Pantagruel

Guide To Using the XML Serializer

Introduction

Lets get two things out of the way immediately.

1) Serialization is complex and rarely will things *just work* without taking some time to review and test your code.

2) People that write technical guides are in the habit of talkng to themselves. They need to be in order to predict all the zany questions that will pop up in the readers' heads.

Given those two facts this guide will often take the form of a F.A.Q. rather than a dry and boring old list of specs. The great thing about it is that I get to decide what *you* will be saying. In the event that you still find questions that I haven't managed to pull from the aether of thought, please feel free to send any additional questions and concers to pgi-support@ancientcraftgames.com

The XML serializer packaged with Pantagruel was designed with the idea that you just throw your data at it and it will magically figure out what to do and later when you want it back it will just as magically barf it all back out just as you left it. The reality is, however, that as a programmer you still need to be aware of what is actually happening and take extra steps, precautions, and responsibilities when designing your data structures to ensure everything works as it should. Nothing is truly free or magical. This guide will explain what the serializer is doing to your precious data and what steps you will need to take ensure nothing unexpected happens to it. If you are generally unfamiliar with the idea of serialization, what it does, and what kinds of attributes can be used to control it it you may want to take some time reading up on it at msdn. If you are very at home with serialization and just need a cheat sheet then feel free to skip the the Appendix at the end for a dry and boring old list of specs.

Modes of Operation

~ Okay, so how do I use this thing?

Aha! See there? I'm putting words in your mouth already. Anyway, it is simply a matter of calling the static methods on the serializer and deserializer classes. When you want to save some data simply use:

```
XmlSerializer.Serialize(object obj, int version, string rootName);
```

The first parameter is the data you want to serialize, whatever that may be. The second value is mostly optional and can be used during deserialization to see if the data your are deserializing can be expected to match the current state of the classes it represents. Whenever you perform major refactoring of your classes this version number should likely be bumped up to avoid trying to deserialize data into fields that may no longer exist. The final parameter is the name of the root XML element. This string must conform to standard XML element naming conventions (no spaces or wierd characters). In the event that you pass a GameObject to this method the rootName parameter will be ignored and the name of the GameObject will be used. There is also an overload of this method that simply takes a GameObject and version number. The XMLDocument returned will contain all of the serialized data and can then be written to file, converted to a string, compressed and sent over the wire or whatever it is you plan to do with the data. When it is time to retreive your data you will want to use:

```
XmlDeserializer.Deserialize(string xml, int maxVersion);
```

The first parameter is the xml text (which can be retrived from an XmlDocument.InnerXml field) and the max version of the data that this deserializer should allow. The object returned will be of the same type that was passed in to the serialize method so you'll need to cast it back to that. There is one other method you'll need to use on certain occations. For reasons that will be explained later, if you passed any object derived from *UnityEngine.Component* (for example Rigibody2D, AudioSource, Transform, or any kind of MonoBehaviour) then you will need to use a special deserializer to handle getting it back out. It takes the form of:

```
XmlDeserializer.DeserializeComponent<T>(string xml,T receiver, int 
maxVersion);
```

The new parameter is of course *receiver*. You should pass an existing Component from a GameObject that is of the same type as was serialized previously. This method will then set all fields and properties to match the state of the serialized data.

~ Sounds simple! So what kinds of things can I serialize with this thing?

Pantagruel is designed to work opaquely. That is to say, you simply feed it some data and it does its thing. Internally it does handle things differently depending on what kind of data it is fed though. And it is important to understand the differences and limitations of those differences. For the most part however you can just given it anything and it will be handled. Primitives like ints and floats, immutables like strings and Vector3, complex user-defined datatypes with circular references, GameObjects. Generic types, List<>, Dictionary<>, delegates. You can even use polymorphic references to classes that were downcast. The only real setback is if a class does not define a default parameterless contructor. In this case it will necessary to supply a serialization surrogate that can handle instatianting the type.

Despite all of these abilities there are still a few issues that need to be considered when serializing certain kinds of data. For the most part the serializer internally is operating in one of several 'modes' so-to-speak. These modes are invisible to you and are based on the context of the kind of data that the serializer is being given. No specific action needs to be taken to ensure the serializer is 'operating in the correct mode'.

Standard Mode:

The first mode of opertation is called *Standard Mode*. The serializer is in this mode when you hand it any kind of primitive or complex data type that isn't a GameObject or Unity Component. User-defined classes, .Net types, int, floats, strings, Lists, Dictionaries, you name it. Heck, even delegates (although it can get quite hairy sometimes so I'd recommend it only in limitied situations). This is meant to work like any other kind of serializer. You can generally just push the data in, supply a version number and a root xml element name and be done with it. When you deserialize the data it will all be as you left it.

However, there are two big caveats: In this mode references to GameObjects and Components will *not* be serialized! This is because there is simply no useful context for them. GameObjects exist in a scene graph hierarchy and need to know where they would belong in that hierarchy. We can't simply serialize the parent object too because we wouldn't know of a good place to stop. Next thing you know you'd have the entire scene serialized! Same goes for Components. They can't exist naked without a GameObject (they can't even be initialized like that!) and we'd need to know the whole GameObject hierarchy all over again just to serialize this one little component reference.

~ *But I really need to preserve these references to my GameObject/Component! What if it's something important like my Game State Manager or the Player's entity?*

The recomended way to handle these things is the *Unity Way*™. Start using components and GameObjects to store all your references to these user-made classes and then serialize the whole GameObject hierarchy (see below). Barring that, you could also use tags and search for these objects and re-attach them after deserialization.

~ *If I serialize an object that has a reference to another still-active object and then deserialize the data will the reference be re-linked to the original copy or will a new one be made?*

If the reference is valid for serialization then the whole object will be serialized and new copies of all objects in the graph will be instatiated. Otherwise the reference will be null. In either case if you want to re-link to already existing objects after deserialization then you will have to handle that yourself.

~ *What about shared or circular references? Will copies be made or will it all be re-linked properly?*

The serializer will properly maintain reference ids in such cases and all shared and circular links will be resolved correctly.

Hierarchy Mode:

This is the most common way of serialzing Unity game states. When you pass a GameObject to the serializer it will preserve the entire hierarchy with that GameObject as the root. All components attached to each GameObject and their data will also be stored. Again, there is a slight caveat: References to GameObjects or Components *outside* of the hierarchy that is being serialized will not be included. And again, this is due to the fact that the only way to store the full context would be to serialize the entire scene. So the moral of the story is if you really want all of your data serialized, either refactor it so that you are serializing the common root GameObject of all that data or use *GameObject.Find()* to look for things to re-attach after deserialization. Other than that, this mode works under the same principles as the standard mode: Shared or circular references will be properly maintained and you can expect a GameObject to pop out of the deserializer that is more-or-less the same as what you put in.

~ What about references to prefabs?

Prefabs are a special case for GameObjects. In this situation what you really want is to store a reference to the prefab resource and not a particular instance of a GameObject. However, Unity doesn't provide a means of determining if a GameObject reference is to a prefab or an in-scene instance. In order for this to work you must attach a *PrefabIdComponent* to the root of your prefab. There are a whole slew of other issues as well. You can read more about it below in the 'Resources' section of this document.

~ What about other resources? You know, Sprites, Maerials, Textures, Audio data, etc...

Again, this is more complex than it sounds. Refer to the 'Resources' section below..

Component Mode:

This is sort of a special case mode. As stated before, you can't expect to deserialize a single component and get one to pop out into the world ready to go. Unity doesn't allow naked components to be created. And even if you could get one such component, there is no way to attach such a pre-existing object to a GameObject. In many cases you would just serialize the whole GameObject and be happy. However, there might be times when it makes more sense to just store the state of a single Component - like if you wanted to send that state over the wire or clone it or store some info that isn't bloated with irrelevant data. It is entirely possible to serialize a single component just like normal. However, when deserialzing you will need to use *XmlDeserializer.DeserializeComponent<T>(T component)* to handle getting things back out. In this case you can pass it a component that is already initialized and attached to a GameObject and the deserializer will simply fill out the fields using the stored data.

How Complex Types are Handled

Pantagruel uses reflection to glean as much information as it can from complex data types. In many cases if your data serialized correctly using Unity's default serializer then you are already in the clear. But just for the sake of review, the following rules apply:

- Serializes all public fields.
- In some cases (UnityEngine.Object and built-in Unity components) serialize all properties with public get/set accessors.
- Ignores all private fields.
- Ignores properties on most data (except the few cases mentioned above).
- Ignores properties that don't have both a public *get* and public *set* accessor.
- Ignores all public fields marked with [NonSerialized] or [XmlIgnore].
- Ignores all properties marked with [XmlIgnore].
- Apply serialization to all base-class data of a class unless that class is marked

[XmlIgnoreBaseType]

• Defers serialization to the object itself if marked with [IXmlSerializable].

• Ignores [Serializable]. This serializer will grab everything it can get its greedy little mits on.

• Circular or shared references are serialized once and then referenced by id after that.

• References to GameObjects or Components that are not part of the hierarchy being serialized will be stored as null.

• Delegates are serialized. The target objects that own the methods they refer to will also be serialized so make sure that's what you want.

• *ISerializationSurrogates* will be considered in the reverse order they were added and before using default serialization methods.

Serialization Surrogates

~ Hey, this is pretty great so far but I'm having trouble with a specific class. It won't serialize correctly or it is serializing too much information.

That's not a question. But fortunately there is still an answer for your concern - Serialization surrogates. It is possible to write a small class that implements the *ISerializationSurrogate* interface and can specifically feed or receive information from the xml serializer. In this way you can translate, filter, combine, or ignore whatever data you see fit for a specific type. As a bonus to help get you started, you can derive from the *SurrogateBase* class to gain access to a lot of helper methods for retreiving data from your objects.

Once the surrogate is properly implemented you simply call the static method *AddTemporarySurrogate()* on the *XmlSerializer* or *XmlDeserializer* just before serializing or deserializing your data. Surrogates will handle serialization and deserialization of the type specified or of any type derived from the type specified. In this way they can be used as fallbacks for derived classes. Surrogates are considered in the reverse order they were added so the newest addition will be used first if viable. This way you can specify surrogates in order of most generic to most specfic to handle a wide variety of special datatypes. It is important to note that surrogates will *not* be considered for primitive types, only complex types.

There are also Activation surrogates that must implement the *IActivationSurrogate* interface and be added to the serializer or deserializer with the static method *AddTempoaryActivationSurrogate()*. Unlike, full-blown surrogates, these just provide a way of creating an instance of an object in the event that the given type does not have a default parameterless constructor.

Also, take note of the name 'temporary' in those methods. They surrogates will only stick around unless you serializer or deserialize something. So you will have to add all of your surrogates in every time you want to use them.

Resources

~ Ok, so what's the deal with references to resources? You know, textures, sprites, audio clips, animator controllers, etc....

So here is where things get real hairy. Unfortunately, Unity does not provide a way to access the internal reference used by those resources so instead, they are all serialized as strings that can be used by *Resources.Load()* at runtime to re-obtain a reference. The big issue with this is that all resources that you want to serialize **absolutely must** be stored inside a *Resources* subdirectory. If you are not familiar with the special Resources directory and the *Resources* utility class used by Unity I suggest you read up on them now.

~ Got it. All assets that are to be serialied must go in a Resources folder or a subdirectory of a Resources folder. Anything else?

Yeah, that's just the beginning. Unity also doesn't provide a way to determine the path that an asset was loaded from during runtime. This means we can't simply obtain a string to serialize. So to combat this, a series of manifest files must be built at edit time. These manifest contain references to every single asset you have stored in a Resources folder and it associates each of them with the path that they can be loaded from at runtime using *Resource.Load()*. Basically, what this means is that every single time you create a new asset, if there is any chance whatsoever that it needs to be serialized, you'll have to place it in a Resources folder and then rebuild the manifest library.

To build a resource manifest library, navigate to the menu Window->Pantagruel-> Resource Manifest and choose one of the two build options. This may take some time if you have a large project with a lot of files. The safe option will also ensure that no repeat directories are generated. After that is done you should be able to serialize any references to resources included in the manifest.

Another thing you will have to do for GameObject prefabs is to attach a *PrefabId* component to the root of the prefab. This is the only reliable way of determining if this GameObject reference was assigned from a prefab at edit-time or not.

~ What about asset bundles

They are not currently support because I have little-to-no experience with them. They will probably be supported in the future once I have a better idea.

~ What Resources tyes are currently supported?

The following resources will be serialized as strings.

- Animation Clip
- Animator Controller
- Audio Clip

- Font
- GameObject Prefab (must have *PrefabId* component on root)
- Material
- Mesh
- PhysicMaterial
- PhysicsMaterial2D
- Shader
- Sprite
- Texture2D
- Texture3D

Anything not on this list will likely fall back on to the default *UntiyEngine.Object* surrogate and will be serialized in place as a unique copy. This will likely cause issues and in such cases I would suggest that you mark the fields as [NonSerialized] and manually re-attach the references in your own code after deserialization. Support for other resources types can be added by editing the files *Constants.cs* and *ResourceManifest.c*. These files are somewhat documented with comments about where and how to add new types. You should also note that the manifest files will store sprite references as both Sprites and Texture2Ds. It's not really an issue with anything but it can make the manifests a bit larger.

Each resource of a given type must compile to a unique resource path. That is to say, if you have two textures with the same name in the same sub-directories relative to a Resources folder then the manifest builder will spit out warnings. This is because it will be impossible to determine which resource you actually wanted to load at runtime since they will both have the same path (Unity, by default always takes the first available in such cases).

Bad Example:	PlayerAssets/Resources/Textures/MainText.png
	NpcAssets/Resources/Textures/MainText.png

In both cases this will compile to 'Textures/MainText.png'. There is no way for *Resources.Load()* to know which you actually want so it will always take the first available resource of the appropriate type.

Good Example:	PlayerAssets/Resources/Player/Textures/MainText.png
	NpcAssets/Resources/Npc/Textures/MainText.png

Good Example:	PlayerAssets/Resources/Textures/PlayerText.png
	NpcAssets/Resources/Textures/NpcText.png

Good Example: PlayerAssets/Resources/Player.png (where this is a Texture asset)

PlayerAssets/Resources/Player.fbx (where this is a 3D model asset)

In the first two examples, the resources will compile to unique names so everything will be ok. In the third example the resources are of a different type so again, it is ok.

One other consideration is the resource object's name. You know, the *name* field that all *UnityEngine.Object*s have. Not the name of the resource file but the actual *UnityEngine.name* field. This absolutely must not change after a manifest is built. This name is the only way the object can be traced back to the manifest at runtime. Keep this in mind especially with Shaders and GameObject prefabs because their names often get changed, sometimes automatically and even in project code just for the sake of easier object tracking in the editor.

Appendix

Here you will find just the facts about Pantagruel's XML serializer. If you are quite familiar with the topic of serialization then everything you really need to know is right here.

Constants.cs

The file found at *Pantagruel/Serializer/Scripts/Constants.cs* contains several important values. The most notable is *Constants.RootPath*. If you change the location of the Pantagruel project folder you must also change this path accordingly to reflect the new location.

Types Subject to Serialization

- Primitives (int, float, double, char, etc) are all able to be serialized directly or as fields of a complex type. Strings are also treated as primitives.
- Enumerations are cast as ints.
- *ISerializable* interface is ignored.
- *IXmlSerializable* interface is honored but not recommended unless you know eactly what you need to do to make it work.
- Complex types are recursively serialized using reflection on fields within.
- Shared references to complex types and circular references are properly maintained.

Special Considerations

- Polymorphism is handled correctly. Types are cast and stored in their true form when serialized.

• Generic types are serializable.

• Arrays, lists, and dictionaries, generic or not are all serialized properly.

• Delegates are subject to serialization. Note that the targets of invoked method calls will be serialized as well so be sure you actually want your delegates serialized.

• Anonymous delegates are not tested with this system yet.

• Events are *not* serialized.

• In order for a type to deserialize automatically it must have a default parameterless constructor. If it doesn't, then a serialization surrogate or activation surrogate will be necessary in order to properly instantiate the object before the deserializer can fill out its data.

• Refences to GameObjects are serialized only if the root object passed is a GameObject. In this case all GameObjects in the hierarchy from that root object will be included along with all attached components of those GameObjects.

• Refences to Components will only be serialized if the root object passed is a GameObject. All components will be serialized as being attached to their respective GameObject within the hierarchy.

• If a Component is passed directly to the serializer then it will be necessary to use *XmlDeserializer.DeserializeComponent<T>()* and pass it an already instantiated instance of the correct Component type in order to deserialize that data again. This is due to the fact that Unity doesn't allow Components to be instantiated by themselves.

Complex Type Fields that are Handled

• Public fields are serialized unless marked with [NonSerialized] or [XmlIgnore].

• Private fields are ignored unless marked with [SerializeField].

• Properties are usually ignored except in the case of anything deriving from *UnityEngine.Object* that isn't a Monobehaviour. In the case of UnityEngine.Objects, properties with public get/set accessors are serialized.

• Base class fields and properties are considered for serialization under the same rules above.

• [XmlIgnoreBaseType] can be used on a class to avoid serializing base class info.

• GameObject references are only serialized if the root object passed was a GameObject and the reference in question points to a GameObject that is within the hierarchy being serialized. If a reference points to something outside the hierarchy or it is not a hierarchy that is being serialized it will simply be null.

• Component references are also only serialized if they point to a Component that is attached to a GameObject that is within the hierarchy being serialized.

Attributes Utilized

- Honors [SerializeField], [XmlIgnore], [NonSerialized] attributes on fields and properties as applies.

- Honors [XmlIgnoreBaseType] on classes.

- Ignores [Serializable] on classes. All types are subject to serialization.

Serialization Surrogates

- Classes that implement *ISerializationSurrogate* can be used to handle special-case serialization (this is how many Unity-specific objects are handled by Pantagruel in fact!).

- New surrogates can derive from *SurrogateBase* to help quickly develop them. It provides several helper methods for reflecting information about the objects you are serializing.

- The *SurrogateBase* has three very useful methods: *GatherFields(), GatherProperties()*, and *GatherFieldsAndProps()*. These can be used to collect info for a datatype you want to build a surrogate for. All three methods also have a 'filter' argument list that can contain a list of names that should be ignored when collecting fields and properties to serialize or deserialize.

- You can supply custom written surrogates to the serializer and deserializer by calling *AddTemporarySurrogate()* before caling Serialize or Deserialize. Afterward, these surrogates will be forgotten so you will need to re-apply them every time you want them to be used (hence the term 'temporary').

- Surrogates are considered in reverse order they were added so you can specify surrogates from most generic to most specific in that order.

- The *StreamingContext.Context* value passed to the surrogates during serialization/deserialization will be either an *XmlSerializer.SerializeContext* or *XmlDeserializer.DeserializeContext* object respectively.

- *IActivationSurrogate* can be implemented on a class and then applied just before deserialization using *AddTemporaryActivationSurrogate()*. These activation surrogates are used to simply create an instance of an object that does not have a default parameterless constructor.

- Activation surrogates are considered in reverse order they were added so that you can supply most generic to most specific type cases in that order.

- Surrogates are applied to complex types only. Primitives, strings, and enumerations will ignore them.

Unity Resources

Important: Each resource of a given type must compile to a unique resource path. That is to say, if you have two textures with the same name in the same sub-directories relative to a Resources folder then the manifest builder will spit out warnings. This is because it will be impossible to determine which resource you actually wanted to load at runtime since they will both have the same path (Unity, by default always takes the first available in such cases).

Bad Example: PlayerAssets/Resources/Textures/MainText.png

NpcAssets/Resources/Textures/MainText.png

In both cases this will compile to 'Textures/MainText.png'. There is no way for *Resources.Load()* to know which you actually want so it will always take the first available resource of the appropriate type.

Good Example: PlayerAssets/Resources/Player/Textures/MainText.png

NpcAssets/Resources/Npc/Textures/MainText.png

Good Example: PlayerAssets/Resources/Textures/PlayerText.png

NpcAssets/Resources/Textures/NpcText.png

Good Example: PlayerAssets/Resources/Player.png (where this is a Texture asset)

PlayerAssets/Resources/Player.fbx (where this is a 3D model asset)

In the first two examples, the resources will compile to unique names so everything will be ok. In the third example the resources are of a different type so again, it is ok.

Other considerations are:

- Prefabs must have a *PrefabId* Component attached to the root GameObject in order to be identified as a prefab and not a regular GameObject. This component automatically removes itself upon instatiation to avoid accidentally attempting to serialize a live object as a prefab resource.

- *Animation Clips, Animator Controllers, Audio Clips, Fonts, GameObject prefabs, Meshes, Materials, PhysicMaterials, PhysicsMaterial2Ds, Shaders, Sprites, Texture2Ds, and Texture3Ds* are currently supported.

- All resources references will be simply stored as a string that identifies the resource location and name within a Resources folder.

- Resources that are referenced *must* exist within a Resources folder so that they can be properly instantiated at runtime during deserialization

- A manifest library must be built at edit time (Window->Pantagruel->Resource manifest) and will only include supported resource types located in Resource folders. Only items in the manifest will be able to serialize and deserialize as resource paths.

- You **absolutely must not** change the root name field (e.x. root GameObject.name for a prefab) of a resource or prefab after you have built a manifest. If you do, the serializer will not be able to link that reference to the correct manifest at runtime and the reference will not serialize properly.

• All manifest files generated are named after the *name* field of the objects they store and not the filename of the resource. Many objects, like GameObject prefabs or Shaders, use different names than the file so this is required to allow linking at runtime since the original resource filename will no longer be accessible.

• **Warning**: Any resource type that is not supported will be serialized in place. Assuming the deserialization even works, the reference will now be to a copy rather than a shared reference.

• You can extend the resources that are supported by the manifest builder and serializer by editing the list in *Pantagruel/Serializer/Scripts/Constants.cs* as well as adding support code within *Pantagruel/SerializerScripts/ResourceManifest.cs*.

Contact Info

Any questions, concerns, bugs, or requests can be directed to

pgi-support@ancientcraftgames.com