Table of Contents

Save Extension Documentation

Save Extension allows your projects to be saved and loaded automatically.

This plugin is for Unreal Engine 4 and has support for versions **4.20** and **4.19**

Intended Usage

Our plugins are designed to work for very different games and needs, but naturally, they are more focused towards satisfying certain needs.

Save Extension in particular has been created to help games with high amounts of content in the world like open world or narrative games that need to save world state with the less amount of work or complexity possible.

As an example, a game like Super Mario probably wouldn't need **Save Extension**, because it doesn't need to store world state. It can do it, but may not be worth it. Other games might have items to be picked, player states, AI, or streaming levels that require this serialization and here's where the strength of **Save Extension** really shines.

Quick Start

Check Quick Start to see how to setup and configure the plugin.

Supported Features

SaveGame tag saving

Any variable tagged as `SaveGame` will be saved.

Full world serialization

All actors in the world are susceptible to be saved.

Only exceptions are for example StaticMeshActors

Asynchronous Saving and Loading

Loading and saving can run asynchronously, splitting the load between frames.
This states can be tracked and shown on UI.

Level Streaming and World Composition

Sublevels can be loaded and saved when they get streamed in or out. This allows games to keep the state of the levels even without saving the game.

If the player exists an area where 2 enemies were damaged, when he gets in again this enemies will keep their damaged state

Data Modularity

All data is structured in the way that levels can be loaded at a minimum cost.

Compression

Files can be compressed, getting up to 20 times smaller file sizes.

About Us

At Piperift we like to release the technology we create for ourselves.

Save Extension has been around for multiple years being used in internal and external projects, and as such, we wanted it to be public so that others can enjoy it too, making the job of the developer easier.

This plugin was designed to fulfill the needs for automatic world saving and loading. Features that are unfortunately missing in the engine at this point in time. Automatic in the sense that any actor in the world can be saved, including AI, Players, controllers or game logic without any extra components or setups.

Installation

Manually

This are the general steps for installing the plugin into your project:

1. Download the last release from here
Make sure you download the same version than your project

2. Extract the folder “SaveExtension” into the **Plugins folder** of your existing project (e.g "MyProject/Plugins")

3. Done! You can now open the project

From Marketplace

Install from the launcher: AVAILABLE HERE

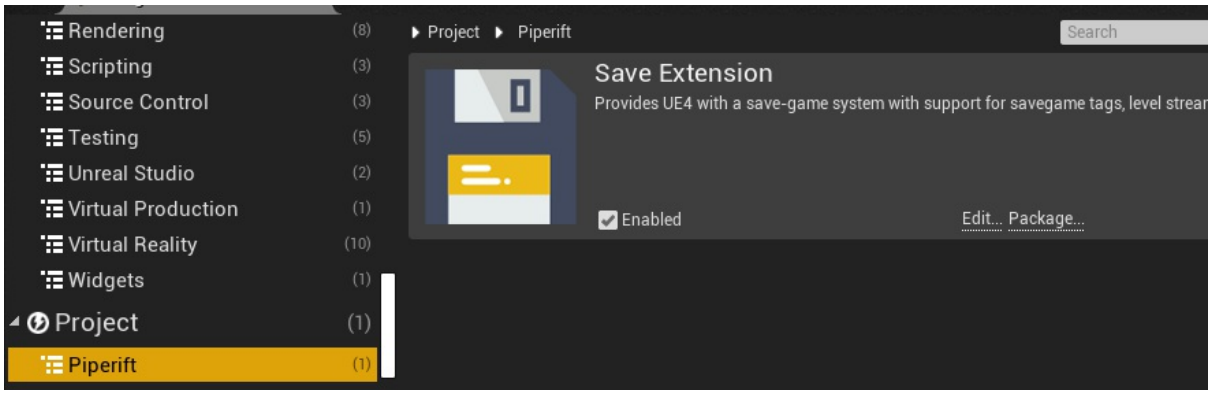
Quick Start

Quick Start will show the basic steps to follow to setup the plugin and start using it at a base level.

Setting Up the Project

We can start by creating an empty project (How to create UE4 projects) or instead using your own. Then installing the plugin from marketplace or inside Plugins folder (See Installation).

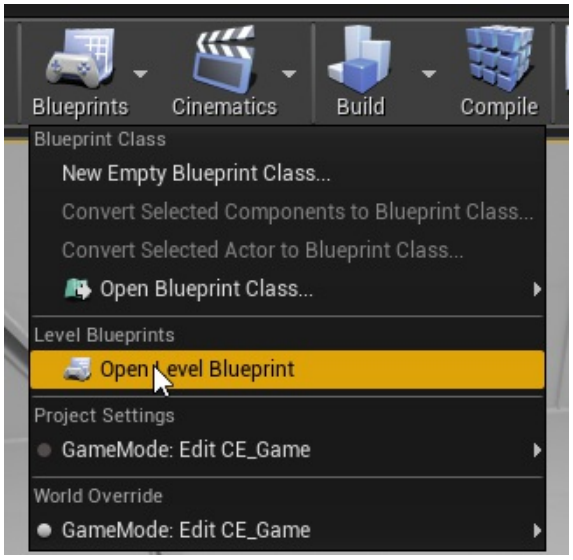
If everything went right, we should see the plugin enabled under *Edit->Plugins->Piperift*



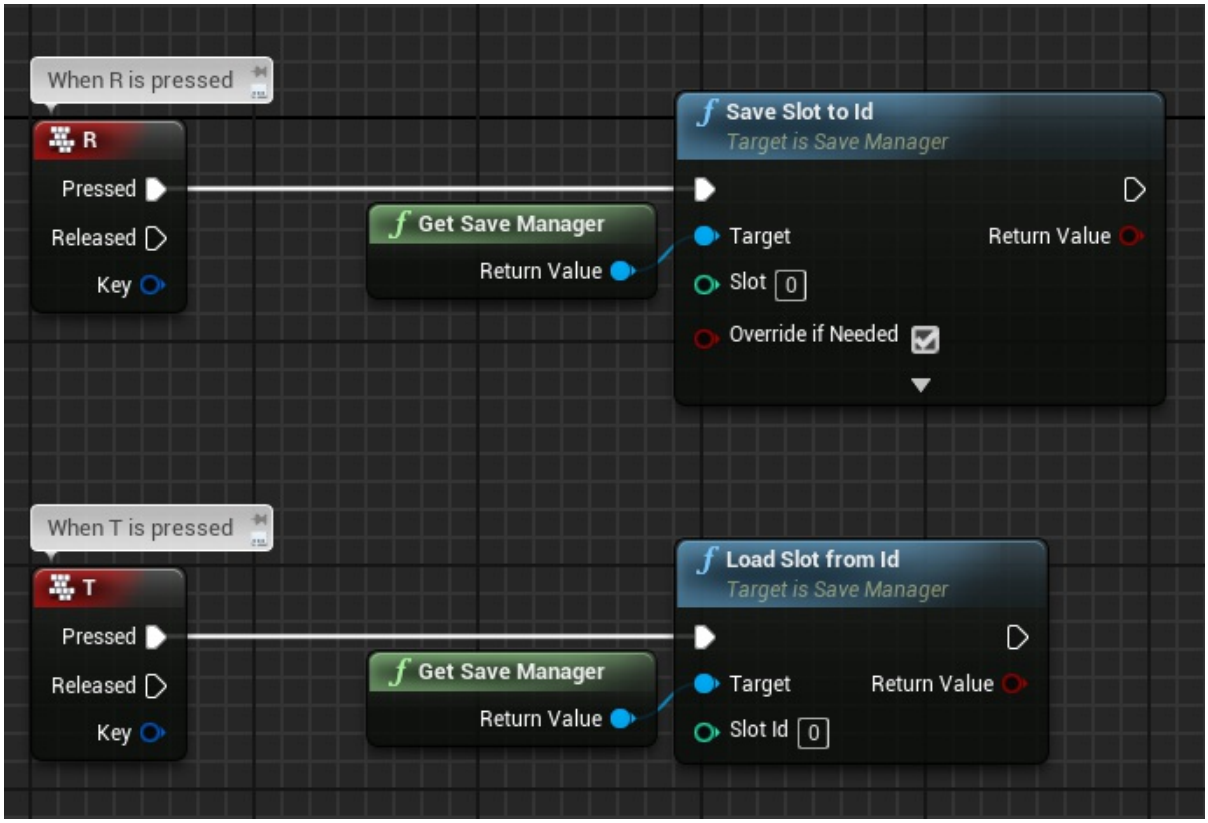
Using the Plugin

Save Extension requires minimum setup. The only thing you have to do is to call Saving and Loading. Therefore **as an example** we will make a level-blueprint **save** when we press **R** and **load** with **T**.

Lets start by opening the level-blueprint of the scene we want to use:



Then we add the following functions:



Save Slot to Id will save the world into slot 0 and override if another save was there.

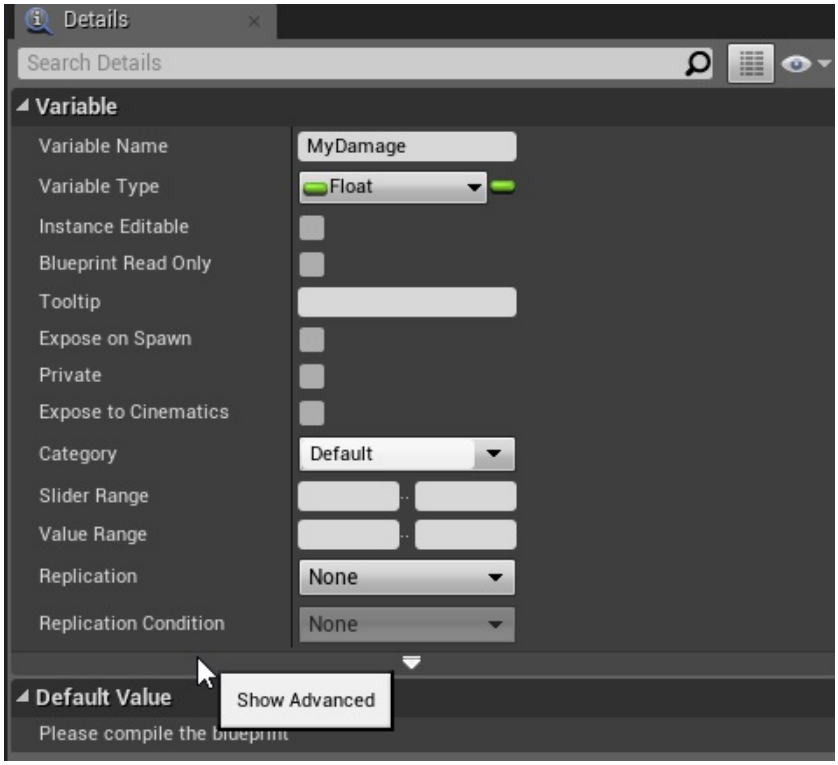
Load Slot from Id will load world from slot 0

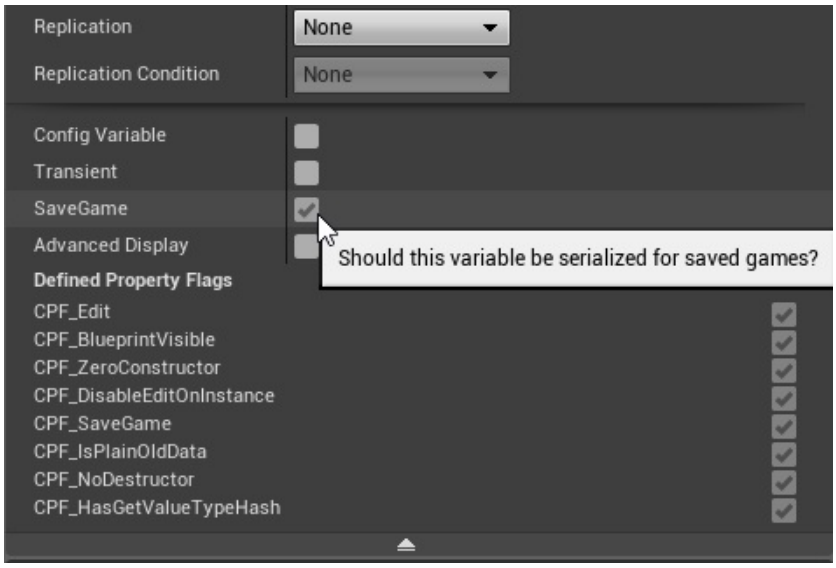
Something to save

Player's rotation and location will always be saved (unless you disable it from Configuration) but we can also add to the scene some actors to be saved. They could save variables, physics, positions, etc.

Saving a variable

To save any variable inside an Actor or Component we can just tick "*SaveGame*" on its properties:


Details
Search Details
Variable
Variable Name
MyDamage
Variable Type
Float
Instance Editable
Blueprint Read Only
Tooltip
Expose on Spawn
Private
Expose to Cinematics
Category
Default
Slider Range
Value Range
Replication
None
Replication Condition
None
Default Value
Show Advanced
Please compile the blueprint


Replication
None
Replication Condition
None
Config Variable
Transient
SaveGame
Advanced Display
Should this variable be serialized for saved games?
Defined Property Flags
CPF_Edit
CPF_BlueprintVisible
CPF_ZeroConstructor
CPF_DisableEditOnInstance
CPF_SaveGame
CPF_IsPlainOldData
CPF_NoDestructor
CPF_HasGetValueTypeHash

Asynchronous

Asynchronous loading and saving is one of the most requested features of the plugin.

Saving and Loading process can be asynchronized, speeding up the process, making it super efficient.

There are currently three types of parallelization:

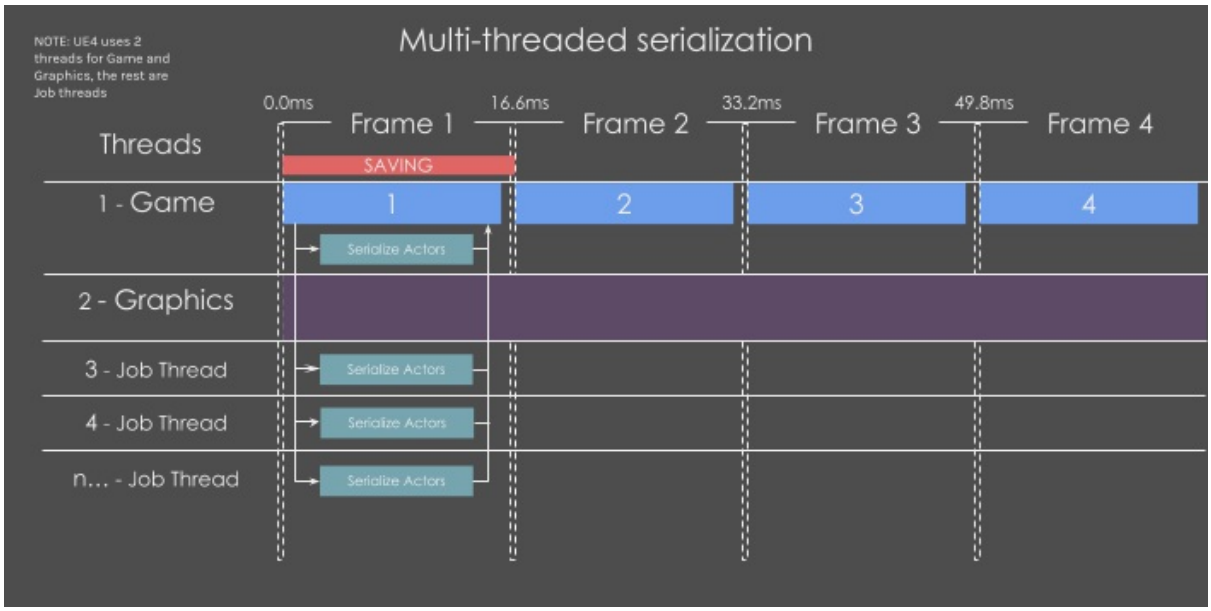
- Multithreaded Serialization
- Frame-splitted Serialization
- Multithreaded Files

All of them can be configured to be used while saving, loading or both.

Multithreaded Serialization

Multi-threaded serialization will split the **serialization** (*data collection from the world*) between all available threads on the CPU.

This means platforms with many cores like modern CPUs will obtain very noticeable performance advantages.

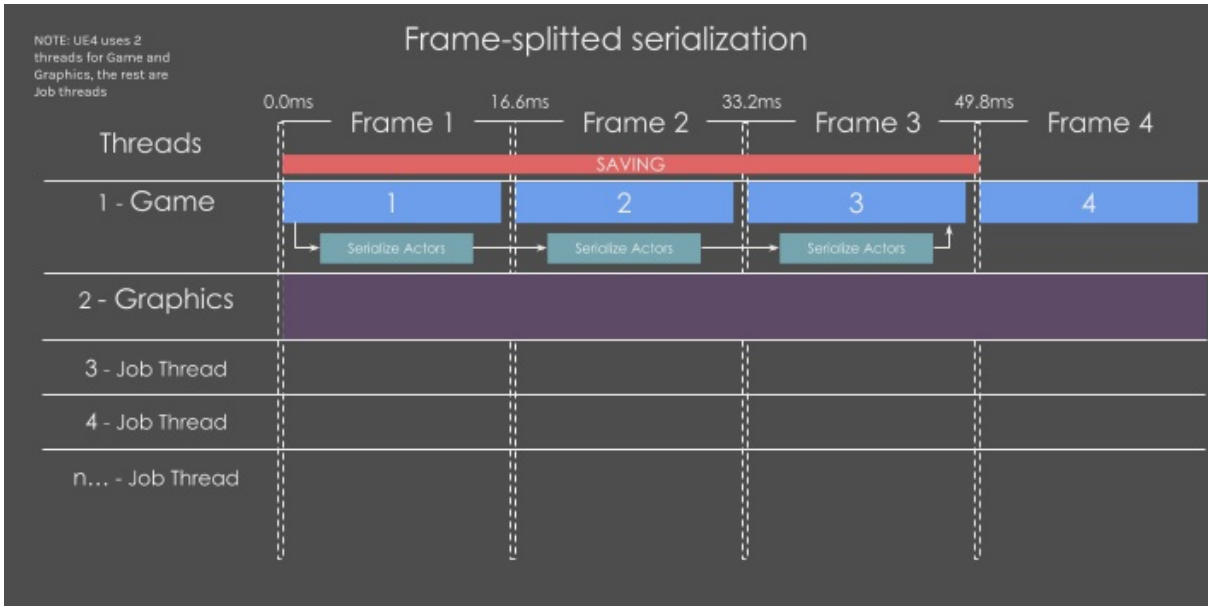


Frame-splitted Serialization

Serialization (*data collection from the world*) will be splitted between multiple frames, taking *MaxFrameMS* (5ms by default) every frame until it finishes.

This method is only available if Multithreaded Serialization is disabled.

> Frame Splitting is not recommended if level streaming saving is enabled. It could be interrupted while loading or saving creating unexpected issues



Multithreaded Files

Files will be compressed and saved/loaded asynchronously on the background.

Its great to avoid the small performance cost of compressing saved games. Usually you would just keep it enabled, but you have the option to disable it.

Configuration

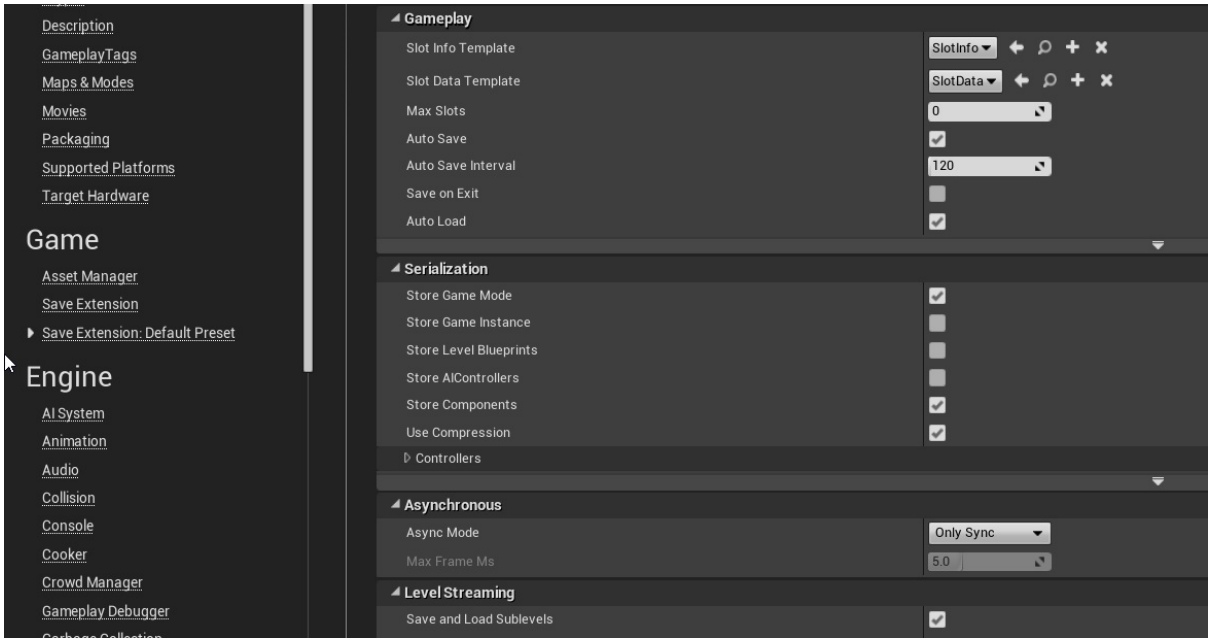
Presets

A preset is an asset that serves as a configuration preset for Save Extension.

Within other settings, presets define how the world is saved, what is saved and what is not.

Default Preset

Under *Project Settings -> Game -> Save Extension: Default Preset* you will find the default values for all presets.



This default settings page is useful in case you have many presets, or in case you have none (because without any preset, default is used).

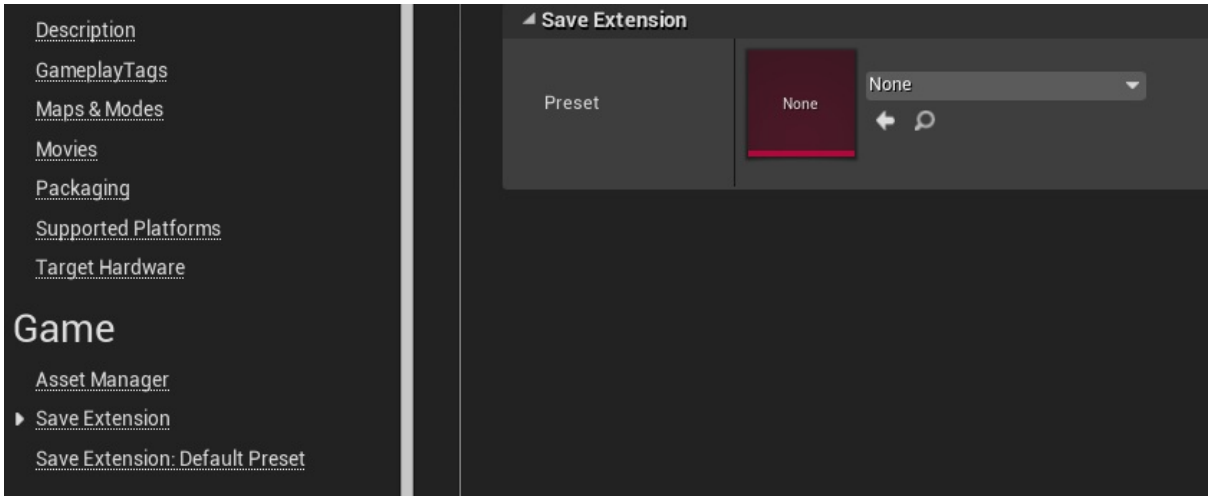
> All settings have defined tooltips describing what they are used for. Check them moving your mouse over the property.

- **Gameplay**: Configures the runtime behavior of the plugin. Debug settings are also inside Gameplay. Check Saving & Loading
- **Serialization**: Toggle what to save from the world.
  - **Compression**: This settings can heavily reduce saved file sizes, but adds an extra cost to performance.
- **Asynchronous**: Should save & load be asynchronous?
- **Level Streaming**: Configures Level Streaming serialization

Custom Presets

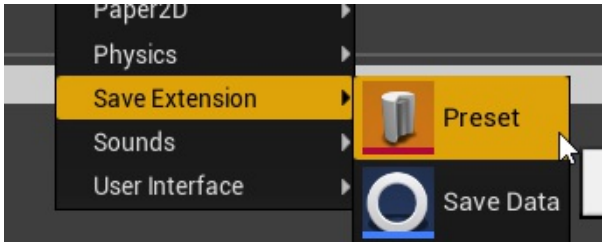
Setting a custom Preset

You can define which preset to use in editor inside *Project Settings -> Game -> Save Extension*



Because presets are assets, the active preset can be switched in runtime allowing different saving setups for different maps or game modes.

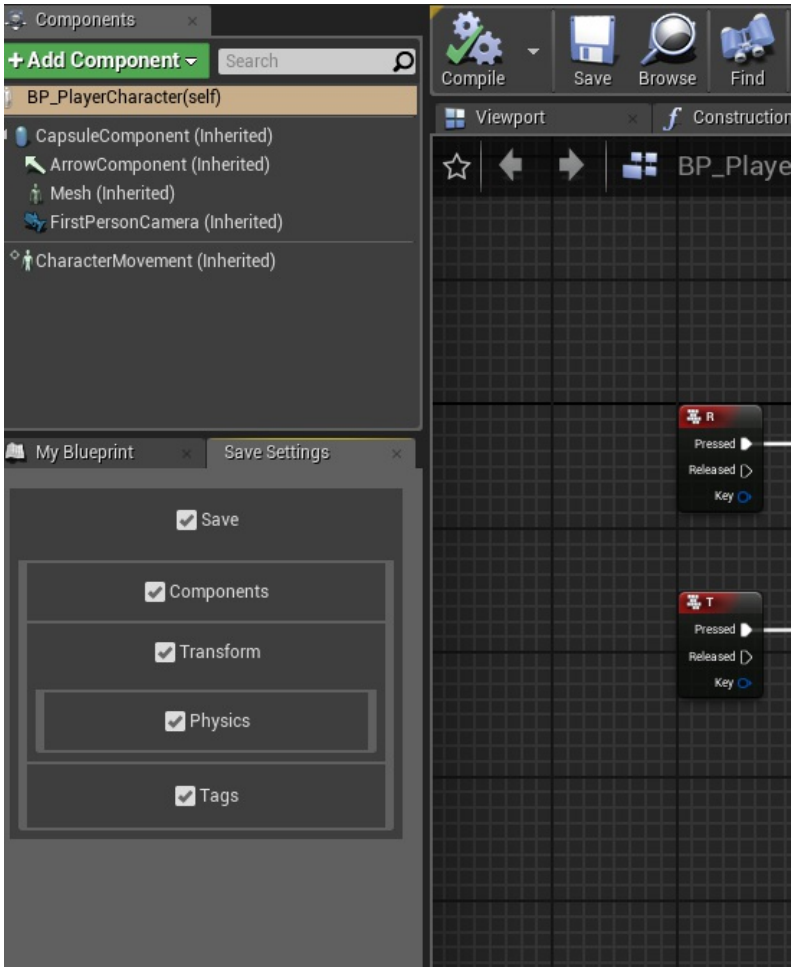
Creating a Custom Preset

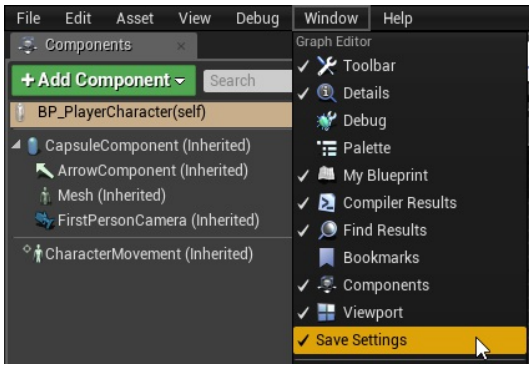


You can create a new preset by right-clicking on the content browser -> *Save Extension* -> *Preset*

Per-actor settings

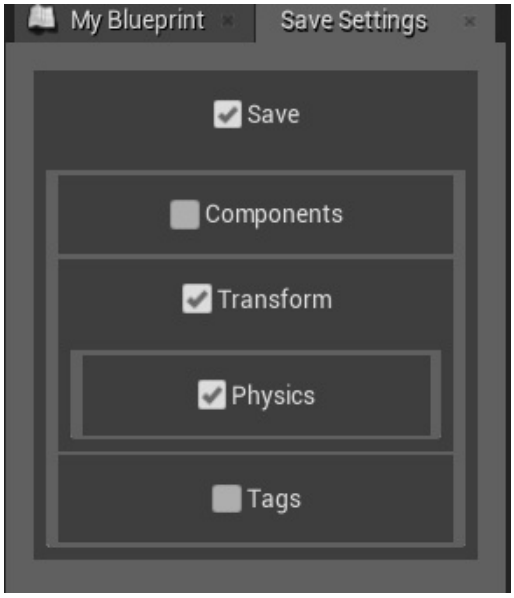
Each actor blueprint can have its own configuration which is edited directly inside the blueprint:





If you can't see *"Save Settings"* window opened it can be manually opened from **Window -> Save Settings**

Save settings



- **Save**: If false, this actor will be completely ignored while saving. *Disable this on all actor classes you don't want to save for performance.*
  - **Components**: Should components be considered for saving? *(Most components will still not be serialized for performance)*
  - **Transform**: Should save position, rotation and scale of this actor?
    - **Physics**: Should physics be saved? **Transform** is required to be enabled to save physics.
  - **Tags**: Should save actor tags?</img>

Level Streaming & World Composition

Save Extension support this engine features.

Saving will adapt, load and unload sublevels seamlessly so that is a sublevel is unloaded, its data is cached and if it gets loaded, its data gets restored.

This means sublevel data is still only saved when the game saves or loads, but their state is persistent in memory.

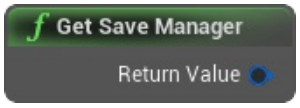
Saving & Loading

Early concepts

Save Manager

The save manager is in charge of loading, saving and all the rest of the logic (auto load, auto save, level streaming...). It will be initialized the first time **GetSaveManager** gets called. Usually at BeginPlay.

GetSaveManager



Slots

When we say **slot** we refer to the **integer that identifies a saved game**.

This slot identifies a saved game even during gameplay, meaning that if an slot gets loaded it will be "active" until we load another slot.

With this we can for example do "*SaveCurrentSlot*" to pick the current slot if any and save it.

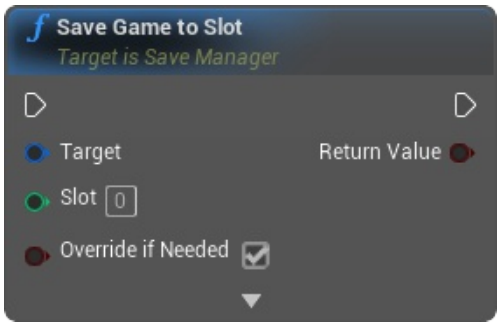
Auto-Save will also save the current slot, and **Auto-Load** will load the last current slot

Saving

To save a game you can just get the Save Manager and then call *SaveGame to Slot*

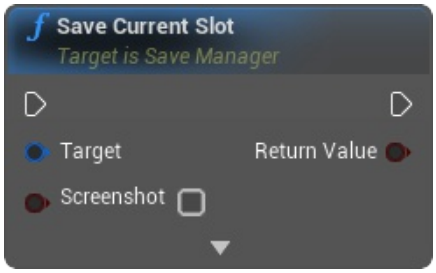
Save to Slot

Save into certain slot



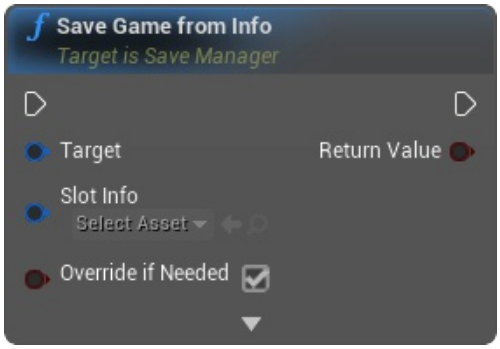
Save Current Slot

Save into the last slot that was loaded



Save Game from Info

Saves a game based on a Slot-Info (Check Slot Infos)



Slot Templates

Slot Info

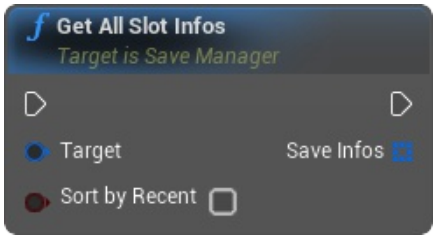
Slot Infos are SaveGame Objects used to identify a saved game. They hold all information that needs to be accessible while the game is **NOT** loaded.

It can for example hold the name of the game, the progress of the player, the current quest or the level.

Slot Infos are also used directly to load or save games. You can load all available infos and make the player decide which one to load from UI.

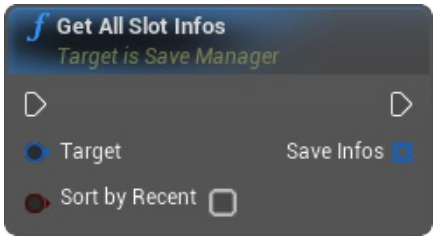
Get All Slot Infos

May be expensive, use with care



Get Slot

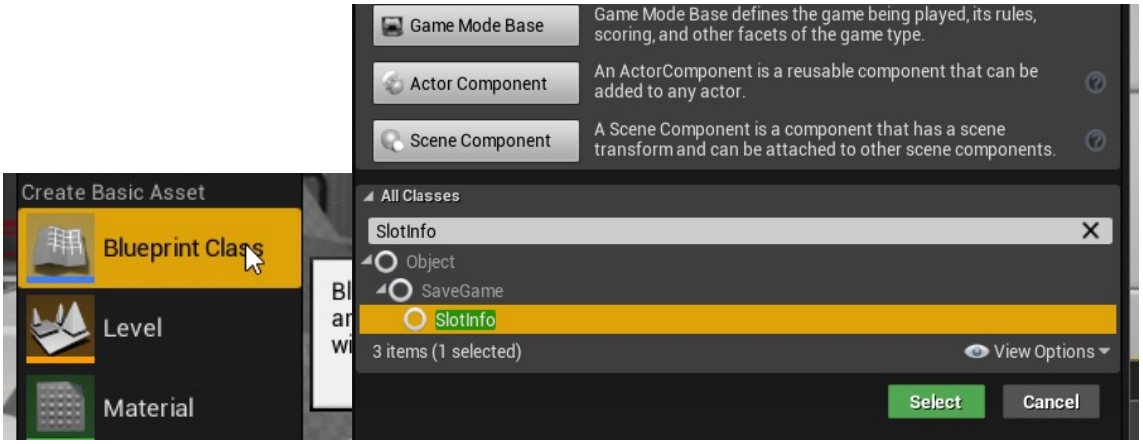
May be expensive, use with care



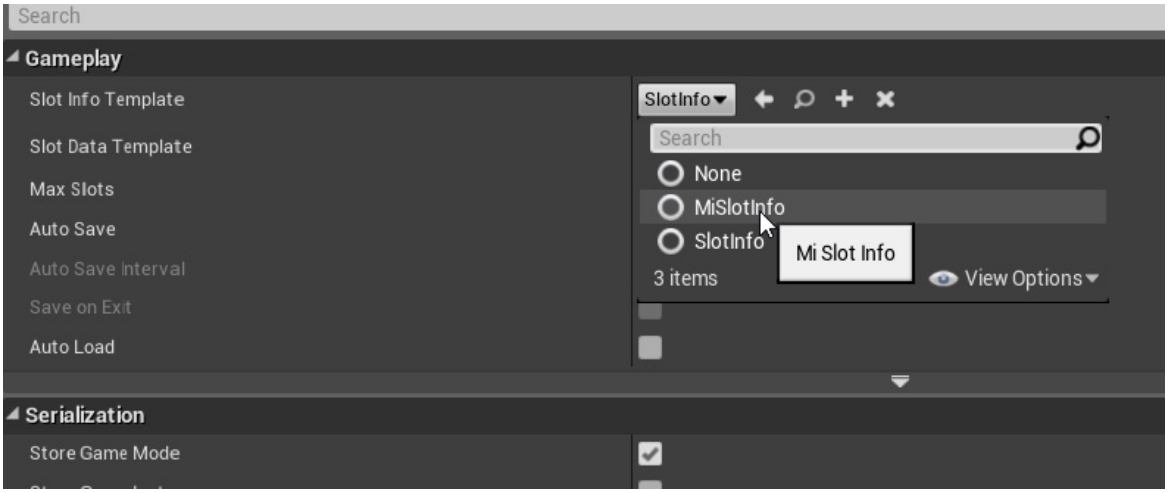
Custom Slot Infos

You can also create your own slot infos and store anything you want on them. Like a normal SlotInfo, your variables will be available when you load the slot (it will only need casting to your class).

1. Create a Blueprint child of *"SlotInfo"*



2. Assign it into your active Preset



Slot Data

Same as Slot Info objects, **SlotData** is a SaveGame Object, meaning all its properties will be saved.

Slot Datas are designed to hold all the information about the world and the player. All information that will be only accessible during gameplay.

Lifetime Component

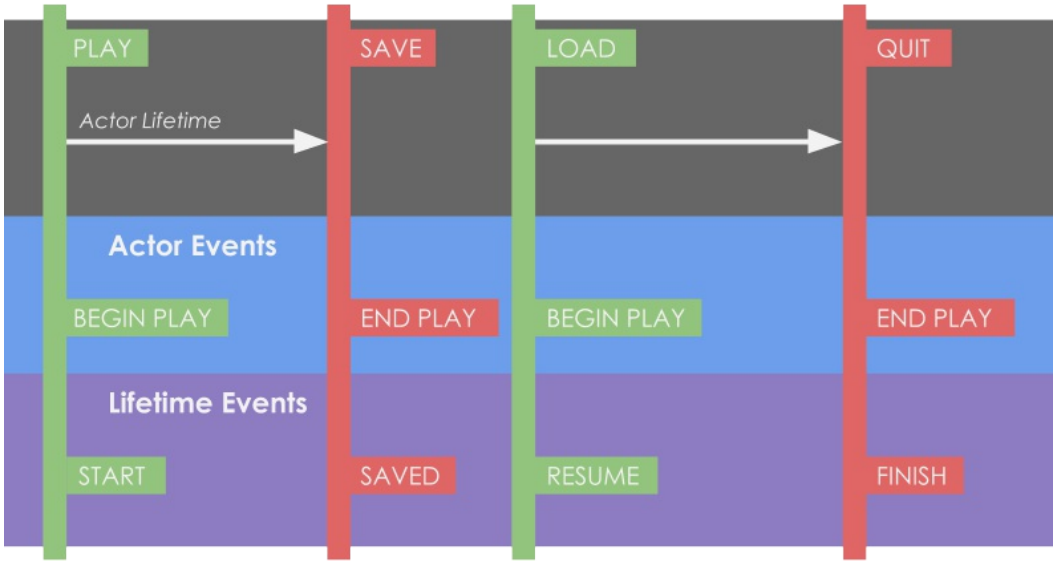
Lifetime components are an optional feature. Their single purpose is to offer **events called in relation to the saved lifetime** of the actor.

They are **not necessary** by any means to save your game since they only add some useful events.

Why can't I just use BeginPlay?

BeginPlay gets called every time game starts for that actor including when game starts, when you load, when the actor didn't exist...

It's not a good representative of the lifetime of the actor. That is why Lifetime events are called only during the actor's lifetime. It doesn't matter if it was loaded, saved, destroyed, etc. It is kind of more deterministic.



Events

Started

Gets called only first time an actor is created.

- When you start a new game
- When you spawn the actor
- Wont be called when you load any game (**Resume** will be called instead)

Saved

Called when this actor is saved

Resume

Called when this actor is loaded. When opening a saved game from any level in any situation

Finish

Similar to EndPlay, but gets called when this actor **gets destroyed during gameplay or at normal endplay**. But wont be called when you load a game and this actor gets destroyed as a consequence.