Servicios en línea

PID_00270021

Rubén Mondéjar Andreu
Carles Pairot Gavaldà

Tiempo mínimo de dedicación recomendado: 4 horas

Universitat Oberta de Catalunya

Rubén Mondéjar Andreu **Carles Pairot Gavaldà**

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Javier Cánovas Izquierdo (2020)

Primera edición: febrero 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

Introducción..... 5

Objetivos..... 6

1. Plataformas de servicios en línea..... 7
1.1. Características principales 7
1.2. Estadísticas 8
1.3. Logros (*achievements*) 10
1.4. Clasificaciones (*leaderboards*) 12
1.5. Persistencia en la nube (*cloud saving*) 13
1.6. *Matchmaking*..... 14
1.7. Otros servicios 16

2. *Cloud computing*..... 18
2.1. Descripción conceptual 18
2.2. Características de los sistemas *cloud*..... 20
2.3. Integración con sistemas *cloud*..... 21
2.3.1. El estándar JSON 22
2.3.2. El protocolo *WebSocket*..... 23
2.3.3. *Server-Sent Events* (SSE) 25
2.3.4. La arquitectura REST 26

3. Proyecto: *Tanks!* En línea..... 27
3.1. PlayFab 27
3.1.1. Servicios orientados al juego 28
3.1.2. Análisis en tiempo real 30
3.1.3. *LiveOps*..... 31
3.1.4. Instalación del PlayFab SDK 32
3.1.5. Utilizando el servicio de *login* de jugadores 34
3.1.6. Utilizando el servicio de estadísticas de jugadores 38
3.1.7. Llamando a la API de PlayFab mediante *Postman*..... 43
3.2. Soluciones a los retos planteados 45

Resumen..... 55

Bibliografía..... 57

Introducción

Después de estudiar y trabajar el mundo de los juegos multijugador, nos queda ver un paso más. Desde hace años, existen plataformas que nos permiten conectar nuestros juegos y disfrutar de una serie de servicios para complementar nuestros proyectos, sean o no juegos multijugador.

Estos servicios son muy conocidos hoy en día y no es difícil encontrar juegos que nos ofrezcan, por ejemplo, una serie de retos a superar en el juego o compartir nuestros logros con amigos.

Por otra parte, es importante conocer cómo conseguimos desplegar esa lógica remota. Aquí es donde entran en juego todos los conceptos y tecnologías del lado servidor: los *backends*, los sistemas distribuidos, las bases de datos, la computación en la nube, etc. Veremos estos conceptos y como los clientes (los propios videojuegos) se integran con ellos.

Para ilustrar mejor todo lo visto en este módulo, trabajaremos con un proyecto basado en la integración con una plataforma *cloud* real, estudiando los servicios de registro de jugadores y el servicio de estadísticas que ofrece. Esto nos permitirá aprender cómo integrar otras funcionalidades o servicios en línea para cualquier tipo de videojuego.

Objetivos

En este módulo didáctico se presentan al estudiante los conocimientos necesarios para conseguir los objetivos siguientes:

1. Estudiar las plataformas de servicios en línea orientados a los videojuegos.

2. Estudiar el concepto de *cloud computing* y cómo aplicarlo en los videojuegos.

3. Integrar las funcionalidades de registro de usuarios y estadísticas en un juego multijugador que ofrece una plataforma real de tipo *cloud* con el motor Unity.

1. Plataformas de servicios en línea

Podemos afirmar sin riesgo a equivocarnos que la mayoría de los jugadores habituales de hoy en día tienen perfiles en las plataformas más conocidas como Steam, Xbox Live o PlayStation Network. Estas plataformas ofrecen servicios con muchas características, tanto para los jugadores como para los propios juegos, incluyendo *matchmaking*, estadísticas, logros, tablas de clasificación y partidas en la nube, entre otros.

Debido a que el uso de estos servicios en los videojuegos actuales es tan frecuente, los jugadores esperan que cada juego, incluso los de un solo jugador, se integre con uno de estos servicios de alguna manera. En este apartado revisaremos los servicios más usuales e interesantes que podemos encontrar.

1.1. Características principales

Con tantas opciones, vale la pena considerar en qué plataforma integrar nuestros juegos. En la mayoría de los casos, la elección se hace basándose en la plataforma en la que se lanza el juego. Por ejemplo, todos los juegos de Xbox One deben estar integrados con el servicio de videojuegos de Xbox Live, aunque en este caso concreto también se nos permite integrarlo en otras plataformas con Windows 10. En este sentido, tenemos varias opciones para ordenadores basados en los sistemas operativos más conocidos, como Windows, MacOS y Linux. Pero sin duda, el servicio más popular en estas plataformas hoy en día es el servicio de Valve Software, Steam. En cambio, en plataformas móviles, tenemos principalmente Game Center para iOS y el Google Play Games para Android e iOS.

Antes de escribir cualquier código específico para un servicio de jugador, se debe estudiar cómo integrar el código en nuestro videojuego. La opción más inmediata podría ser agregar directamente llamadas al código de servicio del jugador donde sea necesario. En nuestro caso, llamaríamos directamente a las funciones de Steamworks SDK en todos los archivos que necesiten usar el servicio de jugador. Sin embargo, esto se desaconseja por un par de razones:

- En primer lugar, esto significa que cada desarrollador del equipo puede necesitar tener un cierto nivel de familiaridad con Steamworks, porque el código que lo utiliza se extenderá a través de su base de código.

- En segundo lugar, y lo que es más importante, esto hace que sea mucho más difícil integrar un servicio de jugador diferente en el juego. Esto es particularmente una preocupación para los juegos multiplataforma, ya que,

como se ha comentado, las diferentes plataformas tienen diferentes restricciones sobre las que se puede utilizar el servicio de jugador.

Las principales características que proporcionan plataformas como Steam son las siguientes:

- **Autenticación**: se proporciona una variedad de diferentes API para administrar la autenticación y la propiedad del usuario.

- **Comunidad**: La API de la comunidad es un conjunto de funcionalidades que permiten acceder a información sobre otros jugadores, incluyendo pero estando limitado a: nombre de usuario, avatar y grupos a los que pertenece actualmente.

- **Estadísticas y logros**: proporciona un método fácil y eficaz de almacenar estadísticas de juego persistentes e información relativa a los logros.

- **Marcadores**: proporciona un conjunto sólido de API enfocadas a que los jugadores puedan ver el marcador o tablas de clasificación de cada juego.

- **Partidas en línea**: Steam Cloud proporciona la forma más sencilla de sincronizar los datos de los juegos guardados en la nube, permitiendo a sus jugadores mantener su progreso en el juego sin molestias al cambiar entre dispositivos o incluso después de un desagradable bloqueo de la computadora.

- ***Matchmaking***: Steamworks proporciona un excelente conjunto de herramientas para *matchmaking* multijugador perfecto tanto para juegos basados en servidores como para juegos orientados al *lobby*.

- ***Networking***: Se proporciona una capa de abstracción de red para tomar la difícil logística de enviar datos a través de Internet.

- **Sistemas antitrampa**: Valve Anti-Cheat proporciona una capa adicional de seguridad en sus competitivas experiencias multijugador. Es muy similar a un escáner de virus y tiene una lista de trucos conocidos que se pueden detectar.

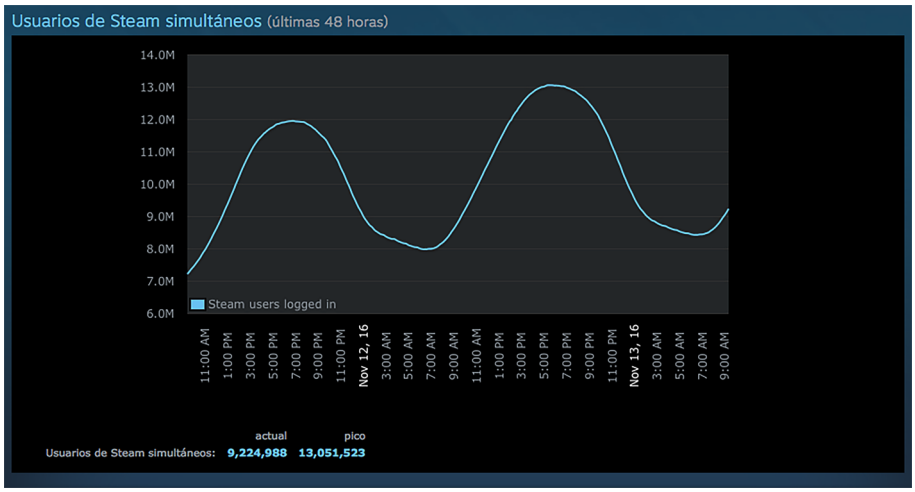
Steamworks.NET fue diseñado para seguir lo más cerca posible de la API nativa Valve C++ API y cuenta con una cobertura del 100 % de la API Steamworks nativa en todas las interfaces.

1.2. Estadísticas

Una característica muy típica y bastante genérica de los servicios en línea es la capacidad de recopilar y mostrar diferentes estadísticas de los jugadores. De esta forma, es posible navegar por nuestro perfil o el de un amigo y ver cómo

se ha completado cada juego; y plataformas como Steam o Google Play Games incluso permiten visualizar estadísticas globales, como por ejemplo el número de jugadores de cada juego durante un periodo de tiempo concreto.

Figura 1. Estadísticas de jugadores simultáneos en Steam



Para tener acceso a estos datos, normalmente hay alguna forma de consultar al servidor para ver las estadísticas del jugador, así como una forma de actualizar y escribir nuevos valores en el servidor. Aunque es viable utilizar siempre el servidor como fuente principal de los datos, de forma que escribimos y leemos la información en remoto, generalmente es una buena idea almacenar en caché los valores en la memoria local del juego.

Todos estos datos generados por los jugadores durante el juego y almacenados en los servidores sirven para el análisis de juegos; y para cada juego en concreto se pueden definir con flexibilidad qué datos se deben recopilar, desde los más genéricos, como horas de juego, progreso actual o puntuación total; hasta detalles más específicos de cada juego, como el porcentaje de uso de *ítems*, armas, tipo de enemigos eliminados, distancia recorrida, carreras ganadas, etc. A partir de estos datos se pueden extraer métricas como la frecuencia de determinados aspectos: cómo los jugadores usan un elemento en particular, cuándo alcanzan un cierto nivel o si realizan alguna acción específica del juego.

De cara al desarrollador, estos datos pueden ser de una gran importancia para mejorar el juego. Por ejemplo, se puede ajustar el nivel de dificultad de ciertos niveles donde los jugadores se encuentran en general en situaciones extremas, ya sea porque el nivel es demasiado difícil o, todo lo contrario, demasiado fácil de completar.

Además de la información individual, estas plataformas presentan diferentes tipos de estadísticas más globales tanto públicas como privadas:

- **Estadísticas globales**: número de jugadores concurrentes, totales, picos del día en los cien juegos con más jugadores.

- **Estadísticas de *hardware* y *software***: el proceso de recolección de datos sobre qué tipo de *hardware* y *software* están usando los jugadores, ya sea de forma automática o por medio de encuestas, ayuda a los desarrolladores a saber quiénes son técnicamente sus jugadores.

- **Geolocalización de los jugadores**: de dónde proceden los jugadores es primordial para poder realizar una estrategia de marketing adecuada, así como planificar un lanzamiento escalonado, en próximas actualizaciones o versiones del juego.

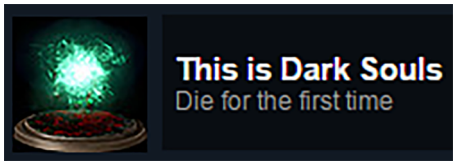
- **Compras *in-game***: saber cuánto y quién ha gastado dinero en un juego es muy importante si el sistema de monetización utilizado se basa en este tipo de estrategia comercial.

- **Informes personalizados**: este servicio también puede generar informes elaborados con toda la información recopilada de cada jugador, como sus logros más difíciles desbloqueados, total de horas jugadas, juegos más jugados, horario favorito para jugar, etc. y enviarlo de forma automática para dar un valor añadido a la plataforma a los ojos del jugador.

1.3. Logros (*achievements*)

Otra característica popular de los servicios de los jugadores son los logros. Aunque muchos otros juegos individuales desarrollan sus propios retos o secretos de forma interna, la primera implementación de un sistema de logros de fácil acceso y multijuegos fue el Xbox Live, con el Xbox 360 Gamerscore, introducido por Microsoft en 2005. Este sistema fue adaptado por otras plataformas y ha ido evolucionando durante estos años, aunque aún destaca por su simplicidad, y, a día de hoy, sigue siendo muy común y demandado por los jugadores habituales.

Figura 2. Logro



Logro

También conocido como trofeo, distintivo, premio, sello, medalla o desafío, siendo un metaobjetivo definido fuera de los parámetros de un juego. Esta funcionalidad intenta incentivar al jugador para alargar la vida útil de los juegos.

Su funcionamiento está muy ligado a cada juego y consiste en que al jugador se le desbloquean en su perfil ciertas hazañas después de conseguir superarlas. Algunos ejemplos de logros incluyen eventos individuales, como derrotar a un

jefe final o acabar el juego en un nivel de dificultad concreto. Otros logros se dan como una estadística acumulativa en el tiempo, por ejemplo, ganar diez combates consecutivos.

Trofeos platino

Una variante de los logros completos fueron los trofeos platino introducidos en PSN, que de una forma más visible muestran que el jugador ha completado todos los logros del juego.

Los logros son una técnica para intentar que el jugador entienda mejor todo lo que le ofrece el juego, animar a los jugadores a explorar con más énfasis o probar estilos de juego completamente diferentes. Los logros permiten a los jugadores comparar su progreso entre sí y fomentar la competitividad entre ellos.

Para entender cómo funcionan los logros, es necesario revisar los elementos básicos asociados a cada uno:

- **Identificador**: es una cadena única generada por la plataforma para referirse al logro en sus clientes de juegos.
- **Nombre**: es un nombre corto del logro (por ejemplo, «Sin Rival»).
- **Puntos**: número de experiencia que se añadirá al perfil del jugador.
- **Descripción**: es una descripción concisa acerca del logro para ayudar al jugador a entender de qué se trata (por ejemplo, «acaba el juego en la dificultad más alta»).
- **Icono**: imagen asociada al logro.

La lista de logros de un juego muestra normalmente el icono, los puntos y el nombre de cada logro para los que han sido desbloqueados; en cambio, los logros bloqueados u ocultos aparecen con menor información.

Cada juego tiene una puntuación máxima (por ejemplo, 1.000 puntos, donde cada logro aporta una parte —entre 5 y 200 puntos, por ejemplo—).

Por otro lado, los logros pueden ser designados como estándares o incrementales. Generalmente, un logro incremental implica que un jugador progrese gradualmente hasta conseguir el logro durante un periodo de tiempo más largo. A medida que el jugador avanza hacia el logro incremental, el juego debe informar del progreso parcial del jugador a la plataforma de servicios. En cualquier caso, una vez informada la plataforma, esta se encarga de validar el logro y retornar la respuesta en caso afirmativo al juego, para que este pueda notificar al jugador en el momento que consigue el logro y pueda progresar al siguiente nivel.

1.4. Clasificaciones (*leaderboards*)

Las **tablas de clasificación** o **marcadores** son una forma de proporcionar clasificaciones para ciertos aspectos de un juego, como por ejemplo, una puntuación o tiempo para completar un nivel en particular.

En general, las tablas o marcadores pueden ser consultados tanto en términos de un *ranking* como en rangos con filtros concretos, por ejemplo, en relación con los amigos del jugador en la misma plataforma.

Por ejemplo, los marcadores en Steam, pueden crearse a través del sitio web de gestión para desarrolladores de Steamworks o pueden crearse mediante programación a través de una llamada de su SDK. Esta forma de trabajar es muy común en el resto de las plataformas.

El proceso típico funciona de la siguiente manera. Al final de un juego (o en un momento que se haya determinado), el juego envía la puntuación del jugador a una o más tablas de clasificación que se hayan creado. Los servicios de juego comprueban si este puntaje es mejor que la entrada de clasificación actual del jugador para la puntuación diaria, semanal o de todos los tiempos. Si es así, los servicios de juegos actualizan las tablas de clasificación correspondientes con la nueva puntuación.

Para recuperar los resultados de un jugador para una tabla de clasificación, se puede solicitar un marco de tiempo (diario, semanal o de todos los tiempos) y especificar si el usuario desea ver una tabla de clasificación social o pública. El servicio de juegos realiza todo el filtrado necesario y, a continuación, envía los resultados al cliente.

Un aspecto interesante de las tablas de clasificación es la posibilidad de asociar o subir contenido generado por el juego asociado con una entrada de tabla de clasificación. Por ejemplo, un *speedrun* de un nivel podría tener una captura de pantalla asociada o un video mostrando cómo se ha conseguido.

Speedrun

Es una competición cuyo objetivo principal es acabar un videojuego lo más rápido posible, generalmente en dificultad máxima.

Alternativamente, un juego de carreras o aventuras podría proporcionar corredores fantasmas de otros jugadores, pudiendo descargarse y competir contra ellos. Esto permite formas de hacer que las tablas de clasificación sean más interactivas que simplemente listar las mejores puntuaciones.

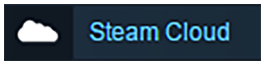
Figura 3. Marcadores y modo Time Trial con fantasma de *Mirror's Edge*

Las tablas de clasificación pueden ser otra forma de impulsar la competencia entre los jugadores, tanto para los aficionados más *hardcore*, que lucharán por el primer puesto en una clasificación pública, como para los jugadores más *casual*, que estarán interesados en comparar su progreso entre sus amigos.

1.5. Persistencia en la nube (*cloud saving*)

El servicio ofrece una manera simple de guardar la progresión de los jugadores de forma remota. En concreto, este servicio permite sincronizar los datos de juego de un jugador en varios dispositivos. Al integrar nuestro videojuego con este servicio, nos permite recuperar los datos guardados para los jugadores, que pueden continuar con el juego en su último punto de guardado.

Figura 4. Un ejemplo de persistencia en la nube es el servicio de Steam Cloud



Por ejemplo, si un juego se ejecuta en una consola, se puede usar el servicio de guardados para permitir que un jugador inicie su juego en otra consola o incluso un ordenador, siempre y cuando ambos permitan conectarse al mismo servicio (por ejemplo, Xbox Live) y luego continuar jugando sin perder ningún

progreso. Este servicio también se puede utilizar para asegurar que el juego de un jugador continúa desde donde lo dejó, incluso si su dispositivo se pierde, se rompe, o lo renueva por uno más moderno.

El espacio que acostumbran a ocupar tiende a ser reducido, por lo tanto, este servicio no tiene un coste adicional por cantidad de datos guardados en la nube. Normalmente, este espacio ocupado se asocia a la cuota del jugador, que está dimensionada convenientemente.

Los juegos pueden seguir trabajando con su forma tradicional de leer y escribir las partidas en local, especialmente cuando el dispositivo del jugador está sin conexión. La idea es que el juego, al detectar la conexión con la plataforma, ejecute un proceso de sincronización para que de forma asíncrona los datos del juego sean consistentes en los servidores remotos.

Obviamente, al utilizar este tipo de servicio, el juego puede encontrar conflictos a la hora de guardar datos, ya que el jugador puede haber jugado en un dispositivo *offline* a la vez que lo hacía en otro dispositivo en línea. En general, la mejor manera de evitar conflictos de datos es cargar siempre los datos más recientes del servicio cuando la aplicación se inicia o se reanuda y se guardan los datos en el servicio con una frecuencia razonable. Sin embargo, no siempre es posible evitar conflictos de datos.

El juego debe hacer todo lo posible para manejar los conflictos de manera que los datos de sus usuarios se conserven y que tengan una buena experiencia. Esta situación también nos la podemos encontrar con otros servicios como marcadores o logros, de forma que se pueda concentrar la sincronización de datos en el mismo proceso, aunque naturalmente es mucho más crítico en el caso de las partidas al tratarse de información más sensible y específica.

1.6. *Matchmaking*

Este mecanismo está muy ligado a los juegos multijugador donde se requiere emparejar o agrupar a los jugadores para realizar partidas equilibradas o no tanto, públicas o privadas, etc. El flujo básico de preparación para jugar un juego multijugador es aproximadamente como sigue:

- El juego busca un *lobby* basado en parámetros personalizables del juego. Estos parámetros pueden incluir modos de juego o incluso nivel de habilidad (si se realiza *matchmaking* basado en habilidades).

- Si se encuentran uno o más *lobbies* adecuados, el juego selecciona uno automáticamente o el jugador puede seleccionar de una lista. Si no se encuentra ningún *lobby*, el juego puede elegir crear uno para el jugador. En cualquier caso, cuando se ha encontrado o creado el *lobby*, el jugador puede unirse a él.

- En el *lobby*, es posible configurar aún más los parámetros del próximo juego, como personajes, mapa, etc. Durante este periodo, otros jugadores probablemente se unirán al mismo *lobby*. También es posible enviar mensajes de chat entre sí, mientras los jugadores están en el mismo *lobby*.

- Una vez que el juego está listo para comenzar, los jugadores se unen a su juego y salen del *lobby*. Normalmente, esto implica la conexión a un servidor de juego (ya sea un servidor dedicado o un servidor de juego). En el caso de no tener servidor, los jugadores empiezan a comunicarse entre sí mediante conexiones *peer-to-peer* antes de salir del *lobby*.

Servidor de juego

Un servidor de juego se encarga de simular el mundo del videojuego permitiendo que los clientes conectados a él mantengan consistente su propia versión del mundo. Así pues, el servidor transmite a los clientes los datos sincronizados a tal efecto. Por el contrario, los **servidores de juego dedicados** se encargan de simular el mundo sin dar soporte a dispositivos de entrada y salida, excepto para su administración. En este caso, pues, los jugadores **no** pueden jugar directamente en un servidor de juego dedicado. Tendrán que hacerlo en alguno de los clientes conectados a este.

A continuación, vamos a estudiar un sistema concreto de *matchmaking* para entender en qué consiste, qué particularidades o técnicas se han definido y cómo se aplica.

Equilibrio en *matchmaking*

Predecir la probabilidad de cada resultado del juego mejora la competitividad *matchmaking*, lo que permite reunir equipos con habilidades más equilibradas.

Tomando como base el sistema de puntuación Glicko, un método para evaluar la habilidad de un jugador en juegos de habilidad, el conocido sistema TrueSkill presenta un algoritmo bayesiano de clasificación y sistema de *matchmaking* desarrollado por Microsoft Research y establecido en los servicios en vivo de Xbox 360. Con TrueSkill, un jugador recién llegado a la plataforma puede clasificarse correctamente en menos de veinte partidas. En TrueSkill el rango de un jugador se representa como una distribución normal $\aleph$ caracterizada por un valor medio de μ (habilidad percibida) y una varianza de σ (la seguridad depositada en el valor μ).

TrueSkill

Utiliza un modelo matemático de incertidumbre para abordar debilidades en sistemas de clasificación existentes.

Cuando se produce una coincidencia, el sistema intenta agrupar individuos según su nivel de habilidad estimado. Si dos individuos compiten cara a cara y tienen el mismo nivel de habilidad estimado con una incertidumbre de estimación baja, cada uno tiene aproximadamente una probabilidad del 50 % de ganar una partida. De esta manera, el sistema intenta hacer cada partida tan competitiva como sea posible.

Con el fin de evitar el abuso del sistema, la mayoría de los juegos clasificados tienen opciones relativamente limitadas para *matchmaking*. Por diseño, los jugadores no pueden jugar fácilmente con sus amigos en los juegos clasificados. Sin embargo, estas contramedidas pueden fracasar debido a técnicas como cuentas alternativas y fallos del sistema en las que cada sistema tiene su propia calificación TrueSkill individual. Para proporcionar juegos menos competitivos, el sistema permite también juegos con partidas sin clasificar, donde se emparejan jugadores de cualquier nivel de habilidad, sin que estos contribuyan a la calificación de TrueSkill.

1.7. Otros servicios

Aparte de todos estos servicios, podemos encontrar algunos similares o más bien específicos de red, que suelen diferir más dependiendo de la plataforma escogida. En este último apartado veremos brevemente en qué consisten los servicios que proporcionan un envoltorio para la comunicación en red entre los jugadores. En el caso de Steamworks o Google Play Games, se proporcionan una serie de funciones para enviar paquetes a otros jugadores creando una red *peer-to-peer*.

Concretamente, esta vez pondremos como ejemplo la API multijugador en tiempo real de Google Play Games. Esta es lo suficientemente flexible como para que podamos utilizarla para reescribir nuestro juego y hacer que todas las instancias envíen los datos a través de la red *peer-to-peer* subyacente.

Un tema importante a tener en cuenta es que para designar al único cliente que actúa como *host* y establecer su autoridad en los datos del juego, este será el encargado de centralizar la comunicación y transmitir datos a los demás participantes conectados a través del sistema de mensajería de datos de Google Play Games.

Además, una vez superada la fase de *matchmaking* para crear la partida, si la lógica de juego se basa en la existencia de un «anfitrión» o «propietario» del juego, será el juego y no la plataforma, el responsable de implementar el algoritmo para determinar quién es el *host*.

Elección de líder

Para escoger al *host* se puede hacer uso de un algoritmo de elección de líder.

Si reescribimos nuestro juego para ello, podemos utilizar los servicios de juegos de Google Play para transmitir datos a los participantes en una partida o permitir a los participantes intercambiar mensajes entre sí. Como en otros servicios, los mensajes de datos pueden enviarse utilizando un protocolo de mensajería fiable o poco fiable proporcionado por los servicios de la plataforma:

- **Mensajería fiable**: la entrega de datos, la integridad y el orden están garantizados. Podemos elegir notificar el estado de entrega mediante una devolución de llamada. La mensajería fiable es adecuada para enviar datos no sensibles al tiempo. También se puede usar mensajería fiable para enviar grandes conjuntos de datos donde los datos se pueden dividir en segmen-

tos más pequeños, son enviados a través de la red y luego reensamblados por el cliente receptor. Este tipo de mensajería puede tener latencia alta.

- **Mensajes poco fiables**: el cliente del juego envía los datos una sola vez (*fire-and-forget*) sin garantía de entrega de datos o datos que lleguen en orden. Sin embargo, la integridad está garantizada, por lo que no es necesario agregar un código de verificación o *checksum*. Este tipo de mensajería tiene una latencia baja y es adecuada para enviar datos sensibles al tiempo. Su aplicación es responsable de garantizar que el juego se comporte correctamente si los mensajes se descartan en transmisión o se reciben fuera de orden.

Obviamente, si un cliente que envía datos no está conectado a los servicios de juego de Google Play o el destinatario no está conectado, el mensaje no se entregará. Para conservar las transmisiones de mensajes y evitar exceder los límites de velocidad, las prácticas recomendadas para enviar datos son las siguientes:

- Enviar mensajes solo a los participantes que requieran esa información, en lugar de transmitir a todos los participantes. Si está enviando un mensaje de difusión, hay que excluir al participante del remitente de la lista de destinatarios de difusión.

- Si se están enviando datos utilizando el protocolo de mensajería fiable, hay que mantener la frecuencia de las transmisiones de mensajes por debajo de un límite razonable (por ejemplo, cincuenta mensajes por segundo). Si se necesita enviar datos con más frecuencia, es recomendable utilizar mensajería no fiable.

2. *Cloud computing*

Desde hace unos años y gracias a la computación en la nube, se ha facilitado mucho a los desarrolladores poder hospedar, gestionar y mantener servicios remotos. Incluso los pequeños estudios pueden permitirse sus propios servidores dedicados. En este apartado veremos en qué consiste este paradigma, y de qué forma podemos aprovecharnos de él para que nuestro juego, parte de él o servicios asociados se ejecuten en servidores remotos.

2.1. Descripción conceptual

Las plataformas en la nube han tomado mucha relevancia estos años gracias a sus ventajas inherentes, de manera que han transformado los servicios de *hosting* tradicionales en otra forma de venderse al gran público, en parte gracias a proporcionar un servicio con bajos costes de gestión y administración de las infraestructuras de *hardware*.

Otra razón importante para adoptar los sistemas en la nube es su capacidad para escalar rápidamente en entornos de rápido crecimiento con cargas de trabajo masivas. Alcanzar la escalabilidad deseada de forma flexible es una tarea compleja, que implica resolver muchos problemas concurrentes, como mantener el equilibrio de carga de los recursos, políticas de replicación y coherencia, y escalabilidad horizontal de almacenes de datos persistentes.

En estos casos, los proveedores de Cloud deben ofrecer *frameworks* con API específicas para servicios en la nube como por ejemplo Microsoft Azure, Amazon EC2 o Google App Engine. Las tecnologías de la nube comenzaron a emerger cuando Amazon irrumpió estelarmente en el mercado en 2006 ofreciendo sus servicios web. De hecho, Amazon se ha convertido en un estándar de nube de facto, aunque ahora muchas otras compañías proporcionan soluciones en un ecosistema empresarial muy activo. Pero existen diferentes tipos de nubes y formas de diferenciarlas según su naturaleza.

De acuerdo con la definición ampliamente utilizada por el NIST (National Institute of Standards and Technology):

> El *cloud computing* es un modelo para habilitar el acceso a la red de forma omnipresente, conveniente y bajo demanda a un conjunto compartido de recursos computacionales configurables (por ejemplo, redes, servidores, almacenamiento, aplicaciones y servicios) que se pueden aprovisionar y liberar rápidamente con un mínimo esfuerzo de gestión o interacción con el proveedor del servicio.

Dependiendo de la capacidad proporcionada al consumidor podemos distinguir entre tres modelos de servicio:

- ***Software* como Servicio (SaaS)**: utilizar las aplicaciones del proveedor que se ejecutan en una infraestructura en la nube. Las aplicaciones son accesibles desde varios dispositivos cliente a través de una interfaz de cliente ligero, como un navegador web (por ejemplo, correo electrónico basado en web) o una interfaz de programa. El consumidor no gestiona ni controla la infraestructura subyacente de la nube, incluyendo la red, los servidores, los sistemas operativos, el almacenamiento o incluso las capacidades de las aplicaciones individuales, con la posible excepción de los ajustes de configuración específicos de la aplicación del usuario.

- **Plataforma como Servicio (PaaS)**: desplegar en la infraestructura de nube las aplicaciones creadas o adquiridas por el consumidor utilizando lenguajes de programación, bibliotecas, servicios y herramientas soportadas por el proveedor. El consumidor no gestiona ni controla la infraestructura subyacente de la nube, servidores, sistemas operativos o almacenamiento, pero tiene control sobre las aplicaciones desplegadas y posiblemente configuraciones para el entorno de hospedaje de aplicaciones.

- **Infraestructura como Servicio (IaaS)**: proveer procesamiento, almacenamiento, redes y otros recursos de computación fundamentales donde el consumidor es capaz de desplegar y ejecutar *software* arbitrario, que puede incluir sistemas operativos y aplicaciones. El consumidor no gestiona ni controla la infraestructura subyacente de la nube, sino que tiene control sobre los sistemas operativos, el almacenamiento y las aplicaciones implementadas, y posiblemente un control limitado de componentes de red selectos (por ejemplo, *firewalls* de *host*).

Según el uso de la infraestructura de la nube, podemos clasificar los modelos de implementación en los cuatro siguientes:

- **Nube privada**: uso exclusivo por parte de una única organización que comprende múltiples consumidores (por ejemplo, unidades de negocio). Puede ser de propiedad, administrada y operada por la organización, un tercero o una combinación de ellos, y puede existir dentro o fuera de las instalaciones.

- **Nube pública**: uso abierto por el público en general. Puede ser de propiedad, administrada y operada por una organización comercial, académica o gubernamental, o alguna combinación de ellas. Se sitúan físicamente en las instalaciones del proveedor de la nube.

- **Nube híbrida**: es una composición de dos o más infraestructuras de nube distintas que siguen siendo entidades únicas, pero que están unidas por una tecnología estandarizada o propietaria que permite la portabilidad de datos y aplicaciones.

2.2. Características de los sistemas *cloud*

El caso *Pokémon GO*

Aún a fecha de hoy, es complicado estimar el número de usuarios potenciales que puede atraer un juego. De hecho, en 2016, Niantic subestimó en una escala de magnitud (x10) los jugadores de *Pokémon GO*. En mayo de 2018 la cifra de usuarios activos de este juego era de 147 millones.

En los primeros días de los juegos en línea, el alojamiento de los propios servidores dedicados era tarea muy compleja, empezando por la adquisición y mantenimiento de grandes cantidades de *hardware*, infraestructura de redes y personal de TI. La estimación de jugadores en el lanzamiento era sumamente importante, ya que una infraestructura sobredimensionada podía significar un coste adicional imposible de mantener. Pero peor aún era quedarse corto y que los jugadores pagasen sin poder conectarse debido a restricciones de procesamiento y ancho de banda.

Pero como hemos visto, la nube ha solventado estos problemas. Gracias al abundante poder de procesamiento disponible de los gigantes de Internet, como Amazon, Microsoft y Google, las compañías de juegos son capaces de manejar estos costes de forma mucho más razonable. Otros servicios de terceros, como Heroku, facilitan el despliegue proporcionando servicios de administración de servidores y bases de datos según sea necesario. Además, existen servicios orientados directamente a videojuegos, como es el caso de Unity Cloud Services o Photon Cloud, que nos permiten hospedar los juegos con servicios más orientados y dedicados a las necesidades concretas de los videojuegos.

A pesar del reducido coste inicial en el lado del servidor, seguimos teniendo algunos inconvenientes potenciales que debemos considerar:

- **Complejidad**: utilizar un grupo de servidores dedicados es una tarea compleja, a pesar de que la nube proporciona la infraestructura y parte del *software* de administración. Pero además se requiere código personalizado de administración de procesos y máquinas virtuales, así como adaptarse a las API cambiantes.

- **Coste**: a pesar de que la nube disminuye el coste inicial y de largo plazo de manera significativa, no es totalmente gratuito. El aumento del interés del jugador puede cubrir el aumento del coste, pero depende de cada caso.

- **Dependencia de terceros**: hospedar el juego en los servidores de Amazon, Microsoft o Unity, significa que el destino descansa en sus manos. Aunque las empresas ofrecen acuerdos de nivel de servicio que garantizan un mínimo de tiempo de inactividad, sigue habiendo un riesgo asociado.

- **Cambios inesperados de *hardware***: los proveedores de hosting generalmente garantizan proporcionar *hardware* que cumple con ciertas especificaciones mínimas. Esto no les impide cambiar *hardware* sin previo aviso, siempre y cuando esté por encima de la especificación mínima. Si de repente introducen una configuración de *hardware* extraña que no se ha probado, puede causar problemas.

Aunque estas desventajas pueden ser significativas, los beneficios a menudo las superan:

- **Servidores confiables, escalables y de alto ancho de banda**: en modelos tradicionales no hay garantías de que los jugadores anfitriones sean los que tengan mejor ancho de banda cuando los otros jugadores quieren jugar. Con alojamiento en la nube y un programa de administración de servidores, se puede activar cualquier servidor bajo demanda para realizar este tipo de tareas y asegurar el buen funcionamiento de la partida.

- **Prevención de trampas**: controlando todos los servidores, es fácil asegurarse de que están ejecutando versiones no modificadas y legítimas de los juegos. Esto significa que todos los jugadores obtienen una experiencia uniforme y fiable. Además, permite no solo clasificaciones reales, sino que los juegos puedan utilizar todos los servicios de la plataforma.

- **Protección de copia razonable**: restringir los juegos a ser ejecutados en servidores dedicados proporciona una forma de DRM (*Digital Rights Management*) *de facto*, no intrusiva. El hecho de no liberar ejecutables de servidor dificulta el despliegue de servidores piratas o el desbloqueo de contenido de modo ilegal. También permite comprobar las credenciales de inicio de sesión para cada jugador, asegurándose de que realmente tiene derecho a jugar al juego.

Como conclusión, en estos últimos años los sistemas *cloud* han ido ganando terreno a las soluciones tradicionales de infraestructura dedicada, principalmente por su confiabilidad, escalabilidad y coste asociado. Estos aspectos son cruciales a la hora de proporcionar la mejor experiencia multijugador a los usuarios de nuestros videojuegos en línea.

2.3. Integración con sistemas *cloud*

El desarrollo de servidores *backend* es un campo en evolución constante y veloz, que consta de un conjunto de herramientas que evolucionan rápidamente.

Por suerte, existen muchos lenguajes, plataformas y protocolos diseñados para hacer la vida más fácil al desarrollador de *backend*. Los *frameworks* modernos de desarrollo web pueden ser vistos como un conjunto de componentes que simplifican la programación HTTP, controlando automáticamente los detalles de transporte de bajo nivel.

Además, en los últimos años, se ha puesto de manifiesto que HTTP es también útil para para la creación de una API estándar vía REST, utilizando los verbos (GET, POST, etc.) y códigos (200, 500, etc.) proporcionados, además de otros conceptos como URI y encabezados.

Este estilo arquitectónico denominado RESTful ha demostrado ser una forma eficaz de aprovechar HTTP, aunque ciertamente no es el único enfoque válido. Actualmente, existe una tendencia clara para que los servicios utilicen API REST y datos JSON, junto con *WebSockets* o *Server-Sent Events* (SSE) para complementar los servicios más interactivos.

Estas son herramientas flexibles y ampliamente aceptadas para el desarrollo de servidores. Es viable elegir diferentes herramientas para nuestra plataforma *cloud* de servicios, pero claramente para poder realizar la integración de nuestros videojuegos se requiere una comprensión básica de HTTP, JSON, REST, *WebSockets* y SSE, así como las principales tecnologías web involucradas.

2.3.1. El estándar JSON

A finales de los años noventa y principios de los 2000, el lenguaje XML (*eXtensible Markup Language*) fue anunciado como el formato universal de intercambio de datos que cambiaría la forma de estructurar documentos. Al principio, muchas tecnologías empezaron a utilizarlo, ya que durante ese periodo fue la única opción para compartir datos libremente.

En la actualidad, JSON (*JavaScript Object Notation*) ha ido ganando terreno como lenguaje de intercambio de datos universal, principalmente gracias a su ligereza (su formato es sumamente simple y ocupa menos caracteres que XML) y la rápida velocidad en su procesamiento (se puede interpretar directamente como un objeto JavaScript).

Los datos en JSON se representan en forma de mapa, almacenando pares de información clave/valor, mientras que en XML estos datos se representan como un árbol, lo que implica que su procesado siempre es más tedioso y es menos eficiente en términos de coste espacial (uso de memoria) y temporal.

A efectos prácticos, JSON se utiliza mucho en los documentos asociados a peticiones (*requests*) web (encabezado HTTP especificando el *content-type* de aplicación como JSON) tanto de escritura (creación POST, actualización PUT o actualización parcial PATCH), como de consulta (GET).

La utilidad principal desde el punto de vista de los videojuegos multijugador es facilitar las tareas de serialización de objetos a documentos textuales y viceversa, entre clientes y servidores. A continuación vamos a ver cómo se trabaja con este formato en C#, a partir de la noción de JSON «estructurado», lo que significa que se debe indicar qué variables se van a almacenar en sus datos JSON creando una clase o estructura. Por ejemplo:

```
[Serializable]
public class MyClass {
    public int level;
    public float timeElapsed;
```

```
    public string playerName;
}
```

Definiendo una clase C# simple que contiene una serie de variables (por ejemplo, *level*, *timeElapsed* y *playerName*) y la marcamos como *Serializable*, lo cual es necesario para poder usar el serializador JSON. A continuación, vamos a crear una instancia de esta clase y serializarla a formato JSON mediante la función *JsonUtility.ToJson()*:

```
MyClass myObject = new MyClass();
myObject.level = 1;
myObject.timeElapsed = 47.5f;
myObject.playerName = "Dr Charles Francis";
string json = JsonUtility.ToJson(myObject);
```

Esto resultaría en la variable *json* que contiene la cadena de caracteres:

```
{"level":1,"timeElapsed":47.5,"playerName":"Dr Charles Francis"}
```

Finalmente, para convertir el JSON de nuevo en un objeto, podemos utilizar la función *JsonUtility.FromJson()*:

```
myObject = JsonUtility.FromJson<MyClass>(json);
```

De este modo, podemos comprender la sencillez de trabajar con esta herramienta, siendo mucho más compleja la tarea de enviar y recibir los datos vía HTTP, REST o *WebSockets* que la de codificarlos en formato JSON.

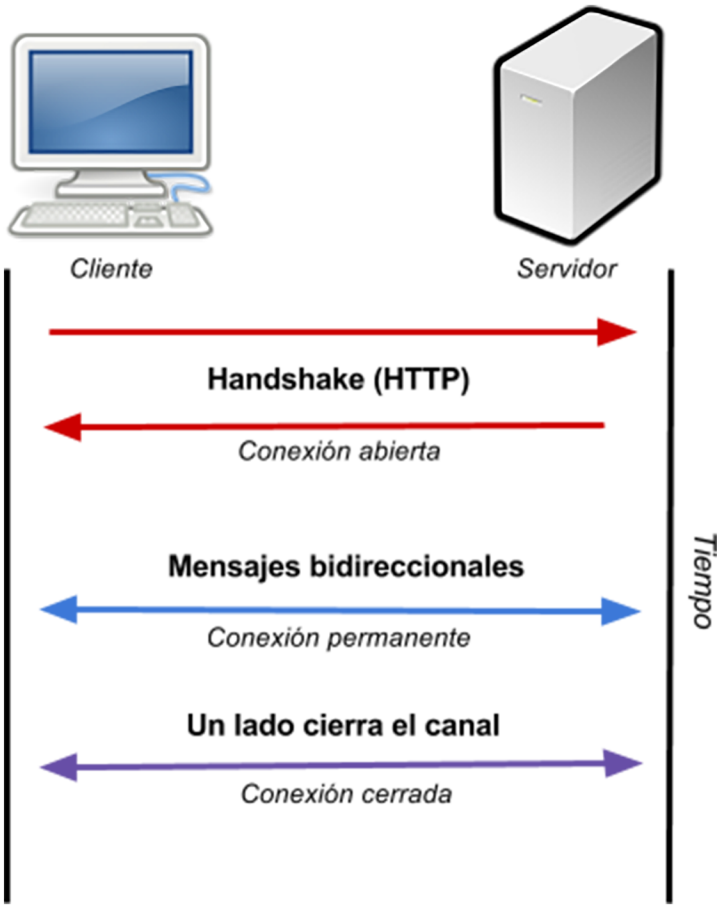
2.3.2. El protocolo *WebSocket*

WebSockets es una extensión de la idea de *socket* tradicional, motivado por las carencias inherentes en el protocolo HTTP, el cual fue diseñado para la World Wide Web y por lo tanto, para ser usado por los navegadores.

Al ser un protocolo específico que funciona de un modo particular, no resulta adecuado para cada necesidad, concretamente nos referimos a cómo HTTP maneja las conexiones, dónde se realiza una conexión por cada solicitud, por ejemplo, para descargar un documento HTML o una imagen. Se abre siempre un puerto/*socket* y se transfieren los datos, cerrando la conexión justo al acabar. Esta continua apertura y cierre de conexiones HTTP puede crear una sobrecarga y para ciertas aplicaciones —especialmente aquellas que requieren respuestas rápidas, interacciones en tiempo real o mostrar flujos de datos— simplemente no funciona.

La otra limitación con HTTP es que pertenece al **paradigma *pull***, en el que es el interesado, en este caso el navegador, quien solicita o extrae información del origen, en este caso los servidores. El problema aquí es que el servidor solo es capaz de enviar datos al navegador en el momento de responder a una solicitud. Esto significa que los navegadores deben consultar al servidor de forma continua para obtener nueva información repitiendo las solicitudes cada tantos segundos o minutos para averiguar si hay nuevos datos.

Figura 5. Diagrama de conexión *WebSocket*



A finales de la década del 2000, surgió un movimiento para añadir *sockets* a los navegadores, siendo en 2011 cuando el protocolo *WebSocket* fue estandarizado. Esto permitió a los desarrolladores usar dicho protocolo, que gracias a su flexibilidad permite crear nuevas formas de comunicación para transferir datos desde y hacia servidores desde el navegador, así como comunicación *peer-to-peer* (P2P) o comunicación directa entre navegadores. La principal diferencia con el protocolo HTTP es que un *WebSocket* permanece abierto al servidor teniendo siempre un canal disponible para la comunicación bidireccional. Esto significa que los datos pueden ser enviados al navegador en tiempo real bajo demanda, cambiando la forma de trabajar del paradigma *pull* al **paradigma *push***.

2.3.3. *Server-Sent Events* (SSE)

Aparte de los *WebSockets*, existe otra metodología para trabajar con el **paradigma *push***, de forma que el servidor envíe datos al navegador: esto es mediante *Server-Sent Events* (SSE).

Hablando de manera estricta, el protocolo HTTP no permite que el servidor envíe información de forma activa. De todas formas, existe una excepción: si el servidor declara al cliente que lo que va a enviar a continuación es un flujo (*stream*) de datos.

En otras palabras, que lo que enviará de forma continua es un *stream* en lugar de un solo paquete de datos. De esta forma, el cliente no cerrará la conexión y se quedará esperando a recibir más datos del *stream* que el servidor le enviará. Un típico ejemplo de esto es la reproducción de vídeo. Esencialmente, el objetivo de este tipo de comunicación es conseguir la descarga de un fichero mediante *streaming*.

SSE utiliza este mecanismo para enviar información de tipo *push* mediante un *stream* al navegador. Evidentemente se basa en el protocolo HTTP y está estandarizado como parte de HTML5 por el W3C (World Wide Web Consortium) y soportado por todos los navegadores excepto Internet Explorer / Edge.

SSE se asemeja bastante a los *WebSockets* en el sentido de que ambos establecen un canal de comunicación entre el navegador y el servidor, y el servidor envía la información al navegador. En líneas generales, los *WebSockets* son más potentes y flexibles, ya que establecen un canal de comunicación *full-duplex*, que permite la comunicación en ambas direcciones; mientras que SSE es un canal unidireccional que solo permite al servidor enviar datos hacia el navegador, porque el *streaming* es esencialmente una descarga. Si el navegador necesita enviar información al servidor, entonces realiza otra petición HTTP.

De todas formas, SSE tiene sus ventajas con respecto a los *WebSockets*:

- SSE utiliza el protocolo HTTP que está soportado por todos los servidores, mientras que *WebSockets* es un protocolo independiente.

- SSE es más ligero y sencillo de utilizar, mientras que el protocolo *WebSocket* es relativamente complejo.

- SSE soporta reconexiones por defecto, mientras que si utilizamos *WebSockets* tenemos que implementar esta funcionalidad nosotros mismos.

- SSE generalmente solo se usa para transferir texto; los datos binarios tienen que codificarse como texto primeramente antes de poderlos transmitir,

mientras que el protocolo *WebSocket* soporta transferencias binarias por defecto.

2.3.4. La arquitectura REST

REST, o *REpresentational State Transfer*, es una propuesta para crear API para aplicaciones de una manera metodológica.

En las aplicaciones web típicas y tradicionales, la creación de *endpoints* REST mediante HTTP es el modo en el que se diseña la gran mayoría de las aplicaciones. Cualquiera de las diversas tecnologías disponibles (por ejemplo, JavaScript) son muy similares en la forma en que reciben las solicitudes de información y luego se responde a estas solicitudes.

REST organiza estas peticiones de forma predecible, usando tipos de operaciones HTTP, o verbos, para construir respuestas apropiadas. Las peticiones se originan en el cliente y los verbos HTTP incluyen GET, POST, PUT, DELETE, entre otros, que corresponden a operaciones esperadas, recuperación de datos, envío de datos, actualización de datos y eliminación de datos, respectivamente.

REST es el modo más extendido de estructurar la API para peticiones web. Fácilmente podemos encontrar ejemplos en muchas aplicaciones web que hacen uso de este sistema y exponen su API para poder realizar peticiones. Un ejemplo típico sería Twitter, que además nos permite ligarlo con *OAuth* para poder utilizar su sistema de usuarios desde nuestra aplicación y acceder posteriormente a sus datos. Esto sería útil para poder tuitear los resultados de nuestras partidas de forma automática desde el juego.

OAuth

Es un sistema que basado en REST para permitir la autenticación usando sistemas de terceros (por ejemplo, Google o Facebook) desde una aplicación.

Pero dado que implica el uso de HTTP, también tiene asociada la sobrecarga de este protocolo. Para la mayoría de las aplicaciones, la información solo necesita ser transferida cuando un usuario realiza una acción. Por ejemplo, al navegar por un sitio de noticias, una vez que el navegador ha solicitado el artículo, el usuario está ocupado leyéndolo y no realiza otras acciones. En estos casos, la práctica habitual suele ser la de cerrar el *socket* para ahorrar recursos en el servidor.

En cualquier caso, si el tipo de interacción es más exigente, como por ejemplo en aplicaciones en tiempo real o transferencias de datos, la combinación HTTP y REST no es la más adecuada y seguramente deberemos optar por otras alternativas, como *WebSockets* o SSE.

3. Proyecto: *Tanks!* En línea

En este proyecto, vamos a integrar el juego *Tanks!* con un conjunto de servicios de una plataforma *cloud* real que proporciona una serie de funcionalidades muy útiles para los videojuegos en línea. La principal motivación de este ejemplo es ver cómo utilizar herramientas actuales para poder añadir funcionalidades remotas en nuestros desarrollos, como comunidades, soporte a retransmisiones en directo, mundos persistentes, algoritmos de física o inteligencia artificial distribuidos, etc. Aunque el propósito del proyecto no es tanto añadir un conjunto de funcionalidades concretas, sirve de justificación para ver cómo integrar nuestro juego con un proveedor de servicios en línea.

3.1. PlayFab

Como hemos visto, actualmente existen diferentes opciones para crear nuestro servidor o servidores, siendo la computación en la nube una de las formas que aportan más flexibilidad y escalabilidad. Sin embargo, el uso de la nube suele venir asociado con un coste económico generalmente mensual, parecido al que encontramos en el *hosting* tradicional, pero más orientado al consumo de la plataforma.

Para este proyecto hemos optado por una solución de tipo *PaaS* (utilizada para desplegar nuestro código), en la que la lógica que requieran nuestros videojuegos se ejecutará de forma remota en los servidores del proveedor, pero disponiendo de ciertas capacidades de administración.

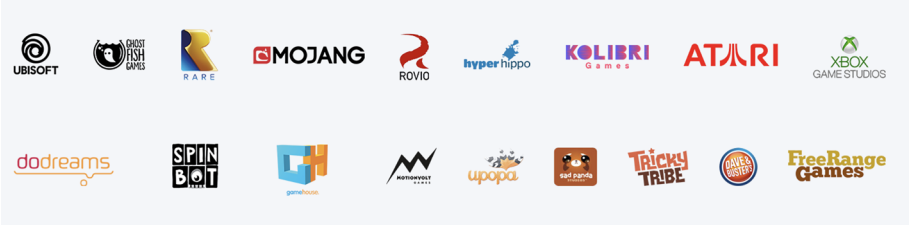
Platform as a Service - PaaS

Plataforma en la nube donde podemos desplegar nuestro código siempre que este esté soportado por el proveedor.

En 2014 se anunció la primera versión del servicio de *backend* para videojuegos **PlayFab** para Unity. Su objetivo era el de proporcionar un conjunto de servicios necesarios para construir videojuegos en línea, así como las herramientas necesarias para gestionar los videojuegos una vez salen al mercado.

Se trata de una plataforma que ha ido evolucionando a lo largo de los años y que goza de un gran éxito por la cantidad y calidad de servicios prestados, así como por su sencillez.

Figura 6. Algunos de los más prestigiosos estudios de videojuegos utilizan PlayFab



En 2018, PlayFab fue adquirida por Microsoft y pasó a denominarse **Microsoft Azure PlayFab**. Actualmente esta plataforma está soportando más de 2.500 juegos y cuenta con más de un billón de cuentas de usuario creadas. Algunos de los juegos que la utilizan son títulos tan conocidos como *Rainbow Six Siege*, *Sea of Thieves*, *Minecraft*, *Angry Birds Seasons*, *Forza Horizon 4* o *Crackdown 3*.

Figura 7. Microsoft Azure PlayFab

Microsoft Azure PlayFab

Microsoft Azure PlayFab proporciona servicios orientados al desarrollo de videojuegos, incluyendo plataformas de distribución de juegos, cuadros de mando para entender cómo está funcionando el juego, servicios para juegos multijugador, monetización, anuncios y comunidades

Monthly Active Users (MAU)

Métrica que utiliza PlayFab, equivalente al número total de jugadores únicos que generan actividad en nuestro juego durante un mes. Si la misma persona juega a nuestro juego en dos dispositivos distintos con una cuenta enlazada (por ejemplo, PC y móvil), cuenta como una única MAU.

Los tipos de licencia de PlayFab incluyen varios planes, siendo el esencial gratuito, pero con limitaciones, entre las cuales no se incluyen características como la segmentación de jugadores, análisis avanzados, tareas programadas, soporte dedicado o acuerdos de nivel de servicio específicos. Aun así, este tipo de licencia es ideal para desarrolladores *amateurs* e *indie* que están empezando y desean probar los servicios que ofrece PlayFab. El resto de los planes son de pago y varían según las necesidades y el número de MAU que utilizan la infraestructura.

Las características más destacadas de PlayFab están divididas en tres áreas, que son **servicios orientados al juego**, **análisis en tiempo real** y **LiveOps**.

3.1.1. Servicios orientados al juego

Los servicios orientados al juego se ponen a disposición del desarrollador para que pueda utilizarlos de una forma sencilla y ágil. De este modo, el desarrollador puede dedicarse al desarrollo del juego propiamente dicho y delega la implementación de funcionalidades que la plataforma ya proporciona de una forma eficiente, eficaz y escalable.

Los servicios incluidos en esta categoría son:

- **Servidores multijugador**. Permiten construir, desplegar y escalar nuestro juego rápidamente en servidores multijugador confiables. Permite escalar dinámicamente de 100 jugadores a 10.000.000 o más, manejando automáticamente los picos de demanda sin impactar en la experiencia de juego. Esto se consigue mediante la utilización de la infraestructura que proporciona Microsoft Azure, que PlayFab utiliza por debajo.

- ***Responsive matchmaking***. Permite ordenar, filtrar y juntar usuarios para conseguir la mejor compatibilidad y diversión. Permite emparejar jugadores con habilidades similares utilizando el algoritmo de *matchmaking* de Xbox Live para incrementar la experiencia de juego del usuario.

- **Autenticación del jugador y cuentas enlazadas**. PlayFab ofrece múltiples opciones de autenticación, de modo que los usuarios pueden utilizar múltiples dispositivos. Algunos de los mecanismos soportados son: *DeviceID* (en el caso de iOS o Android), *username / password*, cuentas de Google, GameCenter, Facebook, Steam, Twitch, etc.

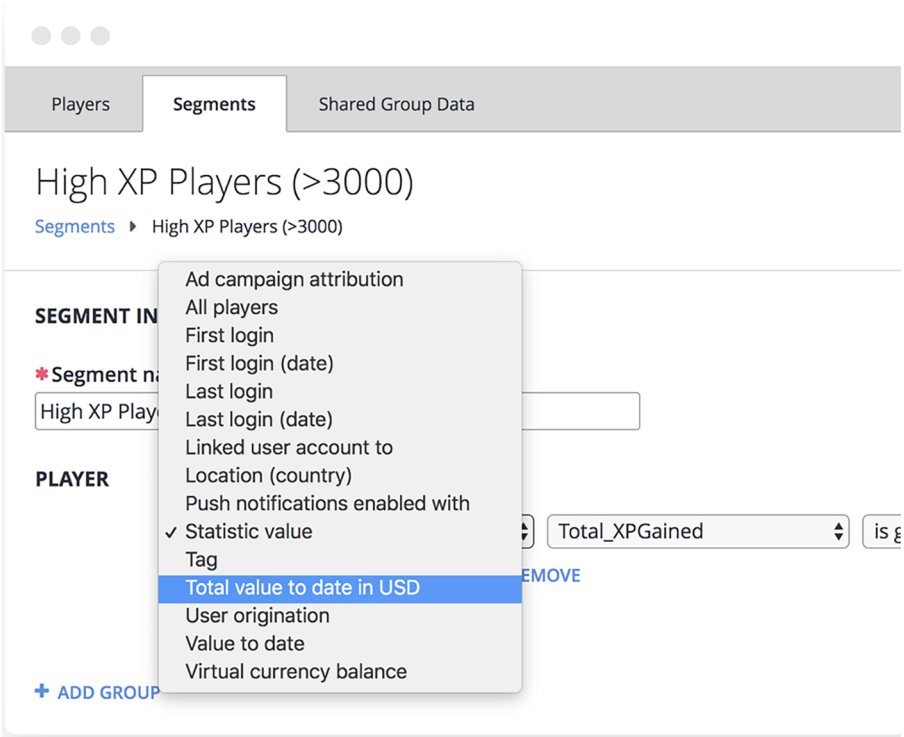
- **Almacenamiento de datos del jugador**. Permite almacenar información asociada con el jugador y el estado del juego en la nube evitando así la pérdida de datos. Se pueden compartir datos entre múltiples juegos y dispositivos, otros jugadores, estadísticas, etc.

- **Segmentación de jugadores**. PlayFab permite ejecutar acciones en tiempo real cada vez que un jugador entra o sale de un segmento.

Segmento

Un segmento permite definir grupos de jugadores y realizar acciones exclusivas en ese grupo. Por ejemplo, nos puede interesar agrupar a todos los jugadores que sobrepasan un umbral de fuerza definido en el juego para ejecutar una acción concreta.

Figura 8. Un ejemplo de segmentación de jugadores



- **Clasificaciones**: se pueden crear clasificaciones de cualquier estadística del jugador, filtrarlas solo por amigos, mostrar la posición actual del jugador o cualquier posición, etc.

- **Caracteres *in-game***: se pueden almacenar múltiples caracteres por jugador. Cada carácter que se configura tendrá sus propios datos, monedas virtuales e inventario.

- **Envío de notificaciones *push***: PlayFab permite enviar manual o automáticamente notificaciones *push* a jugadores individuales o a un segmento concreto.

- **Herramientas de comercio electrónico:** herramientas tales como creación de un catálogo dinámico de ítems, agrupaciones, protección antifraude, envío de regalos o compras entre jugadores, procesado de pagos mediante terceros (Apple Pay, Facebook o PayPal), generación de cupones para cambiarlos por ítems, creación de múltiples monedas virtuales, etc.

3.1.2. Análisis en tiempo real

Este servicio proporciona una serie de cuadros de mando para poder controlar cómo está evolucionando nuestro juego una vez lo hemos puesto en marcha. Esta información es crucial para entender el comportamiento de los jugadores con nuestro juego, permitiendo su mejora y adaptación en versiones posteriores. Los servicios incluidos son:

- **Monitorización de cuadros de mando en tiempo real**. Los datos se refrescan cada 2 minutos.

- **Búsqueda de eventos**. Filtraje a través de la reciente historia de eventos y búsqueda de jugadores específicos, tipos de eventos o condiciones de errores.

- **Exportación de eventos**. Envío de eventos en tiempo real a un *bucket* alojado en Amazon S3, *webhook* o cualquier otro sistema de análisis externo.

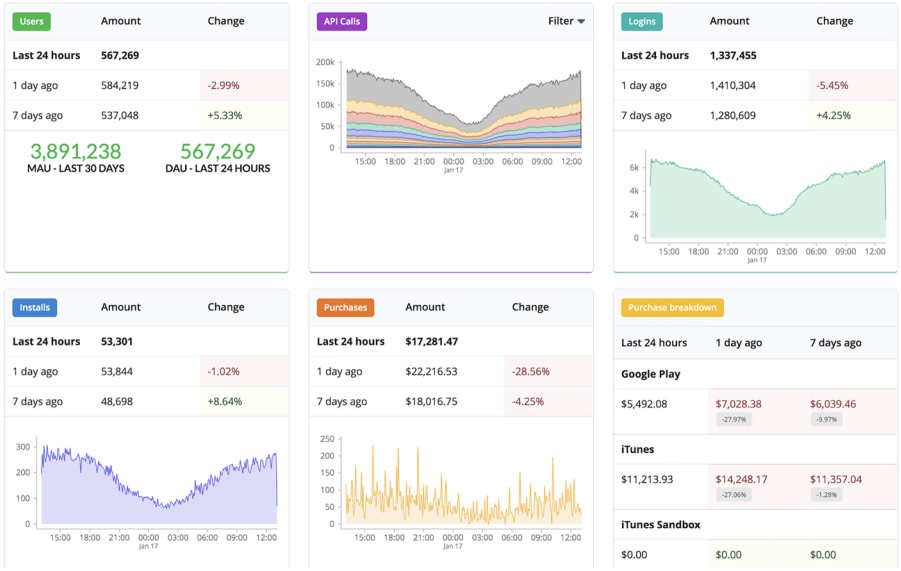
- **Informes de clientes**. Diseñados para proporcionar acceso sencillo a los indicadores clave (*key performance indicators*, KPI) de nuestro juego.

- **Depuración en tiempo real gracias al flujo de eventos**. Permite observar en tiempo real el flujo de eventos de los clientes del juego, servicios de *backend* y extensiones de terceros.

Integración con Amazon S3

PlayFab permite multitud de integraciones con proveedores externos. Por ejemplo, puede exportar los eventos de nuestro juego a un *bucket* alojado en Amazon S3. Esta integración está descrita aquí: https://api.playfab.com/docs/tutorials/ landing-analytics/s3-archive

Figura 9. Ejemplo de los cuadros de mando en tiempo real de PlayFab



3.1.3. *LiveOps*

Mediante el concepto de LiveOps se permite hospedar eventos, ejecutar experimentos y recompensar a los jugadores. Los servicios incluidos son los siguientes:

- **Gestión remota de la configuración del juego**. Se trata de un servicio que permite almacenar la configuración del juego en el servidor para poder modificar el comportamiento del mismo sin actualizar a los clientes.

- **Almacenaje y distribución de ficheros del juego**. Permite actualizaciones remotas de nuestro juego mediante paquetes de nuevos *assets*, DLC (*downloadable contents*) u otros ficheros del juego. El acceso de los clientes del juego se hace globalmente mediante CDN.

- **Disparar acciones según eventos en tiempo real**: se puede recompensar a los jugadores y permitirles un conjunto más rico de acciones enviando por ejemplo notificaciones *push*.

- **Tareas programadas**: se trata de un servicio para programar tareas una única vez o repetidas veces.

- **Publicación de titulares**: se pueden publicar y editar mensajes que verán todos los jugadores.

- **Plantillas de correo electrónico**: se facilita la creación de plantillas de correo electrónico para nuestra base de jugadores del juego. Esto permite dirigirnos a nuestros jugadores, ya sea para responder a sus peticiones, enviarles enlaces que puedan intercambiar por un regalo, etc.

CDN (*Content Distribution Network*)

Se trata de una red distribuida geográficamente de servidores *proxy* y sus centros de datos. Su objetivo es el de proporcionar alta disponibilidad y alto rendimiento distribuyendo el servicio teniendo en cuenta la localización de los usuarios finales.

3.1.4. Instalación del PlayFab SDK

Vamos a empezar a trabajar con PlayFab. Para ello, crearemos un proyecto Unity vacío y nos descargaremos el Unity3D SDK de PlayFab. Nos bajaremos el *asset* **PlayFab Unity Editor Extensions *package***.

Descargar Unity3D SDK

https://docs.microsoft.com/en-us/gaming/playfab/sdks/unity3d/quickstart

Figura 10. Descarga del PlayFab Unity Editor Extensions *package*

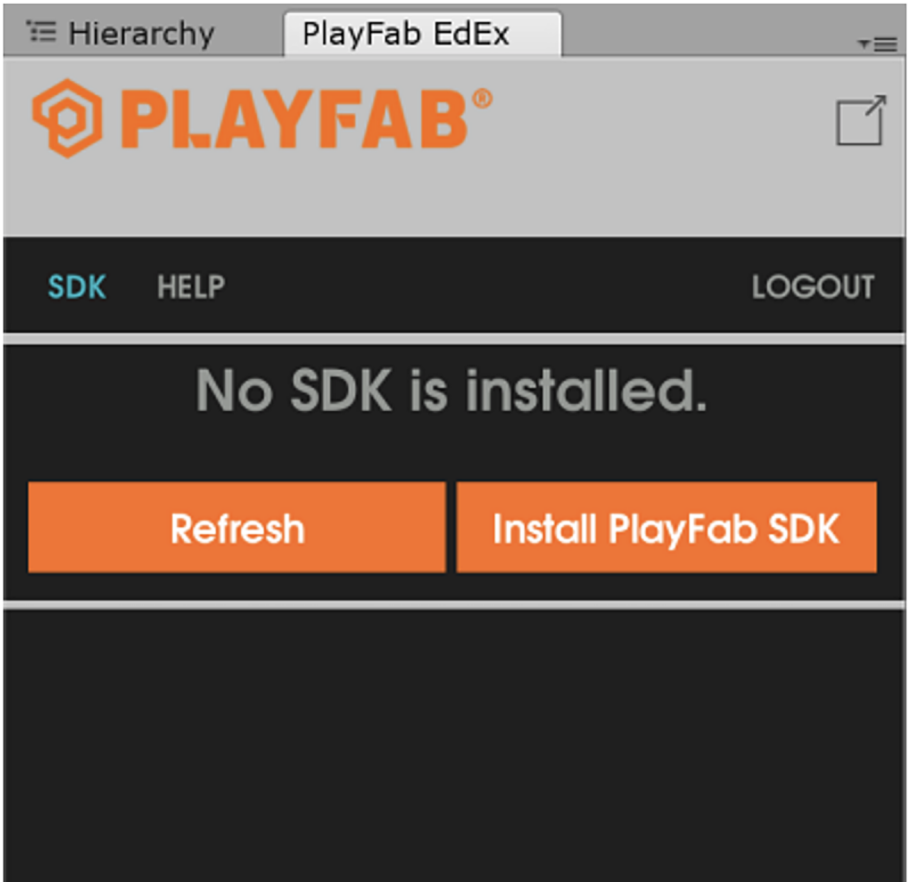
Download PlayFab SDK

The best way to acquire our Unity SDK is via our editor extensions - although you can directly download the Unity 3D SDK from our github page. PlayFab Unity3D SDK.

1. Download and Import the PlayFab Unity Editor Extensions package.
2. To import the the Unity Editor Extensions package, navigate to where the file was downloaded, and double-click on the .UnityPackage file. This will bring up the following window.

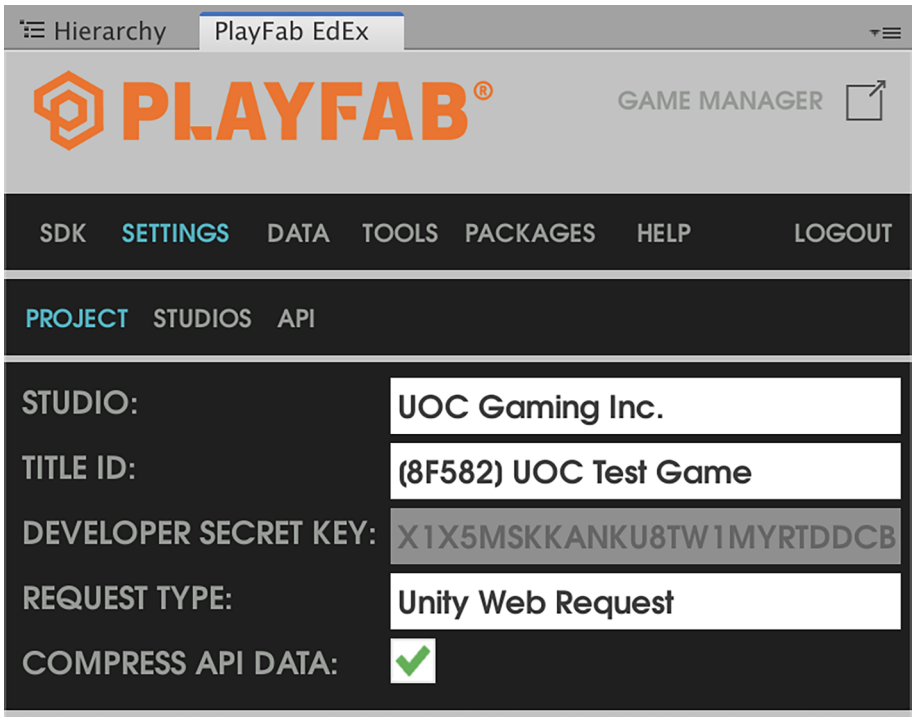
A continuación, importaremos el *asset* a nuestro proyecto Unity. Tiene que aparecer una nueva ventana *PlayFab EdEx*, desde donde nos crearemos nuestra cuenta gratuita de PlayFab y a continuación podremos instalar el *PlayFab SDK*:

Figura 11. Ventana PlayFab EdEx



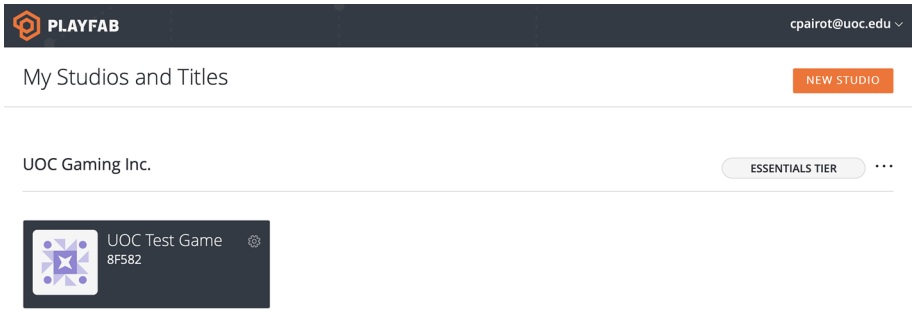
Si todo ha ido bien, iremos a *Settings* y estableceremos los datos de nuestro proyecto.

Figura 12. Settings



En este punto, nos puede pasar que al escoger el Studio nos deje siempre seleccionado el valor erróneo ***_override_***. Para solucionarlo, nos dirigiremos a https://developer.playfab.com/en-US/my-games y crearemos un nuevo Studio. Una vez creado al volver a Unity nos permitirá escogerlo y ya nos aparecerá la *Developer Secret Key* junto con los otros valores.

Figura 13. Listado de estudios y juegos en nuestra cuenta de PlayFab



Después de todos estos pasos ya estamos listos para utilizar PlayFab en nuestro proyecto Unity.

3.1.5. Utilizando el servicio de *login* de jugadores

PlayFab expone todos sus servicios mediante una API web REST. Por una cuestión de simplicidad y seguridad, todas las llamadas a sus funciones son mediante peticiones HTTP POST a través de SSL y todos los datos se envían en formato JSON, con compresión opcional *gzip*.

SSL (*secure sockets layer*)

Protocolo que ofrece comunicaciones seguras a través de Internet mediante la encriptación de los datos. Todos los sitios web que empiezan por el prefijo **https://** lo utilizan.

En este primer ejemplo vamos a utilizar el servicio de *login* de jugadores de PlayFab. Este servicio permite a los usuarios de nuestro juego registrarse o entrar en su cuenta de PlayFab mediante una dirección de correo electrónico y una contraseña.

API de *login* de PlayFab

Podéis encontrar ejemplos de la API de *login* de PlayFab aquí: https://api.playfab.com/docs/tutorials/ landing-players/best-login

Primero, crearemos un nuevo *script* que nos permitirá manejar todo este proceso. Incluiremos los *namespaces* de PlayFab al inicio:

```
using PlayFab;
using PlayFab.ClientModels;
using UnityEngine;

public class PlayFabLogin : MonoBehaviour {
```

En el método *Start()* hay que establecer el campo **TitleId**. Este campo es el que podéis ver en la pantalla de *PlayFab EdEx*.

Figura 14. Campo Title Id

Así pues, lo fijamos en el código solo para evitar que nos falle si este campo no estuviera ya inicializado:

```
    public void Start() {
```

```
        if (string.IsNullOrEmpty (PlayFabSettings.staticSettings.TitleId)) {
            PlayFabSettings.staticSettings.TitleId = "8F582";
        }
```

A continuación, ya podemos hacer llamadas a métodos de PlayFab. En este primer ejemplo llamaremos al método *LoginWithCustomId()*, que se traducirá como una petición HTTP POST a los servidores de PlayFab y que recibe como parámetros de entrada los datos que contendrá la petición en forma de objeto *LoginWithCustomIDRequest*. Es importante destacar que, además, le pasamos dos funciones que deberemos implementar y que tienen la forma de *callback*, es decir, que se llamará a una u otra en función de si el *login* ha tenido éxito (*OnLoginSuccess*) o ha fallado (*OnLoginFailure*).

```
    var request = new LoginWithCustomIDRequest { CustomId =
                "GettingStartedGuide", CreateAccount = true };
    PlayFabClientAPI.LoginWithCustomID (request, OnLoginSuccess, OnLoginFailure);
}
```

Así pues, si el *login* ha tenido éxito se llama a la función *OnLoginSuccess()*:

```
    private void OnLoginSuccess (LoginResult result) {
        Debug.Log ("Congratulations, you made your first successful API call!");
    }
```

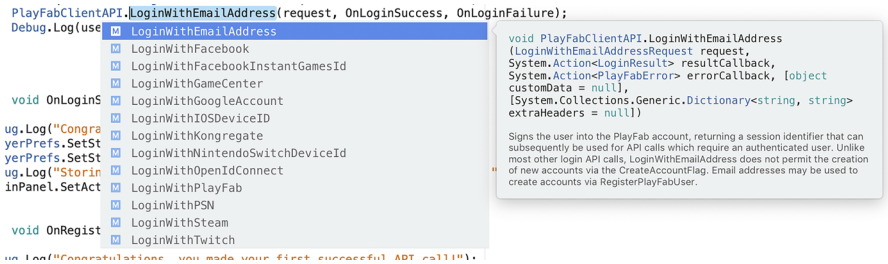
Por el contrario, si este falla se llama a la función *OnLoginFailure()*:

```
private void OnLoginFailure (PlayFabError error) {
    Debug.LogWarning ("Something went wrong with your first API call.");
    Debug.LogError ("Here's some debug information:");
    Debug.LogError (error.GenerateErrorReport());
}
```

Reto 1

Probad el funcionamiento de este *script* en vuestro proyecto de Unity.

Figura 15. PlayFab permite hacer *login* en su sistema mediante distintos proveedores y servicios de terceros



En cualquier caso, el sistema de *login* que hemos utilizado en este ejemplo es solo para efectos demostrativos. Ahora veremos cómo hacer un *login* más tradicional utilizando una dirección de correo electrónico y una contraseña. Para ello, cambiaremos la llamada por *LoginWithEmailAddress()* en el método *Start()*:

```
var request = new LoginWithEmailAddressRequest { Email = userEmail,
                                                 Password = userPassword };


PlayFabClientAPI.LoginWithEmailAddress (request, OnLoginSuccess, OnLoginFailure);
```

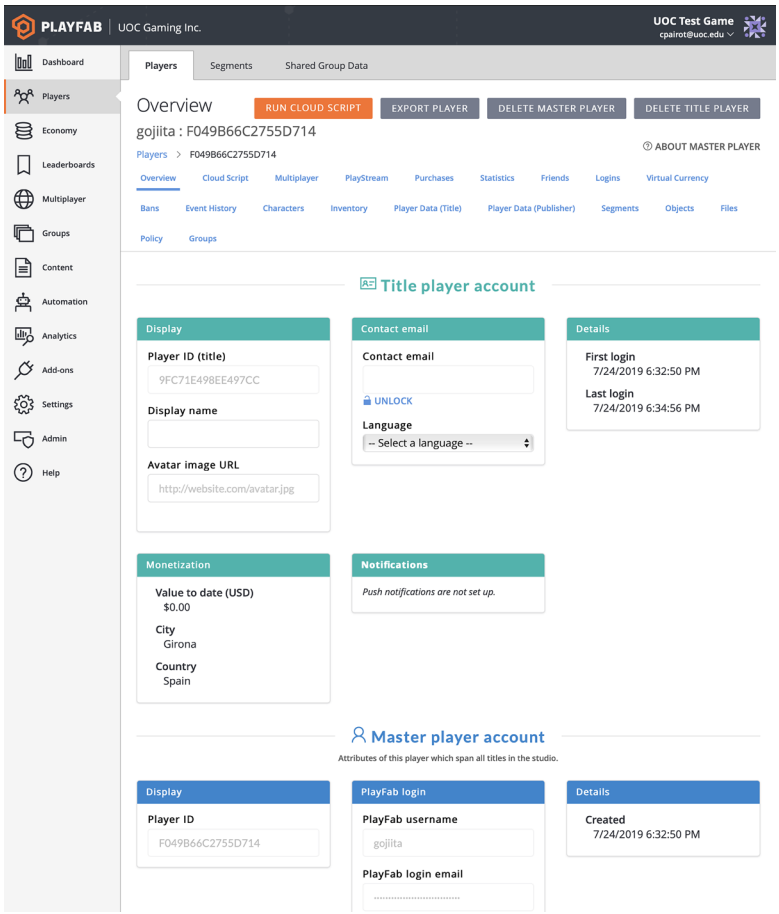
Si os fijáis, el objeto que encapsula los datos que se enviarán a la petición (*LoginWithEmailAddressRequest*) necesita dos variables de tipo *string*, que contendrán la dirección de correo del usuario y su contraseña.

Reto 2

Si el *login* con la dirección de correo y contraseña falla, haced que el usuario se registre automáticamente en el sistema.

Una vez los usuarios se han dado de alta en vuestro juego, podéis consultar en la aplicación web de PlayFab sus datos.

Figura 16. Overview de PlayFab



PlayFab proporciona una clase llamada **PlayerPrefs** que permite almacenar y consultar datos del jugador mediante los métodos *SetString()* y *HasKey()*:

```
PlayerPrefs.SetString ("EMAIL", userEmail);

if (PlayerPrefs.HasKey ("EMAIL")) {
        Debug.Log ("Player's e-mail address is: " + userEmail);
}
```

En este ejemplo, la clave **EMAIL** quedará vinculada a ese jugador para siempre con el valor que establezcamos, permitiéndonos recuperar su valor en cualquier momento desde la nube.

Si nos interesa, podemos borrar una clave concreta de las preferencias del jugador o bien todas las que tenga asociadas con los métodos *DeleteKey()* y *DeleteAll()*:

```
PlayerPrefs.DeleteKey ("EMAIL");  // Deletes the EMAIL key
PlayerPrefs.DeleteAll();          // Deletes all keys
```

Reto 3

Conseguid que, cuando el usuario se haya autenticado en PlayFab por primera vez, vuestro juego ya no os pida más que os autentiquéis en el sistema.

Reto 4

Cread un *canvas* en la escena que os pida el código de usuario, la dirección de correo electrónico, la contraseña y que, haciendo clic en un botón, pase todos estos datos a vuestro *script PlayFabLogin* y que este autentique o registre a vuestro jugador.

3.1.6. Utilizando el servicio de estadísticas de jugadores

Vamos ahora a ver cómo utilizar otro de los servicios de PlayFab. En este caso, el servicio de estadísticas de jugadores. Para ello, continuaremos con el código del ejemplo anterior y empezaremos renombrando nuestro *script* de ***PlayFabLogin.cs*** a ***PlayFabController.cs***. Esto lo hacemos porque ahora vamos a añadir nuevas funcionalidades que ya no están directamente relacionadas con el *login* del jugador y es preferible que el *script* tenga un nombre más genérico.

API de estadísticas de PlayFab

Podéis encontrar ejemplos de la API de estadísticas de PlayFab aquí: https://api.playfab.com/docs/tutorials/ landing-players/player-statistics

Llegados a este punto, vamos primero a crear las nuevas variables de las cuales nos interesará disponer de estadísticas. Vamos a crear, por ejemplo, variables para saber el nivel del jugador, el nivel del juego, la salud del jugador, el daño que ha infligido el jugador, así como la puntuación máxima que ha conseguido el jugador:

```
public int playerLevel;
public int gameLevel;
public int playerHealth;
public int playerDamage;
public int playerHighScore;
```

Además, para gestionar adecuadamente estas estadísticas, nos interesa que solo exista una única instancia de **PlayFabController** en ejecución al mismo tiempo. Para ello, utilizaremos el patrón de diseño *singleton* para conseguirlo.

```
public static PlayFabController PFC;


private void OnEnable() {
    if (PlayFabController.PFC == null) {
        PlayFabController.PFC = this;
```

```
        }
        else {
            if (PlayFabController.PFC != this) {
                Destroy (this.gameObject);
            }
        }
        DontDestroyOnLoad (this.gameObject);
    }
```

Como vemos, en el método ***OnEnable()***, que se llama cuando el GameObject en cuestión se activa, comprobaremos si ya tenemos una instancia de PlayFabController activa o no. Si no la tenemos, fijamos el valor de la variable PFC a esta instancia. En caso contrario, si la instancia es distinta a la nuestra, la destruimos porque ya existe otra. Finalmente, indicamos, mediante el método *DontDestroyOnLoad()*, que no se destruya el GameObject actual al cargar una escena nueva. Con este comportamiento conseguimos que solo haya una única instancia activa al mismo tiempo del *script* PlayFabController, siguiendo correctamente el patrón de *singleton*.

Patrón de diseño *singleton*

En la ingeniería del *software*, el *singleton* es un patrón de diseño que restringe la instanciación de una clase a una única instancia. Resulta útil cuando se necesita un único objeto que coordine las acciones dentro de un sistema.

A continuación, siguiendo la API de PlayFab, crearemos un método *SetStats()* que se encargará de enviar las estadísticas de nuestro jugador al servicio de PlayFab correspondiente:

```
public void SetStats() {
    PlayFabClientAPI.UpdatePlayerStatistics (new UpdatePlayerStatisticsRequest {
       Statistics = new List<StatisticUpdate> {
         new StatisticUpdate { StatisticName = "PlayerLevel", Value = playerLevel },
         new StatisticUpdate { StatisticName = "GameLevel", Value = gameLevel },
         new StatisticUpdate { StatisticName = "PlayerHealth", Value = playerHealth },
         new StatisticUpdate { StatisticName = "PlayerDamage", Value = playerDamage },
         new StatisticUpdate { StatisticName = "PlayerHighScore", Value = playerHighScore }
       }
    },
    result => { Debug.Log ("User statistics updated"); },
    error => { Debug.LogError (error.GenerateErrorReport()); });
}
```

Como vemos, ***PlayFabClientAPI.UpdatePlayerStatistics()*** es como llamaremos al método, pasándole una *request* HTTP (*UpdatePlayerStatisticsRequest*) y los *callbacks* de éxito y error.

En el cuerpo de *UpdatePlayerStatisticsRequest*, le pasaremos las parejas clave/valor equivalentes a las estadísticas que queremos almacenar en PlayFab.

Para recuperar los datos de las estadísticas que tenemos almacenadas en PlayFab para ese jugador en concreto, en primer lugar tendremos que implementar un método *GetStats()* de la siguiente manera:

```
      new GetPlayerStatisticsRequest(),
         OnGetStatistics,
         error => Debug.LogError(error.GenerateErrorReport())
      );
  }
```

Este método simplemente llama al método ***PlayFabClientAPI.GetPlayerStatistics()*** que, si tiene éxito, llamará a nuestra función ***OnGetStatistics()***. Así pues, tendremos que implementar esta función, que se encargará de cargar los valores de las estadísticas provenientes de PlayFab a nuestras variables locales mediante una estructura de tipo *switch*:

```
public void OnGetStatistics (GetPlayerStatisticsResult result) {
    Debug.Log ("Received the following Statistics:");
    foreach (var eachStat in result.Statistics) {
        Debug.Log ("Statistic (" + eachStat.StatisticName + "): " + eachStat.Value);
        switch (eachStat.StatisticName) {
            case "PlayerLevel":
                playerLevel = eachStat.Value;
                break;
            case "GameLevel":
                gameLevel = eachStat.Value;
                break;
            case "PlayerHealth":
                playerHealth = eachStat.Value;
                break;
            case "PlayerDamage":
                playerDamage = eachStat.Value;
                break;
            case "PlayerHighScore":
                playerHighScore = eachStat.Value;
                break;
        }
    }
}
```

Ahora que tenemos todos los métodos para actualizar y obtener las estadísticas, debemos decidir desde dónde llamarlos:

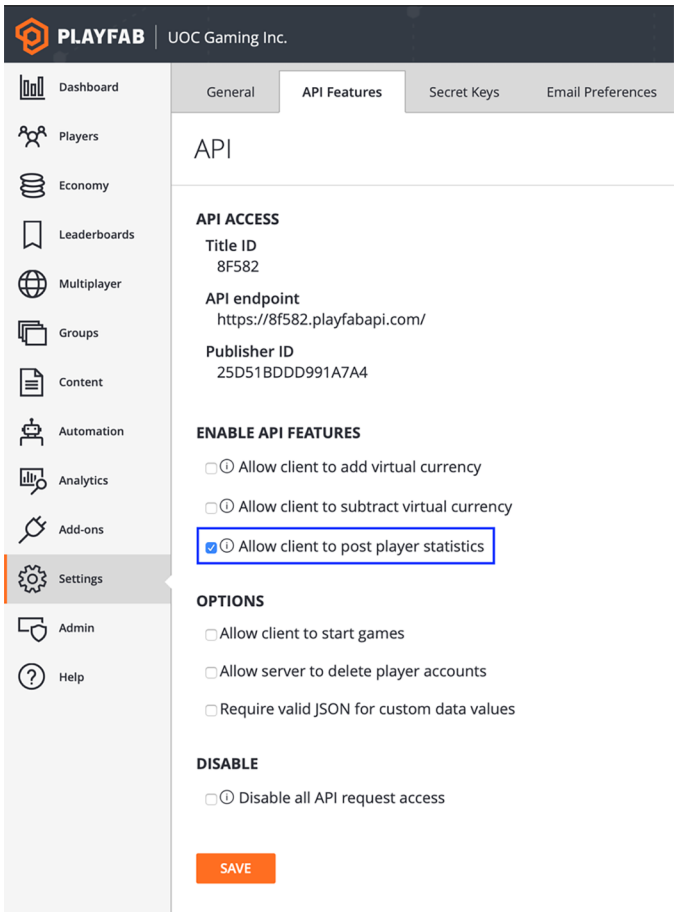
- El mejor momento en el que se puede llamar al método ***GetStats()*** es cuando el jugador se autentica correctamente contra el servicio de PlayFab, y

esto es dentro de nuestro método *OnLoginSuccess()*, así que lo actualizamos en consonancia.

- Podremos llamar al método ***SetStats()*** siempre que necesitemos enviar las estadísticas de los jugadores a la nube. Se puede hacer mediante una tarea programada periódica y se puede llamar desde cualquier *script* del juego, accediendo al *singleton* con ***PlayFabController.PFC.SetStats()***. Para el caso que nos ocupa, podemos llamarlas mediante un botón de la interfaz.

De todos modos, antes de poder utilizar el servicio tenemos que saber que PlayFab, por defecto, permite obtener las estadísticas, pero que para evitar posibles actualizaciones fraudulentas no permite actualizarlas desde los clientes. Es por ello que tenemos que permitir las actualizaciones de las estadísticas por parte de los clientes dirigiéndonos al PlayFab *dashboard* de nuestro juego, menú **Settings** > **API Features** y marcar la opción ***Allow client to post player statistics***.

Figura 17. Allow client to post player statistics



Con esto ya podremos probar nuestro proyecto y si nos dirigimos a las estadísticas del jugador desde el PlayFab *dashboard*, podremos ver como efectivamente estas se han guardado y cualquier actualización (a través del cliente de Unity o bien desde el *dashboard* directamente se propagan adecuadamente).

Figura 18. Podemos ver cómo se actualizan las estadísticas desde Unity Editor

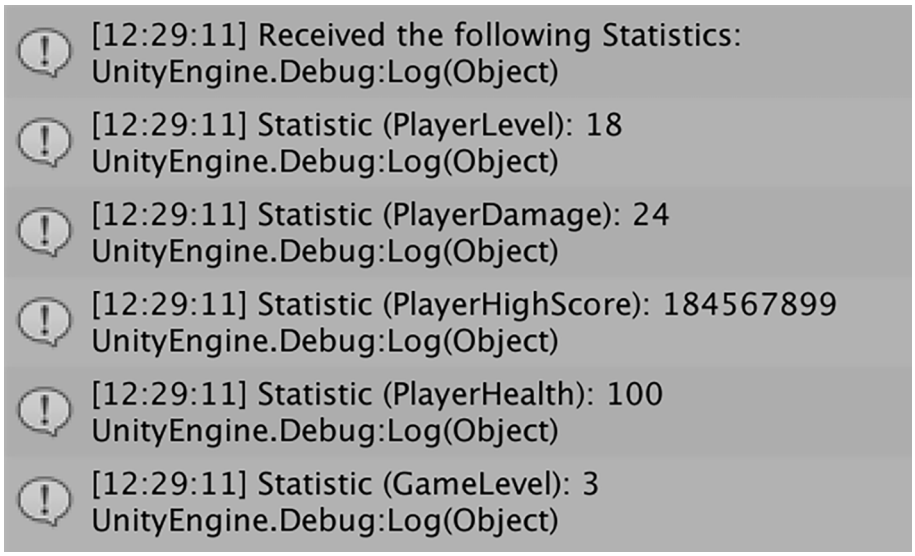
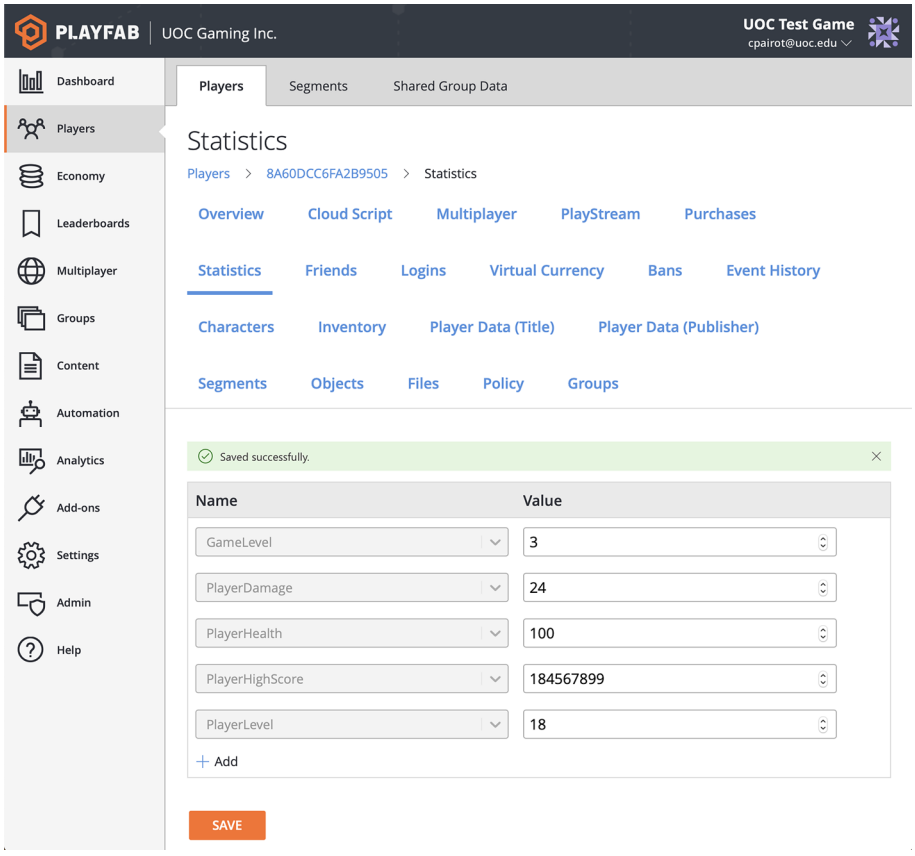


Figura 19. Estadísticas de un jugador en PlayFab



Reto 5

¿Cómo lo haríais para evitar actualizar las estadísticas desde el cliente y evitar así que clientes fraudulentos pudieran actualizar ilícitamente las estadísticas de un jugador?

3.1.7. Llamando a la API de PlayFab mediante *Postman*

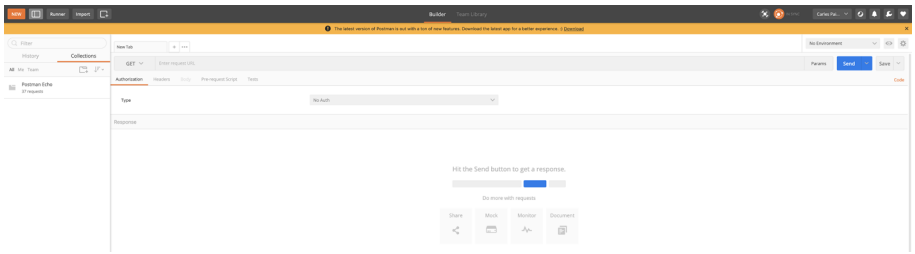
Saber cómo llamar manualmente a los métodos de la API web de PlayFab es de suma importancia cuando estamos integrando el servicio con nuestro juego. Por ejemplo, a veces nos puede interesar saber si un método funciona de la forma esperada, o bien nos interesa llamar a un método y analizar los datos específicos de la respuesta, o bien queremos ver el error que aparece al invocar un método. Existen muchas herramientas que facilitan la interacción directa con API web, pero una de las más populares es el *plugin* **Postman** para Chrome.

Para ello, deberemos seguir los siguientes pasos:

- Obtener el navegador **Google Chrome**.

- Instalar el *plugin* **Postman**. Una vez instalado, veremos una pantalla similar a esta:

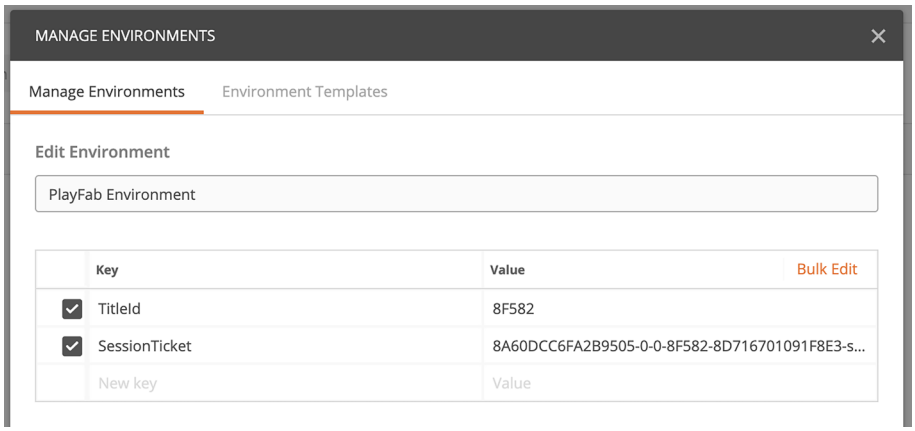
Figura 20. *Plugin* Postman



- Descargar la JSON Collection de PlayFab desde https://api.playfab.com/downloads/postman e importarla a Postman.

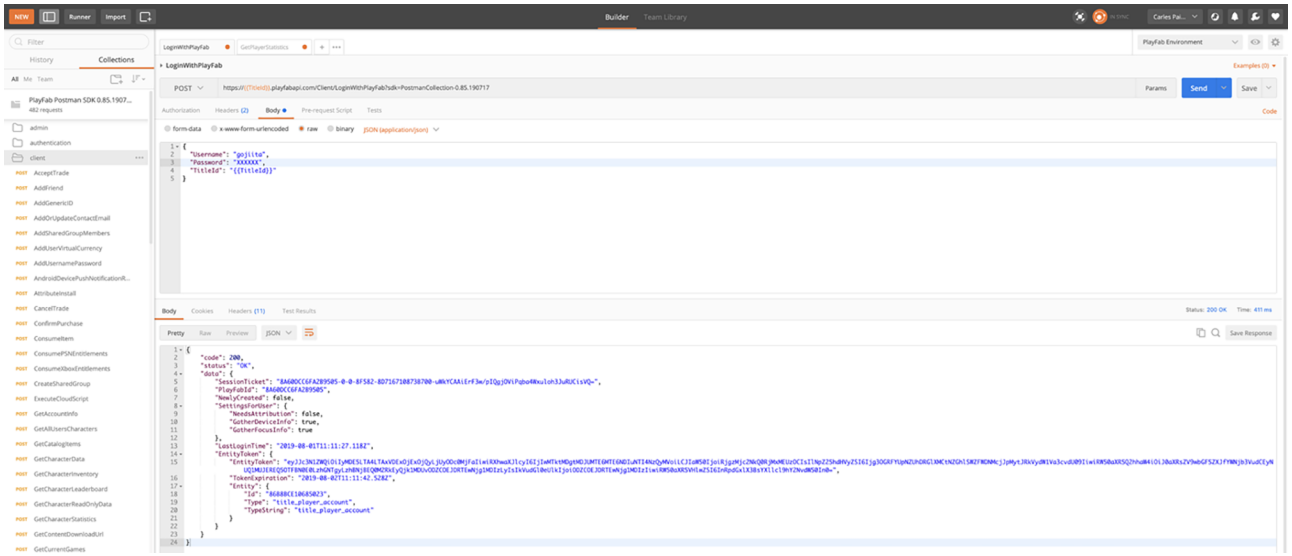
- Crear un nuevo entorno llamado PlayFab Environment y añadirle las variables que se utilizarán en cada llamada que hagamos a la API. En este primer caso, necesitamos como mínimo el ***TitleId***. Estos parámetros los podemos consultar del proyecto Unity que tenemos o desde el PlayFab *dashboard*.

Figura 21. Manage Environments



- Para comprobar que nos funciona correctamente, podemos abrir el método **Client > LoginWithPlayFab** y en la pestaña Body introducir los datos de un usuario (código de usuario y contraseña) que esté registrado en nuestro juego. Si todo va bien, recibiremos una respuesta 200 OK del estilo de esta:

Figura 22. Respuesta 200 OK



- Si queremos probar otros métodos de la API, seguramente se requieran más parámetros que tendremos que informar, como por ejemplo el *SessionTicket*, que tiene una duración máxima de 24 horas, por lo que pasado este periodo de tiempo, tendremos que volver a autenticarnos y actualizar el valor de esta variable.

PlayFab y *Postman*

Tenéis más información acerca de cómo llamar a los métodos de la API de PlayFab desde Postman en este enlace: https://api.playfab.com/docs/tutorials/ execute-playfab-api-via-postman.

Reto 6

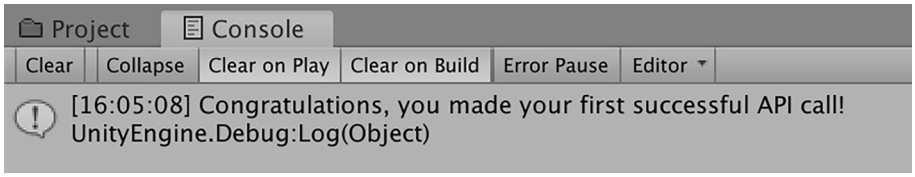
Llamad al método de la API *GetPlayerStatistics()* con Postman y obtened las estadísticas del jugador que hemos grabado anteriormente.

3.2. Soluciones a los retos planteados

Reto 1

Al ejecutar el proyecto Unity, realizamos la primera llamada correcta al servicio de autenticación de jugadores de PlayFab. Si observamos la salida en la consola, el resultado tiene que ser el esperado:

Figura 23. Resultado en consola



Reto 2

Para completar este reto, debemos modificar el comportamiento del método ***OnLoginFailure()*** de modo que, cuando se ejecute, intente crear el nuevo usuario.

Esto se consigue llamando al método ***PlayFabClientAPI.RegisterPlayFabUser()*** pasándole como parámetros la dirección de correo, la contraseña y el código del usuario.

```
private void OnLoginFailure (PlayFabError error) {
    Debug.Log ("User " + userEmail + " does not exist. Registering new player...");
    var registerRequest = new RegisterPlayFabUserRequest { Email = userEmail,
                                    Password = userPassword, Username = username };
    PlayFabClientAPI.RegisterPlayFabUser (registerRequest, OnRegisterSuccess,
                                    OnRegisterFailure);
}


}
```

También debemos registrar los *callbacks* de registro de nuevo usuario correcto (***OnRegisterSuccess***) y de error (***OnRegisterFailure***).

```
private void OnRegisterSuccess (RegisterPlayFabUserResult result) {
    Debug.Log ("Congratulations, new user has been registered!");
}

private void OnRegisterFailure(PlayFabError error) {
    Debug.LogError (error.GenerateErrorReport());
}
```

Reto 3

Para implementar la funcionalidad de recordar al usuario que se ha autenticado en PlayFab en vuestro juego, podemos utilizar la clase ***PlayerPrefs*** de Unity, que permite almacenar parejas clave/valor que quedan automáticamente asociadas al perfil del jugador actual.

Para ello, primero almacenaremos los datos del usuario (dirección de correo electrónico y contraseña) en las *PlayerPrefs* del usuario justo en el momento en que se produzca una autenticación satisfactoria. Esto es, dentro del método ***OnLoginSuccess()***, añadiremos las siguientes líneas:

```
PlayerPrefs.SetString ("EMAIL", userEmail);
PlayerPrefs.SetString ("PASSWORD", userPassword);
Debug.Log ("Storing " + userEmail + " credentials into Player Preferences.");
```

También podemos añadir estas líneas en el método ***OnRegisterSuccess()***, que hemos implementado en el reto 2 y que crea automáticamente al usuario si este no estaba registrado con anterioridad. Pues bien, en este escenario, también nos interesa almacenar las credenciales del usuario para que no nos las vuelva a pedir al iniciar de nuevo el juego.

Finalmente, será en el método ***Start()*** en el que implementaremos la lógica de *login* de usuario si ya tenemos los datos almacenados en las *PlayerPrefs*:

```
if (PlayerPrefs.HasKey("EMAIL")) {
    userEmail = PlayerPrefs.GetString ("EMAIL");
    userPassword = PlayerPrefs.GetString ("PASSWORD");

    var request = new LoginWithEmailAddressRequest { Email = userEmail,
                                                     Password = userPassword };
    PlayFabClientAPI.LoginWithEmailAddress (request, OnLoginSuccess, OnLoginFailure);
    Debug.Log (userEmail + " user logged in automatically.");
}
```

Si analizamos este código, primero comprobamos si dentro de las *PlayerPrefs* tenemos ya almacenado un valor para la clave **EMAIL**. Si lo tenemos, entonces es que previamente ya se han registrado las credenciales del jugador y, por lo tanto, las cargamos en las variables *userEmail* y *userPassword* para proceder, a continuación, a realizar automáticamente el *login* del jugador en PlayFab.

Como nota adicional, si queréis resetear los valores de vuestras *PlayerPrefs*, podéis llamar a su método ***DeleteAll()***.

Reto 4

Este reto consiste en enlazar los distintos métodos del *script* PlayFabLogin/PlayFabController con elementos de interfaz gráfica (UI) de Unity.

Primero, tendremos que añadir métodos en el *script* PlayFabLogin/PlayFabController para que, desde la interfaz, se puedan actualizar los valores de las variables userName, userEmail y userPassword:

```
public void GetUserEmail (string emailIn) {
    userEmail = emailIn;
}

public void GetUserPassword (string passwordIn) {
    userPassword = passwordIn;
}

public void GetUsername (string usernameIn) {
    username = usernameIn;
}
```

También añadiremos un método que será el que se llamará cuando hagamos clic sobre el botón de Login, y que invocará a la API de PlayFab para autenticar al jugador:

```
public void OnClickLogin() {
    var request = new LoginWithEmailAddressRequest { Email = userEmail,
                                                     Password = userPassword };
    PlayFabClientAPI.LoginWithEmailAddress (request, OnLoginSuccess, OnLoginFailure);
}
```

Para acabar con la parte de *scripting*, declararemos una variable pública para poder referenciar el panel de *login* que va a utilizar el script:

```
public GameObject loginPanel;
```

Dentro del Unity Editor, crearemos un *canvas* muy sencillo y añadiremos un panel al que llamaremos LoginPanel. Dentro del LoginPanel, añadiremos tres InputFields (Username, Email, Password) y un Button (Login).

Figura 24. Panel canvas

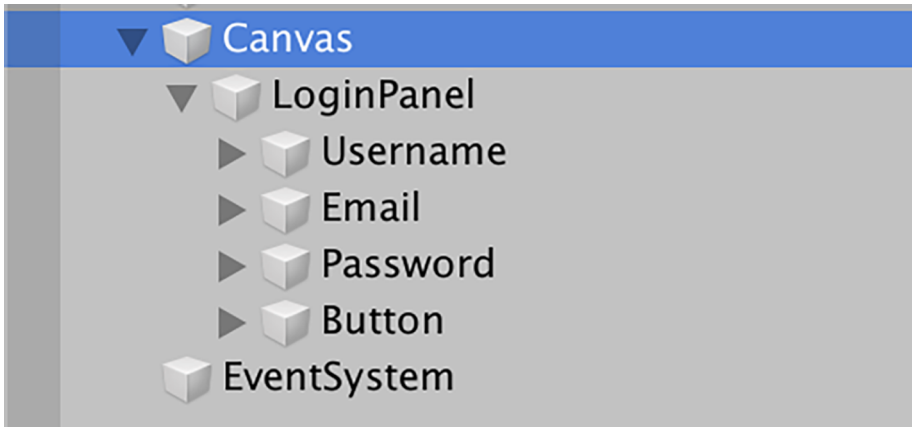
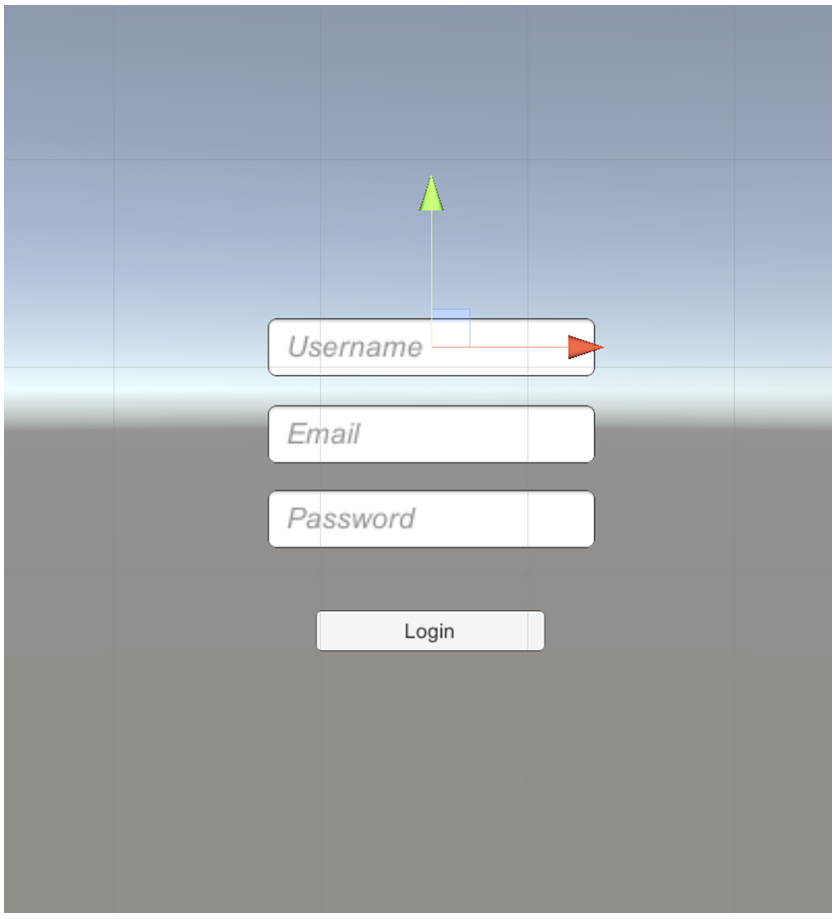


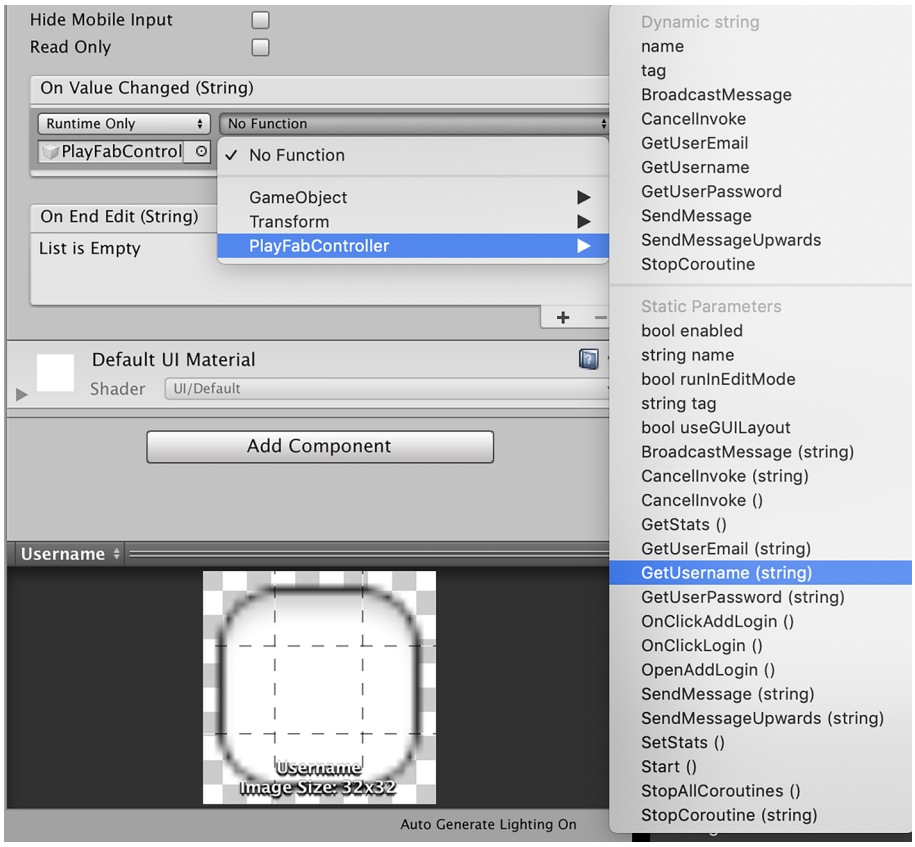
Figura 25. Panel LogIn



A cada uno de los InputFields, les vincularemos en el On Value Changed, el método correspondiente del PlayFabLogin.

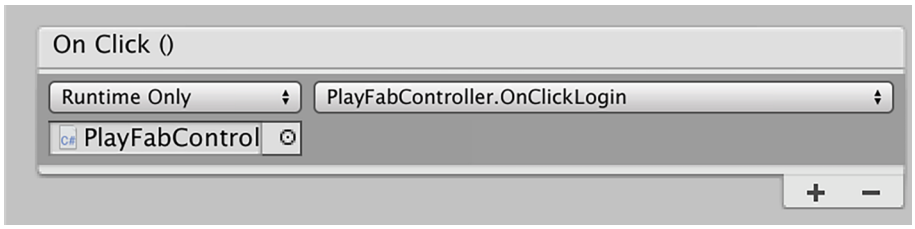
Por ejemplo, en el InputField Username, llamaremos al método GetUsername() de PlayFabLogin/PlayFabController:

Figura 26. InputField Username



Al botón le vincularemos el método OnClickLogin() de PlayFabLogin/PlayFabController:

Figura 27. Configuración del botón



Finalmente, arrastraremos el LoginPanel a la propiedad Login Panel del *script*:

Figura 28. Play Fab Controller

Para darle un toque más profesional, podemos hacer que una vez que el usuario se haya autenticado correctamente, el LoginPanel desaparezca. Para conseguirlo, podemos añadir esta línea en los métodos OnLoginSuccess() y OnRegisterSuccess():

```
loginPanel.SetActive(false);
```

Reto 5

Como hemos visto, la actualización de las estadísticas desde los propios clientes es una práctica eminentemente insegura y desde PlayFab hemos tenido que cambiar el comportamiento por defecto para permitirlo.

Para poder conseguir lo mismo, pero de un modo mucho más seguro, necesitamos que los valores de las estadísticas se actualicen desde el propio servidor. Pero, ¿cómo podemos hacerlo? La respuesta de PlayFab a esta necesidad se llama ***cloud scripts***.

Los *cloud scripts* se ejecutan en la nube de PlayFab y tienen acceso completo a la API de servidor de PlayFab. Lo más importante es que estos *scripts* se ejecutan en el contexto de un jugador autenticado de forma segura, de modo que se pueden utilizar para implementar lógica del juego que esté segura de *exploits* que puedan venir de clientes fraudulentos.

Las funciones de tipo *cloud script* también pueden realizar llamadas a *endpoints* HTTP externos, como por ejemplo bases de datos u otras API privadas, de modo que las hace muy flexibles de cara a la integración con otros sistemas de *backend* ya existentes.

Así pues, para crear nuestro primer *cloud script*, vamos a ir al **PlayFab Dashboard > Automation > Cloud Script > Revision** y vamos a añadir el siguiente código, por ejemplo, al final del fichero que se nos muestra:

```
handlers.UpdatePlayerStats = function (args, context) {
    var request = {
        PlayFabId: currentPlayerId, Statistics: [{
                StatisticName: "PlayerLevel",
                Value: args.Level
            },
                                                {
                StatisticName: "PlayerHighScore",
                Value: args.highScore
            },
                                                {
                StatisticName: "PlayerHealth",
                Value: args.Health
            }]
    };
    // The pre-defined "server" object has functions
    // corresponding to each PlayFab server API
    var playerStatResult = server.UpdatePlayerStatistics(request);
    return {messageValue: "Updated Cloud Stats"}
};
```

Simplemente, lo que hemos hecho ha sido crear una nueva función ***UpdatePlayerStats()*** a partir de la función ya existente *makeAPICall()* para adecuarla a nuestras necesidades. El lenguaje utilizado en los *cloud scripts* es **JavaScript**, pero si nos fijamos, se parece mucho a lo que hemos estado haciendo hasta ahora en C# dentro de Unity.

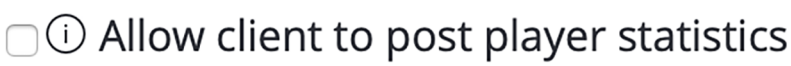
Por un lado, tenemos una primera parte que es la creación de la *request* propiamente dicha, en la que fijamos los valores de las estadísticas que queremos actualizar y, por otro lado, tenemos la llamada a la API del servidor de PlayFab, en este caso, pasándole esta *request*.

Es muy importante que grabéis estos cambios dándole al botón **Save Revision X**, donde la X indica un número de revisión. Pero no es suficiente solo con grabarlo, sino que tenemos que desplegar (*deploy*) esta versión para que se ejecute en el entorno de producción de PlayFab. Para ello, le daremos al botón **Deploy Revision X**.

Figura 29. Botón Deploy Revision X

También podemos aprovechar para desmarcar la opción que permite a los clientes enviar estadísticas, dejando el entorno mucho más seguro: **PlayFab Dashboard > Settings > API Features**.

Figura 30. API Features



Creación de *cloud scripts* personalizados

Para más información acerca de la creación de *cloud scripts* personalizados, podéis ir a este enlace: https://api.playfab.com/docs/tutorials/ landing-automation/writing-custom- cloud-script

Ahora ya podemos ir a Unity Editor para modificar el *script* ***PlayFabController*** para que llame a este *cloud script* (***UpdatePlayerStats()***) cuando tenga que actualizar las estadísticas.

Para llamar a nuestro *cloud script*, lo haremos implementando el método ***StartCloudUpdatePlayerStats()***:

```
public void StartCloudUpdatePlayerStats() {
    PlayFabClientAPI.ExecuteCloudScript (new ExecuteCloudScriptRequest()
    {
        FunctionName = "UpdatePlayerStats",
        FunctionParameter = new { Level = playerLevel, highScore = playerHighScore,
                                  Health = playerHealth }, GeneratePlayStreamEvent = true,
    }, OnCloudUpdateStats, OnErrorShared);
}
```

Como vemos, llamaremos a la función ***UpdatePlayerStats()*** pasándole los parámetros de las estadísticas a actualizar y los *callbacks* a llamar en caso de éxito (***OnCloudUpdateStats***) y error (***OnErrorShared***).

El código de estos dos *callbacks* es el siguiente:

```
private static void OnCloudUpdateStats (ExecuteCloudScriptResult result) {
    Debug.Log(PlayFab.PluginManager.GetPlugin<ISerializerPlugin>
         (PluginContract.PlayFab_Serializer).SerializeObject (result.FunctionResult));
     JsonObject jsonResult = (JsonObject) result.FunctionResult;
     object messageValue;
     jsonResult.TryGetValue ("messageValue", out messageValue);
     Debug.Log ((string) messageValue);
}

private static void OnErrorShared (PlayFabError error) {
    Debug.Log (error.GenerateErrorReport());
}
```

El método OnCloudUpdateStats() simplemente deserializa el valor de retorno (messageValue) de la llamada a UpdatePlayerStats() y lo muestra por la consola.

La última tarea a hacer es cambiar la acción que llamará el botón de actualizar las estadísticas para que, en lugar de llamar al método ***SetStats()*** –que ya no funcionará porque hemos deshabilitado que los clientes puedan actualizarlas directamente– se llame al método ***StartCloudUpdatePlayerStats()***.

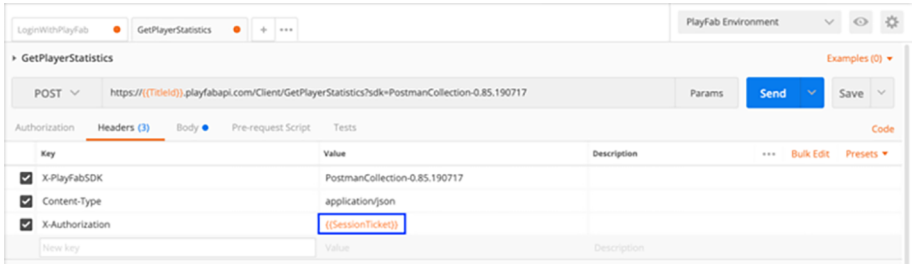
Figura 31. Cambio del método en acción *On Click()*

Reto 6

Para poder invocar al método ***GetPlayerStatistics()***, primero lo abriremos con Postman desde **Client** > ***GetPlayerStatistics()***.

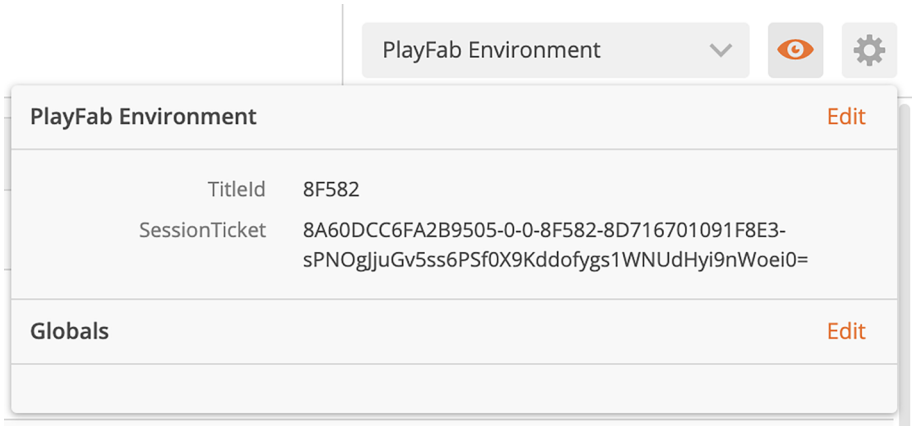
Si nos fijamos, veremos que en la cabecera se nos pide que informemos el parámetro *SessionTicket*, que tendremos que haber obtenido antes del autenticado del usuario llamando a ***LoginWithPlayFab()***:

Figura 32. Llamada a LoginWithPlayFab



Estableceremos, pues, el valor del *SessionTicket* en la variable de entorno correspondiente:

Figura 33. SessionTicket en el PlayFab Environment



Si abrimos la pestanya Body, veremos que en la *request* se le pasa un parámetro llamado *StatisticNames*, que nos permite especificar qué estadísticas queremos obtener. Como las queremos obtener todas, simplemente comentamos este parámetro:

Figura 34. Pestaña Body

Finalmente, realizamos la invocación y si todo ha ido bien, nos tiene que devolver las estadísticas del jugador que habíamos fijado anteriormente:

Figura 35. Estadísticas del jugador

Resumen

En este módulo hemos podido repasar lo que hoy en día son los servicios indispensables para cualquier juego, aunque no se trate ni de un juego en línea ni multijugador. Estos servicios son propios de plataformas muy populares, como el ejemplo de Steam que hemos estudiado. Pero estos servicios se encuentran obviamente en el lado servidor de cada plataforma.

Para entender cómo trabajan estos servicios, hemos presentado el paradigma de la computación en la nube, que hoy en día es la forma más recomendada para realizar este tipo de tareas remotas. Como hemos visto, existen diferentes tipos de nubes, cada una con sus ventajas e inconvenientes. El coste es uno de los principales inconvenientes, compartido con las soluciones de servidor tradicionales, aunque quizás ligeramente más justo, ya que se factura por consumo real.

Existen diferentes alternativas a servicios *cloud* oficiales para Unity, como Microsoft Azure PlayFab. Son servicios de pago que ofrecen una forma sencilla para desplegar nuestro videojuego. Alternativamente podemos buscar otros *assets* que nos permitan integrar nuestro juego con una plataforma concreta o específica del propio *asset* y dotarlo de los servidores dedicados.

Finalmente, hemos recuperado el ejemplo trabajado en Unity en los otros módulos, *Tanks!*, para integrarlo con una plataforma *cloud* real como es PlayFab. Esta plataforma de servicios en línea ofrece muchas funcionalidades que se benefician de la escalabilidad de la nube y de una API de programación sencilla y robusta, basada en tecnologías abiertas como HTTP y JSON.

Hemos visto cómo se utilizan los servicios en la nube de PlayFab, concretamente los servicios de registro de jugadores y autenticación, así como el servicio de estadísticas que para mayor seguridad, debe ser llamado directamente desde el servidor mediante *cloud scripts*.

Bibliografía

Alexandre, Thor (2005). *Massively Multiplayer Game Development 2 (Game Development)*. Rockland, MA: Charles River Media, Inc.

Armitage, Grenville; Claypool, Mark; Branch, Philip (2006). *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. Hoboken, NJ: John Wiley & Sons, Ltd.

Coulouris, George; Dollimore, Jean; Kindberg, Tim (2005). «Distributed systems: concepts and design». En: *International computer science series* (vol. 4). Reading, MA: Addison Wesley.

Jimenez-Rodriguez, J.; Jimenez-Diaz, G.; Diaz-Agudo, B. (2011). «Match-making and case-based recommendations». En: *Workshop on Case-Based Reasoning for Computer Games. XIX Conferencia Internacional sobre Case Based Reasoning*.

Glickman, M. E. (1999). «Parameter estimation in large dynamic paired comparison experiments». *Applied Statistics* (vol. 48, págs. 377-394).

Herbrich, R.; Minka, T.; Graepel, T. (2007). «TrueSkill(TM): a bayesian skill rating system». *Advances in Neural Information Processing Systems* (vol. 20, págs. 569-576).

Tanenbaum, Andrew S.; Van Steen, Maarten (2006). *Distributed Systems: Principles and Paradigms* (2.ª ed.). Upper Saddle River, NJ: Prentice-Hall, Inc.

Yahyavi. Amir; Kemme, Bettina (2013). «Peer-to-peer architectures for massively multiplayer online games: A Survey». *ACM Comput. Surv.* (vol. 46, n.º 1, art. 9).