

Master Server Kit

Documentation

Welcome and thank you for your purchase!

Table of contents

- [Introduction](#)
- [Setup](#)
- [Client API](#)
- [Database](#)
- [F.A.Q.](#)
- [Version history](#)
- [Support and feedback](#)

Introduction

Master Server Kit is a complete library that allows you to host your UNET-based multiplayer game on a dedicated server of your own.

Why?

If you are using [UNET](#) as the networking library for your game, [Unity Multiplayer Services](#) are an excellent way to get player-hosted games up and running without worrying about the details of setting up your own server infrastructure. But, sometimes, you just need to host your game on a dedicated server. Consider the following scenarios:

- **Custom backend:** You need to be in total control of your game backend. This could be for a variety of reasons: custom player authentication, player profile or advanced matchmaking facilities, increased ability for monitoring or maybe you just have the team and expertise needed and prefer to do everything in-house.
- **Server-side authoritative game logic:** Your game needs to be secure against player hacks. In player-hosted games, one of the clients acts as the server. This means that a malicious client has the potential to alter the outcome of the game. Whether this is important or not will depend on the specifics of your game, but sometimes you really need to be safe against this type of cheats. Running the game logic on an independent, authoritative game server that you control is the best way to ensure that.

Running your game on a dedicated server has therefore several advantages over player-hosted games:

- More protection against malicious clients.
- No artificial CCU limit.
- No need for NAT punchthrough or relay servers. Clients are never connected directly to each other but through the master server instead.
- You only pay for setting up and maintaining your servers (which can be your own machines or a third-party cloud provider like [Digital Ocean](#)).

The main drawback is that, of course, you are in charge of everything. You need to pay for the hosting of your servers (unless you physically own them) and make sure they are always available to your players. As they say:

"With great power comes great responsibility."

Master Server Kit has been specifically created to help you with this task. We want you to focus on making a great game instead!

Features

Master Server Kit provides the following features:

- Player registration and authentication (which can be bypassed if desired).
- Guest mode.
- Support for both world-based and room-based games.
- Matchmaking supporting public and private games. Independent game server instances are dynamically spawned and destroyed as needed.
- Zone servers that allow you to distribute the game server load across different machines (useful for load balancing or region-based matchmaking).
- Player properties system that can be used for implementing currencies and achievements with persistent storage in a database. Default implementations are provided for SQLite, MongoDB and LiteDB.
- Chat with support for public and private messages and an unlimited number of channels.
- Complete and extensively documented C# source code. Customers have access to the private repository of the project.
- Compatible with games using UNET, including games that use the Network Manager component.
- Includes demo that showcases the functionality of the kit.

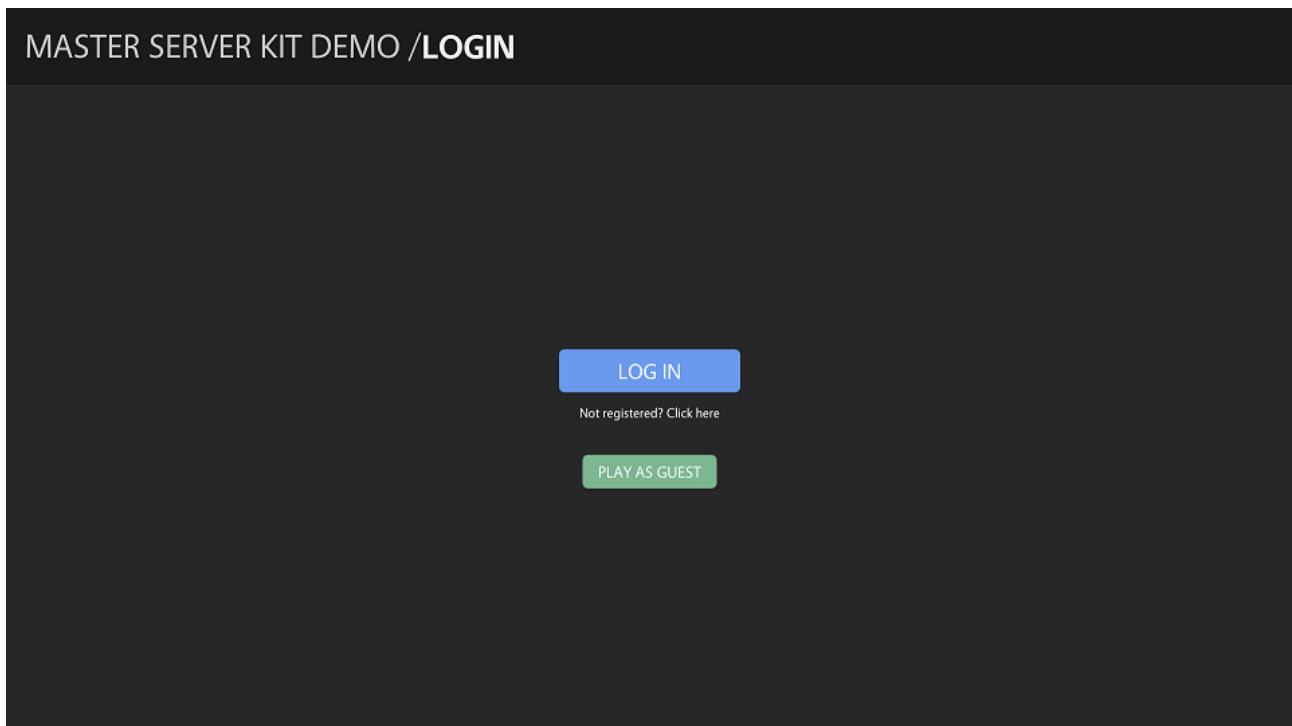
Setup

Demo

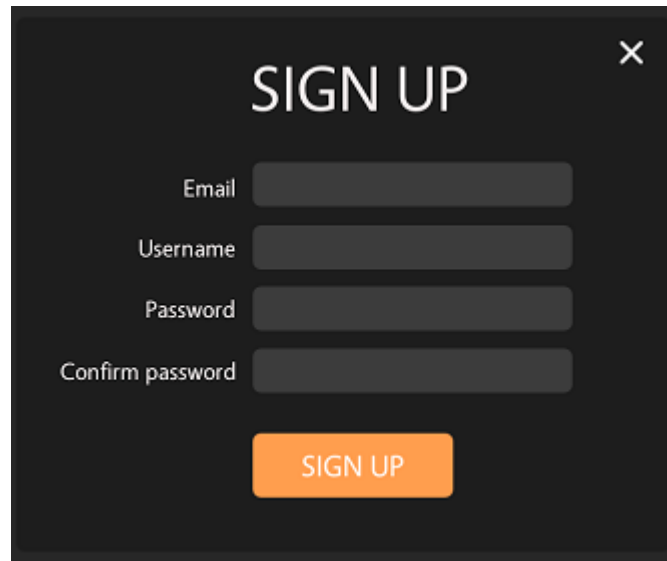
The best way to introduce the functionality provided by Master Server Kit is to play with the included demo. You can download it here:

- [Windows demo](#)
- [Mac OS demo](#)

You can launch it on your computer in order to be prompted with the intro screen:

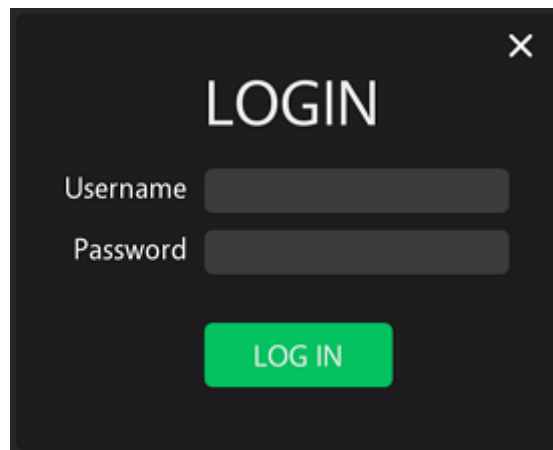


You can register a new user:



A dark-themed modal window titled "SIGN UP" with a close button (X) in the top right corner. It contains four input fields labeled "Email", "Username", "Password", and "Confirm password". Below the fields is an orange "SIGN UP" button.

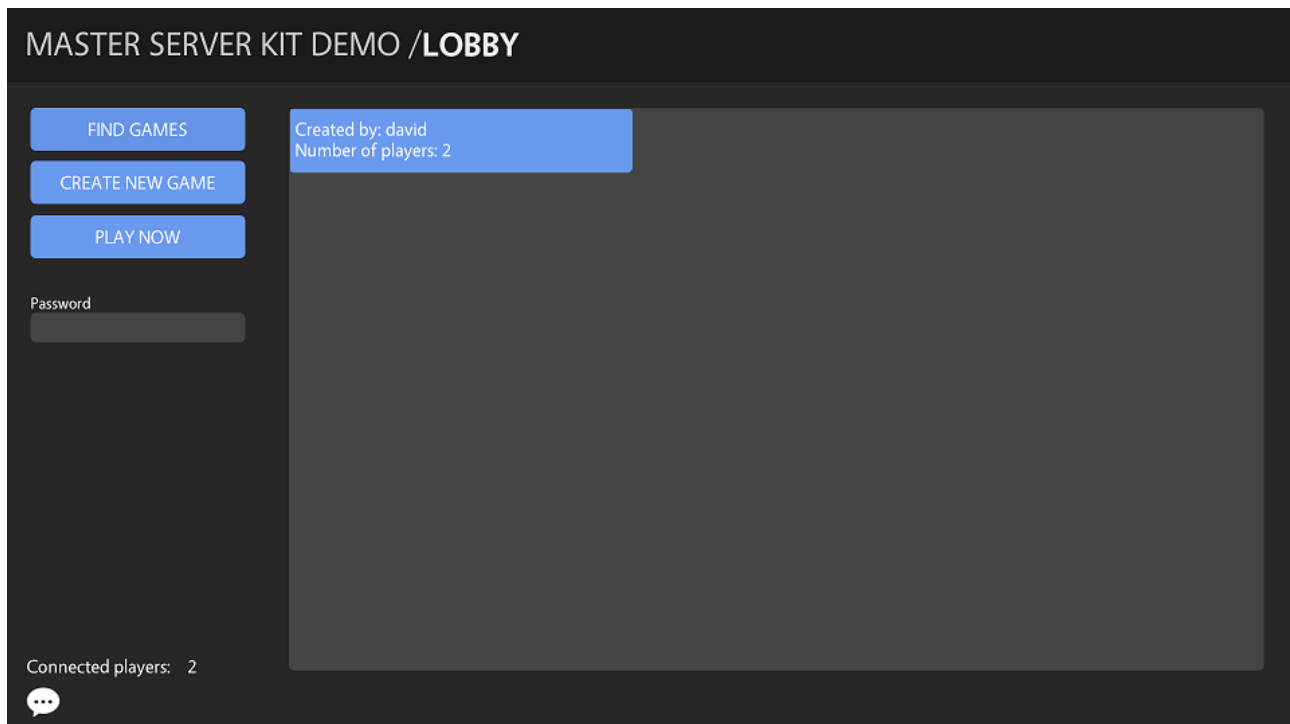
And log in with an existing user:



A dark-themed modal window titled "LOGIN" with a close button (X) in the top right corner. It contains two input fields labeled "Username" and "Password". Below the fields is a green "LOG IN" button.

You can also log in as a guest.

Once successfully logged into the master server, you will enter the lobby screen:

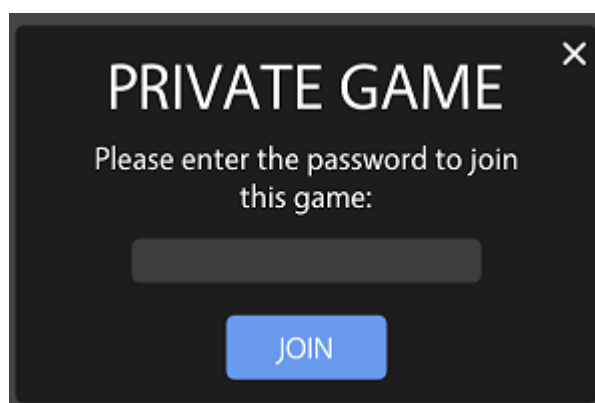


From this screen, you can perform the following actions by clicking on the appropriate buttons:

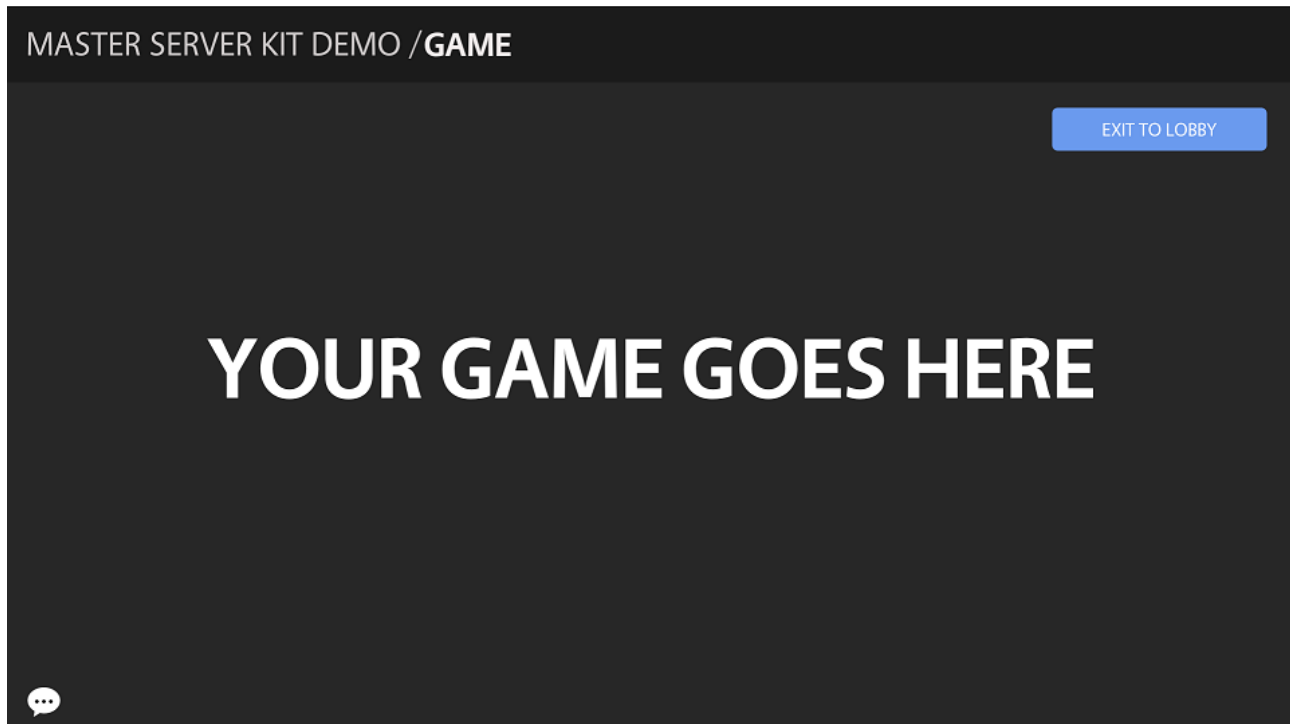
- **Find games:** Lists all the available games on the master server.
- **Create a new game:** Creates a new game (which will be public or private depending on the contents of the password input field). While the demo always creates 2-player games, the kit supports any number of players. We plan to extend the demo with the option to create games with different capacities in a future update.
- **Play now:** Joins an available game or creates a new one automatically if none was found.
- **Join game:** Joins the corresponding game.
- **Chat:** Chat with all the other players connected.

You can see this functionality is fundamentally equivalent to the one provided by [Unity Multiplayer Services](#)'s matchmaking, with the difference that everything is running on a dedicated server here.

If you try to join a private game you will be prompted with this dialog, where you need to introduce the game's password:



Once you have successfully joined a game, you will enter the game screen:



This screen is just a placeholder for demonstration purposes; you would implement your authoritative game logic here. You can chat with the other players in the game/match/room and also have the option to exit the game and return to the lobby. When no players are left in a game server, it is automatically shut down by the master server.

How to build the demo

The best way to understand how Master Server Kit is structured and how it works is to build the accompanying demo and run it locally on your machine. In order to do that, you will need to generate four binaries:

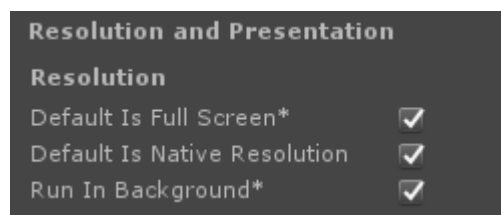
- The **master server**: This server is responsible for authenticating existing players, registering new players into the system and managing the available games. You can think of it as the "brain" in Master Server Kit, acting like a centralized lobby.
- The **zone server**: This server is responsible for managing the game server instances in a given zone. Zones are a useful abstraction that allow you to distribute the game server load across different machines (which is useful for load balancing or region-based matchmaking), meaning you can have several zone servers running a different subset of all the available game servers.
- The demo's **game server**: This server is responsible for managing the server-side game logic of a single game/match/room of your game.
- The demo's **game client**: This is the binary that your players will launch in order to play your game.

You should generally be able to use the provided master server and zone server mostly with no changes for your game, although you always have the option of extending or customizing them if

needed because the kit comes with its complete source code included. The game server and game client are where you come in: it is your responsibility to program the multiplayer logic specific to your game server and client. The demo provides an example game server and client that you can use as an official reference for your own game.

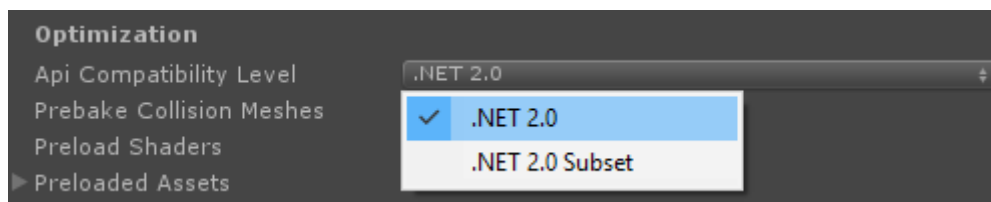
In order to build the demo so that it can run locally on your development machine, follow these steps:

- Import Master Server kit into your project. It is highly recommended you use the [latest stable version](#) of Unity.
- Make sure the **Run In Background** option in *Build Settings/Player Settings/Resolution and Presentation* is enabled. This is very important when testing everything locally on your development machine to make sure the servers keep running even when they do not have the system focus.



It is also a good idea to make sure the **Display Resolution Dialog** option in *Build Settings/Player Settings/Standalone Player Options* is disabled for convenience, but this is not strictly required.

- Change the **Api Compatibility Level** option in *Build Settings/Player Settings* from **.NET 2.0 Subset** to **.NET 2.0** (this is only needed by the default SQLite database implementation) as follows:



If you decide you want to use a different database implementation and remove the files related to SQLite from your project, you will be able to use the **.NET 2.0 Subset** option as is usual in most Unity projects.

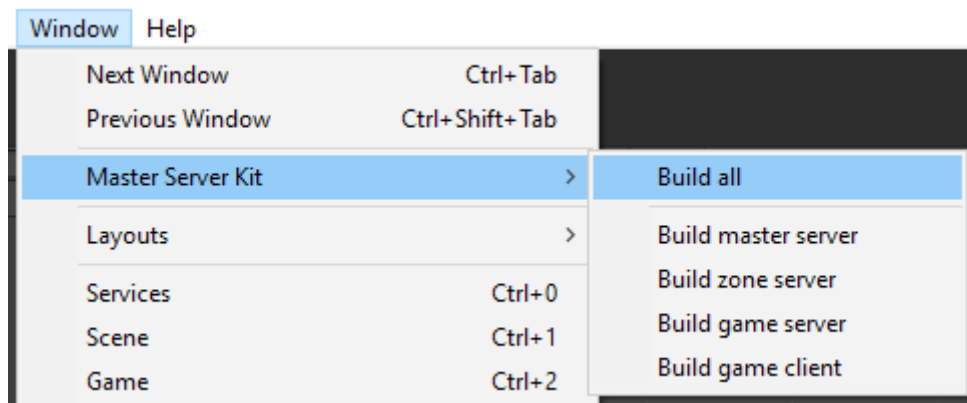
Note

If you want to completely remove SQLite from your project, you will need to delete the following files: the `SQLiteDatabaseProvider.cs` file and the DLLs needed by SQLite (`Mono.Data.Sqlite.dll`, `System.Data.dll`, `sqlite3.dll`, `sqlite3.def`, `libsqlite3.so`).

- Configure your server settings as appropriate in the appropriate components located in the `MasterServer`, `ZoneServer`, `GameServer` and `GameClient_Start` scenes. By default they point to localhost (127.0.0.1), which is just fine for testing things locally on your development machine. You will need to change the value of the *Path to binary* field in the Zone Server

component so that it contains the path on your system where the game server binary is located. In our computer, that would be "*GameServer.exe*".

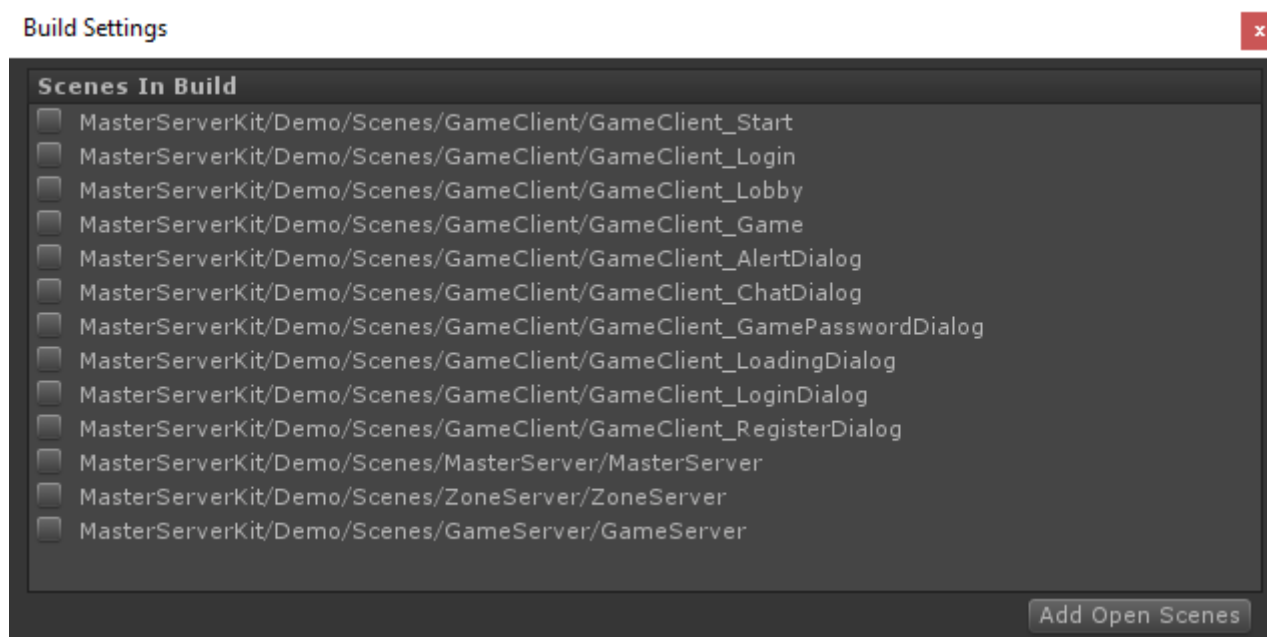
- Select the *Build All* option located in the *Window/Master Server Kit* menu. This will generate the four binaries automatically. By default, the binaries will be located at the root of your Unity project in a folder named Builds.



This menu option is provided via an editor script located in `Demo/Scripts/Editor/Builder.cs`.

- Launch the master server, the zone server and any number of game clients (in that order). You should now be able to test the demo. The game servers will be automatically spawned and destroyed by the zone server as players create and leave new games rooms in the lobby (this is precisely the reason behind the existence of the *Path to binary* field, so that the zone server knows which binary to launch when a new game room is created).

You can also generate the builds manually from the build settings by selecting the appropriate scenes for each binary:



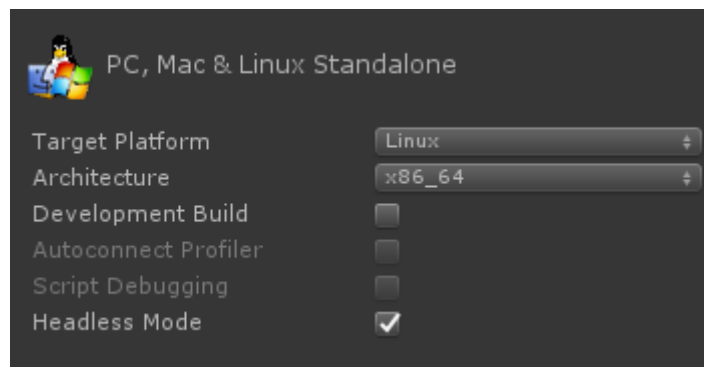
More specifically, you will need:

- The MasterServer scene for the master server.
- The ZoneServer scene for the zone server.
- The GameServer scene for the demo's game server.
- All the scenes located in the Demo/Scenes/GameClient folder for the demo's game client.

By default, the players' data is stored in a SQLite database, but the kit also provides implementations for MongoDB and LiteDB. You do not generally need to worry about the details of interacting with the database, as this is something handled by the kit.

Deploying to a production server

When deploying the server binaries to a production environment, we recommend building the authentication server, master server and game server as Linux headless. This allows you to launch them as command line programs with no graphics, which is particularly convenient for servers (where no graphics are required).



Master Server Kit is known to work on a [Digital Ocean](#) VPS, but any well-respected cloud service with support for Linux servers able to run Unity instances should be fine. Please note that we do not provide technical support for deployment issues.

In order to get the servers deployed to a Digital Ocean VPS, you can follow these steps:

- Set up your Digital Ocean server. You can follow the official tutorial [here](#). Not all the steps are strictly necessary; for example, you can do everything as the root user at the beginning, but at some point it is recommended to create an administration user with more limited privileges for security reasons.
- Update the firewall rules on your server to allow remote connections to the ports used by the kit. Ours look something like this (with the default configuration):

To	Action	From
--	-----	----
OpenSSH	ALLOW	Anywhere
8000	ALLOW	Anywhere
9000:9100/tcp	ALLOW	Anywhere
9000:9100/udp	ALLOW	Anywhere
OpenSSH (v6)	ALLOW	Anywhere (v6)
8000 (v6)	ALLOW	Anywhere (v6)
9000:9100/tcp (v6)	ALLOW	Anywhere (v6)
9000:9100/udp (v6)	ALLOW	Anywhere (v6)

- Configure your server settings in the appropriate components located in the MasterServer, ZoneServer, GameServer and GameClient_Start scenes. Change the IP address fields to point to the public IP address of your server. You will also need to change the value of the *Path to binary* field in the Zone Server component so that it contains the path on your server where the game server binary is located. In our server, that would be "*game_server.x86_64*".
- Build the server binaries (master server, zone server and game server) for Linux 64-bit in headless mode.
- Zip the server binaries and upload them to your Digital Ocean server. For example, by using the scp command:

```
scp DigitalOceanBuilds.zip root@DIGITAL_OCEAN_IP:~/DigitalOceanBuilds.zip
```

- Log into your Digital Ocean server and go to the location where you uploaded the .zip file containing the server binaries. Unzip it (you may need to install the unzip utility on your server):

```
unzip DigitalOceanBuilds.zip
```

- Give execution permission to the extracted binary files:

```
chmod +x master_server.x86_64
chmod +x zone_server.x86_64
chmod +x game_server.x86_64
```

- Finally, run the master server and zone server in the background (so they keep running 24/7).

```
./master_server.x86_64 &
./zone_server.x86_64 &
```

The game server instances will be spawned and destroyed automatically by the zone server as appropriate.

With non-root users, you may need to use the sudo command as appropriate at some of the steps mentioned above.

Client API

The ClientAPI class is the client-side entry point to all of the functionality available in Master Server Kit. It provides the following methods:

FindGameRooms

```
void FindGameRooms(  
    List<Property> includeProperties,  
    List<Property> excludeProperties,  
    Action<SpawnedGameNetwork[]> callback)
```

Description

Finds all the registered game rooms. You can do filtering on the game rooms that have the properties you are interested in.

Parameters

- **includeProperties:** The properties that the game rooms should have.
- **excludeProperties:** The properties that the game rooms should not have.
- **callback:** The callback to execute when the rooms have been found. This callback will receive the list of returned game rooms as a parameter.

CreateGameRoom

```
void CreateGameRoom(  
    string name,  
    int maxPlayers,  
    string password,  
    Action<string, int> successCallback,  
    Action<CreateGameRoomError> errorCallback)
```

Description

Creates a new game room with the specified properties.

Parameters

- **name:** The name of this room.
- **maxPlayers:** The maximum number of players allowed in this room.
- **password:** The password of this room (can be null for public rooms).
- **successCallback:** The callback to execute when the room has been successfully created. This callback will receive both the IP address and the port number of the spawned game server instance as parameters.
- **errorCallback:** The callback to execute when the room has not been successfully created. This callback will receive the error as a parameter.

JoinGameRoom

```
void JoinGameRoom(  
    int roomId,  
    string password,  
    Action<string, int> successCallback,  
    Action<JoinGameRoomError> errorCallback)
```

Description

Joins an existing game room with the specified properties.

Parameters

- **roomId:** The unique identifier of the room to join.
- **password:** The password of the room to join (can be null for public rooms).
- **successCallback:** The callback to execute when the room has been successfully joined. This callback will receive both the IP address and the port number of the spawned game server instance as parameters.
- **errorCallback:** The callback to execute when the room has not been successfully joined. This callback will receive the error as a parameter.

PlayNow

```
void PlayNow(  
    Action<string, int> joinCallback,  
    Action<string, int> createSuccessCallback,  
    Action<CreateGameRoomError> createErrorCallback)
```

Description

Joins an existing available game room or creates a new one if none is available.

Parameters

- **joinCallback:** The callback to execute when an available room has been successfully joined. This callback will receive both the IP address and the port number of the spawned game server instance as parameters.
- **createSuccessCallback:** The callback to execute when a room has been successfully created. This callback will receive both the IP address and the port number of the spawned game server instance as parameters.
- **createErrorCallback:** The callback to execute when a room has not been successfully created. This callback will receive the error as a parameter.

JoinGameServer

```
void JoinGameServer(  
    string ip,  
    int port,  
    Action onConnected)
```

Description

Joins the game server with the specified IP address and port number. Use this method if you are not using the Network Manager component available in UNET.

Use the callback to transition between your lobby scene and your game scene as appropriate in the context of your game.

Parameters

- **ip:** The IP address of the game server to join.
- **port:** The port number of the game server to join.
- **onConnected:** The callback to execute when the game server has been joined.

JoinGameServer (when using the Network Manager component)

```
void JoinGameServer(  
    string ip,  
    int port)
```

Description

Joins the game server with the specified IP address and port number via the specified [Network Manager](#) object. Use this method if you are using UNET's high-level API for your game's networking code.

The transition between your lobby scene and your game scene will be performed automatically by the Network Manager object.

Parameters

- **ip:** The IP address of the game server to join.
- **port:** The port number of the game server to join.

Database

Introduction

Master Server Kit includes an authentication system that can be used to securely log players into the master server. This system can be bypassed if you are not interested in the authentication functionality (the *authenticationRequired* flag in the Configuration class controls this).

The authentication system uses a database for storing the players' data. The kit provides three default database implementations:

- [SQLite](#)
- [MongoDB](#)
- [LiteDB](#)

By default, the SQLite implementation is used. You can easily switch the database implementation used by the DatabaseService class:

```
public static class DatabaseService
{
    /// <summary>
    /// Default database provider.
    /// </summary>
    private static IDatabaseProvider database = new SQLiteProvider();

    // ...
}
```

Replace the database line with

```
private static IDatabaseProvider database = new MongoDBProvider();
```

in order to use the MongoDB database.

Replace the database line with

```
private static IDatabaseProvider database = new LiteDBProvider();
```

in order to use the LiteDB database.

If you are interested in using a different database, it is easy to do so by creating your own provider. The provider needs to implement the IDatabaseProvider interface as appropriate:

```
/// <summary>
/// Custom database provider implementation.
/// </summary>
public class CustomDBProvider : IDatabaseProvider
{
    /// <summary>
    /// Performs any initialization-related logic.
    /// </summary>
    public void InitializeDatabase()
    {
        // ...
    }
}
```



```

    /// <summary>
    /// Registers a new user with the specified properties in the system.
    /// </summary>
    /// <param name="email">The new user's email address.</param>
    /// <param name="username">The new user's name.</param>
    /// <param name="password">The new user's password.</param>
    /// <param name="onSuccess">Delegate to be executed when the request is
successful.</param>
    /// <param name="onError">Delegate to be executed when the request is not
successful.</param>
    /// <returns>Async operation for the request.</returns>
    public IEnumerator Register(string email, string username, string password,
Action<string> onSuccess, Action<RegistrationError> onError)
    {
        // ...
    }

    /// <summary>
    /// Logs the specified user in the system.
    /// </summary>
    /// <param name="username">The user's name.</param>
    /// <param name="password">The user's password.</param>
    /// <param name="onSuccess">Delegate to be executed when the request is
successful.</param>
    /// <param name="onError">Delegate to be executed when the request is not
successful.</param>
    /// <returns>Async operation for the remote request.</returns>
    public IEnumerator Login(string username, string password, Action<string>
onSuccess, Action<LoginError> onError)
    {
        // ...
    }

    /// <summary>
    /// Gets the specified integer property from the specified user.
    /// </summary>
    /// <param name="username">The user's name.</param>
    /// <param name="key">The property's key.</param>
    /// <param name="onSuccess">Delegate to be executed when the request is
successful.</param>
    /// <param name="onError">Delegate to be executed when the request is not
successful.</param>
    /// <returns>Async operation for the remote request.</returns>
    public IEnumerator GetIntProperty(string username, string key, Action<int>
onSuccess, Action<string> onError)
    {
        // ...
    }

    /// <summary>
    /// Sets the specified integer property from the specified user to the
specified value.
    /// </summary>
    /// <param name="username">The user's name.</param>
    /// <param name="key">The property's key.</param>
    /// <param name="value">The property's value.</param>
    /// <param name="onSuccess">Delegate to be executed when the request is
successful.</param>
    /// <param name="onError">Delegate to be executed when the request is not
successful.</param>
    /// <returns>Async operation for the remote request.</returns>
    public IEnumerator SetIntProperty(string username, string key, int value,
Action<int> onSuccess, Action<string> onError)

```

```

{
    // ...
}

/// <summary>
/// Gets the specified string property from the specified user.
/// </summary>
/// <param name="username">The user's name.</param>
/// <param name="key">The property's key.</param>
/// <param name="onSuccess">Delegate to be executed when the request is
successful.</param>
/// <param name="onError">Delegate to be executed when the request is not
successful.</param>
/// <returns>Async operation for the remote request.</returns>
public IEnumerator GetStringProperty(string username, string key,
Action<string> onSuccess, Action<string> onError)
{
    // ...
}

/// <summary>
/// Sets the specified string property from the specified user to the
specified value.
/// </summary>
/// <param name="username">The user's name.</param>
/// <param name="key">The property's key.</param>
/// <param name="value">The property's value.</param>
/// <param name="onSuccess">Delegate to be executed when the request is
successful.</param>
/// <param name="onError">Delegate to be executed when the request is not
successful.</param>
/// <returns>Async operation for the remote request.</returns>
public IEnumerator SetStringProperty(string username, string key, string
value, Action<string> onSuccess, Action<string> onError)
{
    // ...
}
}

```

Note these functions are intended to be called asynchronously as coroutines. This is particularly convenient when interacting with a database, as there will usually be a degree of latency when accessing it (or maybe the database is not accessed directly but through a web service instead; the provided MongoDB implementation does precisely this).

Node.js + MongoDB

The following section applies only to the MongoDB database implementation. This implementation does not access the database directly, but via a server that implements a REST-based API. This server is written in [Node.js/Express](#) and uses a [MongoDB](#) database, so a fundamental knowledge of these technologies is recommended.

Initial setup

In order to set up the server on your machine, the first thing you need to do is to install the required dependencies:

- [Node.js](#)

- [MongoDB](#)

The latest stable releases work just fine.

The next step is to unzip the contents of the PersistentDataServer/PersistentDataServer.zip file to a desired location on your machine (be careful not to extract it inside your Unity project, as that will cause build errors). The server files have the following structure:

- **app/ folder:** Contains the complete source code of the server.
- **package.json file:** Contains a list of all the dependencies for the server.
- **server.js file:** Main source code file for the server.

You will need to run this command from a terminal

```
npm install
```

the first time in order to install the required dependencies (note that a **node_modules** folder will be created).

At this point, you should be able to run the server locally on your machine. The **app/config/config.js** is particularly interesting because it allows you to tweak the server configuration (the database name and secret, the number of cards contained in a purchasable card pack, etc.).

In order to run the server, you just need to run the following terminal command

```
node server.js
```

from the folder where you extracted the server files. Note that you also need to have a running instance of MongoDB (in a second terminal); in order to do that you need to run the following terminal command

```
mongod
```

You should now be able to use the server functionality from the demo game. Feel free to expand the provided functionality as you need for your game!

Important

If you are on a Windows machine, you may need to add the MongoDB folder (e.g., C:\Program Files\MongoDB\Server\3.2\bin) to [your system PATH](#) in order to run the **mongod** command from the terminal. Or, alternatively, just navigate to that location manually from the terminal and execute the command directly from there.

The following programs are incredibly useful for server development and testing:

- [Postman](#) to test the REST API.
- [Robomongo](#) to manage your MongoDB database.

Deploying the server in production

It is very convenient to have the server running locally on your machine during development, but sooner or later you are going to need to deploy it in a production environment.

You can choose between hosting your own server or use an existing, third-party hosting service. Well-known options for deploying the server in a production environment are [AWS](#), [Digital Ocean](#), [Heroku](#) and [OpenShift](#). Choosing one over the other is a balance between ease of setup, experience of your team and pricing/scalability concerns that need to be carefully considered on every particular case. This means it is both a technical and a business decision that does not have a single, universally valid answer.

REST API

The server provides the following REST-based API:

POST /api/register

Creates a new player.

Parameters:

- email (string): The player's email. Must be unique.
- username (string): The player's name. Must be unique.
- password (string): The player's password.

Example successful response:

```
{
  "status": "success",
  "token":
  "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfYWQiOiIiN2YyOTRmZjU3Mjk3ZTdjaYzYxN2E2OGUiLCJlbWVpbCI6ImRwLmFob251bjZAZ21haWwY29tIiwidXNlcm5hbWUiOiJEYXZpZDYiLCJleHAiOjE0NzYxMjA0NDcsImhhdCI6MTQ3NTUxNTY0N30.11_RzALcCLz0WSj-5ltoLHbtwxzHI4De4uRGbS81pxw"
}
```

POST /api/login

Logs an existing player into the server.

Parameters:

- username (string): The player's name.
- password (string): The player's password.

Example successful response:

```
{
  "status": "success",
  "token":
  "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfYWQiOiIiN2UyYzk5MTU3Mjk3ZTdjaYzYxN2E2ODgiLCJlbWVpbCI6ImRwLmFob251bkBnbWVpbC5jb20iLCJ1c2VybmFtZSI6IkpXZWhwIiwidXNlcm5hbWUiOiJEYXZpZDYiLCJleHAiOjE0NzYxMjA0NDcsImhhdCI6MTQ3NTUxNTY0N30.11_RzALcCLz0WSj-5ltoLHbtwxzHI4De4uRGbS81pxw",
  "username": "David"
}
```

All the API endpoints always return a "status" string indicating if the request was successful ("success") or not ("error"). In the case of an error, you can obtain more details by examining the "message" string.

The API provides security by leveraging [JSON Web Tokens](#). When a player successfully logs into the server, he receives a token that uniquely identifies him on the system and that can be used to securely authenticate later calls to the API (if you decide to extend and/or customize it).

The code for accessing the REST API lives in the MongoDBProvider class, which you can find at the Core/Scripts/Core/DatabaseProviders folder.

F.A.Q.

What platforms are supported by Master Server Kit?

Master Server Kit works on all the platforms supported by UNET. For the servers, you will need a machine running Windows, Mac OS or Linux (Linux is recommended because you can build the binaries in headless mode, which is particularly convenient for servers). For the client, you can target both desktop and mobile platforms. UNET on WebGL currently has some bugs, so we do not officially support this platform at this moment. As the situation improves, we will update this section.

Do I need a knowledge of programming and network programming?

Yes. Master Server Kit is a C# library built on top of UNET, so familiarity with both is highly recommended. If you are not familiar with network programming, a good starting point are the [official tutorials](#) from Unity.

Do I need a knowledge of server administration?

Yes. You are responsible for deploying your game to your servers (your own or hosted by a third-party), so a basic familiarity with server administration will definitely be helpful. The included demo is hosted on a \$5/month [Digital Ocean](#) VPS, which works just fine for development and testing purposes. Please note that we do not provide technical support for deployment issues.

If I purchase the kit, will you also provide the hosting for my game?

No. You are responsible for setting up and paying for the hosting of your servers. Master Server Kit only provides the foundational backend code to help build your infrastructure. Please note that we do not provide technical support for deployment issues.

Will Master Server Kit help me with programming the logic of my multiplayer game?

No. Master Server Kit is specifically focused on the backend side of things. Helping with the game logic itself (and related aspects such as client-side prediction or entity interpolation) is therefore completely outside the scope of the kit.

How many players does Master Server Kit support?

This is a question that cannot possibly be answered in a general, definitive way. There are no hardcoded limits on the kit itself, but the actual number will depend on several game-specific considerations such as the network bandwidth usage of your game, the capabilities of your server and any limitations inherent to UNET.

Version history

Version 1.10:

- Added MySQL database provider implementation.

Version 1.09:

- Added a new version of the ClientAPI's CreateGameRoom method that accepts a list of properties.
- Added hideWhenFull field to the game server component that allows game servers to be removed from the matchmaking results once they are full.

Version 1.08:

- Upgraded project to Unity 5.5.2.
- Implemented a "no database" provider as an in-memory alternative for player data storage.
- Added a default path to binary to the zone server demo scene.
- Added an explicit maximum number of connections to the master server.
- Improved the error handling when spawning a new game server instance.
- Removed redundant networkManager property from the game server.

Version 1.07:

- Fixed bug that prevented new game server instances from being spawned after a while due to an accumulation of internal helper connections not being properly cleaned up.

Version 1.06:

- Upgraded project to Unity 5.5.1.
- Removed the authentication server and integrated its functionality into the master server.
- Added the concept of zone servers, which allow you to distribute the game server load across different machines (useful for load balancing or region-based matchmaking).
- Removed the configuration file. All the server settings can be modified visually via Unity's inspector now.

Version 1.05:

- Several improvements in the matchmaking API to simplify its usage.
- Replaced *authenticationEnabled* flag with *authenticationRequired* in the configuration.
- Added *maxPlayers* property to the configuration.
- The database does not store the “is logged in” status of a player anymore; the master server does now.

Version 1.04

- Upgraded project to Unity 5.5.0.
- Improved the default support for games using UNET's high-level [Network Manager](#).
- The master server now resets the 'logged in' flag of all the registered users upon launch.
- Trying to join an expired game/match/room results in a proper error now.
- Several improvements and refinements in the API and the organization of the kit.

Version 1.03

- Implemented a player properties API.
- Updated the demo's UI.
- Fixed bug that caused spawned game server instances to not be properly destroyed sometimes.

Version 1.02

- Generalized database support for player data storage and added three default implementations: SQLite, MongoDB and LiteDB.
- Added configuration flag to enable/disable authentication.
- Players cannot be logged in the master server more than once at the same time with the same credentials now.
- Improved the behavior of UI dialogs in the accompanying demo.

Version 1.01

- Implemented in-lobby and in-game chat with support for public and private messages and an unlimited number of channels.

Version 1.0

- First release.

Support and feedback

If you have any questions or suggestions, please do not hesitate to let us know! We are happy to help you and we want Master Server Kit to be the best kit for developing master servers in Unity. You can reach us at support@spelltwinegames.com. Please make sure to include your invoice number for technical support requests and please note that we do not provide support for deployment issues.

Access to the private repository of the project

If you are interested in having access to the private repository of the project, please send us your invoice number and [GitLab](#) username to the support email and we will be happy to set you up.