Please Initially read the “Short Description to Git_serialization.pdf”, this document addendum to it.

Scene “1. ByOdinClassicEditorOnly”

Use the standard functionality of the Odin Serializer, which give a possibility to store objects like a “GameSettings” in Editor only (t.1 & t.2 from aims), because for resolving and storage reference use the “ReferenceResolver classes” (IExternalStringReferenceResolver or IExternalGuidReferenceResolver) which based on the use of methods from AssetDatabase class (namespace UnityEditor).

Serialization did in “two steps”:

1. Serialization the “plain part(members)” of the initial object to JSON, but the ref to assets stored in the internal collection (“List<UnityEngine.Object>”)
2. The item from collection (ref to unity assets) transformed by reference resolver to “plain data” (string or GuiD)

Summary: It may be useful for some custom Editor applications only.

Note. To OdinSerializer:

Exist two mains classed:

- SerializationUtility class to stored the C# plain class (objects derived from System.Object), and therefore I used methods from it in this Example
- UnitySerializationUtilityMod class - you should use it, if it is required to store a UnityObject (objects derived from UnityEngine.Object) like a ScriptableObject

I describe it more detail below.

Note. Common to ALL the next variants:

To have a possibility to resolve the reference to assets in Editor and in Build, I use the next **“common procedure”** (for store reference & restore object):

1. I create the “dictionary” Dictionary<key, value>, where the key is a string - for name of SO (for store reference), the value is Type of SO (to have possibility restore the ref to this SO).
2. All assets were put in special folder in Resource folder.
3. Before use for serialization, I Fill the “dictionary” by a command, like as: Resources.LoadAll<LevelSO> (“”).ToDictionary((level) => level.name, (level) => level);
4. “To store ref to SO” – I use simply get the name of the SO.
5. “To restore object from ref” – I use this dictionary (with some checks) to get ref to object from the reference.

All variants below differ only by methods to combine the serialization process with this **“common procedure”**. Some variants are more elegant (use additional generic class and so on) and give more possibilities, other simpler to realize, but harder to maintain.

All next variants were ordered by “complexity ascending” (demands more “wrapping code”).

Scene "2. BySpecialSO_Unity_Odin" ("simple to realize")

Use the special SO (like a GameSettingsSpecialSO) – This is simple SO, which was wrapped by special methods, which realized the ISerializationResoverRefUnityObject interface.

This interface is an analogue of the classic ISerializationCallbackReceiver Unity interface. The Unity interface will be called on all serialization's calls ("anywhere and anytime"), the proposed interface will use only when you store data. It gives a more control and less utilization of resources.

2.a) It can be used with **Unity Serializer** (w/o any use of Odin Serializer) to store reference:

```
if (data is ISerializationResoverRefUnityObject resoverRefUnityObject)
{
    resoverRefUnityObject.BeforeSerialize();
}
string strJSON = JsonUtility.ToJson(data);
```

"data" the stored object. And to restore:

```
JsonUtility.FromJsonOverwrite(str, data);
if (data is ISerializationResoverRefUnityObject resoverRefUnityObject)
{
    resoverRefUnityObject.AfterDeserialize ();
}
```

Note. When we work with objects derived from UnityEngine.Object we must use at restore the "FromJsonOverwrite" (or "analogue" of it), like in this Example.

To store the string with ref to SO I added special member to this SO:

```
[SerializeField,HideInInspector] private string _levelSOName;
```

The original member will also be serialized

```
[SerializeField] private LevelSO _levelPlayer;
```

but its value is not used (and will be "override" after deserialization)

2.b) It can be used with **Odin Serializer** (it this case it gives more possibilities to control Serialization):

I little extend the possibilities of **Odin** Serializer:

- I added the additional NonOdinSerialized attribute (Unity Serializer not react on it)

```
[SerializeField, NonOdinSerialized] private LevelSO _levelPlayer;
```

- And created the SpecialSerializationPolicy ("SpecialUnity") which can react on that attribute and skip members from serilization.

to store reference, I use:

```
SerializationContext myContext = new SerializationContext()
{
Config = new SerializationConfig() { SerializationPolicy =
SpecialSerializationPolicy.SpecialUnity }
};
if (data is ISerializationResoverRefUnityObject resoverRefUnityObject)
{
resoverRefUnityObject.BeforeSerialize();
}
UnitySerializationUtility.SerializeUnityObject(data, ref bytes, ref
unityReferences, FormatData, serializeUnityFields: true, context: myContext);
```

In this case Odin Serializer does not give any warnings and give the nicer to peoples JSON format, but this format non canonic and not compatible with standard Unity JSON.

Summary: In general the "result: use the Unity or Odin serializer will be the same: It gives a possibility to store & restore SO in Editor and in Build (all tasks completed), but you must all logic to support resolving reference put to the stored objects (it's not very elegant).

Note. Common to All to next variants:

To separate the logic which supported the resolving reference from data of stored object. I created the special generic class

```
UniversalScriptableObjectStringReferenceResolver<T> :
IExternalStringReferenceResolver where T : ScriptableObject
```

Which derived from the standard Odin interface – IExternalStringReferenceResolver

Where the T used as the Type of member of the stored object with ref to an asset (in my examples it will be the LevelSO or ComplexitySO).

Scene "3.ByOdinPlainClass" (nicer but not ideal)

It is work good but only with C# plain class with ref to assets (e.g. in my case it was a **GameSettings** objects), therefore I use the methods from SerializationUtility class, to store ref:

```
SerializationContext serContext = new SerializationContext()
{
StringReferenceResolver = new
UniversalScriptableObjectStringReferenceResolver<ComplexitySO>(),
};
byte[] bytes = SerializationUtility.SerializeValue(data, FormatData,
serContext);
File.WriteAllBytes(filePath, bytes);
```

And to restore object:

```
DeserializationContext desContext = new DeserializationContext()
{
StringReferenceResolver = new
UniversalScriptableObjectStringReferenceResolver<ComplexitySO>(),
};
byte[] bytes = File.ReadAllBytes(filePath);
return SerializationUtility.DeserializeValue<T>(bytes, FormatData, desContext);
```

remind the "SerializationUtility class" does not support the correct restore UnityObjects (deserialization of UnityObjects like "FromJsonOverwrite")

Summary: Therefore, it realizes all tasks, the "stored object" contain only data (except "testing code" in my example).

Scene "4.ByOdinUnityObject" (may be stop on that)

In this case as object to store was used the GameSettingsSimpleSO (SO with only contains the data w/o any special code to support serialization). Odin serializer, as I told early. give a possibility to serializer the UnityObject, for that exist the special methods in UnitySerializationUtility class. But initially this class was created to support serialization any

different UnityObject in Editor (Odin Inspector) include different Prefab mode and so on (It is a very hard task, and I didn't understand all aspects of that and limitation, but as you know the possibilities of Odin Inspector is Great).

Therefore "for stability and good management (and may be another reasons)" this class support only the resolvers of IExternalIndexReferenceResolver Type, which used to store the reference in the internal List<UnityObjects > collection only , which exist only in current session of Editor.

Note (from Odin doc for UnitySerializationUtility class) "Note that setting the IndexReferenceResolver on contexts passed into methods on this class will have no effect, as it will always be set to a UnityReferenceResolver."

So, I made my own variant of UnitySerializationUtility class and (**!!!**) I tested it to store the UnityObject with link to SO only (like a GameSettingsSimpleSO), other type reference to assets I did not test (**!!!**) (and not thought about it, all code under the Apache license and you can continue to make it better).

I made additional modifications to original Odin Serializer code (trying to minimize changes to original code):

- I created class UnitySerializationUtilityMod and put it in the OdinSerializer namespace and added to OdinSerializer Assembly definition.
- I created the analogue of methods SerializeUnityObject and DeserializeUnityObject from original of UnitySerializationUtility class
- The main difference of these methods, they doesn't use the UnityReferenceResolver and can work with custom "ReferenceResolver" (IExternalStringReferenceResolver or IExternalGuidReferenceResolver)
- All other left without changes except I was must to copy to my class the ICache GetCachedUnityWriter and GetCachedUnityReader from original class because they have a private access modificator and original class (UnitySerializationUtility) is static.

As a result, I can use to serialize the next code:

```
SerializationContext serContext = new SerializationContext()
{
StringReferenceResolver = new
UniversalScriptableObjectStringReferenceResolver<LevelSO>(),
};
byte[] bytes = default;
UnitySerializationUtilityMod.SerializeUnityObject(data, ref bytes, FormatData,
serializeUnityFields: true, context: serContext);
```

And for deserialize the next:

```
DeserializationContext desContext = new DeserializationContext()
{
StringReferenceResolver = new
UniversalScriptableObjectStringReferenceResolver<LevelSO>(),
};
UnitySerializationUtilityMod.DeserializeUnityObject(data, ref bytes,
FormatData, desContext);
```

Summary: It works like Variant 3 but can correctly deserialize the UnityObjects (like "FromJsonOverwrite").

Scene "5.ByOdinByPhaseCompexClass" (additional possibilities)

In this example for store, I use the GameSettingsComplex object, which contains data members and many (😉 only two) members with ref to asset. For simplicity was take the C# plain class and only two members (the number of members of that type can be any and the C# plain can be replaced by SO).

This variant uses the same two-phase serialization like in the Variant 1 (I didn't remember why I decided take that variant as base for this variant, not took the "one phase variant"), but main point of this Variant that for resolving reference to UnityObject it is use the chain of resolvers (like in Binary Serialization):

```
private static string FileName((string fileBase, string fileExt) fileName,
 int num) => fileName.fileBase + num + fileName.fileExt;

public static void SaveStrRef((string fileBase, string fileExt) fileName,
List<Object> unityReferences)
{
SerializationContext serContext = new SerializationContext()
        {
        StringReferenceResolver = new
        UniversalScriptableObjectStringReferenceResolver<ComplexitySO>()
                {NextResolver = new
                UniversalScriptableObjectStringReferenceResolver<LevelSO>()},
                };
        for (int i = 0; i < unityReferences.Count; i++)
        {
        byte[] bytes = Serialize(unityReferences[i], serContext);
        File.WriteAllBytes(FileName(fileName, i), bytes);
        }
}
```

Every resolved reference will be store in the separated file because the method on second phase must receive the List<Objects> from first phase. The Deserialization also must do in two phases in backward order: initially call the

```
public static void LoadStrRef((string fileBase, string fileExt) fileName,
List<Object> unityReferences, bool clearUnityReferences = true)
```

and only after that the call the

```
        _gameSettingsComplex = OdinCallsExample.LoadTwoSteps <GameSettingsComplex>
(FilePathMainFile, unityReferences);
```

To combine the restored ref with other members in one object.

In this example the serialized data will be consist of 3 files (one main file from first phase and two from second phase).

Files:

The install.log describe the process of the installation of the environment, OdinSerializer and make a Build (the main steps and warning which existed).

Folders (in Scripts):

- ObjectsToStore – scripts the stored classes
- OdinModifications – modification to original code of Odin Serializer

- Serialization – special classes to work with Odin Serializer, contains the main class – “OdinCallsExample.cs”. which contains all methods for Serialization
- SO – scripts the stored SO classes and SO as a “reference SO”
- Test – scripts to support the interfaces in all Test Scenes, which call the methods from “OdinCallsExample.cs”.