gamedev-wunder9l.blogspot.com

December, 2019

Overview of different serialization approaches

Seems every game development faces a problem: how to serialize and deserialize data? This question occurs in two main scenarios: saving/loading and network communication (how to send data to server or computer). In both cases you want to save some game data (items, players or perhaps game state or action) and restore this data later (might be on other server).

Most common solutions

What approaches does Unity suggest for that? I have found these variants:

- PlayerPrefs (link);
- String serializers (JSON or XML);
- Binary serilization;

Let me take short overview of each of them.

PlayerPrefs

It is one of the most simple way to store data in Unity. It uses built-in system PlayerPrefs that allows to save data in system key-value storage. It has only three functions to store base types of data (int, float and string). PlayerPrefs is not about performance or data transferring. Pros and cons:
+ Simple to use
+ Built-in
- Does not support data transferring
- Does not support complicated data types (as classes)
- Pure performance

String serialization (json, xml)

String serialization has an advantage over other methods: it can be easily read by human. So you can see how does your data serialize. Unity supports several helpers to work with these format (XMLSerializer is Microsoft's wrapper, but Unity supports it). You can combine JSON/XML with previous PlayerPrefs to store object (serialize your object and store it as string). But unfortunately these serializers have bad performance and serialized data has a large size. Pros and cons:
+ Simple to use
+ Human-readable formats
+ Support complicated object serialization
+ Open standards that have a lot of realizations (cross-language support)
- Large serialized size
- Pure performance

Binary serialization

Using of binary serialization allows to avoid problems with performance and large size of serialized data. It is also easy to use (similar to json serializing). There is a class BinaryFormatter that helps to serialize data. But unfortunately it does not provide cross-language support. It means that you are limited to C#

and if your game requires for a example a server you have to write a server on C#. If it is not a problem binary serialization is a good choice. Pros and cons:
+ Simple to use
+ Well performance (in comparison with previous approaches)
+ Support complicated object serialization
+ Small size of serialized data (in comparison with string serialization)
- No multi-language support

Other ways...

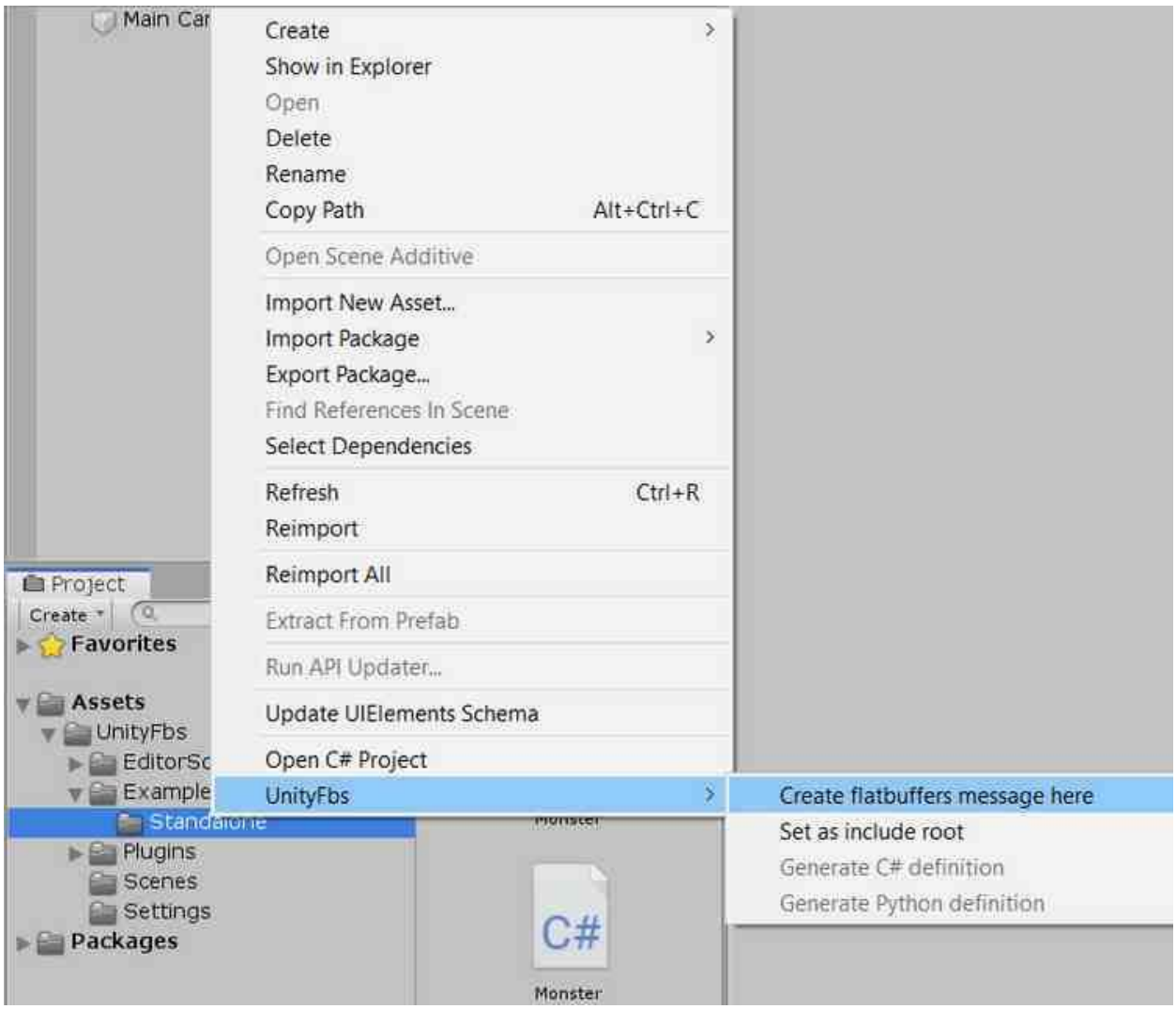
Well... what if none of mentioned variants suits you. What can you do else? Google introduced their approaches to data exchange with messages. There are two protocols developed by Google: protobuf and flatbuffers. Flatbuffers is more later protocol and it has some advantages over protobuf. Both these protocols are much more efficient than previously mentioned approaches. You can find performance tests on this page.

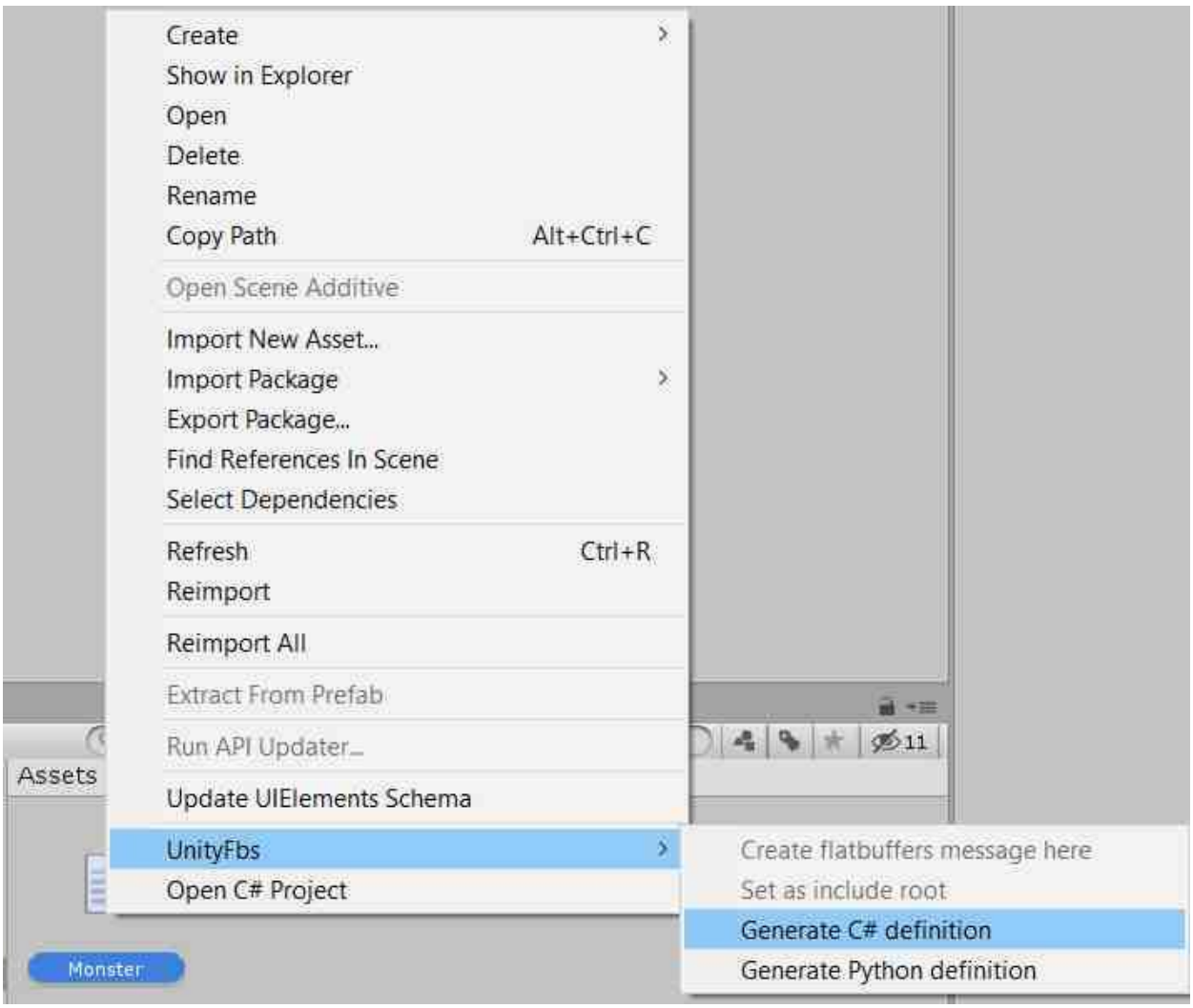
To my astonishment, there is pure support of these protocols in Unity. You can find some articles about using protobuf in Unity but mostly they use old-built release version of protobuf-net and it is frustrating. Also it is not as convenient as previously mentioned approaches.

About flatbuffers I have found one good article (this one) but the problem with using flatbuffers still stays on. For example Unity Editor does not support creating flatbuffers-files (*.fbs). You have to create them in other ways (for example from console or by Windows Explorer). Another inconvenience is that for compiling (translating from .fbs messages to C# or other language code) you have to run a console app (flatc) with a number of arguments. For example typical command to compile Message.fbs to Message.cs with selecting include directory is:

I have decided to use flatbuffers in my project because of these advantages:
+ Great performance
+ Small size of serialized data
+ Cross-language support

To make my life easier I have decided to develop a tool/plugin for Unity to facilitate using flatbuffers. Next part of this article contains detail overview of this tool (I called it UnityFbs). A few screenshots how it looks:


Main Car
Create
Show in Explorer
Open
Delete
Rename
Copy Path Alt+Ctrl+C
Open Scene Additive
Import New Asset...
Import Package
Export Package...
Find References In Scene
Select Dependencies
Refresh Ctrl+R
Reimport
Reimport All
Extract From Prefab
Run API Updater...
Update UIElements Schema
Open C# Project
UnityFbs
Create flatbuffers message here
Set as include root
Generate C# definition
Generate Python definition
Project
Create
Favorites
Assets
UnityFbs
EditorSc
Example
Standalone
Plugins
Scenes
Settings
Packages
C#
Monster


Create
Show in Explorer
Open
Delete
Rename
Copy Path Alt+Ctrl+C
Open Scene Additive
Import New Asset...
Import Package
Export Package...
Find References In Scene
Select Dependencies
Refresh Ctrl+R
Reimport
Reimport All
Extract From Prefab
Run API Updater...
Update UIElements Schema
UnityFbs
Open C# Project
Create flatbuffers message here
Set as include root
Generate C# definition
Generate Python definition
Assets
Monster

FlatBuffers for Unity

In previous part we have considered different ways to serialize and deserialize data. In this chapter I tell about tool that facilitates using of flatbuffers in Unity.

What should it do? I have highlighted three main points. First of all it should enable to **create .fbs messages** from Unity Editor. Just as right-click action. Next, it should allow to **compile .fbs messages to C# classes** (and some other languages). And finally, it should be **flexible** and allow user to tune tool for his needs.

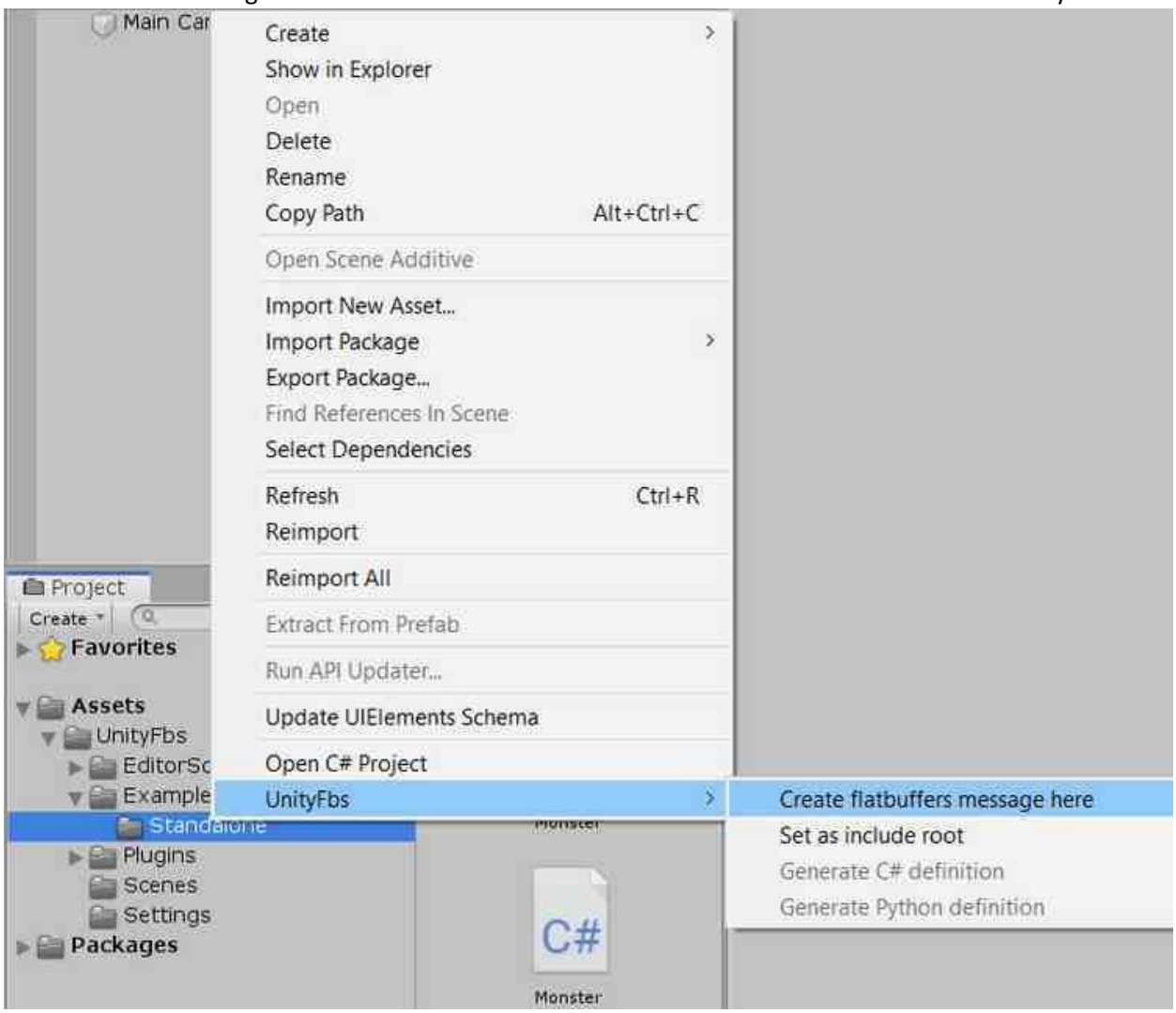
Creating .fbs messages

This point is quite easy: we need to add a new menu item "Create .fbs" to Assets menu. You can do it easily by adding a MenuItem attribute to your method. Additionally you can add a validation method that enables/disables menu item. I used this opportunity to check that there is a selected folder (when we need to create a new file). More information you can find here.

```
[MenuItem("Assets/Create/FlatBuffers message")]
[MenuItem("Assets/UnityFbs/Create flatbuffers message here")]
static void CreateFbsMessage() {
CreateFbsFile();
}

[MenuItem("Assets/UnityFbs/Create flatbuffers message here", true)]
static bool CreateFbsMessageValidation() {
return GetSelectedDirectory() != null;
}
```

I think there is nothing else worth to be mentioned. Just take a look at how it looks in the Unity Editor:



Compiling .fbs-message to C#

That is one of the most interesting part. How should it work? Do we need to download a compiler sources and integrate it into Unity? What should we do on new release? How can we provide the user with latest update?

Facing these questions I have chosen another approach. I have decided to use a compiled binary file (.exe for Windows) and allow user to provide a path to binary to use (it will be discussed in chapter about flexibility).

Of cause this approach also has some disadvantages (for example delegating work to external executable file is not brilliant thought) but it is easy to implement, easy to configure, does not require to integrate to existing code, code has small size (especially as it is executed as Unity Editor plugin - it should compile fast), and it performs well.

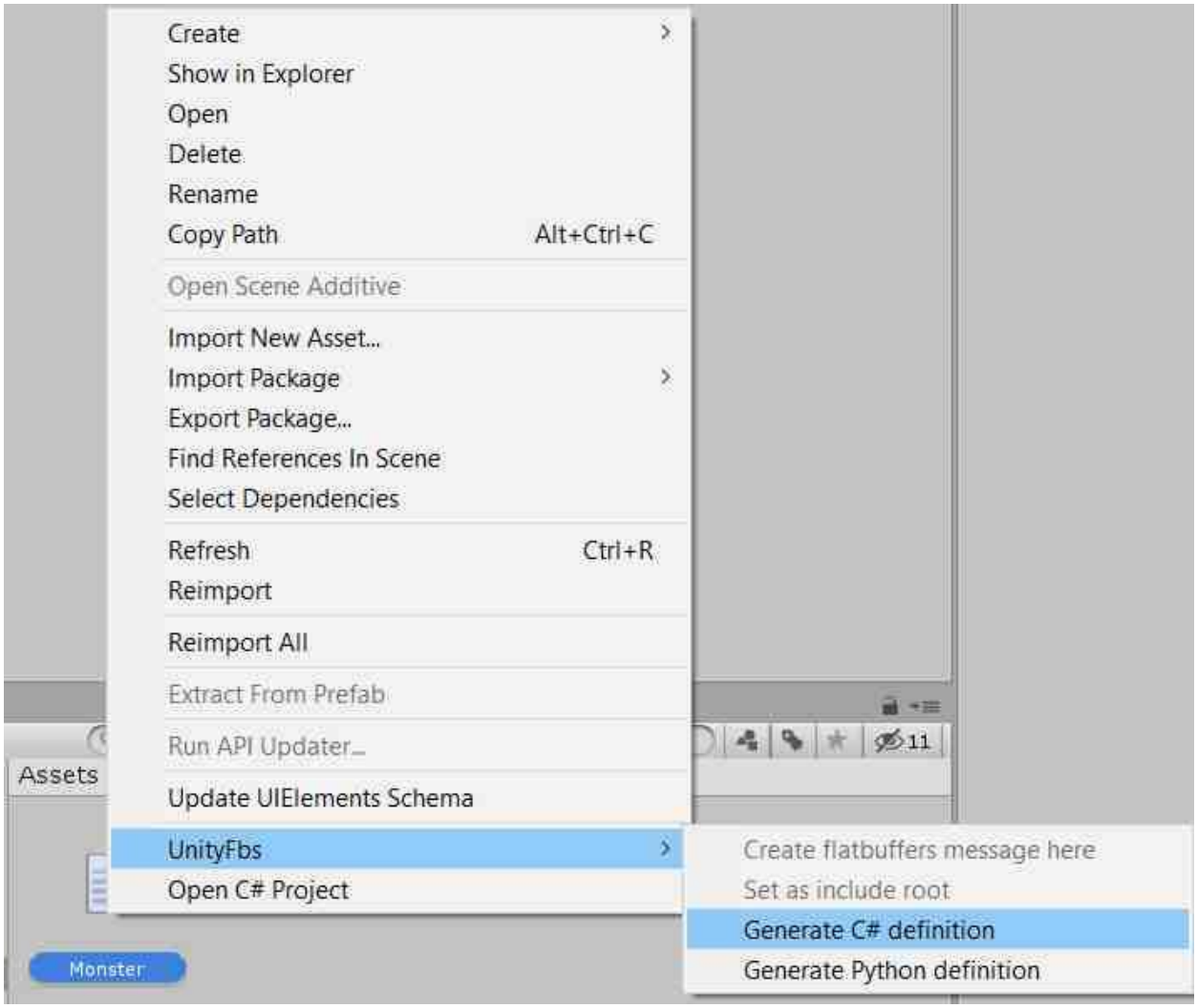
So the plan is:

1. On right-button click on file check whether this file or files (you can select multiple files) could be compiled (check its extension it should be .fbs)
2. Suggest to compile to different languages (C# at least)
3. Call a compiler program with arguments defined at previous step (and additional user-defined arguments if there are)
4. Profit!

First and second step could be easily done with the same method as previously:

```
[MenuItem("Assets/UnityFbs/Generate C# definition")]
private static void UnityFbsGenerateCSDefinition() {
GenerateDefinition(GeneratedOutputEnum.cs);
}

[MenuItem("Assets/UnityFbs/Generate C# definition", true)]
private static bool UnityFbsGenerateCSDefinitionValidation() {
return CheckFbsMessageSelected();
}
```



Seems there is no other variant than to add same methods for each supporting languages. But all these

methods are calling internal method GenerateDefinition that creates class ExeRunner and pass to it files to be compiled and target language.

ExeRunner itself gets additional arguments for compiler depending on selected target. These arguments are taken from Settings that can be changed by user (how user can set these arguments will be discussed later). ExeRunner puts together all arguments, compiler path and files need to be compiled and form a command which is running as a new process:

```
string arguments = MakeArguments(inputFiles, output);
var process = new Process {
StartInfo = new ProcessStartInfo {
FileName = Path.GetFullPath(flatcPath),
Arguments = MakeArguments(inputFiles, output),
UseShellExecute = false,
RedirectStandardOutput = true,
CreateNoWindow = true
}
};
var started = process.Start();
```

Flexibility

Flatbuffers compiler has various parameters. You can find list of its parameters here. Compiler is named **flatc** and further I also will use this term. As you can see there are a lot of options. Some of them make sense only for specific language (for example

```
--gen-onefilemeans generating single output file for C# and Go).
```

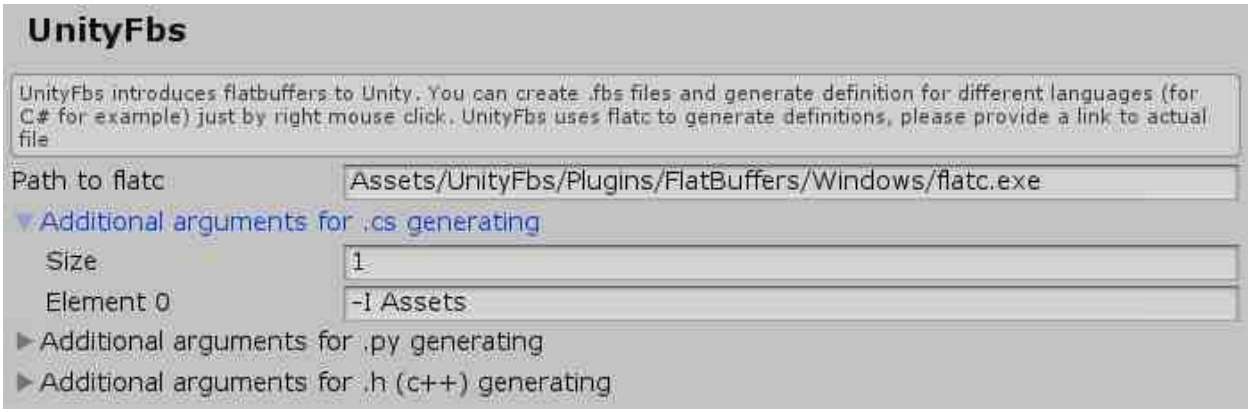
To make my solution flexible I have decided to make possible at least:

1. choose a compiler (set path of flatc)
2. set additional arguments for each supported language
3. change these settings user-friendly

First point is very important because of two aspects. First of all, **security**: you may not trust me and suspect me of being spoofed an original flatc by a malicious file. The second point is supporting new releases of compiler: you just need to download and compile new flatc file instead of previous one. Of cause for convenience reasons I have added compiler for different platform (Windows, MacOS and Linux) to package but it fine if you will your binary, (you can find them in archive FlatcCompillersForPlatforms.zip.
Nevertheless how you get your flatc-compiler you have to set path to it in settings (just click right button on compiler file and select “UnityFbs->Set as flatc-compiler file”).

I think that it is discussed enough about user-defined arguments for compiler. So, let jump to the third point.

How can we allow user to change Settings of plugin (and thus change behaviour of plugin in Editor)? It became clear to me that such settings have to be somewhere in main menu: **Edit** > **Project Setting** (read more). And Unity gives us that opportunity with SettingsProvider! It remains only to use this tool:



One small feature...

Compiler can take an inlude directory path as one of argument. This is path to folder where included .fbs messages stored (you can read about include more detaily on flatbuffers page). In short, if you include into your .fbs messages other .fbs messages (for example warrior.fbs includes weapon.fbs) you have to inform compiler where to find these includes files.

I have thought that it would be convenient to set include directory just as right-click on folder. For example:

You also can choose a flatc-compiler with the same way

You can find sources on github: https://github.com/Wunder9l/UnityFbs/

Usage example

There is an example of PingPong game. It is not about fascinating gameplay it is just demonstration of flatbuffers usage. First of all, there are three .fbs-files (BallCoordinates, MoveDirection and PlayerAction) that we need to compile into c# and python (.cs and .py respectively). Do not forget that you can choose a flatc-compiler (although there is default one for each of platforms: Windows, MacOS, Linux).

After compiling you have to move python files into server-part folder. (Scripts->Server->FbsCompilled).
It is time to prepare server:

1. Install python and pip
    1. Optionally you can create python environment:

```
python -m venv /path/to/new/virtual/environment
```

2. Install requirements from file "requirements.txt" (with the help of pip):

```
pip install -r requirements.txt
```

3. Run flask server with the help of script run_server.bat. There should be output:Pay attention to string "Running on http://127.0.0.1:5000/". It is default port and you server will serve only on localhost. In example this string is hardcoded (NetworkPlayer::host).

```
 * Serving Flask app "server.py"
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

After starting server you will be able to play with it in Play mode. Just click start. You can see that every message from server is logged in Debug console.

That's it. I have added FbsSerializer for message serializing/deserializing.

At StartGame Unity's client send a coordinates of ball (relative to computer-player's board) and its move direction. On server-side handler tries to calculate where the board should be and sends an action command back to client.

Thank you for your time!