# Easy Game Server- Manual de uso

Este manual de uso ha sido creado para indicar y facilitar al usuario final de la librería el proceso que debe realizar de cara a instalar y utilizar la herramienta, a modo de documentación del programa.

## Índice

### Contenido

# Introducción

**Easy Game Server** es una herramienta para el motor de videojuegos Unity que permite a los desarrolladores de videojuegos implementar de manera sencilla un sistema multijugador en línea, mediante una arquitectura de **red de servidores** y **paso de mensajes**, para el videojuego multijugador que estén realizando.

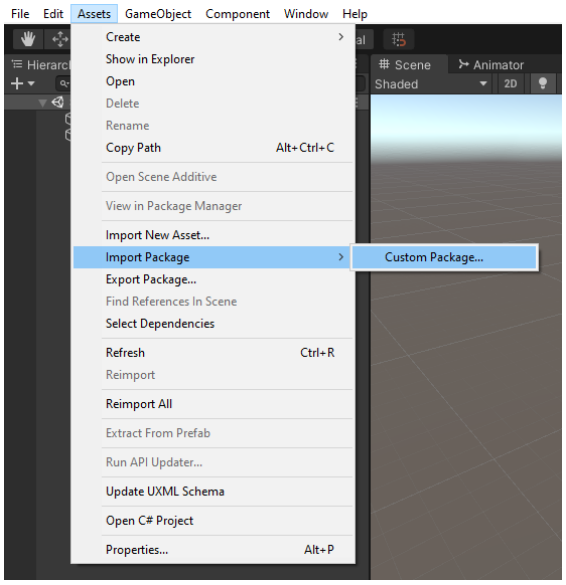
El sistema se compone de tres componentes principales:

- Master Server: Servidor maestro **único** que gestiona las conexiones al servidor y crea las partidas en las que posteriormente participarán los clientes.

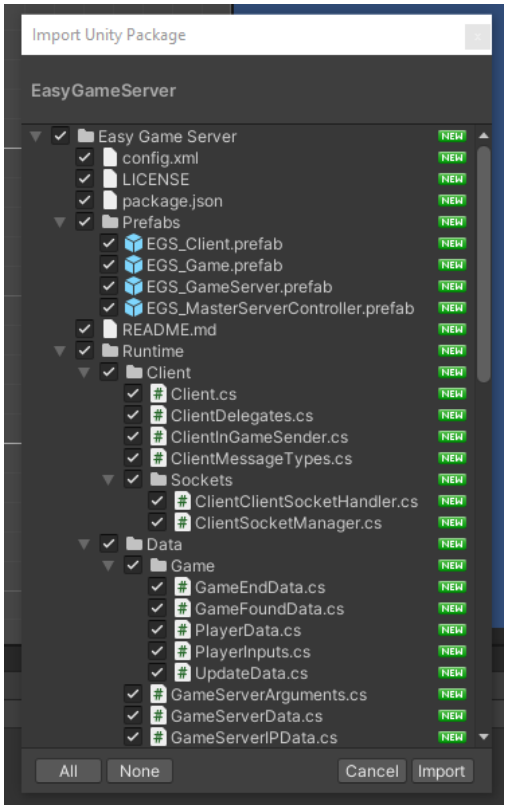
- Game Server: Servidor de juego que solo gestiona la partida que tiene asignada. Puede haber tantos *Game Servers* ejecutándose como número de partidas máximas especificadas por el usuario.

- Client: Cliente que representa al jugador y que se conecta al *Master Server* para iniciar sesión y buscar partida, y posteriormente al *Game Server* asignado para jugar la partida que ha encontrado.

# Cómo instalar la herramienta

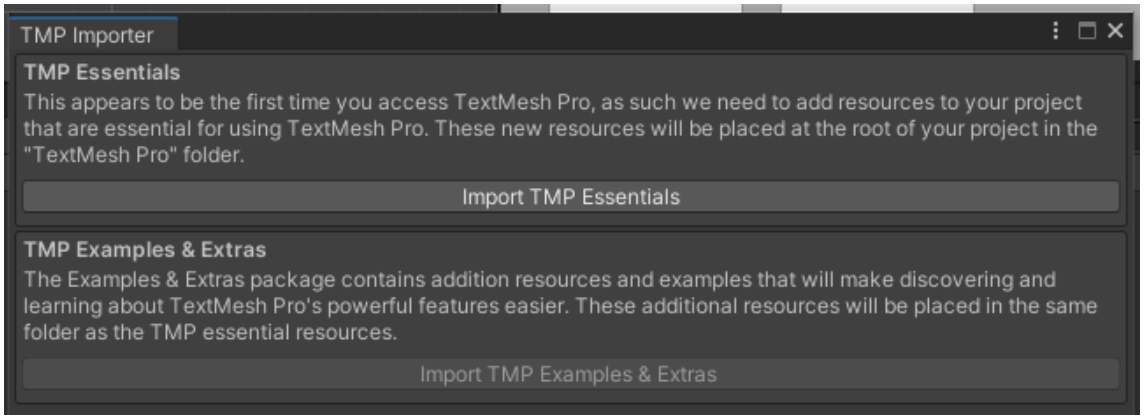



Lo primero que debemos hacer es ir al desplegable ***Assets*** del menú superior de *Unity* y buscamos la opción ***Import Package*** -> ***Custom Package***. Buscamos el paquete en nuestra computadora y lo seleccionamos.



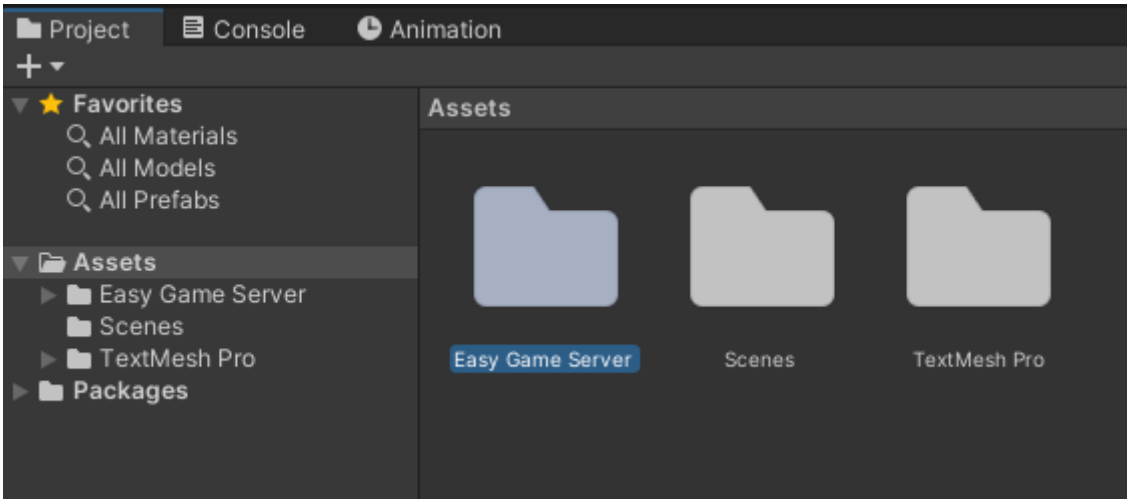
*Unity* nos permite seleccionar qué archivos queremos importar. El paquete incluye dos escenas que pueden utilizarse como *Master Server* y *Game Server*, y varios prefabs que deberán estar añadidos en el proyecto como veremos más adelante.

Por otro lado, **Easy Game Server** depende de **TextMeshPro** ya que se incluyen textos de esta librería en las escenas base que ofrece nuestra herramienta. Si ya lo tuviéramos instalado, podemos no seleccionar la importación de la carpeta *TextMeshPro*.




En caso de tener que añadir **TextMeshPro**, nos aparecerá esta ventana más adelante, donde simplemente debemos dar clic en ***Import TMP Essentials***.

Una vez seleccionados los archivos a importar y tras pulsar el botón ***Import***, *Unity* extraerá los contenidos del paquete en una nueva carpeta del proyecto llamada **Easy Game Server**.



# Cómo utilizar la herramienta

Vamos a dividir este manual de uso según los tres componentes principales mencionados previamente (**Master Server**, **Game Server** y **Client**), ya que cada uno requiere una configuración específica. Pero antes, vamos a explicar el despacho de eventos y a configurar los parámetros que utilizará la herramienta

## Despacho de eventos en el hilo principal

Para poder tratar con los objetos de *Unity* desde los hilos de la librería, se ha implementado un despacho de eventos (clase ***MainThreadDispatcher***) donde los hilos pueden insertar el código que deseen para ejecutarse en la siguiente iteración del despacho de eventos.

Aunque no es una ejecución inmediata, el procesamiento es lo suficientemente rápido para que no sea un problema gestionar los datos de la aplicación, pero a veces es necesario añadir una capa extra de control.

```
/// <summary>
/// Class MainThreadDispatcher, that will execute code on the main thread.
/// </summary>
Script de Unity (3 referencias de recurso) | 39 referencias
public class MainThreadDispatcher : MonoBehaviour
{
    #region Variables
    [Header("Dispatcher")]
    [Tooltip("Static instance of the dispatcher")]
    private static MainThreadDispatcher instance;

    [Tooltip("Queue that will store events to execute")]
    private static readonly Queue<Action> eventQueue = new Queue<Action>();
    #endregion

    #region Unity Methods
    /// <summary>
    /// Method Awake, called on script load.
    /// </summary>
    Mensaje de Unity | 0 referencias
    void Awake()
    {
        // Instantiate the singleton.
        if (instance == null)
        {
            instance = this;
            DontDestroyOnLoad(instance);
        }
        else
        {
            Destroy(this);
        }
    }
```

Para utilizarlo y ejecutar el código que se necesite en el hilo principal de Unity, debe llamarse a la función **RunOnMainThread** como se muestra en el siguiente ejemplo.

```
MainThreadDispatcher.RunOnMainThread(() => {
    pauseViewGameobject.SetActive(false);
    opponentLeftViewGameobject.SetActive(true);
    Invoke(nameof(ChangeToEndGame), 2);
});
```

Todo el código escrito entre las llaves (**{}**) se ejecutará en el hilo principal de *Unity*.

## Configuración de la herramienta Easy Game Server

En el paquete se incluye un archivo ***config.xml*** que debe ir en todas las builds que utilicen la librería:


Imported Object

Config (Text Asset)

```
<?xml version="1.0" encoding="UTF-8"?>
<EasyGameServer>
        <server>
                <debug-mode-console>1</debug-mode-console>
                <debug-mode-file>2</debug-mode-file>
                <max-connections-queue>10</max-connections-queue>
                <max-games>5</max-games>
                <time-between-rtt>1000</time-between-rtt>
                <disconnect-timeout>15000</disconnect-timeout>
  </server>
        <game-server>
                <path>C:\\Users\\Samue\\Desktop\\URJC\\TFG\\Builds\\Game Server\\Easy Game Server.exe</path>
        </game-server>
        <networking>
                <server-ip>192.168.1.45</server-ip>
                <base-port>21721</base-port>
                <connection-tries>3</connection-tries>
        </networking>
        <game>
                <players-per-game>2</players-per-game>
                <calculations-per-second>60</calculations-per-second>
        </game>
</EasyGameServer>
```

Los parámetros que en este archivo se definen son leídos por las distintas instancias y configuran el funcionamiento de la herramienta.

## Parámetros configurables

### *server/debug-mode-console*

Nivel de debug de la consola del servidor, que indica cuánta información se muestra en la consola.

**Niveles de debug**

- *No_Debug* = -1.
- *Minimal* = 0.
- *Useful* = 1.
- *Extended* = 2.
- *Complete* = 3.

### *server/debug-mode-file*

Nivel de debug del log persistente, que indica cuánta información se muestra en el archivo que se guarda con cada ejecución del **Master Server**.

Los niveles de debug son los mismos que para el parámetro anterior *debug-mode-console*.

### *server/max-connections-queue*

Máximo número de conexiones que pueden estar a la espera de conectarse al **Master Server**.

### *server/max-games*

Número máximo de partidas que se pueden jugar a la vez. Por definición, también será el número máximo de **Game Servers** que pueden ejecutarse de forma simultánea.

### *server/time-between-rtt*

Tiempo en milisegundos que transcurre entre peticiones de ***Round Trip Time*** que envían los servidores a las conexiones activas.

### *server/disconnect-timeout*

Tiempo en milisegundos que debe transcurrir sin que se haya recibido un ***Round Trip Time*** de vuelta por una conexión activa para darla por desconectada.

### *game-server/path*

Ruta que indica donde se encuentra, en nuestro explorador de archivos, el ejecutable del **Game Server**. De esta manera, el **Master Server** sabe qué proceso ejecutar.

### *networking/server-ip*

IP donde se establecerá el servidor. Podemos utilizar tanto IP local como IP pública, pero es importante que todas las instancias tengan el mismo valor en su ***config.xml***.

### *networking/base-port*

Puerto base donde se vinculará el **Master Server** para las escuchas. Este parámetro es especialmente importante, ya que **Easy Game Server** utiliza este puerto base para el **Master Server** y los siguientes ***max-games*** puertos para los **Game Servers**.

Ejemplo: Utilizamos de puerto base el 21721 e indicamos un número máximo de 10 partidas. Easy Game Server utilizará los puertos 21721 – 21731 (ambos inclusive).

### *networking/connection-tries*

Número de intentos de conexión que los clientes harán para conectarse a ambos servidores antes de desistir.

### *game/ players-per-game*

Número de jugadores por partida.

*game/calculations-per-second*

Número de veces por segundo que el **Game Server** ejecutará sus cálculos y les enviará la información a los **Clientes**, y número de veces por segundo que los **Clientes** le enviarán información sobre sus inputs al **Game Server**.

# Master Server

## Configuración del Master Server

La librería **Easy Game Server** incluye por defecto una escena ***MasterServerScene*** que contiene toda la base necesaria para el servidor maestro.

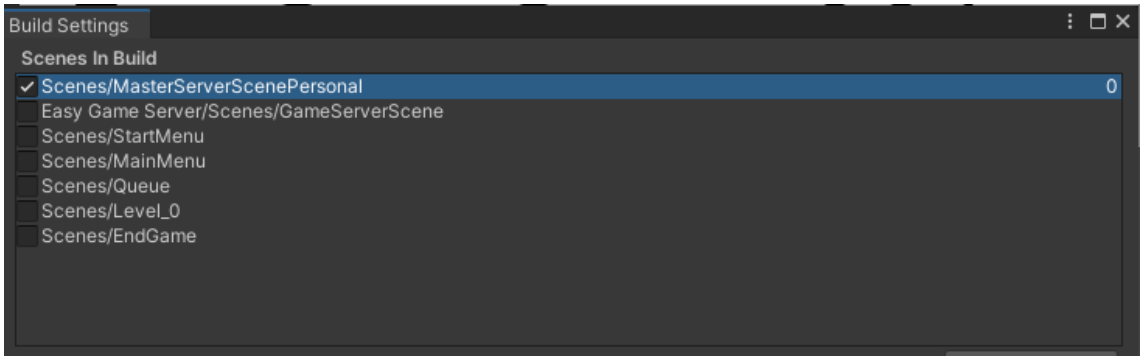
El prefab ***EGS_MasterServer*** debe estar siempre en una build de **Master Server**.


Esta escena puede duplicarse para interactuar mediante los **delegados definidos** y/o añadir más funcionalidades al **Master Server**.


En este ejemplo, vemos como la escena se ha duplicado y se ha añadido un script ***Master Server Controller*** con una variable ***Level Scene Name***. Más adelante se explica la razón.

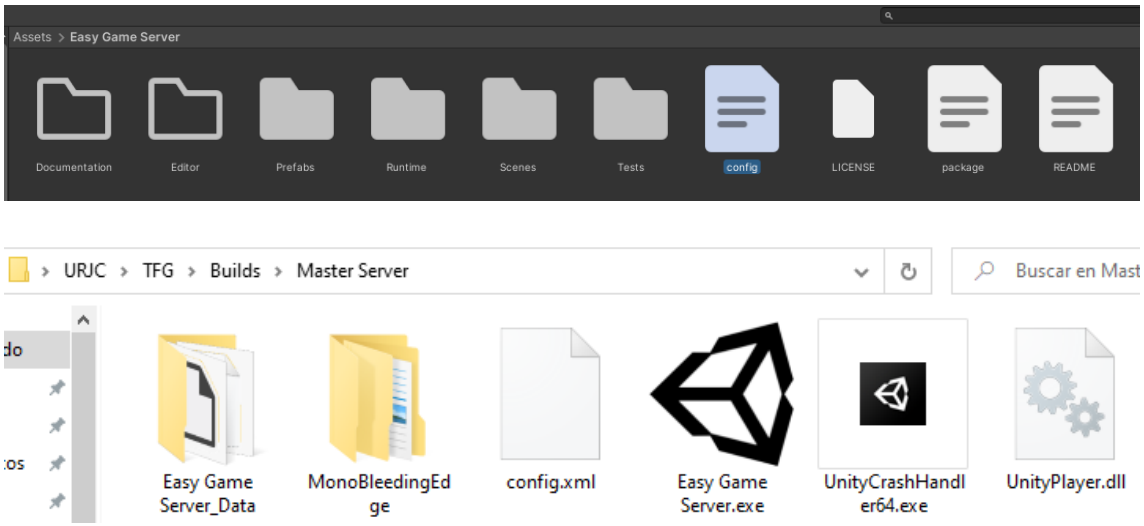
## Despliegue del Master Server

Para desplegar nuestra build del **Master Server**, vamos al apartado ***Build Settings*** de *Unity* y elegimos la escena correcta. De base, solo será necesario añadir la escena que incluye el prefab ***EGS_MasterServer***.




Para este manual de uso, a partir de ahora, trataremos los casos de despliegue como un proyecto real. De esta manera, la escena seleccionada es ***MasterServerScenePersonal*** (la escena duplicada que hemos visto anteriormente).

En el paquete se incluye un archivo ***config.xml*** que debe ir en todas las builds que utilicen la librería, por lo que lo copiamos y pegamos en la ruta de la build.




Con este último paso ya estaría listo el **Master Server**.

## Utilización del Master Server

Cuando queramos iniciar el servidor, simplemente ejecutamos la build del Master Server y nos aparecerá lo siguiente (de base, sin personalizar).


Vemos tres botones con los que podemos interactuar:

- Start Server: Inicia el servidor y comienza a admitir conexiones entrantes.
- Shutdown Server: Desconecta las conexiones actuales y apaga el servidor.
- Autoscroll: Si está marcado, cada mensaje que se pinte en el Server Log moverá hacia abajo todo el texto para ver el último que ha llegado.

**IMPORTANTE [1]**: Si el servidor no está iniciado, no admitirá conexiones y por tanto los clientes no podrán conectarse.

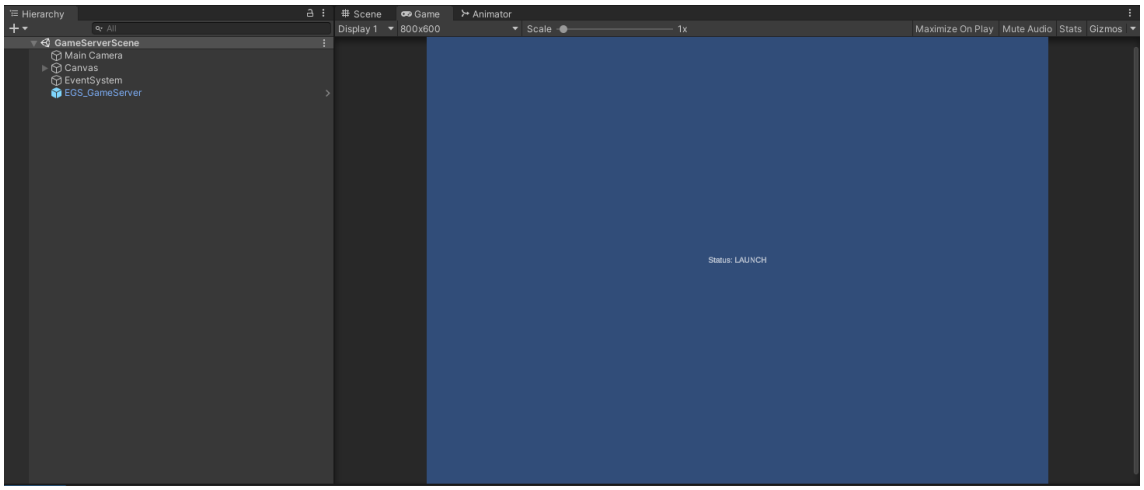
**IMPORTANTE [2]**: El log se escribe en el PersistentDataPath, en la ruta "LocalLow/{company_name}/{product_name}/logs".

# Game Server

## Configuración del Game Server

La librería **Easy Game Server** incluye por defecto una escena ***GameServerScene*** que contiene toda la base necesaria para un servidor de juego.

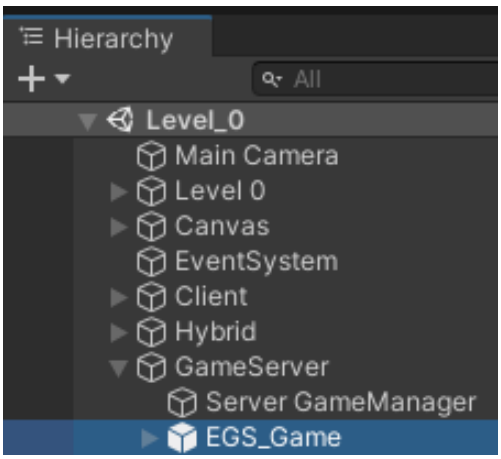
El prefab ***EGS_GameServer*** debe estar siempre en una build de **Game Server**.

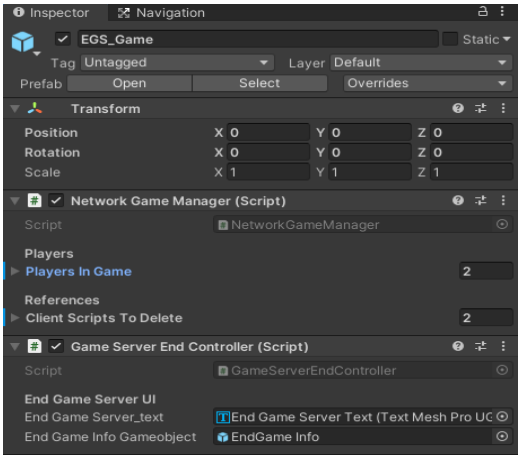



Esta escena, al igual que la del **Master Server**, puede duplicarse para interactuar mediante los **delegados definidos** y/o añadir más funcionalidades al **Game Server**.

En este ejemplo, la escena, se utiliza tal cual viene en el propio paquete, sin añadir nada extra.

Por otro lado, debemos tener una escena (al menos) donde se jugarán las partidas, esta escena la compartirán tanto **Clientes** como **Game Servers**. Esta escena debe contener el prefab ***EGS_Game***.



Por defecto, la librería **Easy Game Server** intentará acceder a una escena cuyo nombre sea "*Level*", pero podemos modificar el nombre de la escena a nuestro gusto e indicárselo a la herramienta.

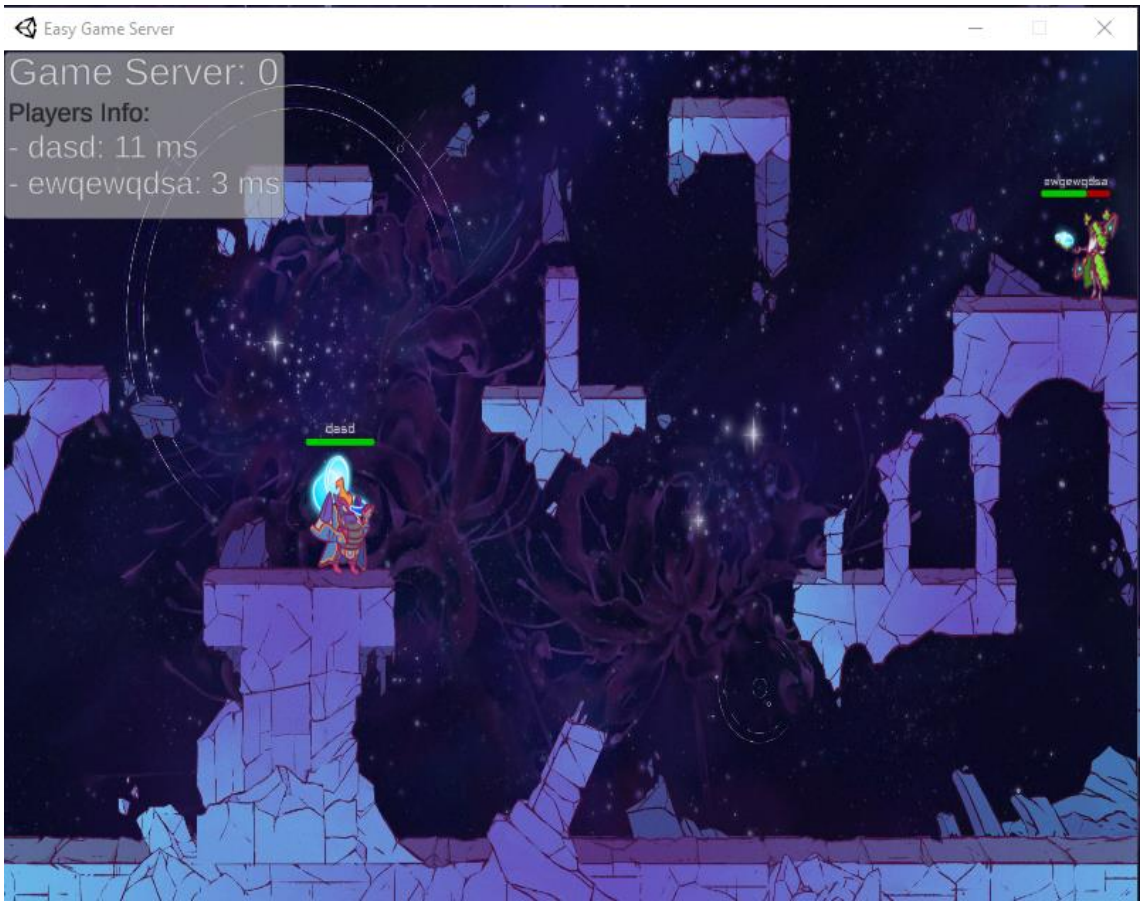
```
using UnityEngine;

/// <summary>
/// Class MasterServerController, to control some behaviour on the Master Server.
/// </summary>
Script de Unity (1 referencia de recurso) | 0 referencias
public class MasterServerController : MonoBehaviour
{
    #region Variables
    [Header("Levels")]
    [Tooltip("Name of the scene where the level is")]
    [SerializeField]
    private string levelSceneName;
    #endregion

    #region Unity Methods
    /// <summary>
    /// Method Start, that is called before the first frame update.
    /// </summary>
    Mensaje de Unity | 0 referencias
    private void Start()
    {
        MasterServerDelegates.onGameFound = SetSceneName;
    }
    #endregion

    #region Class Methods
    /// <summary>
    /// Method SetSceneName, to establish the scene name for the Game Found.
    /// </summary>
    /// <param name="gameFoundData">Data of the Game Found</param>
    1 referencia
    private void SetSceneName(GameFoundData gameFoundData)
    {
        gameFoundData.SetLevelSceneName(levelSceneName);
    }
    #endregion
}
```

Previamente habíamos nombrado la escena **MasterServerScenePersonal** que contenía un script ***Master Server Controller*** con una variable ***Level Scene Name***. Este es el script en cuestión.

Se puede observar que mediante un delegado establecemos el nombre de la escena del objeto ***GameFoundData***, que le dirá al **Game Server** y a los **Clientes** a qué escena deben cambiar para la partida.

Este script es un ejemplo realmente sencillo y se utiliza para solo un nivel de juego, pero… ¿y si tuviéramos múltiples niveles de los cuales se escoge uno aleatorio? ¿o si los usuarios pudieran elegir qué nivel jugar antes de buscar partida? Sería posible programar un sistema que acabe utilizando el delegado ***onGameFound*** para asignar el nombre de la escena elegida.

Por otro lado, el prefab ***EGS_Game*** contiene una pequeña interfaz que muestra información sobre el **Game Server**, indicando el ID de este y el nombre y ping de los **Clientes** conectados.




## *Inicializar la lista de jugadores de la partida*

La clase ***NetworkGameManager*** contiene una lista de jugadores (clase ***NetworkPlayer***) que debe inicializarse llamando a la función ***InitializeNetworkGameManager***.

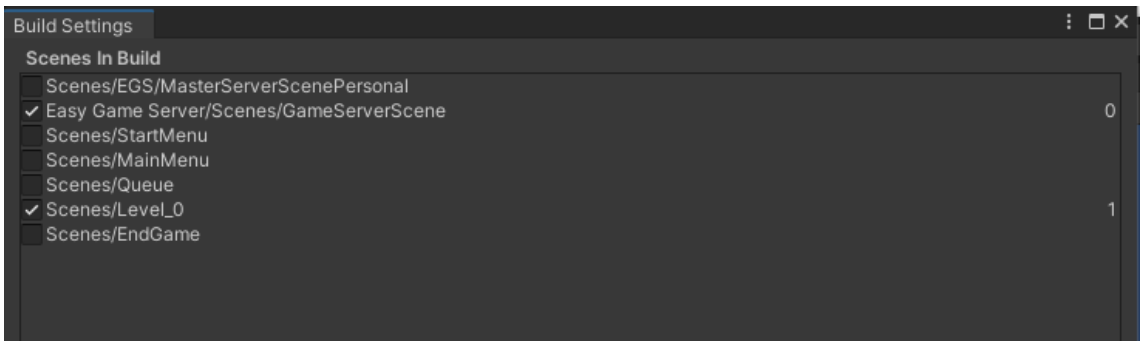
```
List<NetworkPlayer> gameServerPlayers = new List<NetworkPlayer>();
List<PlayerClientController> clientPlayers = new List<PlayerClientController>();

foreach (UserData user in usersForThisGame)
{
    InstantiateNewPlayer(user, gameServerPlayers, clientPlayers);
}

if (EasyGameServerControl.instance.instanceType.Equals(EasyGameServerControl.EnumInstanceType.GameServer))
    NetworkGameManager.instance.InitializeNetworkGameManager(gameServerPlayers);

if (EasyGameServerControl.instance.instanceType.Equals(EasyGameServerControl.EnumInstanceType.Client))
    ClientGameManager.instance.SetPlayers(clientPlayers);
```

Como se ve en la imagen, se instancian los jugadores y se inicializa el ***NetworkGameManager*** con la lista de jugadores que recibe.

## Despliegue del Game Server

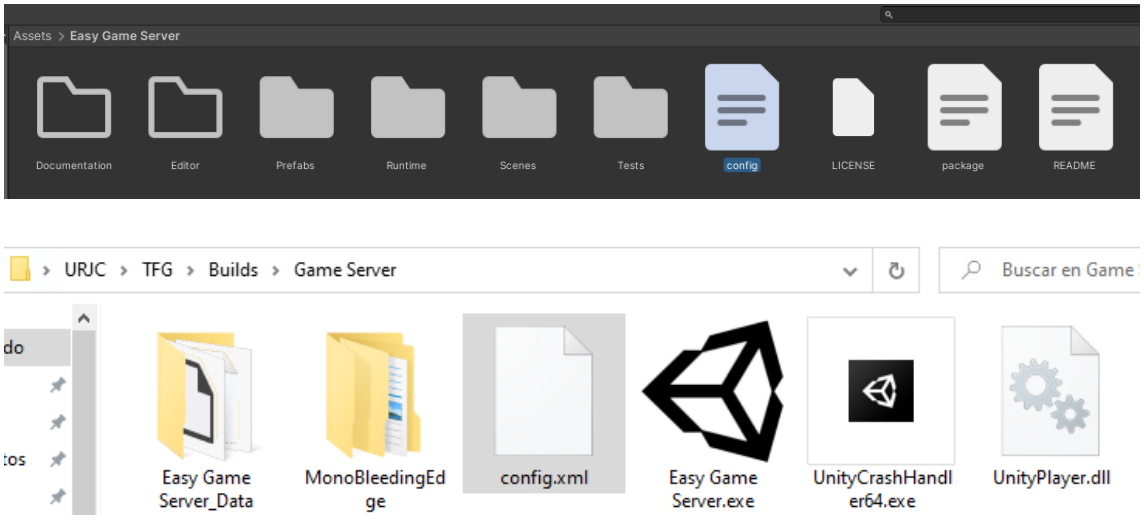
Para desplegar nuestra build del **Game Server**, vamos al apartado ***Build Settings*** de *Unity* y elegimos las escenas correctas. De base, será necesario añadir dos escenas:

- La escena del **Game Server**, que incluye el prefab ***EGS_GameServer***.
- La escena de juego, que incluye el prefab ***EGS_Game***.




En la imagen, se observa como se han elegido las escenas ***GameServerScene*** (escena del **Game Server**) y ***Level_0*** (escena de la partida).

De nuevo, una vez tengamos la build, llevaremos una copia del archivo ***config.xml*** a la ruta de la build.




De esta manera ya tendríamos listo el **Game Server**.

## Utilización del Game Server

El **Game Server** no requiere de intervención alguna por nuestra parte y jamás debemos ejecutar manualmente su build.

En su lugar, es el **Master Server** quien crea instancias del **Game Server** con los argumentos necesarios para comunicarse entre ellos.

Para que pueda crear estas instancias, debemos definir el *path* o ruta del Game Server en el ***config.xml***.

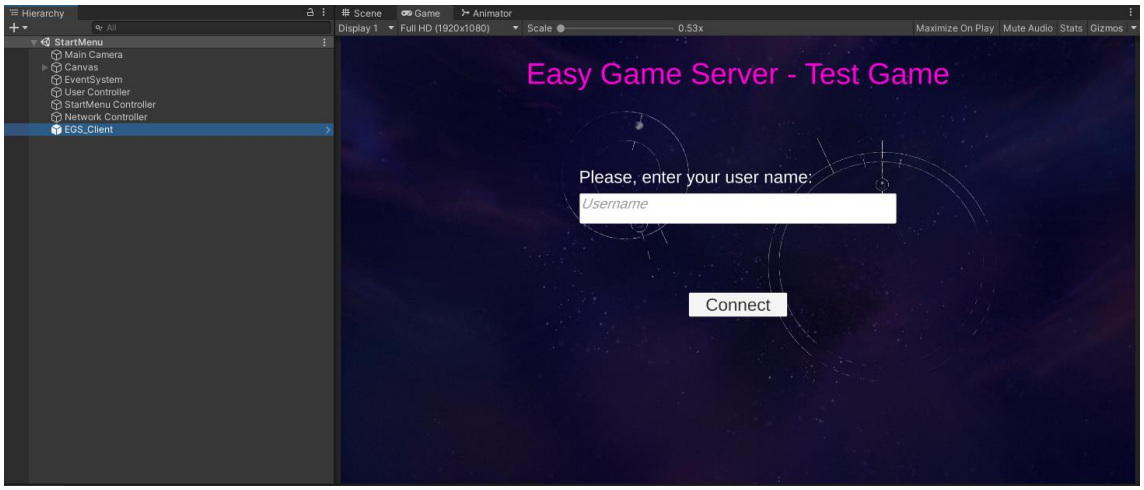
```
<game-server>
    <path>C:\\Users\\Samue\\Desktop\\URJC\\TFG\\Builds\\Game Server\\Easy Game Server.exe</path>
</game-server>
```

# Client

## Configuración del Cliente

Para la configuración del **Client**, el proceso es notablemente más complejo, ya que la mayoría de las funciones deben llamarse por código.

El prefab ***EGS_Client*** debe estar siempre en una build de **Client** y en la primera escena a ejecutar.




En este proyecto de prueba vemos una escena de inicio que contiene al prefab **EGS_Client** y varios controladores propios del proyecto de prueba.

Vamos a dividir este apartado de configuración en subapartados, según avanza el flujo lógico de un videojuego multijugador online.

### *Preparación de los datos a enviar*

#### *Datos de Usuario*

La clase UserData es modificable y podemos añadir las variables que necesitemos.

```
/// <summary>
/// Class UserData, that contains the structure of an user in the server.
/// [MODIFIABLE].
/// </summary>
[System.Serializable]
99+ referencias
public class UserData : BaseUserData
{
    #region Variables
    [Header("Game")]
    [Tooltip("Selected Character ID")]
    [SerializeField]
    protected int selectedCharacterID; // You can delete this! It was for the prototype.
    #endregion
```

En este caso, se ha añadido una variable protegida serializada para comprobar qué personaje ha escogido el jugador.

Cada variable que añadamos, debe llevar la anotación **[SerializeField]** para que se envíe en los mensajes de intercambio de información entre servidores y clientes.

*Inputs de los jugadores*

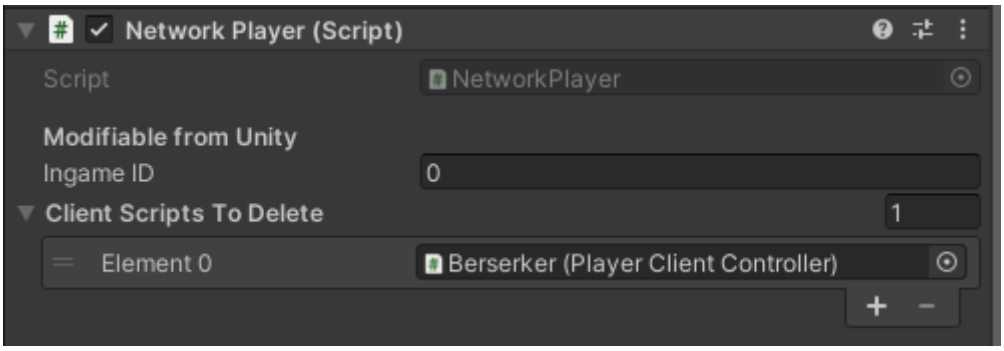
La clase PlayerInputs es modificable y podemos añadir las variables que necesitemos.

```
/// <summary>
/// Class PlayerInputs, that defines how inputs are stored.
/// [MODIFIABLE]
/// </summary>
12 referencias
public class PlayerInputs
{
    #region Variables
    [Header("Player Inputs")]
    [Tooltip("Player's ingame ID")]
    [SerializeField]
    private int ingameID;

    [Tooltip("Array of player boolInputs.")]
    [SerializeField]
    private bool[] boolInputs;
    #endregion
```

No debemos eliminar las variables actuales. y de nuevo las variables añadidas deben llevar la anotación **[SerializeField]** para que se envíe en los mensajes de intercambio de información entre servidor y clientes.

*Script NetworkPlayer*

El ***GameObject*** que represente a los jugadores en la escena de juego, debe llevar el script **NetworkPlayer** asignado, y la variable ***ingameID*** debe indicar el identificador del jugador para la partida que juega. Es decir, si son dos jugadores, indicar quién es el 0 y quién es el 1.




Además, este script solo se ejecutará en el **Game Server**, ya que se destruirá si se está ejecutando en otro tipo de instancia, y cuenta con una lista de scripts para eliminar donde podemos indicar aquellos que no queremos que se ejecuten en el **Game Server**.

### *Conexión al Master Server*

Para que el **Client** establezca una conexión al **Master Server**, debemos llamar desde nuestro código al script ***Client***, al método ***ConnectToServer***. Podemos hacerlo escribiendo lo siguiente:

```
Client.instance.ConnectToServer();
```

Por lo general la mayoría de scripts a los que podemos acceder en la librería **Easy Game Server**, utilizan el patrón *Singleton* para establecer una única instancia pública accesible desde cualquier script.

Si el **Master Server** está disponible, el **Client** se conectará y comenzará el intercambio de mensajes.

### *Desconexión del Master Server*

Para que el **Client** se desconecte del **Master Server**, debemos llamar desde nuestro código al script ***Client***, al método ***DisconnectFromServer***. Podemos hacerlo escribiendo lo siguiente:

```
Client.instance.DisconnectFromServer();
```

Si el **Client** está conectado en ese momento, pedirá al **Master Server** desconectarse y, tras recibir la confirmación, se desconectará.

### *Búsqueda de partida*

Para que un **Client** entre en cola de partida, para poder jugar una, debe indicarlo al servidor ejecutando la siguiente línea:

```
Client.instance.JoinQueue();
```

### *Dejar de buscar partida*

Para que un **Client** salga de la cola de búsqueda de partida, debe indicarlo al servidor ejecutando la siguiente línea:

```
Client.instance.LeaveQueue();
```

### *Cambio de escena a la escena de juego*

Cuando un **Client** encuentra partida, se reciben los datos en un objeto ***GameFoundData***.

Existe un delegado ***OnGameFound*** que permite aprovechar esa información y detectar el momento exacto para ejecutar código propio en el Client como, por ejemplo, un cambio a la escena de juego, o a un lobby, etc.

Es importante tener en cuenta que los jugadores siempre deben cambiar a la escena de juego en algún momento y que estén cuando el **Game Server** haya cambiado al nivel, quien cambiará una vez todos los jugadores estén conectados a él.

### *Cambio al Game Server*

Cuando el **Game Server** correspondiente se haya iniciado, avisará al **Master Server**, quien a su vez ordenará a los **Clients** que participen en la partida a cambiar de conexión al **Game Server**.

### *Inicio de partida*

Una vez todos los jugadores se hayan conectado, se iniciará la partida, y el **Game Server** avisará tanto a **Clients** (jugadores) como al **Master Server**, enviándoles un objeto ***UpdateData***, que habitualmente es el que se utiliza para el intercambio constante de información, pero en este caso el que se envía cumple la función de datos de inicio de partida.

Automáticamente el **Game Server** comenzará a enviar información al **cliente**.

### *Envío de inputs al Game Server*

Para que el cliente envíe sus inputs al Game Server para realizar los cálculos, debe utilizarse el delegado ***OnGameSenderTick*** y asignarle una función que envíe un ***NetworkMessage*** al **Game Server**.

Un ejemplo de ello es la siguiente función:

```
1 referencia
public void SendInputsToServer()
{
    PlayerInputs playerInputs = new PlayerInputs(ingameID, inputs);
    string jsonMessage = JsonUtility.ToJson(playerInputs);
    Client.instance.SendMessage(GameServerMessageTypes.PLAYER_INPUT, jsonMessage);
}
```

Que se asigna al delegado de la siguiente manera:

```
if (EasyGameServerControl.instance.instanceType.Equals(EasyGameServerControl.EnumInstanceType.Client))
{
    isThisPlayer = (ingameID == Client.instance.GetUser().GetIngameID());

    if (isThisPlayer)
    {
        ClientDelegates.onGameSenderTick = SendInputsToServer;
    }
}
```

### *Fin de partida*

Una vez la partida haya acabado, según las condiciones que hayamos definido, el **Game Server** avisará tanto a **Clients** (jugadores) como al **Master Server**, enviándoles un objeto ***GameEndData***.

Una vez reciban este mensaje, los **Clients** enviarán una petición al **Game Server** para volver al **Master Server**.

### *Desconectarse de una partida en curso*

Si un jugador quiere desconectarse de la partida, podrá hacerlo si se ejecuta el siguiente fragmento de código:

```
Client.instance.LeaveGame();
```

Además, si decide cerrar el juego directamente, el **Game Server** lo detectará y lo desconectará, indicando esta información al **Master Server**.

### *Vuelta al Master Server*

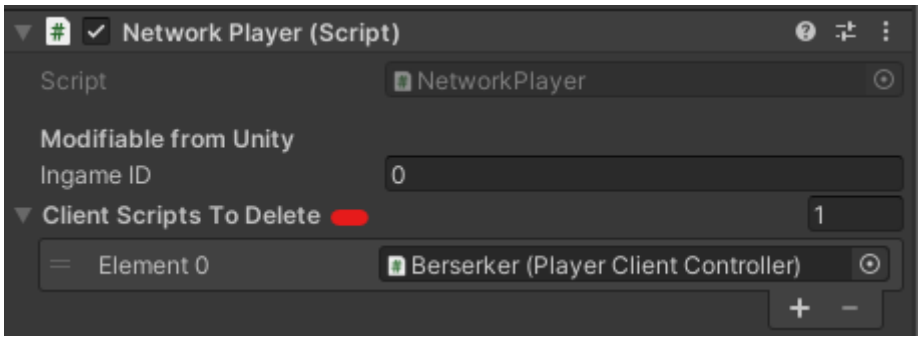
Cuando el **Game Server** devuelva el mensaje que enviaron los **Clients** para volver al **Master Server**, cambiará la conexión de los **Clients** del **Game Server** al **Master Server**, y podrán volver a buscar partida o lo que nosotros deseemos.

### *Distinguir entre Client y Game Server en la escena de juego*

Cuando queremos ejecutar código solo en **Cliente** o solo en el **Game Server**, tenemos dos formas de hacerlo:

#### *Scripts a eliminar*

Como hemos visto previamente con el script ***Network Player***, algunos objetos de la herramienta incluyen listas de objetos a eliminar de la ejecución del **Game Server**.




#### *Comprobaciones en código*

Podemos definir en código, en los métodos ***Start*** o ***Awake*** que nos proporciona *Unity*, las siguientes comprobaciones:

Solo **Cliente**

```
if (EasyGameServerControl.instance.instanceType.Equals(EasyGameServerControl.EnumInstanceType.Client))
```

Solo **Game Server**

```
if (EasyGameServerControl.instance.instanceType.Equals(EasyGameServerControl.EnumInstanceType.GameServer))
```

Ejemplo de uso:

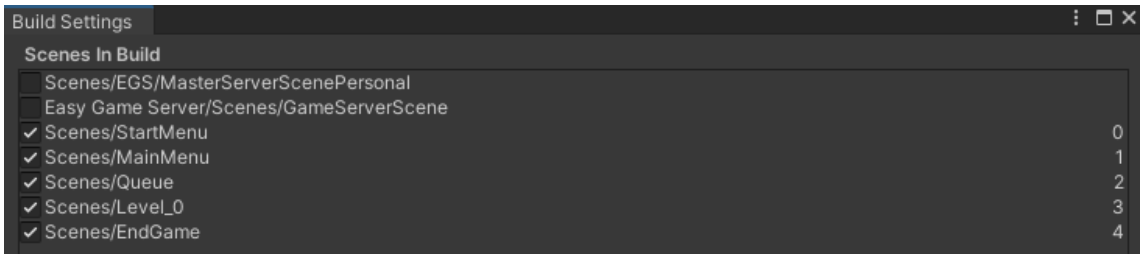
```
/// <summary>
/// Method Start, executed before the first frame.
/// </summary>
Mensaje de Unity | 0 referencias
private void Start()
{
    if (EasyGameServerControl.instance.instanceType.Equals(EasyGameServerControl.EnumInstanceType.Client))
    {
        // Delegates.
        ClientDelegates.onGameStart -= OnGameStart;
        ClientDelegates.onGameStart += OnGameStart;

        ClientDelegates.onLeaveGame -= OnLeavingGame;
        ClientDelegates.onLeaveGame += OnLeavingGame;
    }
    else if (EasyGameServerControl.instance.instanceType.Equals(EasyGameServerControl.EnumInstanceType.GameServer))
    {
        Destroy(this);
    }
}
```

En el ejemplo anterior podemos ver como en un script que solo queremos que se ejecute en **Cliente**, asignamos a los delegados que nos proporciona la herramienta las funciones que nos interesan, y en el caso del **Game Server**, este script directamente se destruye.

## Despliegue del Client

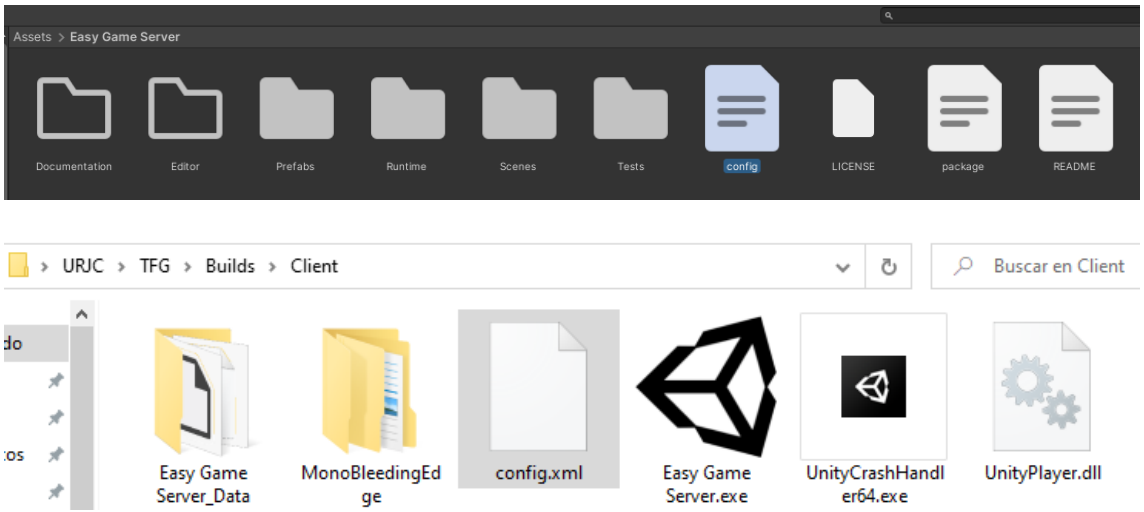
Para desplegar nuestra build del **Client**, vamos al apartado ***Build Settings*** de *Unity* y elegimos las escenas correctas. De base, será necesario añadir dos escenas:

- La primera escena, que debe incluir el prefab ***EGS_Client***.
- La escena de juego.




En la imagen, se observa como se han elegido las distintas escenas y se incluye la escena ***Level_0*** (escena de la partida).

De nuevo, una vez tengamos la build, llevaremos una copia del archivo ***config.xml*** a la ruta de la build.




De esta manera ya tendríamos listo el **Client**.

## Utilización del Client

El **Client** es el juego en sí, por lo que su ejecución debe ser la deseada por nosotros para la experiencia de los jugadores.

# Sistema de delegados

**Easy Game Server** cuenta con un sistema de delegados que en ciertos momentos definidos de la ejecución del programa permite inyectar código externo. De esta manera, es posible personalizar la ejecución de la herramienta y adaptarla a las necesidades de cada videojuego.

## Delegados del Master Server

Los delegados que pueden utilizarse en el **Master Server** están definidos en la clase ***MasterServerDelegates*** y son:

### *OnMessageReceive*

**Definición**:

```
public static Action<NetworkMessage> onMessageReceive;
```

**Parámetros de tipo**:

- NetworkMessage: Mensaje recibido no predefinido.

**Ejecución**:

Cuando el **Master Server** recibe un mensaje que **no viene predefinido por la herramienta**.

### *OnMasterServerStart*

**Definición**:

```
public static Action onMasterServerStart;
```

**Parámetros de tipo**:

No recibe.

**Funcionamiento**:

Cuando se inicia el **Master Server**, antes de empezar a recibir jugadores.

### *OnMasterServerShutdown*

**Definición**:

```
public static Action onMasterServerShutdown;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando se cierra el **Master Server**.

### *OnUserJoinServer*

**Definición**:

```
public static Action<UserData, bool> onUserJoinServer;
```

**Parámetros de tipo**:

- UserData: Usuario del **Cliente** que se conecta al **Master Server**.
- bool: Valor booleano que indica si el cliente está volviendo al **Master Server** tras una partida (**true**) o si se acaba de conectar (**false**).

**Ejecución**:

Cuando un **cliente** se conecta al **Master Server**.

### *OnUserRegister*

**Definición**:

```
public static Action<UserData> onUserRegister;
```

**Parámetros de tipo**:

- UserData: Usuario que se registra en **Master Server**.

**Ejecución**:

Cuando un usuario es registrado en el **Master Server**.

### *OnUserConnect*

**Definición**:

```
public static Action<UserData> onUserConnect;
```

**Parámetros de tipo**:

- UserData: Usuario que se establece como conectado en el **Master Server**.

**Ejecución**:

Cuando un usuario se guarda como conectado.

### *OnUserDisconnect*

**Definición**:

```
public static Action<UserData> onUserDisconnect;
```

**Parámetros de tipo**:

- UserData: Usuario que se establece como desconectado en el **Master Server**.

**Ejecución**:

Cuando un usuario se guarda como desconectado.

### *OnUserDelete*

**Definición**:

```
public static Action<UserData> onUserDelete;
```

**Parámetros de tipo**:

- UserData: Usuario que se elimina del **Master Server**.

**Ejecución**:

Cuando un usuario va a ser eliminado del **Master Server**.

### *OnUserJoinQueue*

**Definición**:

```
public static Action<UserData> onUserJoinQueue;
```

**Parámetros de tipo**:

- UserData: Usuario que entra a la cola de búsqueda de partida.

**Ejecución**:

Cuando un usuario entra a la cola de búsqueda de partida.

### *OnUserLeaveQueue*

**Definición**:

```
public static Action<UserData> onUserLeaveQueue;
```

**Parámetros de tipo**:

- UserData: Usuario que sale de la cola de búsqueda de partida.

**Ejecución**:

Cuando un usuario sale de la cola de búsqueda de partida.

### *OnUserDisconnectToGameServer*

**Definición**:

```
public static Action<UserData> onUserDisconnectToGameServer;
```

**Parámetros de tipo**:

- UserData: Usuario que se desconecta **PROVISIONALMENTE** del **Master Server** para ir al **Game Server**.

**Ejecución**:

Cuando se envía a un usuario la orden para desconectarse del **Master Server** para ir al **Game Server** que le corresponde tras haber encontrado partida.

### *OnUserLeaveGame*

**Definición**:

```
public static Action<UserData> onUserLeaveGame;
```

**Parámetros de tipo**:

- UserData: Usuario que ha abandonado su partida**.**

**Ejecución**:

Cuando un usuario abandona su partida en el **Game Server**, ya sea por decisión propia o por desconexión.

### *OnGameServerCreated*

**Definición**:

```
public static Action<int> onGameServerCreated;
```

**Parámetros de tipo**:

- int: Identificador (id) del **Game Server** que se ha creado.

**Ejecución**:

Cuando un **Game Server** se crea y se conecta al servidor.

### *OnGameServerReady*

**Definición**:

```
public static Action<int> onGameServerReady;
```

**Parámetros de tipo**:

- int: Identificador (id) del **Game Server** que está listo para recibir a jugadores.

**Ejecución**:

Cuando un **Game Server** está listo para recibir a los jugadores.

### *OnGameServerClosed*

**Definición**:

```
public static Action<int> onGameServerClosed;
```

**Parámetros de tipo**:

- int: Identificador (id) del **Game Server** que se desconecta y se cierra.

**Ejecución**:

Cuando un **Game Server** se desconecta del **Master Server** y se cierra.

### *OnGameFound*

**Definición**:

```
public static Action<GameFoundData> onGameFound;
```

**Parámetros de tipo**:

- GameFoundData: Objeto que contiene la información de la partida que se ha creado con los jugadores que estaban esperando en la cola.

**Ejecución**:

Cuando se crea una partida debido a que hay suficientes jugadores en cola.

### *OnGameStart*

**Definición**:

```
public static Action<UpdateData> onGameStart;
```

**Parámetros de tipo**:

- UpdateData: Objeto que contiene la información de la partida que acaba de empezar.

**Ejecución**:

Cuando se inicia una partida.

### *OnGameEnd*

**Definición**:

```
public static Action<GameEndData> onGameEnd;
```

**Parámetros de tipo**:

- GameEndData: Objeto que contiene la información de la partida que ha terminado.

**Ejecución**:

Cuando finaliza una partida.

### *OnReceiveClientRTT*

**Definición**:

```
public static Action<int, long> onReceiveClientRTT;
```

**Parámetros de tipo**:

- int: Identificador del usuario cuyo **Client** ha devuelto el ***Round Trip Time***.
- long: Tiempo en milisegundos que ha tardado en ir y volver el ***Round Trip Time***.

**Ejecución**:

Cuando se recibe un ***Round Trip Time*** de un **Client** conectado.

### *OnReceiveGameServerRTT*

**Definición**:

```
public static Action<int, long> onGameServerRTT;
```

**Parámetros de tipo**:

- int: Identificador del **Game Server** que ha devuelto el ***Round Trip Time***.
- long: Tiempo en milisegundos que ha tardado en ir y volver el ***Round Trip Time***.

**Ejecución**:

Cuando se recibe un ***Round Trip Time*** de un **Game Server** conectado.

## Delegados del Game Server

Los delegados que pueden utilizarse en el **Game Server** están definidos en la clase ***GameServerDelegates*** y son:

### *OnMasterServerMessageReceive*

**Definición**:

```
public static Action<NetworkMessage> onMasterServerMessageReceive;
```

**Parámetros de tipo**:

- NetworkMessage: Mensaje recibido no predefinido.

**Ejecución**:

Cuando el **Game Server** recibe del **Master Server** un mensaje que **no viene predefinido por la herramienta**.

### *OnClientMessageReceive*

**Definición**:

```
public static Action<NetworkMessage, GameServerServerSocketHandler, Socket> onClientMessageReceive;
```

**Parámetros de tipo**:

- NetworkMessage: Mensaje recibido no predefinido.
- GameServerServerSocketHandler: Controlador del socket de servidor del **Game Server**.
- Socket: Socket del **Cliente** que envía el mensaje.

**Ejecución**:

Cuando el **Game Server** recibe de un **Client** un mensaje que **no viene predefinido por la herramienta**.

### *OnGameServerCreated*

**Definición**:

```
public static Action onGameServerCreated;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando se inicia el **Game Server**, antes de conectarse al **Master Server**.

### *OnGameServerShutdown*

**Definición**:

```
public static Action onGameServerShutdown;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando se va a cerrar el **Game Server**.

### *OnMasterServerRefusesConnection*

**Definición**:

```
public static Action onMasterServerRefusesConnection;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando se produce un intento fallido de conectar al **Master Server**.

### *OnCantConnectToMasterServer*

**Definición**:

```
public static Action onCantConnectToMasterServer;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando tras varios intentos de conexión finalmente no ha logrado conectar al **Master Server**.

### *OnConnectToMasterServer*

**Definición**:

```
public static Action onConnectToMasterServer;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando el **Game Server** se conecta al **Master Server**.

### *OnReadyToConnectPlayers*

**Definición**:

```
public static Action onReadyToConnectPlayers;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando el **Game Server** ya está listo para recibir conexiones de **Clientes** de los jugadores que deben conectarse a él.

### *OnUserJoinServer*

**Definición**:

```
public static Action<UserData> onUserJoinServer;
```

**Parámetros de tipo**:

- UserData: Usuario del **Cliente** que se conecta al **Game Server**.

**Ejecución**:

Cuando se conecta un jugador al **Game Server**.

### *OnUserConnect*

**Definición**:

```
public static Action<UserData> onUserConnect;
```

**Parámetros de tipo**:

- UserData: Usuario del **Cliente** que se registra como conectado al **Game Server**.

**Ejecución**:

Cuando se registra como conectado un jugador en el **Game Server**.

### *OnUserDisconnect*

**Definición**:

```
public static Action<UserData> onUserDisconnect;
```

**Parámetros de tipo**:

- UserData: Usuario que se establece como desconectado del **Game Server**.

**Ejecución**:

Cuando se registra como desconectado un jugador.

### *OnUserDisconnectToMasterServer*

**Definición**:

```
public static Action<UserData> onUserDisconnectToMasterServer;
```

**Parámetros de tipo**:

- UserData: Usuario que se desconecta del **Game Server** para volver al **Master Server**.

**Ejecución**:

Cuando se envía a un usuario la orden para desconectarse del **Game Server** para volver al **Master Server** tras finalizar la partida o salir de esta.

### *OnAllPlayersConnected*

**Definición**:

```
public static Action onAllPlayersConnected;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando se han conectado todos los jugadores previstos para jugar la partida en este **Game Server**.

### *OnGameStart*

**Definición**:

```
public static Action<UpdateData> onGameStart;
```

**Parámetros de tipo**:

- UpdateData: Objeto que contiene la información de la partida que acaba de empezar.

**Ejecución**:

Cuando se inicia una partida.

### *OnGameEnd*

**Definición**:

```
public static Action<GameEndData> onGameEnd;
```

**Parámetros de tipo**:

- GameEndData: Objeto que contiene la información de la partida que ha terminado.

**Ejecución**:

Cuando finaliza una partida.

### *OnPlayerSendInput*

**Definición**:

```
public static Action<NetworkPlayer, PlayerInputs> onPlayerSendInput;
```

**Parámetros de tipo**:

- NetworkPlayer: Objeto que contiene la información del jugador que envía los datos.
- PlayerInputs: Objeto que contiene los inputs del jugador.

**Ejecución**:

Cada vez que un **Cliente** envía los datos sobre sus inputs.

### *OnPlayerLeaveGame*

**Definición**:

```
public static Action<NetworkPlayer> onPlayerLeaveGame;
```

**Parámetros de tipo**:

- NetworkPlayer: Objeto que contiene la información del jugador que abandona la partida.

**Ejecución**:

Cuando un jugador abandona la partida.

### *OnTick*

**Definición**:

```
public static Action<UpdateData> onTick;
```

**Parámetros de tipo**:

- UpdateData: Objeto creado para contener la información de una actualización del estado de la partida.

**Ejecución**:

Cada vez que se produce un *Tick* en el **Game Server**. Un *Tick* ejecuta todos los cálculos del Game Server y se produce cada ***TICK_RATE***, que es una variable definida por el parámetro ***calculations-per-second*** en el ***config.xml***.

### *OnProcessPlayer*

**Definición**:

```
public static Action<NetworkPlayer, UpdateData, long> onProcessPlayer;
```

**Parámetros de tipo**:

- NetworkPlayer: Objeto que contiene la información del jugador que se procesa.
- UpdateData: Objeto que contiene la información de una actualización del estado de la partida.
- long: Valor del parámetro ***TICK_RATE***, que es el tiempo entre *Ticks* o cálculos del **Game Server**.

**Ejecución**:

Cada vez que se deben procesar los datos de un jugador.

**IMPORTANTE**: Este delegado debe llamarse para poder ejecutar los cálculos del jugador.

### *OnRTT*

**Definición**:

```
public static Action<long> onRTT;
```

**Parámetros de tipo**:

- long: Tiempo en milisegundos que tardó el anterior ***Round Trip Time*** en ser enviado por el **Master Server** hasta su recepción.

**Ejecución**:

Cada vez que se recibe un ***Round Trip Time*** para devolver al **Master Server**.

### *OnReceiveClientRTT*

**Definición**:

```
public static Action<int, long> onReceiveClientRTT;
```

**Parámetros de tipo**:

- int: Identificador del usuario cuyo **Client** ha devuelto el ***Round Trip Time***.
- long: Tiempo en milisegundos que ha tardado en ir y volver el ***Round Trip Time***.

**Ejecución**:

Cuando se recibe un ***Round Trip Time*** de un **Client** conectado.

## Delegados del Client

Los delegados que pueden utilizarse en el **Client** están definidos en la clase ***ClientDelegates*** y son:

### *OnMessageReceive*

**Definición**:

```
public static Action<NetworkMessage> onMessageReceive;
```

**Parámetros de tipo**:

- NetworkMessage: Mensaje recibido no predefinido.

**Ejecución**:

Cuando el **Client** recibe un mensaje que **no viene predefinido por la herramienta**.

### *OnUserCreate*

**Definición**:

```
public static Action<UserData> onUserCreate;
```

**Parámetros de tipo**:

- UserData: Datos del usuario que acaba de ser creado.

**Ejecución**:

Cuando se crean los datos del usuario, para poder introducirle los parámetros necesarios.

### *OnServerRefusesConnection*

**Definición**:

```
public static Action onServerRefusesConnection;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando se produce un intento fallido de conectar al **Master Server**.

### *OnCantConnectToServer*

**Definición**:

```
public static Action onCantConnectToServer;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando tras varios intentos de conexión finalmente no ha logrado conectar al **Master Server**.

### *OnConnect*

**Definición**:

```
public static Action<EasyGameServerControl.EnumInstanceType> onConnect;
```

**Parámetros de tipo**:

- EasyGameServerControl.EnumInstanceType: Tipo de instancia a la que se está conectando el **Cliente**: **Master Server** o **Game Server**.

**Ejecución**:

Cuando el **Client** se conecta a un servidor.

### *OnJoinMasterServer*

**Definición**:

```
public static Action<UserData> onJoinMasterServer;
```

**Parámetros de tipo**:

- UserData: Datos del usuario del **Cliente** una vez los ha tratado y devuelto el **Master Server**.

**Ejecución**:

Cuando el **Master Server** conecta el **Cliente** y le devuelve su usuario.

### *OnJoinGameServer*

**Definición**:

```
public static Action onJoinGameServer;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando el **Game Server** conecta el **Cliente**.

### *OnDisconnect*

**Definición**:

```
public static Action onDisconnect;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando el **Cliente** se desconecta del **Master Server**.

### *OnUserDelete*

**Definición**:

```
public static Action<UserData> onUserDelete;
```

**Parámetros de tipo**:

- UserData: Datos del usuario del **Cliente**, que han sido eliminados del servidor.

**Ejecución**:

Cuando se eliminan los datos de usuario del **Cliente**.

### *OnPrepareToChangeFromMasterToGameServer*

**Definición**:

```
public static Action<string, int> onPrepareToChangeFromMasterToGameServer;
```

**Parámetros de tipo**:

- string: IP del **Game Server** a la que conectarse.
- int: Puerto del **Game Server** al que conectarse.

**Ejecución**:

Cuando se solicita al **Cliente** que cambie del **Master Server** al **Game Server**.

### *OnChangeFromMasterToGameServer*

**Definición**:

```
public static Action<string, int> onChangeFromMasterToGameServer;
```

**Parámetros de tipo**:

- string: IP del **Game Server** al que se conecta.
- int: Puerto del **Game Server** al que se conecta.

**Ejecución**:

Cuando el **Cliente** cambia del **Master Server** al **Game Server**.

### *OnLeaveGame*

**Definición**:

```
public static Action onLeaveGame;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando el **Cliente** abandona la partida que está jugando.

### *OnPrepareToChangeFromGameToMasterServer*

**Definición**:

```
public static Action<string, int> onChangeFromMasterToGameServer;
```

**Parámetros de tipo**:

- string: IP del **Master Server** para conectarse.
- int: Puerto del **Master Server** para conectarse.

**Ejecución**:

Cuando se solicita al **Cliente** que cambie del **Game Server** al **Master Server**.

### *OnChangeFromGameToMasterServer*

**Definición**:

```
public static Action<string, int> onChangeFromGameToMasterServer;
```

**Parámetros de tipo**:

- string: IP del **Master Server** al que se conecta.
- int: Puerto del **Master Server** al que se conecta.

**Ejecución**:

Cuando el **Cliente** cambia del **Game Server** al **Master Server**.

### *OnReturnToMasterServer*

**Definición**:

```
public static Action<UserData> onReturnToMasterServer;
```

**Parámetros de tipo**:

- UserData: Datos del usuario del cliente tras volver al **Master Server**.

**Ejecución**:

Cuando el **Cliente** vuelve al **Master Server** y este le envía de vuelta la información de su usuario actualizada.

### *OnServerClosed*

**Definición**:

```
public static Action onServerClosed;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando el **Master Server** se cierra y desconecta a todos los **Clientes** activos.

### *OnGameFound*

**Definición**:

```
public static Action<GameFoundData> onGameFound;
```

**Parámetros de tipo**:

- GameFoundData: Objeto que contiene la información de la partida que se ha creado para el **Cliente**.

**Ejecución**:

Cuando se crea una partida para el **Cliente**.

### *OnGameStart*

**Definición**:

```
public static Action<UpdateData> onGameStart;
```

**Parámetros de tipo**:

- UpdateData: Objeto que contiene la información de la partida que acaba de empezar.

**Ejecución**:

Cuando se inicia una partida.

### *OnGameEnd*

**Definición**:

```
public static Action<GameEndData> onGameEnd;
```

**Parámetros de tipo**:

- GameEndData: Objeto que contiene la información de la partida que ha terminado.

**Ejecución**:

Cuando finaliza una partida.

### *OnGameSenderTick*

**Definición**:

```
public static Action onGameSenderTick;
```

**Parámetros de tipo**:

No recibe.

**Ejecución**:

Cuando el ***ClientInGameSender*** envía información al Game Server de los datos del **Cliente**.

### *OnGameUpdateReceive*

**Definición**:

```
public static Action<UpdateData> onGameUpdateReceive;
```

**Parámetros de tipo**:

- UpdateData: Objeto que contiene la información de una actualización del estado de la partida.

**Ejecución**:

Cuando el **Client** recibe un mensaje por parte del **Game Server** con el estado actual de la partida.

### *OnAnotherPlayerLeaveGame*

**Definición**:

```
public static Action<PlayerData> onAnotherPlayerLeaveGame;
```

**Parámetros de tipo**:

- PlayerData: Objeto que contiene la información de un jugador que ha abandonado la partida en curso.

**Ejecución**:

Cuando otro jugador (**Client**) ha abandonado la partida actual.

### *OnRTT*

**Definición**:

```
public static Action<long> onRTT;
```

**Parámetros de tipo**:

- long: Tiempo en milisegundos que tardó el anterior ***Round Trip Time*** en ser enviado por el **Master Server** hasta su recepción.

**Ejecución**:

Cada vez que se recibe un ***Round Trip Time*** para devolver al **Master Server**.

# Dudas sobre la herramienta

Para cualquier duda sobre la herramienta, apartados que puedan no estar definidos en este manual de uso, o posibles fallos de la propia herramienta, contactar al siguiente correo:

**samuelriosdev@gmail.com**

Muchas gracias por utilizar la herramienta **Easy Game Server**, y espero que te sea de gran utilidad para tus videojuegos multijugador online.