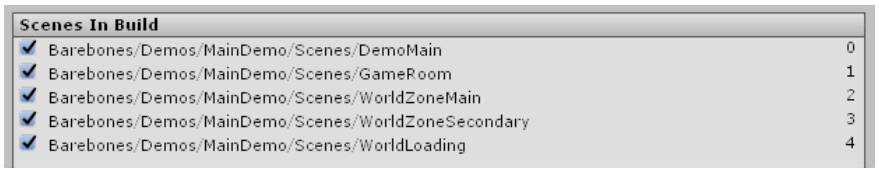
Warning

This document is only a small snapshot of full documentation, which is located in the Github page:
https://github.com/alvyxaz/barebones-masterserver/wiki

Launching The Demo

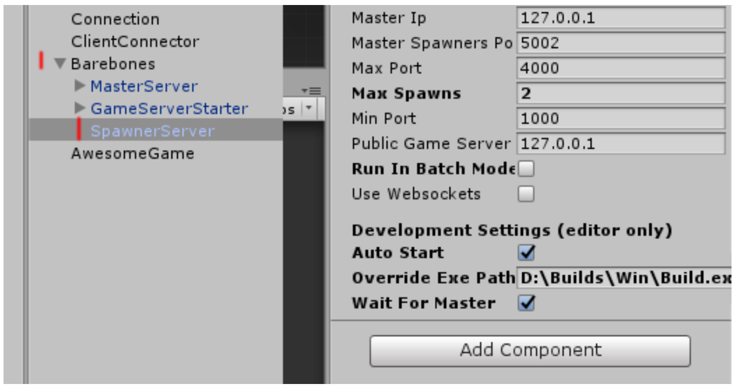
Navigate to folder `Barebones/Demos/MainDemo/Scenes` and add all of those scenes to your build. Make sure `DemoMain` is the first one



Scenes In Build	
✓ Barebones/Demos/MainDemo/Scenes/DemoMain	0
✓ Barebones/Demos/MainDemo/Scenes/GameRoom	1
✓ Barebones/Demos/MainDemo/Scenes/WorldZoneMain	2
✓ Barebones/Demos/MainDemo/Scenes/WorldZoneSecondary	3
✓ Barebones/Demos/MainDemo/Scenes/WorldLoading	4

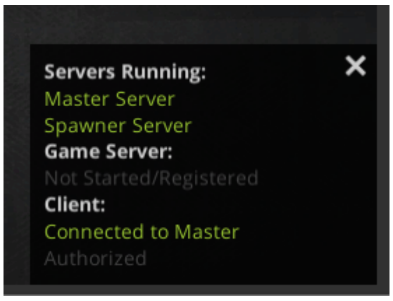
Open the `DemoMain` scene, in the hierarchy, locate `Barebones > SpawnerServer`, and change the **Override Exe Path** property in the inspector to your build path (for example `D:/Build/Win/Build.exe`). *This path will be used to start a game server process when you're in the editor.*

Now make a standalone build to the same path.



Master Ip	127.0.0.1
Master Spawners Po	5002
Max Port	4000
Max Spawns	2
Min Port	1000
Public Game Server	127.0.0.1
Run In Batch Mode	☐
Use Websockets	☐
Development Settings (editor only)	
Auto Start	✓
Override Exe Path	D:\Builds\Win\Build.ex
Wait For Master	✓

Now, with `DemoMain` scene in the editor, you should be able to hit a **Play** button in the editor, and the example should work. A HUD at the bottom right should show an indication that Master and Spawner servers were started, and the client has connected to the master:

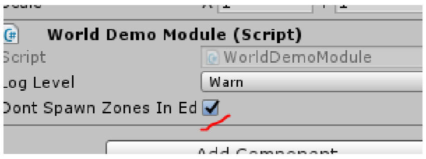


At this point, you should be able to log into the game, and create a new room. You should also be able to start a client from the build you just made, and connect to the new room.

Starting a World Demo

If you've tried to open the world demo, you might have received an error indicating that a specific zone cannot be found. This is because you need to start zone servers. There are two ways to do it.

If you're starting a master server from the editor, there's a flag on the **WorldGame** object, called `DontSpawnZonesInEditor`. If you uncheck it, world demo will start game zones together with the master server, on editor.


World Demo Module (Script)

Script: WorldDemoModule

Log Level: Warn

Dont Spawn Zones In Ed: ✓

Another approach could be to start the master server from the build, and add a `-spawnZones` flag to the command line arguments. You would start your server with a command similar to this:

```
./Build.exe -bmMaster -bmSpawner -spawnZones
```

⚠️ If you run master server from a build and connect to it with a client in the editor, you might encounter a problem, because client in the editor will try to start another master server. To avoid this, you can unset a flag **AutoStartInEditor** on the **Master** object (in hierarchy `Barebones > MasterServer`)

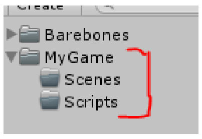
Getting Started

This page will guide you through the steps you can take to start working on your game and back-end server.

Preparations

1. Create a new folder in your project asset's root, and name it **MyGame**
2. Create two more folders within MyGame, called **Scripts** and **Scenes**

⚠️ If there's already a folder named **MyGame** in the `Barebones/Demos` folder, you can delete it. It contains the outcome of this tutorial.

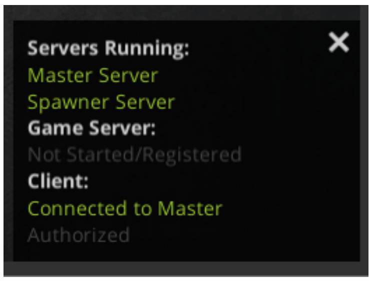


Setting Up Main Screen

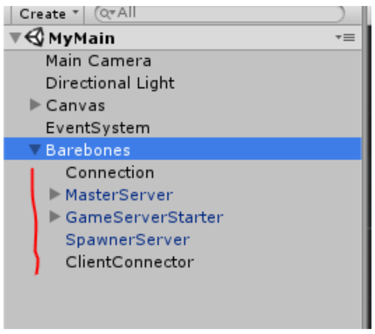
To make things faster, a pre-made scene is provided within the asset, and it's located at `Barebones/MasterFramework/Scenes -> MainEmpty`. When you work on your game, you're not forced to use this scene, but it's very convenient to make a copy of it and customize it the way you want.

Open the scene, hit `File > Save Scene As`, name it **MyMain** and save it to `MyGame/Scenes`

If you hit "Play" in the editor, you should see something like this at the bottom right:


```
Servers Running:
Master Server
Spawner Server
Game Server:
Not Started/Registered
Client:
Connected to Master
Authorized
```

It means that Master and Spawner servers are started, and client (you) is connected to master server. You might be wondering "Why and how these servers are started?" Take a look at the hierarchy:



Here's what these objects are there for:

- Barebones - object, used to conveniently separate framework related stuff from the level
- Connection - this object contains a component which connects client to Master Server
- MasterServer - this is a component, which starts the master server. It has an **Auto Start In Editor** option, which is why it starts automatically when we hit a "play" button in the editor.
- GameServerStarter - contains a component, which starts a game server. It also contains an **Auto Start** option, but it should be **Not Selected** in the main screen. Technically, in the editor, when in main screen, it does pretty much nothing. Only when you start server through command line, it picks up necessary arguments provided, and starts a game server.
- SpawnerServer - this script starts a server, which is responsible for spawning new game processes
- ClientConnector - takes care of connecting a player to game server. Depending on your game-joining flow, you might need to write your own connector

If you tried to log in , you probably got an error saying that there's no profile factory. We'll fix this in the next step

Profiles Factory / Shared Class

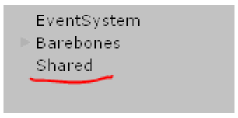
When your clients log in to master server, profiles module will try to construct a profile for each of them. Which is more, clients should also be able to construct their own profile, so that they can subscribe to various observable properties.

This means that client and server should share the same logic for creating profiles. Create a new script called **MyGameShared** in the Scripts folder. and paste the code below:

```
using UnityEngine;
using Barebones.MasterServer;

public class MyGameShared : MonoBehaviour {

    void Awake () {
        // Let the module know which factory to use
        ProfilesModule.SetFactory(ProfileFactory);
    }

    public static ObservableProfile ProfileFactory(string username)
    {
        // Create a profile and add coins property to it
        var profile = new ObservableProfile(username);
        profile.AddProperty(new ObservableInt(MyProfileKeys.Coins, 10));
        return profile;
    }
}

public class MyProfileKeys
{
    public const int Coins = 0;
}
```

This component simply sets a profiles factory (which is a simple method) when it's awakened. Our profile factory simply adds one property, called coins, and sets its default value to 10. Check out player profiles section of the documentation for more info on the matter.

This script should be added to your main screen and, later, to every other screen of your game (mainly to make debugging easier). In this example, it's added to component named **Shared**



Game Scene Setup

To guide you through game scene setup, we'll be using a small game, made with uNET, called **MiniGame**. You can find it at `Barebones/Demos/MiniGame`.

MiniGame represents a regular uNET game, where you host server and connect to it via NetworkManagerHUD.

If you're not familiar with uNET, I recommend you go through a tutorial in the official unity's manual

Overview

General steps to setting up a game scene are as follows:

1. Add game server object (**IGameServer**) - it should handle receiving passes from clients, disconnecting players that are not allowed and etc.
2. Add server **starter script** - this should be a script that reads command line arguments and starts the game server accordingly (example: `GameServerStarter.cs`)
3. Add **connector script** - a script that connects client to game server, when client opens the scene. (example: `UnetConnector.cs`)
4. Add **MasterServer** prefab - this is not necessary, but recommended for faster development. It will start master server automatically, so that you don't need to.

Copy the Game Scene

Open the **MiniGame** scene, hit `File > Save Scene As`, and Save to to your scenes as you did with the main scene. In this example, we'll call it **MyGame**

Implementing IGameServer interface

Our game server is only valid when it implements `IGameServer` interface. If you're using uNET HLAPI, general implementation is already written for you in script **`UnetGameServer.cs`** , this is what we'll be using to set things up faster.

1. Create a new script in `MyGame/Scripts` , and call it **MyGameServer.cs**

2. Make sure your newly created class extends `UnetGameServer` . To do that, you'll need to implement two methods: `OnServerUserJoined` and `OnServerUserLeft` . These methods are called when authenticated user joins and leaves a game.

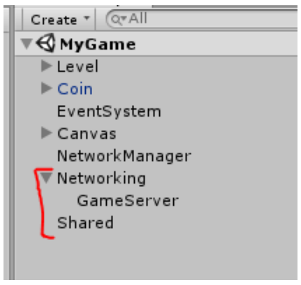
3. Call `MiniNetworkManager.SpawnPlayer(client.Connection, client.Username, "knife")` in the `OnServerUserJoined` . As you might have guessed, this will spawn player's character when player successfully joins the game. OnServerUserJoined is called when the player has officially joined the game.

   Here's what your script should look like:

```
using Barebones.MasterServer;

public class MyGameServer : UnetGameServer {

/// <summary>
/// Called, when authenticated user sends a valid access token
/// </summary>
/// <param name="client"></param>
protected override void OnServerUserJoined(UnetClient client)
{
    MiniNetworkManager.SpawnPlayer(client.Connection, client.Username, "knife");
}

protected override void OnServerUserLeft(UnetClient client)
{
}
}
```

4. Create a new object in the root of the hierarchy and name it **Networking**
5. Create a new empty game object as a child of **Networking** **object and call it* ***GameServer**
6. Attach a newly created component to it.

7. Add the **MyGameShared** component (the one you created earlier) to the scene as you did in the main scene.

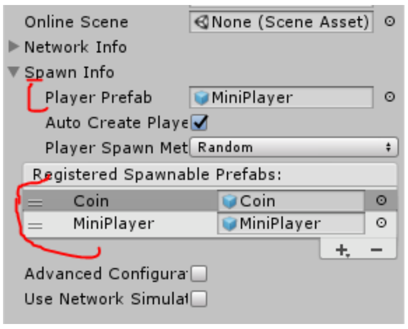


At this point, if you try to run the game, it will behave as it did earlier, except for a thrown error, which says that there's no EventfulNetworkManager in the scene, that's expected, and will be fixed in the next step

Enhancing The Network Manager

We now need to replace the standard NetworkManager with **EventfulNetworkManager**. It is pretty much the same as the regular NetworkManager, but instead of forcing you to override methods, it triggers events, to which other components can subscribe. MyGameServer component we've created earlier is one of those subscribers.

1. Select `NetworkManager` object in the hierarchy, click **Add Component** button, and add `EventfulNetworkManager`. Now there will be two network managers.
2. Remove the `MiniNetworkManager` component by right clicking and selecting `Remove component`
3. Add the MiniPlayer and Coin prefabs to the list of spawnables (you can find prefabs at `Barebones/MiniGame/Resources/Prefabs`):


Online Scene: None (Scene Asset)
Network Info
Spawn Info
Player Prefab: MiniPlayer
Auto Create Playe: ✓
Player Spawn Met: Random
Registered Spawnable Prefabs:
Coin: Coin
MiniPlayer: MiniPlayer
Advanced Configura: ☐
Use Network Simulat: ☐

⚠️ If you try to launch the game now, your ***player character will not be spawned***. This is because in order to allow players to join, game server must first be registered to Master, and then opened to public. In the next steps, we'll use some of the "helper components" to automate this process for us.

Joining The Game

General steps for client to join a game are as follow:

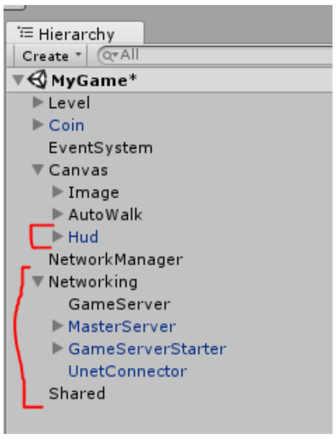
1. Connect to Master
2. Log in
3. Retrieve access key
4. Send access key to game server

When you are developing your game, you probably want to test every iteration of changes as fast as possible. Normally you'd have to start the server, build a client, connect and login with the client and etc, just to get into the game.

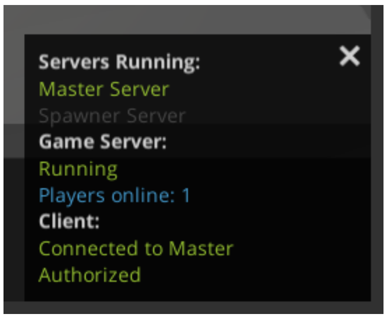
If you're using **uNET* *for your games networking, there are a few scripts and prefabs that can automate this process for you, so navigate to `Barebones/MasterFramework/Prefabs`

1. Add **MasterServer** to scene. This script will automatically start master server when you hit "play" button in the editor
2. Add **GameServerStarter** prefab to scene- it will automatically start your game server (after master server starts)
3. Add **UnetConnector** prefab to scene - it has a flag, called **Auto Join If Host**, which, as the name suggests, will try to automatically join the game, by first connecting to master and logging in.
4. Add **Hud** prefab to the **Canvas** object in the hierarchy. It will help you see if the servers were started successfully.

After adding these prefabs, here's what your scene hierarchy should look like:



✅ At this point, if you start the game in editor, Master server should start first, then game server, and after that - a user should join the game. This should be confirmed in your hud:



Servers Running:
Master Server
Spawner Server
Game Server:
Running
Players online: 1
Client:
Connected to Master
Authorized

ℹ️ It's highly recommended that you take a look at how `GameServerStarter` and `UnetConnector` scripts work, especially if you're not using uNET, or if you're aiming for a custom flow of events. These scripts will give you a general idea of how you can setup your own development process

Persistent Coins

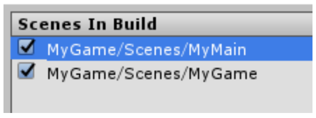
Even though we have added a coins property to the our custom profile, we never actually used it. To do that, we have to first retrieve the profile, read the value we want, and update it when user picks up a coin.

To do that, open the `MyGameServer` script, and modify `OnServerUserJoined` method like this:

```
/// <summary>
    /// Called, when authenticated user sends a valid access token
    /// </summary>
    /// <param name="client"></param>
    protected override void OnServerUserJoined(UnetClient client)
    {
        // Retrieve a profile first
        ProfilesModule.GetProfile(client.Username, MasterConnection, profile =>
        {
            if (profile == null)
            {
                // We failed to retrieve players profile, let's disconnect him
                DisconnectPlayer(client.Username);
                return;
            }

            // Get coins property
            var coinsProperty = profile.GetProperty<ObservableInt>(MyProfileKeys.Coins);

            // Spawn player (the same method we used earlier)
            var player = MiniNetworkManager.SpawnPlayer(client.Connection, client.Username, "knife");
            player.Coins = coinsProperty.Value;

            // Subscribe to the coins change listener.
            // It will be invoked on server, when client picks up a coin.
            player.OnCoinsChanged += () =>
            {
                // Update clients profile by setting a different coins value
                coinsProperty.Set(player.Coins);
            };
        });
    }
```

Making Sure Everything Works

Build the game with two of your scenes, make sure that main scene is first.


Scenes In Build
☑ MyGame/Scenes/MyMain
☑ MyGame/Scenes/MyGame

When you're in your game scene, hit "Play" in the editor. As before, it should start master and game servers automatically. You should be able to start the client you just built, and connect to the game.

Creating Modules

This section will guide you through the process of creating a custom master server module.

Creating modules is a preferred way of extending Master Server functionality. Modular approach can help you reuse components in other games, and it makes testing and debugging easier.

In this tutorial you will learn to:

- Send messages from client to server
- Handle messages (packets) from client
- Store data in the clients session
- Add new module to Master Server

Defining Operation and Property Codes

When clients and servers exchange messages, they need to specify what kind of a message (message type) they send - this is done by setting the **OpCode** (operation code) when constructing the message.

You could specify random numbers from your head, but it's recommended to have all your opCodes in one location. For this example, we will store them in the class:

```
    public class MyOpCodes
    {
        public const int GetPersonalInfo = 0;
        public const int SavePersonalInfo = 1;
    }
```

⚠ Some of the opCodes are occupied by the framework, but they start from 32000, so it's unlikely you will ever collide with them

We will also need to define property codes, which will be used when setting properties of `IPeer`.

```
    public class MyPropCodes
    {
        public const int PersonalInfo = 0;
    }
```

Creating a Custom Packet

When sending information, it's not always enough to send a simple string. When you need to send more complex data, you can make a class which extends `SerializablePacket`. You'll have to manually write the data to binary writer and read it with binary reader.

```
    public class PersonalInfoPacket : SerializablePacket
    {
        public string Name;
        public int Age;

        public override void ToBinaryWriter(EndianBinaryWriter writer)
        {
            writer.Write(Name);
            writer.Write(Age);
        }

        public override void FromBinaryReader(EndianBinaryReader reader)
        {
            Name = reader.ReadString();
            Age = reader.ReadInt32();
        }
    }
```

ℹ️ Serializing data manually is not the only way to do this. Depending on the platforms you are going to support, you can use other libraries, such as protobuf-net, or json.net. To avoid AOT and reflection code, framework does not use any of them, and relies on manual serialization for maximum compatibility.

Sending the message

Code below demonstrates how you can send two types of messages:

- Reliable message without response
- Reliable message with expected response

Since these methods are static, you can add them anywhere you want, and invoke on client

```
    public static void SavePersonalInfo(int age, string name)
    {
        // Construct the packet
        var info = new PersonalInfoPacket()
        {
            Age = age,
            Name = name
        };

        // Create a message
        var msg = MessageHelper.Create(MyOpCodes.SavePersonalInfo, info.ToBytes());

        // Send a reliable message to master server
        Connections.ClientToMaster.Peer.SendMessage(msg, DeliveryMethod.Reliable);
    }

    public static void GetPersonalInfo()
    {
        // Create an empty request message
        var msg = MessageHelper.Create(MyOpCodes.GetPersonalInfo);

        // Send message to master server, and wait for the response
        Connections.ClientToMaster.Peer.SendMessage(msg, (status, response) =>
        {
            // Response received

            if (status != AckResponseStatus.Success)
            {
                // If request failed
                Debug.LogError(msg.ToString());
                return;
            }

            // Success
            var info = response.DeserializePacket(new PersonalInfoPacket());

            Debug.Log(string.Format("I've got info. Name: {0}, Age: {1}", info.Name, info.Age));
        });
    }
```

Usually, when you want to retrieve something from the server, you make a request and expect a response. This is demonstrated in the `GetPersonalInfo` method.

Sometimes, you might want to send a simple notification. Example of this can be found in `SavePersonalInfo` method. It sends a simple save request, and doesn't expect to get a response in return.

Creating a Module Which Handles Requests

Your modules should extend `MasterModule` class. It will force you to implement method `Initialize`. This method will be called when master server starts, and all of the modules dependencies can be resolved.

In the example below, our module has one dependency, which is registered by calling `AddDependency<AuthModule>();`. AuthModule is not really used in the example, it's just there for demonstration purposes.

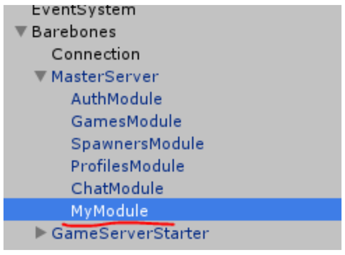
Adding dependencies with `AddDependency` will make sure that `Initialize` method is not called until master server initializes dependencies first.

```
using Barebones.MasterServer;
using Barebones.Networking;
using UnityEngine;

/// <summary>
/// Our custom module
/// </summary>
public class MyModule : MasterModule
{
    private AuthModule _auth;
    private IMaster _master;

    void Awake()
    {
        // Register dependency. Initialize method will only be called
        // when modules in dependency list have been initialized
        AddDependency<AuthModule>();
    }

    /// <summary>
    /// Called, when all dependencies are met and master server is about to start
    /// </summary>
    public override void Initialize(IMaster master)
    {
        _master = master;
        _auth = master.GetModule<AuthModule>();

        // Add client message handlers,
        _master.AddClientHandler(new PacketHandler(MyOpCodes.GetPersonalInfo, HandleGetInfo));
        _master.AddClientHandler(new PacketHandler(MyOpCodes.SavePersonalInfo, HandleSaveInfo));

        // Listen to login event in the auth module
        _auth.OnLogin += OnLogin;

        Debug.Log("My module has been initialized");
    }

    protected virtual void OnLogin(ISession session, IAccountData data)
    {
        Debug.Log("MyModule was informed about a user who logged in");
    }

    private void HandleGetInfo(IIncommingMessage message)
    {
        var info = message.Peer.GetProperty(MyPropCodes.PersonalInfo) as PersonalInfoPacket;

        if (info == null)
        {
            // If there's no info
            message.Respond("You have no profile info", AckResponseStatus.Failed);
            return;
        }

        // We found the info
        message.Respond(info, AckResponseStatus.Success);
    }

    private void HandleSaveInfo(IIncommingMessage message)
    {
        // Deserialize packet
        var info = message.DeserializePacket(new PersonalInfoPacket());

        // Update the property value
        message.Peer.SetProperty(MyPropCodes.PersonalInfo, info);

        // Get the session (for no reason)
        var session = message.Peer.GetProperty(BmPropCodes.Session) as ISession;
        Debug.Log("Server saved personal info from user: " + session.Username);
    }
}
```

To handle a message from client, you'll need to add a handler by calling `master.AddClientHandler()`

When Master Server starts, it automatically "scans" the scene for all of the components that extend `MasterModule`.

In general, it's convenient to add your modules as children of MasterServer component, like this:



+ Add a custom footer