

Serializer

Serialization tools can be found in namespace `Core.Serialization`.

Data Serialization

Low level tool to serialize information.

Overview

In GFF, serialization is done with an abstract `Serializer` that uses a symmetric declaration pattern (symetric read and write). Here is as an example how to describe the serialization of an object:

```
public class SerializableObject : ISerializable
{
    private int intField;
    private EnumType enumField;
    private SerializedObject child;

    public void Serialize(Serializer serializer)
    {
        this.intField = serializer.SerializeValue("Integer", this.intField);
        this.enumField = serializer.SerializeAsValue("Enum", this.enumField);
        this.child = serializer.Serialize("Child", this.child);
    }
}
```

Symetric serialization: It is important to send your value as parameter **and** to retrieve it from the return value of serialization methods. This way the system can handle Read and Write operation for reference types and value types in a safe way.

You can then serialize it in any C# stream using a serializer (for now only `JsonSerializer` exists):

```
using (MemoryStream stream = new MemoryStream())
{
```

```

var objectToSerialize = new SerializableObject();
using (var serializer = new JsonSerializer(stream, JsonSerializer.SerializeMode.Writer))
{
    objectToSerialize.Serialize(serializer);
}

var derializedObject = new SerializableObject();
using (var serializer = new JsonSerializer(stream, JsonSerializer.SerializeMode.Reader))
{
    derializedObject.Serialize(serializer);
}
}

```

API

This section describe how to serialize data in a `Serialize` method.

Object Serialization

You can serialize any object instance, array or list of instances that implement the `ISerializable` interface using the `Serialize` methods.

```

T Serialize<T>(string key, in T serializableObject) where T : ISerializable;
T[] Serialize<T>(string key, in T[] array) where T : ISerializable;
List<T> Serialize<T>(string key, in List<T> list) where T : ISerializable;

```

Value Serialization

You can serialize any value type, array or list of value types using the `SerializeValue` methods.

```

T SerializeValue<T>(string key, in T value);
T[] SerializeValue<T>(string key, in T[] array);
List<T> SerializeValue<T>(string key, in List<T> list);

```

Serialize simple objects as value

Some data structure are simple enough to be serialized as value but are not value type. This special cases are handled using the `SerializeAsValue` methods. For now these methods can serialize enum as integer or string, C# type as string or event Godot `RandomNumberGenerator`. You can also implement the `IValueConvertible` interface to create a specific conversion on types you own.

```

T SerializeAsValue<T, TValue>(string key, in T serializableObject) where T :
IValueConvertible<T, TValue>;
Type SerializeAsValue(string key, in Type type);
RandomNumberGenerator SerializeAsValue(string key, in RandomNumberGenerator rng);
T SerializeAsValue<T>(string key, in T enumValue, bool serializeAsInteger = false) where T :
struct, Enum;
Queue<T> SerializeAsValue<T>(string key, in Queue<T> enumQueue, bool serializeAsInteger =
false) where T : struct, Enum;

```

Versioning

Handling version in serialization is mandatory if you want to handle backward compatibility in your game saves. You can write a version number to any `Serialize` method by using the `SerializeVersion` method. If no version is found the method return 0.

```

public void Serialize(Serializer serializer)
{
    int version = serializer.SerializeVersion(1);

    this.Legacyfield = serialize.Serialize("Legacyfield", this.Legacyfield);
    if (version >= 1)
    {
        this.Newfield = serialize.Serialize("NewField", this.NewField);
    }
}

```

The version number is unique per type so you can have different version number in inheritance level of a class hierarchy.

Save Manager

High level manager that handle game saves.

Game Save System

The `SaveManager` contains a list of save slots. It allows you to create, delete or edit any save slot. A save slot is represented by a single archive file in the save folder (it is a zip file using the `sav` extension).

Save Info

When you retrieve a save slot you get a `SaveInfo` reference. This structure contains the metadata about the save slot content. These metadata are stored as a config file in the game slot archive. You can add any custom metadata to the save (allowing you to gather information about the save without deserializing everything to display it in menus).

Metadata are organized in sections, the Main section contains all general metadata. Then there is one section per save entry.

Save Entries

You can store any instance that implement `ISerializationEntry` in a save slot as an entry. It creates one file per entry in the archive (plus an associated section in the metadata).

Use the `WriteEntry` and `ReadEntry` to store and retrieve entry instances from a save slot.

Post Load Pass

Sometimes you need to serialize references to other serialized objects and you want to resolve these references during loading to just have a C# reference to use in your code. `PostLoad` pass just serve this purpose, it calls the `PostLoad` method on all instances that implements `ISerializableWithPostLoad` interfaces. This call is done after the deserialization process so all instances exists and have valid deserialized data.

PostLoad methods are called in reverse order than the Serialize methods (from child to parent).

Revision #5

Created 2026-07-22 15:32:47 UTC by Adrien Allard

Updated 2026-08-12 20:46:36 UTC by Adrien Allard