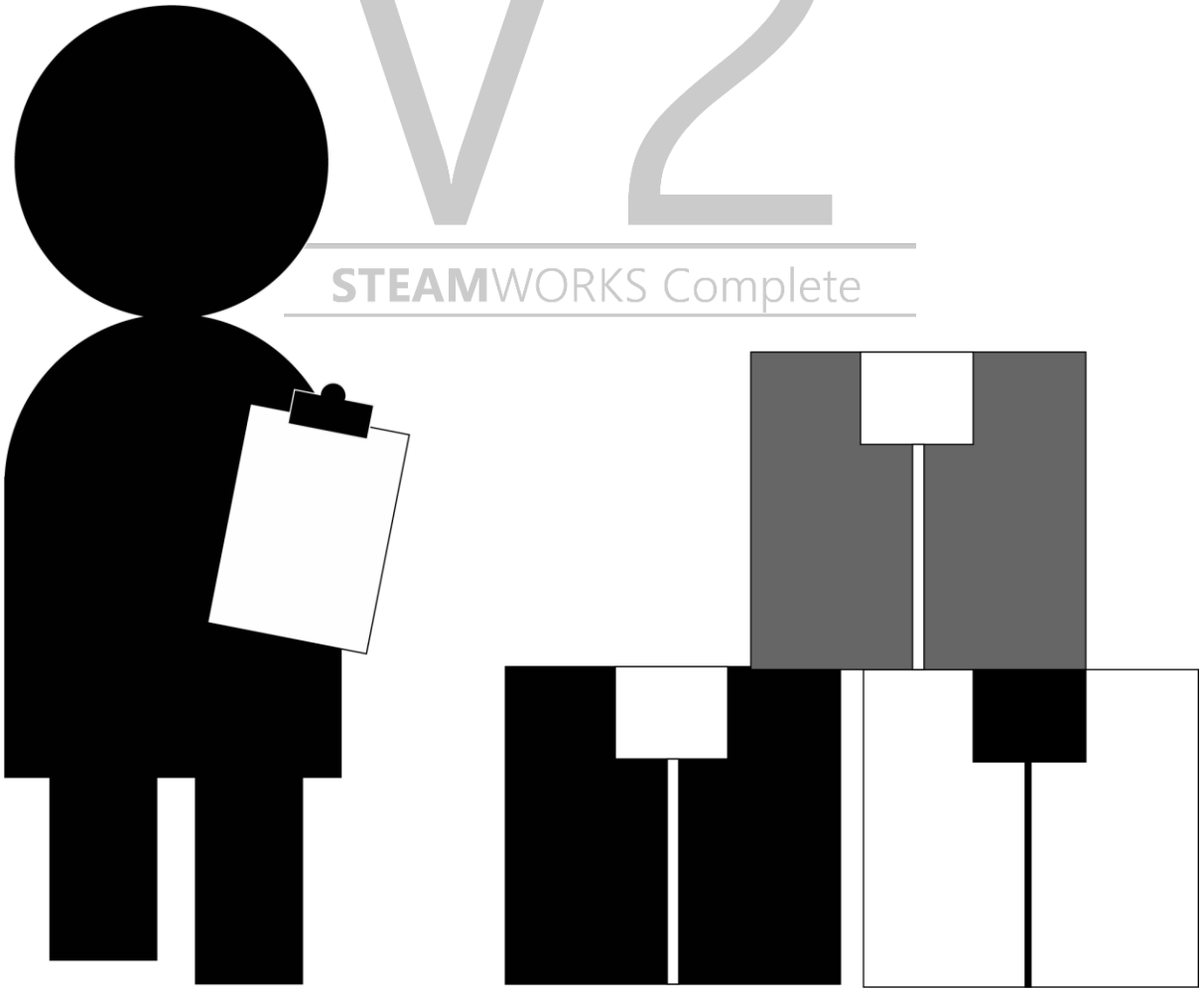
V2

STEAMWORKS Complete



V2.2

Topics new to this release will be prefixed with '**' for example **Group System would indicate that Group System is new since the last update.

Contents

Overview

Heathen Engineering's Steamworks V2 is a set of tools, systems and editor extensions designed to make integrating Steam Client API significantly easier, faster, and more robust.

Heathen's Steamworks V2 (or simply V2 for simplicity) is built on Steamworks.NET. Steamworks.NET is itself a .NET wrapper around Valve's Steam Client API and is a true conversion keeping to the same form and use as defined by Valve's API. The benefit to Steamworks.NET over options is that code snippets, examples, documentation and decades of work from the Steam developer community are directly applicable to Steamworks.NET because it is a 1 to 1 conversion.

The benefit of Steamworks.NET as outlined above is also the main pain point for people integrating Steam's Client API with their Unity game. The source Steam Client API is written in C/C++ using a style of naming and design principles foreign to most Unity developers. Facepunch with its C# focus is slightly simpler however as noted it deviates from Valve's approach and is not a Unity Editor extension rather it's an open-source C# wrapper. Since it deviates from Valve's approach existing code samples and the Steamworks Developer network is of limited use.

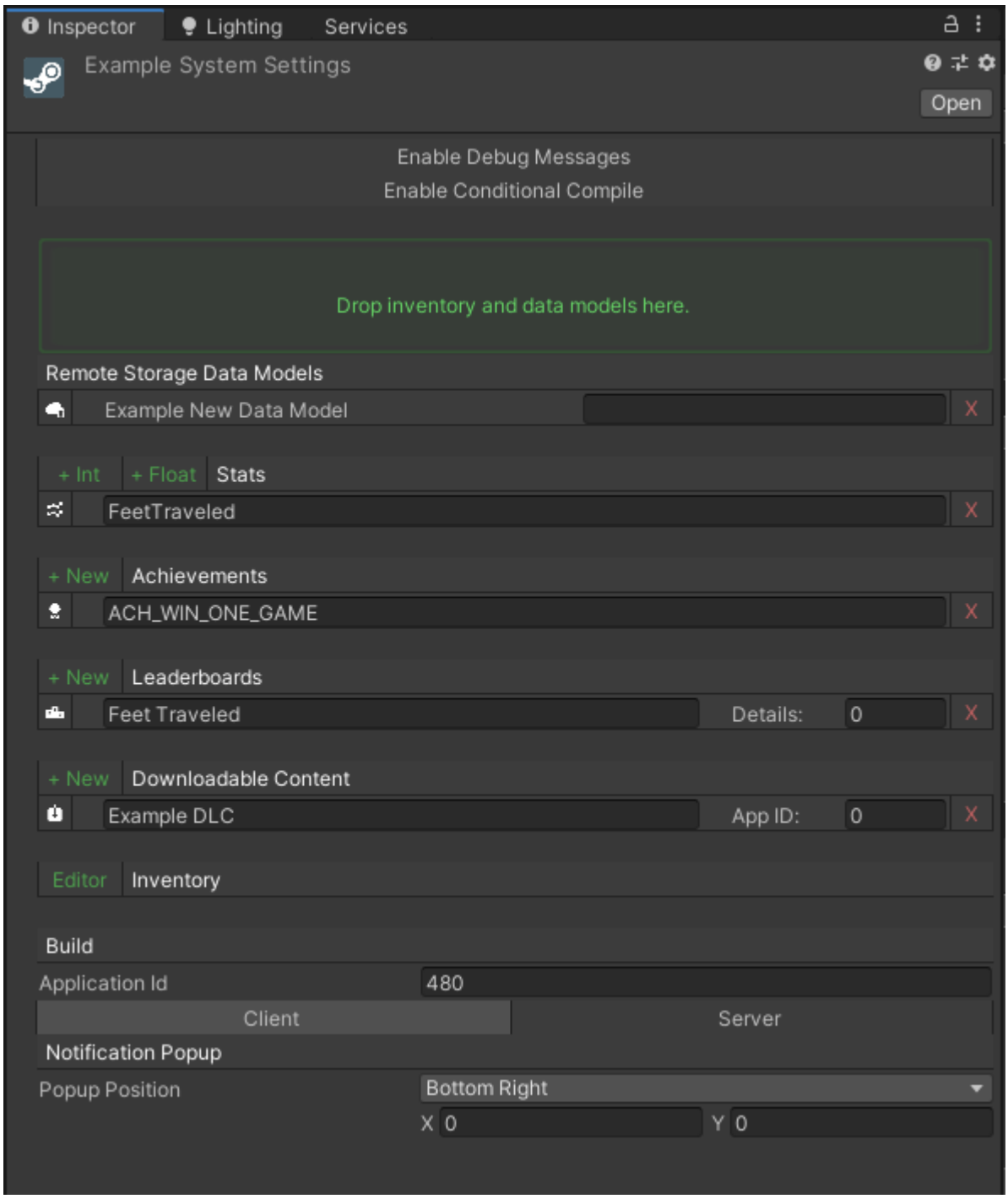
Heathen Engineering's approach with its Steamworks assets has been to take the strength of Steamworks.NET and the parody it has with Valve's original APIs and extend that with Unity centric tools, editor extensions and more to make Steamworks.NET + Heathen the best possible option for Unity developers looking to integrate with Steam. You retain the strength of leveraging raw Steam API when and where you may wish while having the benefit of not just C# focused but Unity focused tools and extensions as well as systems battle tested across many games from many Unity developers.

Rather your new to Steam, Unity or game development in general or a seasoned veteran well capable with the Steam APIs, Heathen's V2 can greatly accelerate your project and help you produce a more robust product that better leverages the services offered by Valve. Heathen's asset does not prevent you from using raw API calls in conjunction with its own extensions and tools. Many features of Steam API can be handled within V2 with little or no coding at all … in the same respect everything is built with respect to the programmer and the need to extend and expand. Modular, extensible, and supported by a large community of fellow developers. Heathen's Steamworks V2 is the best solution for Unity developers looking to ship on the Steam platform.

The remaining sections of this document will be divided into the “modules” of the Steamworks asset. As you will have noticed on the Unity Asset store, we offer 3 levels, being:

1) Foundation
   A free asset containing the basics: including
   a. User Data: name, avatar, status, etc.
   b. Stats
   c. Achievements
   d. Networking transports (Mirror and MLAPI)
2) Player Services (includes Foundation)
   Delivering the most frequently used tools: including
   a. Authentication (client and server)
   b. Clan (groups)
   c. Downloadable Content (DLC)
   d. Leaderboard
   e. Remote Storage (cloud save)
3) Complete (includes Player Services and Foundation)
   Covering the full Steam Client API with powerful tools for virtual every Steam feature including:
   a. Game Server
   b. Inventory (microtransactions, marketplace, etc.)
   c. Matchmaking (lobby, server browser, etc.)
   d. User Generated Content (workshop)
   e. Voice
   f. Custom debugging tools

Quick Start



The above is a screen grab of the Example Steam Settings object from the example scenes. It was taken within the Steamworks Complete package so may have fields you do not and is fully populated with features you may not have access to if you're on Foundation or Player Services.

The following steps will help you create a settings object from scratch.

1) Create a new Steam Settings object in your project folder by right clicking in your Project tab and selecting
   Create > Steamworks > Settings
2) Under the build section in the settings inspector enter your “App Id” as provided by Valve
3) Create a Client Api System on a suitable game object and reference the settings object you created in step 1

Congratulations you are now integrated with Steam Client API. If you run the simulation, you will see that the Steam API initializes, and that the Local User Data object housed under the settings object populates with your Steam user info during play.

IMPORTANT

The Client Api System is a critical component that should not be destroyed or reloaded. Heathen Engineering suggests using a model such as bootstrapping and housing the Client Api System on a Game Object in that scene.

What is bootstrapping?

It means your first scene in your project is ultra-light and does nothing but display a loading screen and initialize system level managers and interfaces … such as the Client Api System. Since this scene is only ever loaded once it avoids the pit falls of putting such managers on Game Objects housed in the title scene or similar where it would be reloaded multiple times often resulting in damaged references.

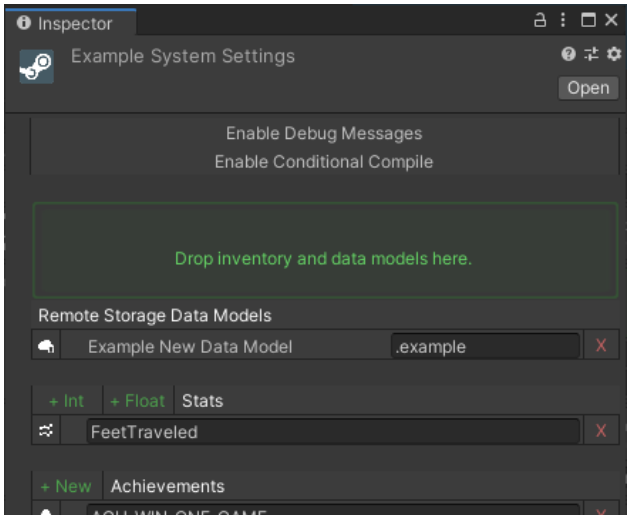
Bootstrapping is a useful mode rather or not you have such managers. It’s a logical place for your loading screen, error dialog, system validation, splash screen, main camera and any other system level object that should be first and persist through out your game session.

Features

Foundation

Steam Settings

The core of the Steamworks system for all modules is always the Steam settings object. Every feature and every component can be accessed from this one object. The settings object is a Scriptable Object meaning it can be referenced at dev time and across all scenes.



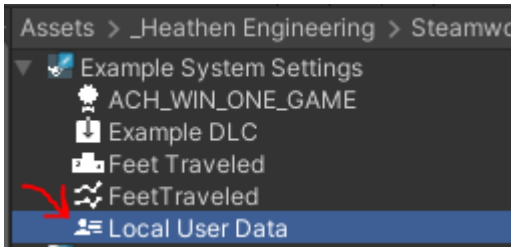
The core functionality of the settings object is housed in the client and server members and these are available by instance reference and static member e.g.

SteamSetting.Client or SteamSettings.Server
as well as
steamSettings.client or steamSettings.server …. Where steamSettings is a variable of type SteamSettings.

Note that features and functionality found in the client are not available for use with servers … similarly features and functionality found in the server is not available for use with the clients. Most of your work is likely to be the client object.

User Data

User Data is a type of object managed by the Steam Settings object. User Data contains … user data 😊 namely … names 😊 but also the user's icon or avatar and persona state information. For the local user the User Data object is always available, you can find the object in your project folder under the Steam Settings object named Local User Data.



You can create a reference to drag and drop this to in your scripts via

public UserData userData;

Get User Data for another user

User Data can be fetched for Steam Users the local user is “aware” of by calling SteamSettings.Client.GetUserData(...)

Note that Valve does not allow you to pull just any users data at any time. The Valve system has a concept of who the user should be able to know based on various conditions. For example, if the user is in a lobby, server or loading a leaderboard where the user is present or if the user is the friend of or in a clan or group with the user then the object will populate normally. In any other condition the object is likely to remain empty or be missing information such as persona state.

Using User Data

User Data objects also contain short cuts to common user related functions such as OpenChat and OpenTrade and access to rich presence data via GetRichPresenceValue and GetRichPresenceValues

You can track changes to the user … assuming Valve determins you should be able to … such as a friend or a lobby member … via the events provided.

Friends

You can list the users friends by simply calling SteamSettings.Client.ListFriends(…) this will return a list of User Data obejcts as described above.

Stats and Achievements

Create stats and achievements directly on the Steam Settings object’s inspector; you will notice new sub-objects created under the settings object and these can be referenced in behaviours easily.

```
public AchievementObject myAchievment;
public StatObject myStat;
```

Each object contains fields and functions suitable for it. For example the AchievementObject has a simple Unlock() method that can be called to unlock the achievement for the user or rather send a request to the Steam client to do so.

Please note that your configuration in the Steam Developer portal for each stat and achievement impacts what can and cannot be done by the Client API. For example, if you set the stat or achievement such that it can only be written to by a “Trusted” source then the client API will not be able to set the value rather you will have to use the Steam Web API to set it from a trusted web server. This is used to prevent players from cheating and setting the value to whatever they please with a hacked or spoofed client.

Note you can ask Valve to refresh your client’s view of all stats and achievements at any time by calling SteamSettings.current.RequestCurrentStats(); this is an asynchronous call meaning the stats and achievements won’t immediately be updated when this call is completed. The SteamSettings.current.evtUserStatsReceived event will be raised when the call is completed.

Note that stats and achievements must be created in the Steam Developer Portal and published before Unity will be able to access them. Your Unity stats and achievements are simply proxies the Client Api System will use them to store information read from Valve’s backend and methods on them will be routed through the Steam Client API to Valve for execution. This model is typical of all Steam API based systems and objects.

Networking Transports

You will find several Unity packages in the HeathenEngineering/Steamworks/Framework/Foundation/Networking folder 1 for each networking API we have integrated with. These transports enable for example Mirror to leverage the SteamNetworking APIs for P2P and Client Server based networking.

Please note that we provide transports, Mirror, MLAPI, etc. are your networking APIs used to sync data between Game Objects, etc.

All transports provided are ports of the community created transports for each respective network interface unless otherwise noted. Heathen modifications are limited to only the changes required to work with Heathen's tools such as removing references to SteamManager and replacing it with references to SteamSettings (Heathen). We also where applicable modify the transports to work with both Steam Client API and Steam Game Server APIs thus enabling Client/Server structures to work as well as classic Peer to Peer

Player Services

Authentication

This is a simple static class that handles Steam Authentication. Note that Steam Authentication is typically only used with a Steam Game Server or a custom Web server but can be used virtually anywhere. For a Steam Game Server, a player authenticating with a server is how the Steam Game Server system on Valve's back end knows how many players are on a server.

Note that you probably do not need to use this. A user cannot initialize the Steam API with your App ID unless they are both a valid Steam user and own a legitimate license to your app. Authentication to prove who a person is or that they own your app is thus moot as if the API is initialized then you already know this to be true.

Also do note that Authentication has separate methods for Client and Server, the functions are the same, but the calls are unique between each.

Finally, the basic concept of Steam Authenticaiton. In short a user or client will generate a Ticket via Authenticaiton.ClientGetAuthSessionTicket() this returns a Ticket which contains a byte[] named Data. It is that data that you will send to your target server or whoever else it is that needs authenticate you. The receiving party if they have access to the Steam API via our asset can use for example Authentiation.ServerBeginAuthSession(data, userId, callback) … call back in this case is simply an action e.g., a method that takes a param of type Session … such as

```
public void HandleSessionCallback(Session session)
{}
```

assuming that your call to start a session would be

```
Authentication.ServerBeginAuthSession(data, userId, HandleSessionCallback);
```

Clans or Groups

Steam Clans or more often called Steam Groups is a social networking structure similar to multi or cross game guilds.

The main use of this is to list the users clans e.g. SteamSettings.Client.clanSystem.RefreshClanList() this will update the SteamSettings.Client.clanSystem.clans list. It is possible to join clan chats, iterate over clan members and perform other actions via the SteamClan objects found in the list.

Downloadable Content (DLC)

DLC is an incredibly power and simple system that depending on your implementation within your game can be very easy to set up. Note that a DLC is defined in your Steam Developer portal where it will be issued an App Id.

You can then create a DLC object in your Steam Settings object like how you created a stat or achievement. Once done our system will be able to fetch data about the DLC such as rather or not the user owns it … or in Valve terms is "subscribed". Its recommended that you call UpdateStatus on the DLC object just before checking its IsSubscribed value, but you shouldn't do this to frequently e.g., every frame as that might get you rate limited.

As with stats and achievements you can create a reference to DLC objects such as

```
public DownloadableContentObject dlc;

...
```

```
dlc.UpdateStatus();
if(dlc.IsSubscribed)
    //The user owns this

…
```

Note that you can also check download progress and similar however in most modern games the game always contains all its content and DLC is simply a license to check and see if the game should unlock that content or not so there is nothing to download and that would be the usual recommendation. If you do want to handle downloading and installing DLC you can check the IsDownloading and IsDlcInstalled members.

Leaderboards

Leaderboards work like stats and achievements in that they can be defined in your Steam Developer portal and set to only allow write operations from a "Trusted" source as a mechanism to prevent cheaters. If you set your Leaderboard to write = "Trusted" in the portal, then you cannot set the score of the Leaderboard with our kit or the Steam Client API in general rather you will need to use the Steam Web API on a trusted web server to set the value.

Assuming your leaderboard is designed to allow client write operations you can use methods on the LeaderboardObject to set the score and data for the user via UploadScore.

To create a new LeaderboardObject select your SteamSettings object and click the + New button by the Leaderboard header. If you do not see this header then you do not have Player Services or Complete and need to upgrade to use Leaderboards.

Understanding Leaderboard Data / Details

Leaderboards can store additional data per entry, either via the User Generated Content system or by packing the data into an array of ints. The more common and faster solution is to pack the data into the array of ints and you will find overloads of the UploadScore. Note that when our system reads scores from the board it will check for a data array with an index if the one you specify in the MaxDetailEntries field on the LeaderboardObject.

For Example:
Lets assume you have a leaderboard that you want to store the players class, race, and level on with each score entry. Your game should convert this data to ints so lets assume you have done so and your data can be expressed as an int[3]. You will want to configure your LeaderboardObject such that MaxDetailEntries = 3 and when you upload data you will use
leaderboardObject.UploadScore(score, dataArray, method);

Reading Leaderboard Data

Heathen provide a few tools to help you read leaderboard entries such as the LeaderboardDIsplayList which helps you render query results from a particular board to a UGUI element. You can also do this manually very simply by calling one of the Query Entries methods on the LeaderboardObject and handling the evtQueryResults event. The evtQueryResult event returns a LeaderboardScoresDownloaded parimiter which contains information about the execution e.g., failed or not, rather the result set contains the local user's score and then of course a List of ExtendedLEaderboardEntry records which will have the entry data score, etc and any details.

Fetching UGC based details

The ExtendedLeaderboardEntry object returned by the query contains helpers for accessing various bits of data including the UGC file name. You can use the entry.StartUgcDownload to start the

download process which will raise the evtUgcDownloaded event and optionally invoke a custom callback on completion. The name of the file can be used with standard IO to load the file as you would a local file from that point.

We have an additional helper new to V2.2 which will handle loading the target file into a object for you. ExtendedLeaderboardEntry.GetAttachedUgc<T>(callback);

Remote Storage

The remote storage system allows you to easily save and load any serializable data you like to the Steam cloud servers for the user. This system handles syncing data for the app between the cloud and the local disc for faster read/write times.

You can access the Remote Storage System via SteamSettings.Client.remoteStorage. The Remote Storage System has features such as RefreshFileList which will build a list of FileAddress available to the user in the files member on the Remote Storage System object. This can be used to "browse" the files saved.

You can also pre-define your data models e.g., the shape and structure of your files in the system as Scriptable Objects and use tools on the Remote Storage System to fetch the appropriate data model for a given file by the files name. This is useful when your game has multiple save types such as a systemSettings, userPreferences, and save profiles all with unique data models.

See the Data Model section for more information

Data Model

Heathen provides you with a simple base class called Data Model to help you define custom data model objects.

What is a Data Model?

It's a Scriptable Object that defines the shape of and provides a memory storage point for the current loaded file. By shape of we mean what fields does it have and what data types are those fields.

How do I define a Data Model?

You simply create a class or structure that has the fields you want to save in it as public members. Be sure that fields are serializable and that the class or structure has the [Serializable] decoration applied … e.g.

```
[Serializable]
public struct MySaveData
{
  public int level;
  public string characterName;
  public bool someOtherValue;
}
```

Once you have your data type defined you can apply it to a Data Model simply by

```
[CreateAssetMenu(menuName = "Steamworks/Examples/Data Model")]
public MySaveDataModel : DataModel<MySaveData>
{}
```

Note there is no need to write any code in the body of MySaveDataModel … all necessary parts have been provided for by the base class. You can see a working example of this in the ExampleNewDataModel.cs located in the (2-A) Remote Storage example folder.

Note that each data model defines an "extension" which is simply a string that is expected to be at the end of each file name. This is how the system knows which file belongs to which model. If you leave this blank, then all files will be assumed to match this model. If you populate it with for example ".settings" then only files whose name ends with ".settings" will be belonging to this data model by the Remote Storage System.

Note DataModels contain a list of all available files that match there extension. When Remote Storage System poles Valve for files it will match each by extension and store the results in the dataMode.availableFiles collection as a FileAddress. FileAddresses contain information such as the files name and any other metadata such as time stamps available from Valve.

How to Save Data?

Once you have a DataModel it provides all the helper functions you will need. You can simply call dataModel.Save(filename); note this is a file name not a location the location is determined by Valve … so dataModel.Save("systemsettings"); will name the file system settings. Also note that I did not need to specify the extension if an extension is specified it will automatically apply it to the end if missing.

Also note that I didn't pass in any data to be saved … the system reads the data to be written to the storage medium from the DataModel object itself … e.g. dataModel.data is a variable of the data type you specified and will contain the data to be written to the storage system.

How to Load Data?

You have multiple options for loading data but the simplest is to LoadFileAddress using a FileAddress object such as found in the availableFiles collection
or
LoadFileAddress using a string name of the file to load. You can also load data from a JSON string or byte[] and can load file addresses asynchronously if desired.

As with save you don't need a return of the data type in question, information loaded will be stored in dataModel.data. The data member of your model represents the currently loaded data for this file type.

How to load multiple files?

As noted, we store the current loaded file in the dataModel.data member so if you want to hold multiple files in memory at once you will need to move them over to another storage point in memory … for example

for each file address loaded
collection of datatype add dataMode.data
then iterate to read the next file

Complete

**Steam Game Server

New in V2.2 due to changes in Steamworks.NET the port structure has changed. As of the time of this writing there are known issues with Steamworks SDK v1.51 which Steamworks.NET v15 is based on. We have implemented a basic workaround that should solve for most issues and will update this further as soon as a satisfactory change to the underlying assemblies has been released.

Heathen's Client Api System can initialize Steam API either as a proper client or a headless "server" and direct the callbacks through the proper updates. You will notice in your Steam Settings inspector that you have "Server" settings as well as Client settings.

The Server settings let you configure your build as a "Steam Game Server" this means that your server build will connect to Steam CMD and register its self on Steam's network. Steam will issue the server a CSteamID which can be used for server discovery and by the SteamNetworking and SteamNetworkingSockets APIs.

Note that you do not have to use SteamNetworking or SteamNetworkingSockets to register a server as a Steam Game Server. Steam Game Server is simply a mechanism to make Valve's backend aware of your game server so you can avail of other features of the Steam API such as the Steam Game Server Browser and matchmaking systems.

Heathens tools for Game Servers includes the GameServerBrowser component. You can add this component to a GameObject and use it to browse for Steam Game Servers related to your app. An example of this in action can be seen in the Game Server demo scene.

Inventory

Steam Inventory is perhaps the single most valuable system Valve provides and one of the most complex. In a nutshell Steam Inventory is a way of defining digital goods such that Valve can sale them for you on its store (optionally) store them for your player in Steam Inventory, allow players to trade them in or out of game (optional) and allow players to market them out of game (optional). Let's look at each core feature of a Steam Inventory Item on its own before we go over Heathen tooling around the feature.

MTX or Micro Transaction

Steam Inventory items can be sold on the Steam store and the Steam Inventory system and its schema (more on that later) is how you define these items and their prices. Since items are sold through Steam and housed in Steam Inventory you do not need a server to manage this greatly reducing the overhead of running a MTX system. This also means you are not on the hook for payment processing, GDPR (data protection), etc.

A user would navigate your store, select, and purchase items and you would see the funds much like you do for sales of games or DLC. You can also list your items in your game and initiate a sale from within your game … the sale will always be handled and completed though through the Steam Client e.g., Overlay for example.

Items live in Steam Inventory

You know when you get trading cards that place on your Steam account where stuff collects … its can be organized by game as well. This is your player's inventory, and this is where Steam Inventory items once, generated, purchased or traded for will end up. You can query the items in here that belong to your game or a related app and then use them in your game.

This is much more powerful than a simple collection of junk. Here are some useful features of Steam Inventory Items.

- Items can be stacked and split … that is you can have a pile of 872435 gold and you can split that into a smaller stack of 500 to trade to your friend.
- Items can be consumed … that is an item could be a potion that is 1 time use
- Items can be exchanged without the need of a server … that is you can define crafting recipes that say 4 iron and 50 gold can be exchanged for 1 iron long sword. Or you can say 1 treasure chest can be exchange for a random set of items where what drops has a probability matrix attached.

With these features you can create crafting systems, loot boxes, consumable potions and player economies that can either include real money MTX or not. As we will see in the next major feature you can also support player marketing.

Player Market

This is where you allow your players to buy and sale the items, they own on the Seam Marketplace. You get a small cut of the transaction and your player keeps the rest. A solid option for adding to your revenue stream without dealing with MTX is to allow your players to buy and sale the items they earn in your game on the marketplace.

Note you can control per item rather or not an item can be sold on Steam store, players can sale on Marketplace and or players can trade with each other.

Generators

This is a feature that enables you to generate items without the need of a web server backend to act as a trusted broker. Generators can also be used with trusted web servers to handle generating items by probability, generating tags (more on that later) and much more.

A generator is simply an item that resolves to 1 or more other items … and yes you can have a generator that resolves to 1 or more other generators so long as at some point down the line it ends up as a set of true items.

Tag generators are another concept … in short a tag is a modifier to an item that persist with the item … so for example you could apply the “proc:fiery” tag to your iron sword … its still an item type iron sword it simply has the tag fiery associated with it. Note that tags have 2 parts … the tag name and the tag value for example we could have a tag “proc” that has 4 types “fire”, “ice”, “earth”, “lightning” … tag generators can handle the probability of one of these over the other.

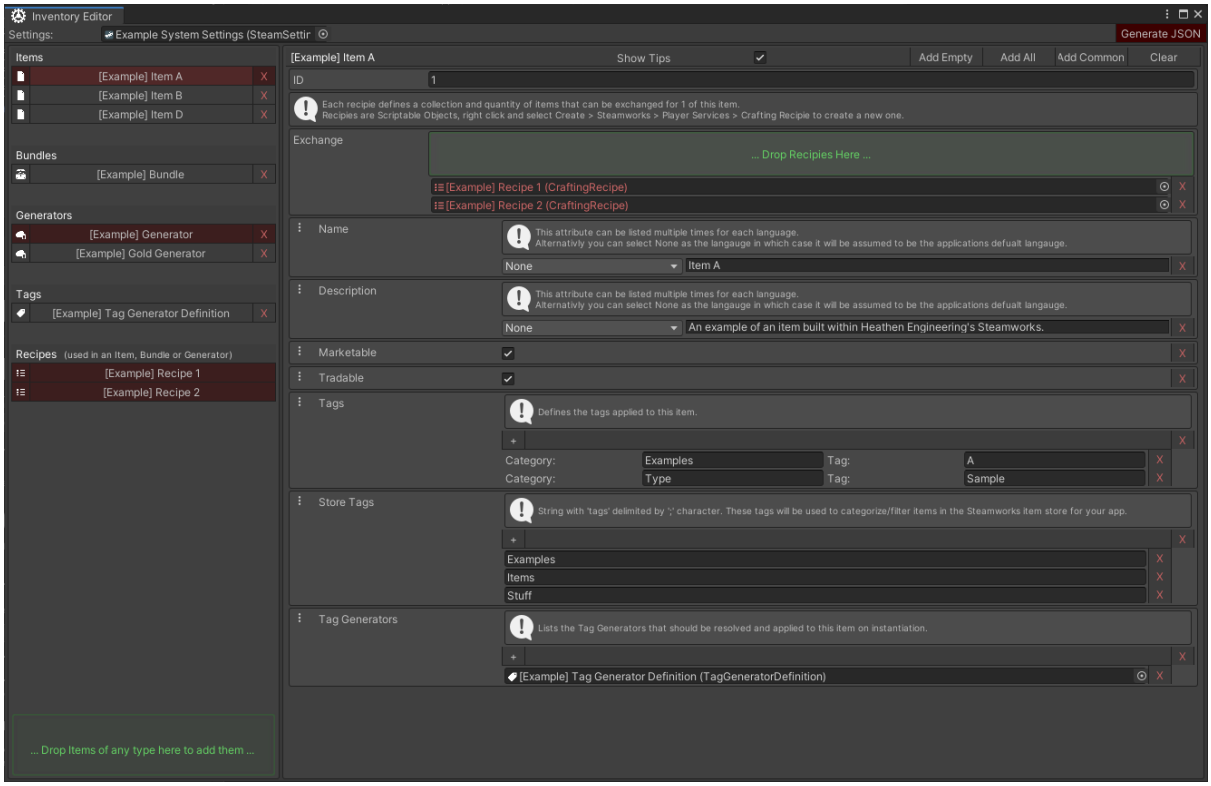
Tags also apply to for example rarity e.g.,
Tag = quality
values = common, uncommon, rare, epic, and legendary

Now when you generate a sword say from a crafting recipe it could have a % chance to generate as any of the above.

Heathen Editor Tools

Access the Steamworks Inspector in the main menu under Steamworks > Inspector to access the Inventory Editor.

Heathen has a number of tools to help you design and generate your item definition JSON … this is how Valve’s Steam Developer Portal is informed about your items … you create a JSON that describes each item’s features in a rather complex schema.



Thankfully Heathen has a visual tool to help you create this JSON object and identify trouble spots ahead of time. In the image above you can see we have several different types of items defined and a few (in red) have logical errors with them. Once all errors have been resolved we could click the "Generate JSON" button in the upper right and have a good starting point for our definitions.

Note this tool helps you get started if you have particularly complex conditions you may want to further edit the JSON file and this will not harm your Unity side definitions.

Creating Item Definitions

Most item types such as generators and recipes are generic however the Item Definition is not. You are expected to create a custom data type for each of your items to hold game specific data. To do so simply create a new class derived from InventoryItemDefinition and populate it with the fields and information you want and need in your game. Note that this is a Scriptable Obejct type so don't forget to decorate it with a CreateAssetMenu so you can create it in your project folder.

You can see an example in the ExampleInventoryItemDefinition.cs located in the (2-C) Inventory demo folder. Once you have defined your item definition types you can create them as you would other objects in your project and drag and drop them to the Inventory editor or SteamSettings object to add them to the system.

Using Inventory at runtime

With your items defined they will appear in your SteamSettings object and Heathen's Client API System will handle the initialization and load of inventory on demand. This means you should call SteamSettings.Inventory.RefreshInventory() before performing actions on the inventory items or if you think the inventory has recently changed. You do not want to call this to frequently or Valve will rate limit the transaction meaning it will ignore them for a bit.

With your inventory refreshed the Item Defnitions you have created will serve as your interface to each item and can be used to perform all relevant actions on the item such as checking the sum or count … sum being the number of units of this item and count being the number of instances … note an instance can have a quantity greater or less than 1 so not every instance equates to 1 unit. Common features include

- Consume (count)
  This tells the system to consume or delete X number of items
- Start Purchase (quantity)
  This tells the system to (if valid for this item) add this number of items to the shopping cart and open it for the user
- Craft (recipe)
  This tells the system to (if supplies are available) exchange the goods defined in the recipe for one of these items … assuming it is valid to do so

Tips

Play Time Generators, Promos and other tools can be used to generate Items for players as they play the game, but these cannot generate specific items only items that can be determined by Valve's backend to be valid at that time.

This is a matter of security … in short Inventory items have potential real-world value so Valve will never trust a game client to generate items on its own. Play Time Generators are basically ruling you tell Valve so it can know when its aloud to generate items for the player. If you want to generate specific items on demand, then you must use a trusted web server and the Steam Web API to do so.

The specific design needs of your game will determine if you can implement inventory without a dedicate backend server or if you require a backend server to meet your specific needs. As with many aspects of Steam limitations are placed on Client API for security sake e.g. trusted Stats, Achievements, Leaderbaords and any form of inventory generations out side pre-defined generators.

Matchmaking

Matchmaking aka Steam Lobby, this is probably the most used system of Steam API. Leaderboard, Stats and Achievements are more common but are simple values. The matchmaking system lets players create lobbies to gather and share data to agree on gameplay conditions. It includes a simple chat system to help communication and tools to help notify users when a server is ready to connect to.

See the Lobby note for details

Note the lobby system is a match making system, it is not intended to be a server discovery system. Its queries are by design meant to return as few lobbies as is relevant for the search parameters and organize them such that the optimal lobby is returned first (usually). The goal of a lobby search is to get the player into a group with other players that are optimal for the game that player wants to paly and for where that player is geographically.

Note the MatchmakingTools stores the lobbies it is joined to in the lobbies collection. In the examples below I will use MatchmakingTools.lobbies[0] this simply means the first Lobby in the list but you should use the index of the lobby you want this is only to simplify the examples.

***Creating a lobby*

New to v2.2 are additional tools for managing multiple lobbies and understanding the type and purpose for each lobby. You now have 3 options for creating a lobby.

** MatchmakingTools.CreatePublicLobby(…)

This short cut creates a normal lobby of type public.

** MatchmakingTools.CreateGroup(…)

This short cut creates a group lobby of type invisible. See the Group System section for more information.

MatchmakingTools.CreateLobby(…)

This method will create a new lobby according to the settings you provided and join the player to it. You can specify the lobby type here … note that you can change a lobby type after creation so for example you could create the lobby as private, set up its metadata and when ready set the lobby to public.

All the lobby create methods have an optional “callback” see the note for details

Note that the evtLobbyCreated event will be invoked when the process is complete so you can choose to use the callback or event as you see fit.

Joining a lobby

MatchmakingTools.JoinLobby or SteamMatchmaking.JoinLobby this assumes you have the lobby ID available in order to tell the methods which lobby to join.

Note this will join the user to a new lobby it will not remove the user from any current lobbies. Our system manages multiple lobbies however most games only deal with 1 lobby at a time so you may wish to call LeaveAllLobbies first to insure only lobby is managed at a time.

Searching for lobbies

You can search for a lobby in 1 of 2 ways, both methods use the LobbyQueryParameters object to describe the filter that should be applied to the search. The LobbyQueryParameters is an important object rather or not you are filtering your search results. The parameters member onQueryComplete is an Action that will be invoked when the query is completed, and this is how you get the results of a query. As with other callback type models we also raise a UnityEvent named evtLobbyMatchList with the same data so you can use either approach to get the results of a query.

FindMatch

This is a traditional linear search and will return any results found via the onQueryComplete action of the search parameters.

QuickMatch

This works like FindMatch but on its first iteration it will tighten the search distance to the minimal (Close) and over each iteration up to 4 searches it will lose the condition up to the max of (Worldwide). As with FndMatch the onQueryComplete action will be raised when complete e.g. when a lobby is found.

Note that QuickMatch does not auto join the lobby for you, if you want to immeaditly join the lobby you can simply call MatchmakingTools.Join(result[0].lobbyId) … assuming result has any values

***Group System*

New to v2.2 the group system is simply a built-in tool that manages 1 invisible lobby as the “group” lobby. Note that Valve allows a user to be a member of 1 “normal” lobby and up to 2 “invisible” lobbies. This can be used for any purpose such as merging groups of players together and similar. In this case we have built a system that helps you manage 1 invisible lobby as a “group” aka a “party” … not to be confused with Valve’s Player Party system.

The idea is that you can call CreateGroup to create this lobby and we will track it as MatchmakingTools.groupLobby; Here your player can group up with friends and use the lobby chat system to communicate. The host of the group lobby can search for a “normal” lobby to join that will fit the whole group and can inform its fellow group members which to join and when.

Note that “invisible” lobbies (despite type name) can be searched for in a lobby query. So it is possible to search for a group to join. We have created 2 metadata fields to help you narrow your searches.

- z_heathenMode
  normal lobbies will set this to “general” and group lobbies will set this to “group”
- z_heathenType
  this indicates the Steam lobby type e.g. public, private, friend or invisible. Remember group lobbies are always “invisible” but general lobbies can be public, private or friend.

With this you can easily search for a lobby my its “mode” and create a means for players to search for a group to join or a lobby.

Note that a user can be a member of 3 lobbies, 2 of which would be invisible so you have a 2nd invisible lobby you can use for other purposes.

For example if you wanted to merge 2 groups of player's you can have 1 create a new invisible lobby via the MatchmakingTools.CreatLobby(…) method and let the other player's know to join this new lobby. You can then decide to maintain the old invisible lobbies as a sort of "private group" vs "public group" system or you can dispose of the former group lobbies.

You can set any lobby to be a group lobby by simply setting its IsGroup = true. This will update the lobby type to invisible and will set the MatchmakingTools.groupLobby reference if its not already set. If you set this parameter to false it will remove this lobby from the MatchmakingTools.groupLobby field but will not change its type.

To change a lobby type you can now simply set its type field e.g. myLobby.Type = ElobbyType.k_ELobbyTypePublic. This will set the MatchmakingTools.normalLobby and MathcmakingTools.groupLobby fields as required. Note that setting a lobby as invisible doesn't make it the groupLobby but setting any lobby to a general type will set it as the "normalLobby".

Setting Lobby Metadata

You can set the metadata of a lobby by using its indexer … e.g.

MatchmakingTools.lobbies[0]["this Field"] = "this value";

This would set the first lobby's metadata to have a field named "this Field" and to have a value of "this value". Note that this will create the metadata field if missing or update it if present. Metadata is always a string-to-string key value pair. You can fetch a list of all metadata on a lobby by calling the GetMetadataEntries on the Lobby object.

Note that only the owner of a lobby can set its metadata, but everyone can read it rather or not they are a member of the lobby.

Getting the lobby owner

MatchmakingTools.lobbies[0].Owner this returns the LobbyMember value of the "owner" or "host" of the lobby. The LobbyMember object provides tools for fetching the metadata related to that player and well as that player's User Data object

Get the local user's LobbyMember object

MatchmakiingTools.lobbies[0].User this returns the LobbyMember value if any of the local user

Set local user metadata

MatchmakingTools.lobbies[0].User["this field"] = "this value"

This works just like setting metadata on the lobby, only the user in question can set its metadata and anyone in the lobby can read that data … non-members of the lobby cannot read a member's metadata

User Generated Content

User Generated Content aka UGC aka Workshop items, this is the system for creating and uploading data to Steam from a user … most frequently used to handle player mods. The tools can be accessed via SteamSettings.UGC and new items can be created with a single line call

Creating a new item

SteamSettings.UGC.CreateItem(…)

This method takes as input all of the required data for creating a new UGC item which is as follows

- The target app
  This is usually the app that is creating it e.g., your game
- Title
  The title of the UGC item as it will appear in the workshop
- Description
  again as it appears in the workshop
- Change Note
  This is simply an annotation such as “initial release”
- Content Folder Path
  This is a folder path not a file path, it is the folder that contains the content that Valve will upload. Note it should not be a zip, Valve will handle scanning the folder, bundling up all the data and uploading it for you … just direct it to the target folder nothing else.
- Preview Image Path
  This is the image that shows as the thumbnail and main image on the item … this is a file path and should point to a JPG or PNG … keep in mind there are size limits check the Steam documentation to keep on top of what that is but in general JPG are a lot smaller so more appropriate to use here.
- File Type
  This is usually k_EWorkshopFileTypeCommunity which is the normal workshop sort of item, See the Steamworks Documentation for EWorkshopFileType to see all the options … there are 16 in total
- Visibility
  This defaults to k_ERemoteStoragePublishedFileVisibilityPrivate and effects who can see the item from the start … the expectation is a user uploads from your game, goes into there workshop, edits as they see fit and then sets it public when ready.
- Completion Callback
  This gets invoked on completion and is a way to be notified when the process is complete, evtItemCreated or eveItemCreatFailed will be invoked as well as any callback provided

Updating an item

This requires a few extra steps,

1) Call the SteamSettings.UGC.StartItemUpdate method providing it with the ID of the file to be updated and the app ID the file belongs to. This will return a UGCUpdateHandle_t … keep this handy you will need it for future actions
2) Call any (or all) of the update methods you require to change values on the file … each of this will require the update handle from step 1 as its first parameter … your options are as follows
   Notes as to each one’s use are available in your IDE via the tooltip / intellisence
   a. AddItemKeyValueTag
   b. AddItemPreviewFile
   c. AddItemPreviewVideo
   d. GetItemUpdateProgress
   e. RemoveItemKeyValueTags
   f. RemoveItemPreview
   g. SetItemContent
   h. SetItemDescription
   i. SetItemMetadata
   j. SetItemPreview

k. SetItemTags
l. SetItemTitle
m. SetItemUpdateLanguage
n. SetItemVisibility
o. UpdateItemPReviewFile
p. UpdateItemPreviewVideo

3) When you are happy with the changes call SubmitItemUpdate … this again requires the update handle as its first parameter.

Getting items

There are several ways of getting or fetching items from Valve depending on your objectives … follows are a few options

- Get Subscribed Items
  This returns a list of PublishedFileIds representing each item the user is subscribed to you can use these IDs to get additional information about each item
- Workshop Browser
  This is a mono behaviour meant to be added to a Game Object and to help you with browsing for and displaying UGC items see the demo scene for a working example
- Manually via the WorkshopItemQuery object … this object simplifies the process of building up a query and managing the handles. You can see this object in use in the Workshop Browser. Note this object handles paging that is loading in a finite number of values and iterating over that page by page.
  when you have built up your query you can run it via query.Execute and pass it a callback to be invoked when its complete. This callback will have access to the query.ResultsList

Voice

Steam Voice is a simple tool for capturing, compressing, and decompressing audio from the user's hardware. To use the system, you will need to add the SteamworksVoiceManager to a Game Object in your scene this behaviour will handle recording audio and compressing it for transport as well as decompressing audio received and feeding it to an audio source for playback.

Note the (3-D) Voice demo scene demonstrates the use of this system and does so as a "loop back" that is it pipes the evtVoiceStream event into the PlayVoiceData input of the manager so that your own voice plays back to you with a delay approximately the same as your buffer + the processing time.

Configuring the manager

The manager has 3 modes it can sample with

- Optimal: which is the suggested mode
- Native: which is determined by the user's system
- Manual: where you can dial in the desired sample rate

In addition, you can specify streaming which effects the nature of the audio clips produced for playback.

Buffer length is used to buffer up input data before sending it out via the voice stream. This is the time in seconds to gather before sending. Note by sending we mean invoking the evtVoiceStream event.

Sending Audio

First know that its up to your networking system of choice as to how you get the audio data to the receiving system. The tool we provide is concerned with gathering the data and compressing for send only.

When the tool is recording it will fill up its buffer and then invoke the evtVoiceStream event, your network system should listen on this event and send the byte[] it passes along as voice data. The array is already compressed and ready for sending

Receiving Audio

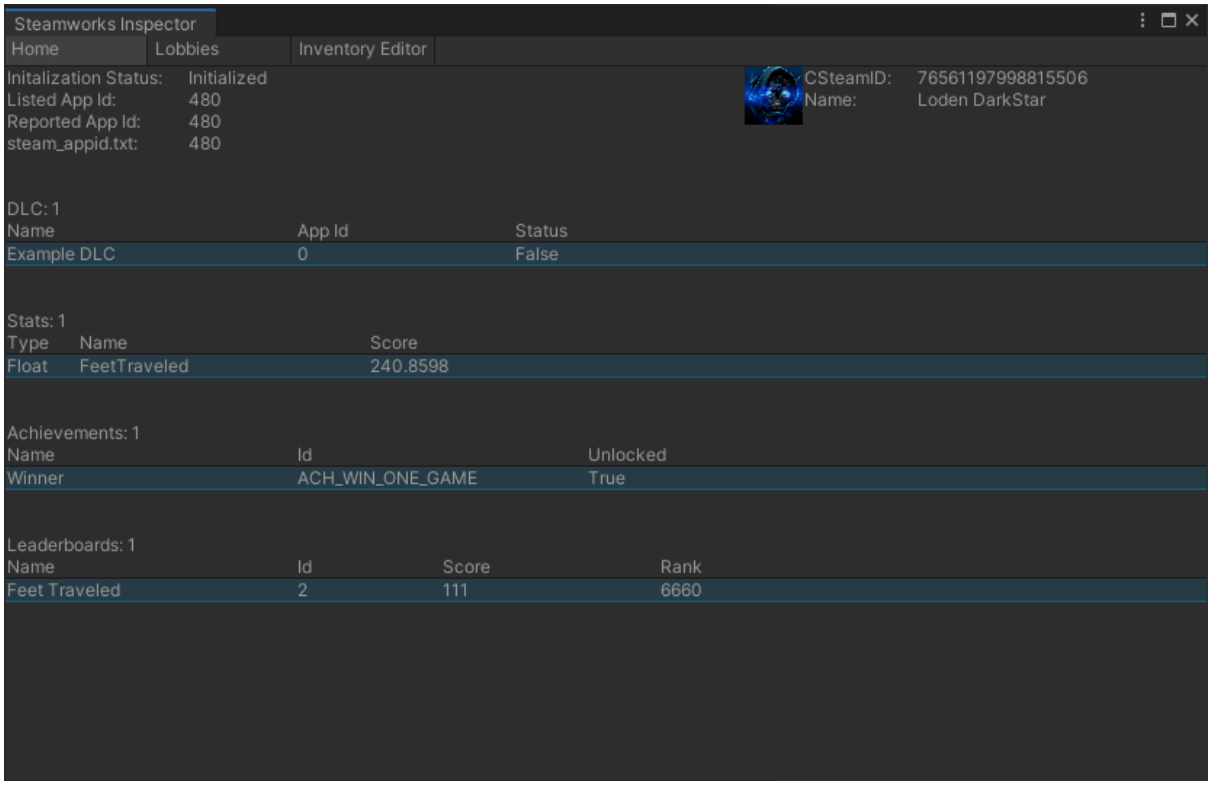
As with sending how you get the byte[] containing the data into your system from the source is up to your networking tools.

When you do receive the byte[] you should call the manager's PlayVoiceData passing the byte[] buffer into its parameter. This will handle decompression and preparation as an audio clip and feeding that data into the configured Audi Source for play back.

Tools and Procedures

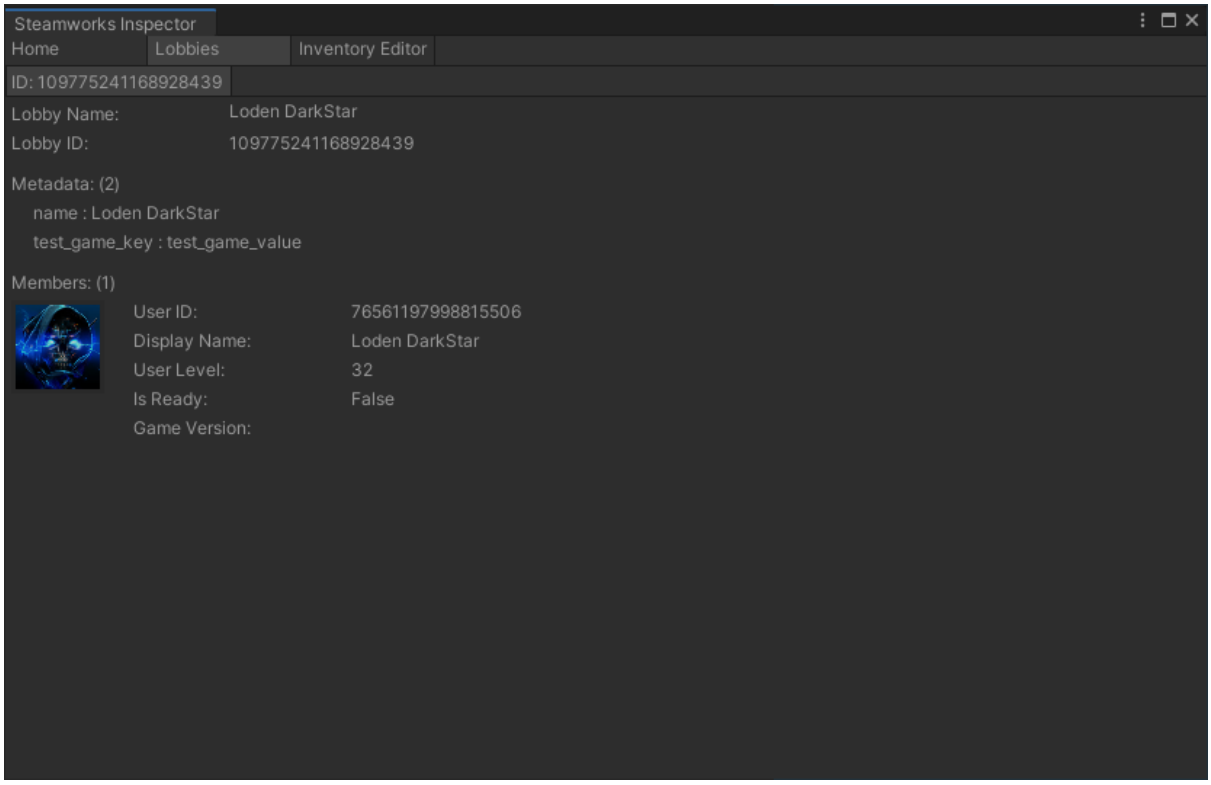
Steamworks Inspector

This is a custom editor tool accessible from the main menu under the Steamworks > Inspector option.

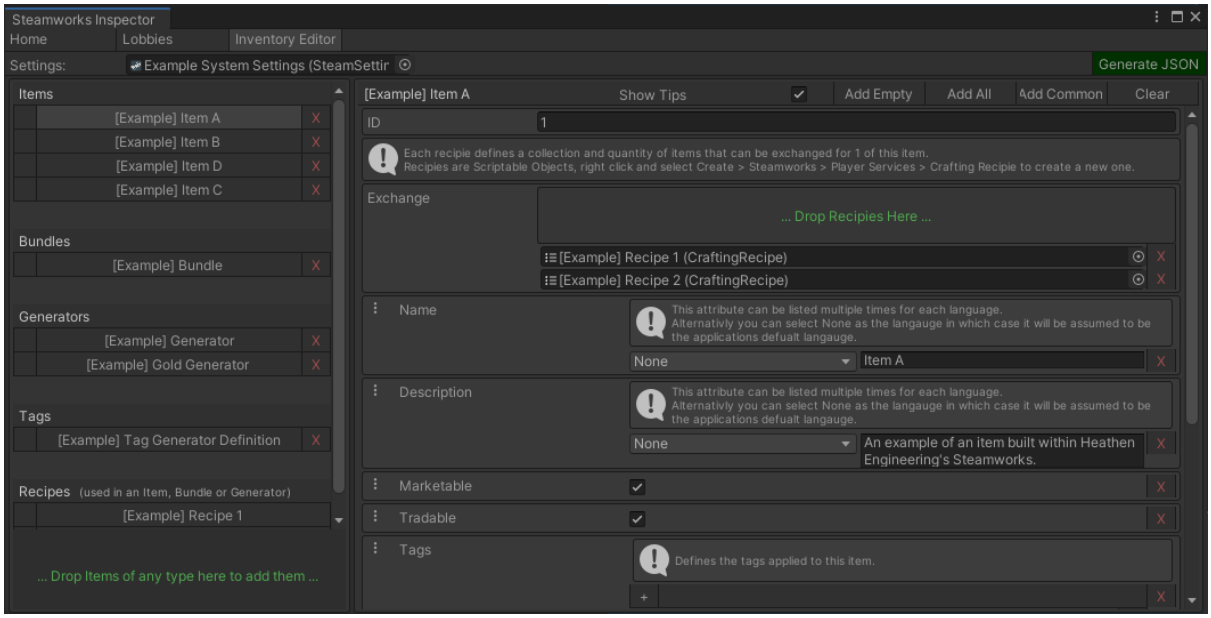


The Home screen of the Steamworks Inspector displays detailed data about the Steam API as it is initialized. In the upper left corner will display the App ID as seen from your configured Steam Settings object, as reported by the Steam API and as listed in the steam_appid.txt file. To the right of the screen is the user data for the local user.

4 sections follow the header data including DLC, Stats, Achievements and Leaderboards. In each row the inspector will list the configured objects of that type as well as the stats and values reported by the Steam API at that time.



The Lobbies tab will display each lobby the local user is a member of along with details about he lobby including all the metadata found on the lobby and each member's user data found on the lobby.



The Inventory Editor tab helps you configure the item definition JSON file required by the Steamworks developer portal. The tool displays the items you have defined and attached to your SteamSettings object and exposes the elements configured for each item. You can click the Generate JSON button in the upper right of the screen when it is green to generate the JSON file. Note this tool will help identify common errors in your design but cant catch all conditions, you may need to edit your JSON file manually to adjust or add features as required by Valve.

Code Documentation

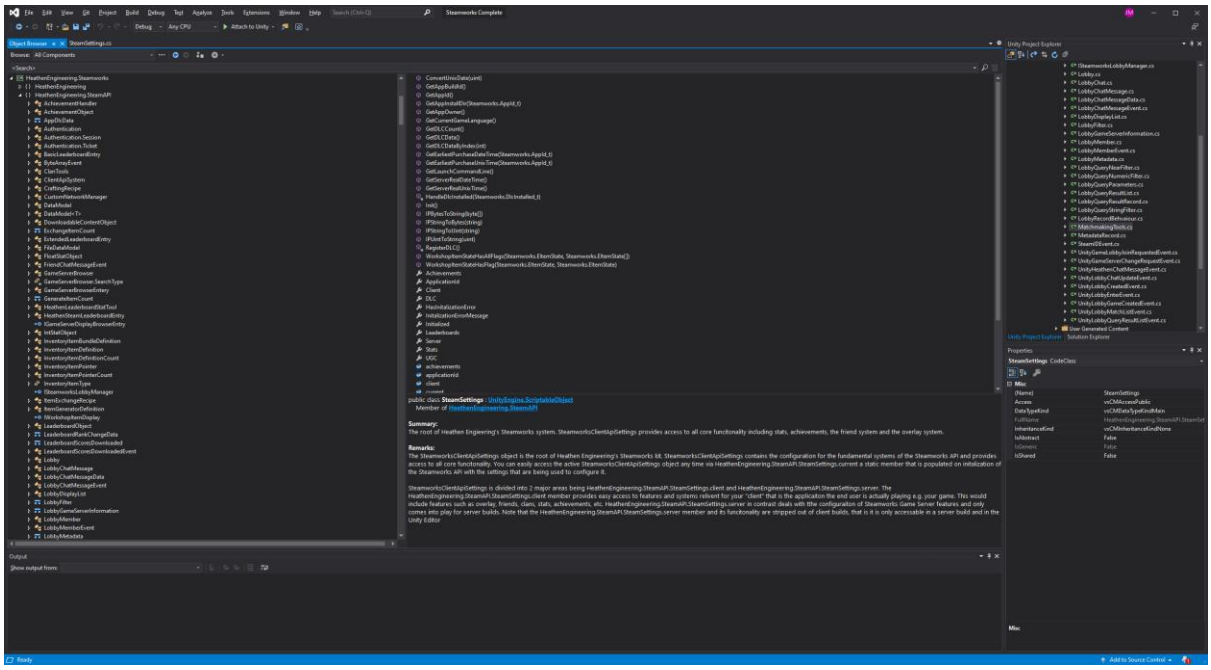
This is not a Heathen Tool but a valuable tool all the same and one we support with the way we have structured our asset. To use the tool simply look in your IDE (Visual Studio is recommended and free and has a powerful Unity plugin).

The simplest to way to for example browses the code documentation is to open up the Object Browser. You can do this by clicking in the main menu View > Object Browser … the default hot key is Ctrl+W then press J

Next choose the assembly you want to browse for Steamworks its name is HeathenEngineeing.Steamworks

Now you can expand each namespace and browse through all of the objects, selecting an object will show you the code documentation for it e.g., our remarks and summary as well as every member within it … clicking one of the members such as a function or variable will show the code documentation for it.

There is a tremendous amount of valuable information here that can help you understand the structure of the code you are working on and the remarks can add a bit more context to each.



Secondly you can use code documentation while writing your own scripts. Visual Studio includes an "auto complete" like feature called intellisense. It will predict what you are looking for based on context and what you have typed so far and show you options, you can mouse over options to see the remarks and summary for each option. This is useful for reading the remarks on each parameter to a function you are calling so you can see what was intended to be used in that function.

We are working on extending every object and every object's member with this additional information as means to help you learn more without needing to search a web site, document or ask on a forum or chat.

Terminology

steam_appid.txt

a simple text file containing only your app id. This file is used when you want to initialize the Steam API from outside of the Steam client … such as simulating in the Unity Editor. The file must be in the "working directory" of the game so in the case of the Untiy Editor it must be in the project root beside the Assets folder not within it.

You would also use this for server builds along with SteamCMD, if you do not have the steam_appid.txt located in the root of a builds folder and it is not launched from within Steam then the Steam API will close the program and attempt to relaunch it from the Steam Client.

Remember any time you change this in the Unity Editor you need restart the editor and visual studio before it will take effect.

Note V2 automatically edits this file for your project when you change the App ID in our settings object.

App Id

A simple number issued to you by Valve. Many things get an app id but the most common is your game itself. When you register as a Steam Developer and pay your fees Valve will issue you an app ID … use that in your settings and in your steam_appid.txt file.

CSteam Id

A simple number as type ulong issued by Valve for each user, lobby, and various other uniquely identifiable objects.

Lobby

Many networking APIs have the term lobby and what they really mean is a "room" or simple/general server connection that acts like a chat room or gathering point. The main difference between Steam's lobby and these rooms is that Steam Lobby is not a network connection, it does not require a server and has nothing at all to do with networking. You can and players often do join lobbies in single player games using lobbies as a sort of "party" system.

Lobbies have many purposes, but the main purpose is to gather players together and help them communicate so that they may agree upon the conditions of a game session and then start said session. So, this occurs before a player has connected to any network backend (as a rule). Classically players would leave the lobby once they had connected to a play session as the features of the lobby at that point are superseded by the features of a network connection.

Callback

In the context of this document a callback is an Action that is you can pass in a method that takes the required parameters and it will be called back at the appropriate time. For example if the method Foo(Action<int> callback) was called then you could pass in a method that takes an int as a parameter … such a method might look like

```
private void HandleFoo(int value) {}
```

so, your call would be

Foo(HandleFoo);

This is a common approach to notify a caller when an asynchronous task has completed and is like using a Unity Event only the callback is specific to that call. Typically, Heathen also invokes a Unity Event so you can use either model (Event or Callback)

Common Use-cases

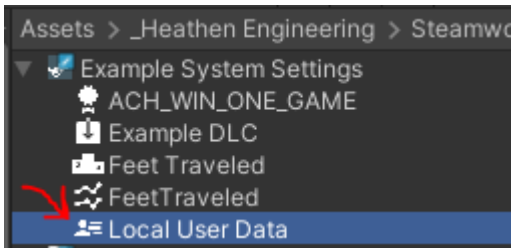
Get user information

This process can be applied to getting the user as in your game's player's name, icon/avatar or other Steam persona information.

You have a number of ways of getting at this information. As far as best practices each method has its strengths and weaknesses, for most projects the difference from one to another is negligible e.g. it's a matter of preference as to which you use.

UserData Reference

The SteamSettings object your running as e.g. the one you referenced on your Client Api System has an object under it called "Local User Data"



This object's data type is UserData so you can create a reference to it in any of your scripts by simply typing

public UserData localUserData;

You can now access the user's name, avatar and more via localUserData.DisplayName for example.

SteamSettings Reference

The SteamSettings object your running … as noted above … the one you referenced on your Client Api System has a reference to the UserData object under it with it its members.

public SteamSettings settings;

you can now access the user's name and more via settings.client.user.DisplayName for example.

Static Reference

When your Client Api System initializes the SteamSettings object you referenced on it, it will have set a static variable named current with the value of the settings it use to initialize …

This is accessed by fields such as Client which are static themselves. So you can access the data with no reference at all via

In your scripts you can simply type:

SteamSettings.Client.user.DisplayName for example

Unlock an Achievement

As with most things in Steam there are multiple ways to accomplish this. Each has its advantages and disadvantages in some cases but for most projects the difference from one to the other is negligible.

NOTE: if your achievement is set to trusted then it must be set by the Web API from a server that is using a developer token … see https://partner.steamgames.com/doc/webapi/ISteamUserStats#SetUserStatsForGame for more information

For achievements that are set to allow the client to unlock them see the following options.

ArchievmentObject Reference

AchievementsObjects located under your SteamSettings object can be referenced in your scripts for example.

public AchievementObject myAchivement;

once you have done this you can unlock the achievement by

myAchievement.Unlock();

SteamSettings Reference

The SteamSettings object your running … as noted above … the one you referenced on your Client Api System has a reference to the achievement object under it with it its members.

public SteamSettings settings;

You can now access the achievements via

```
foreach(var achievement in settings.achievements)
  //DO WORK
```

The achievements member is a List of AchievementObject, if you know the object's index or its name then you can look it up as you would any list member.

Static Reference

SteamSettings.Achievments is a reference to the current SteamSettings object list of AchievmentObjects.

Set the value of a Stat

As with most things in Steam there are multiple ways to accomplish this. Each has its advantages and disadvantages in some cases but for most projects the difference from one to the other is negligible.

NOTE: if your stat is set to trusted then it must be set by the Web API from a server that is using a developer token … see https://partner.steamgames.com/doc/webapi/ISteamUserStats#SetUserStatsForGame for more information

For stats that are set to allow the client to set them see the following options.

StatObject Reference

StatObjects located under your SteamSettings object can be referenced in your scripts for example.

```
public StatObject myStat;
```

once you have done this you can unlock the achievement by

```
myStat.SetIntStat(value);
or
myStat.SetFloatStat(value);
```

depending on the data type of the stat

SteamSettings Reference

The SteamSettings object your running … as noted above … the one you referenced on your Client Api System has a reference to the stat objects under it with it its members.

```
public SteamSettings settings;
```

You can now access the achievements via

```
foreach(var achievement in settings.stats)
  //DO WORK
```

The stats member is a List of StatObject, if you know the object's index or its name then you can look it up as you would any list member.

Static Reference

SteamSettings.Stats is a reference to the current SteamSettings object list of StatObjects.

**Set leaderboard score and data (client)

Leaderboard's work a lot like stats but have a few additional features. To simply set the user's score on a leaderboard you can use the leaderboard object as it appears in your inspector. Typically you would reference this on whatever behaviour component (script) you plan on having drive the score.

public LeaderboardObject leaderboard;

Once done you can simply drag and drop the board to that field in the Unity inspector.

Note that you can set leaderboards to be writable by the client API or only by a trusted server. This breakdown assumes the leaderboard is writable by the client API. This does mean anyone can write to the board so do expect cheaters to write to the board at some point.

leaderboard.UploadScore(value, method);

That is all that is required to upload the score. It is however not uncommon at all for people to not see the expected score appear. The most common reason is the method you passed in. in most cases you will use "keep best" and its very common that you have your board sorting the wrong direction. To test if your set up is correct we recommend submitting a non-0 value such as 10. Then submitting 9 and then again 11, now observer the board, which value was keep, does that match your expectation? If not change the sort ordering on the board.

Score Details

Next we will cover uploading a score and additional data to the board. This additional data can be used by your game client to do all sorts of things such as indicating what character or build the player used when earning the score. The simplest approach to this is to configure the board to use detail entries. This can be done in the inspector or via code … e.g.

leaderboard.maxDetailEntries = 1;

This would indicate that this board will have 1 detail entry per player entry. A "detail" is simply an integer and is set or returned as an int[]. You can then pack data into this integer as you see fit.

As an extra here is an example of tightly packing a lot of data into a single integer. Lets pack our player level, class and score into a single value.

```
int detail = 0;
byte[] bytes = BitConvertter.GetBytes(detail);


//a 32 bit int has 4 bytes so you can pack 4 8bit aka byte values here more than enough for most data
bytes[0] = level; //as long as max level is < 254
bytes[1] = class; //as long as you have less than 254 classes
```

we can even pack bigger data in here like a ushort (max value 65,535) often large enough for score or time, etc.

```
byte[] scoreBytes = BitConvertter.GetBytes[playerScore]; //assumes playerScore is a ushort
bytes[2] = scoreBytes[0];
bytes[3] = scoreBytes[1];
```

Now we convert it back to an integer for upload

```
detail = BitConverter.ToInt32(bytes);
leaderboard.UploadScore(value, new int[] { detail }, method);
```

UGC Attachment

The second method for applying additional details to an entry is to use the AttachLeaderboardUGC option. This can be used in conjunction with Score Details. This method will store a file to the local user's remote storage, mark the file as shared and attach a handle to it on the entry.

Heathen handles all the steps for you in a single call

```
leaderboard.AttachLeaderboardUGC(filename, object, callback);
```

- filename
  This is the name of the file on the user's remote storage, this should be unique
- object
  This is any JsonUtility serializable object, we will serialize this for you and save it to the users remote storage
- callback
  An optional callback, this will get invoked when the process is complete

Reading this data is just as simple, Heathen handles the multiple steps required for you on the ExtendedLeaderboardEntry. Note that the local user's entry is always keep up to date for you on the leaderboard itself. E.g.

```
leaderbaord.userEntry.GetAttachedUgc<T>(callback);
```

- T should be the JsonUtility serializable type used to save the attachment
- callback is technically optional but is how you get the results, so you want to set this to a value.

The callback is of the form Action<T result, bool error>
If any error occurs result will be the default value for that type and bool will be true.

Start purchase of an item (MTX/Inventory)

For any Steam Inventory Item that has been correctly configured to be sold on the Steam store you can start a purchase process from within the game. Note that this does require that pricing be configured on the item such that the item can show in the steam Store.

Steam Settings method

SteamSettings.current.client.inventory.StartPurchase(itemsToPurchase);

The client object within Steam Settings contains an inventory object this is where all of your inventory functions can be found. The StartPurchase method takes a collection of InventoryItemDefinitionCount … this simply indicates what item and how many of that item. Assuming all items are correctly configured the Steam Overlay will be opened to the users cart.

This method similar to having the user create a basket or cart and then sending that collection of goods to be checked out.

Item Reference method

Item Definitions and Bundle Definitions contains a StartPurchase method which will start a purchase for that specific item or bundle. This method is more akin to a “buy now” button.

Getting the results

There is no direct method know that the items which started the purchase where actually purchased. This is because a purchase can be started from out side the game and purchased started in the game can be modified before they are purchased.

Indirectly though the SteamSettings.current.client.inventory.evtItemInstancesUpdated event is raised when any data changes with items including when a sale is completed and item quantities updated. You can use this event to know that something has changed and can then check the quantities on items or if you prefer a more rigorous check you can refresh the inventory on this event being raised to insure your in-memory state matches the current state of the Valve backend.

**Search for a lobby

Use the MatchmakingTools.FindLobbies(…) method.

The FindLobbies method takes a LobbyQyeryParameters filter permitter which is used to narrow the results of a search. The members of that object are listed below.

- useDistanceFilter
  This indicates that the query should use the distance option you specify in distanceOption
- distanceOption
  How far out should the query search
- useSlotsAvailable
  This indicates that the query should use the slots available option you specify in the requiredOpenSlots
- requiredOpenSlots
  This is how open slots you need in a lobby … useful if you're looking to join a group up to this lobby
- maxResults
  What's the maximum number of lobbies you want returned
- nearValues
  These are numeric values you want to keep near but don't require to be a precise match
- numberValues
  These are integer values and the match method you want to use on them
- stringValues
  These are string values and the match method you want to use on them

FindLobbies will raise the MatchmakingTools.evtLobbyMatchList event with its results or alternatively you can pass in a callback on the final paramiter.

Callback is a delegate (pointer to a method/function) the signature is as such

Action<LobbyQueryResultList results, bool bIOFailure>

Meaning the first parameter is of type LobbyqueryResultList and the second is a Boolean indicating failure e.g. if the bool is true an error occurred.

You can pass a callback in either using expression or by defining the method traditionally and passing it. E.g.

```
public void Foo(LobbyQueryResultList results, bool bIOFailure)
{
    //Write your code here to run when the process comletes
}

and then
MatchmakingTools.FindLobbies(filter, Foo);

or using expression
MatchmakingTools.FindLobbies(filter, (results, bIOFailure) =>
{
  //Write your code here to run when the process comletes
});
```

**Update or read Lobby Metadata

You can easily set or get the metadata of the lobby via a few methods. The simplest is to use the indexer. To do so you need to get the lobby you want to read data for … so for example

string value = MatchmakingTools.normalLobby[“fieldname”];

normal lobby would be any public, private or friends only lobby you have joined. Valve only allows a player to be a member of 1 such lobby so the multiple lobby concept doesn’t apply to them.

Setting the value works the same way

MatchmakingTools.normalLobby[“fieldname”] = “Hello World”;

Note that we do not need to create the field nor test if it exists. Steam will always return a string when asked fore a field, an empty string indicates no value. Similarly it will create or update an existing field as required.

There are occasions when you want to iterate over fields you can do this via the GetMetadataEntries method e.g.

Dictionary<string, string> metadata = MatchmakingTools.normalLobby.GetMetadataEntries();

You can then use the dictionary how you please. Note the dictionary is a copy of the values at that time it will not update automatically and setting values in it will not update the lobby. To set values you muse use the indexer as shown above and to get the updates you must call GetMetadataEntries again.

**Update or read Lobby Member Metadata

This process is very similar to updating or reading lobby metadata in that we have provided a simple indexer to access the information. The main task is then selecting which member you want to read metadata from. Note that you can only assign metadata for the local user.

We have provided a quick tool for fetching the local user's member as well as fetching the owner's member.

var user = MatchmakingTools.normalLobby.User;

var owner = MatchmakingTools.normalLobby.Owner;

Please note that these fields will search for the appropriate LobbyMember when called so if your making multiple calls to these members you should cache the value as shown above and use the cashed reference e.g. user or owner as shown above.

Setting the metadata on the user then is

user[“fieldname”] = “Hello World”;

and reading data is

string value = user[“fieldname”];