Em Item Plugin Guide

Version 1.0.0
10/21/2025
Etymorph Studios, LLC.

Working Unreal Engine Versions
5.3
5.4
5.5
5.6

Table of Contents

Features

- Item and Inventory Replication
- Flexible Subsystem with customizable Factory Objects
- Data Driven Item System
- Data Driven Inventory System
- Item Reflection System – AKA Item Spawning system.
- Item Lifecycle Management System
- UI Abstraction – Ready to use interfaces and widgets
- Drag & Drop Operations – Used with UI interactions
- Extensible Containers System – Allowing for customizable inventory systems.
- Save and Load System – Items and Inventories can be saved and loaded.

Plugin Code Objects

U Objects

* UEmInventoryContainer
* UEmItemInfoObj
* UEmItemParserBase
* UEmItemParser
* UEmItemSpawnerBase
* UEmItemSpawner
* UEmSaveLoadManagerBase
* UEmItemSaveLoadManager

Inventory Container Types

* UEmInventoryContainerHotBar
* UEmInventoryContainerSimple

Actor Components

* UEmHotBarManagerComponent
* UEmInventoryComponent
* UEmSpawnItemComponent
* UEmInventorySaveIdComponent

Data Asset

* UEmInventoryContainerAsset
* UEmInventoryContainerSimpleAsset
* UEmInventoryContainerHotBarAsset
* UEmInventoryAsset
* UEmItemAsset
* UEmItemsInWorldAsset

Actor

* AEmItem

World Subsystem

* UEmItemSubsystem

Developer Settings

* UEmItemDeveloperSettings

Interfaces

* UEmItemEntryWidgetInterface

User Widget

* UemInventoryListWidget

Drag Drop Operation

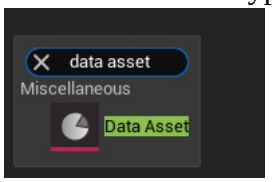
* UEmItemDragDropOperation

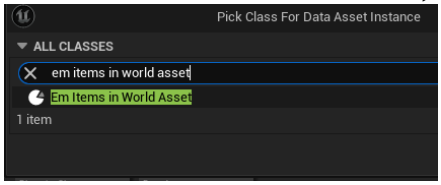
Blueprint Library

* UEmItemInventoryLibrary
* UitemDragDropLibrary

Prerequisite

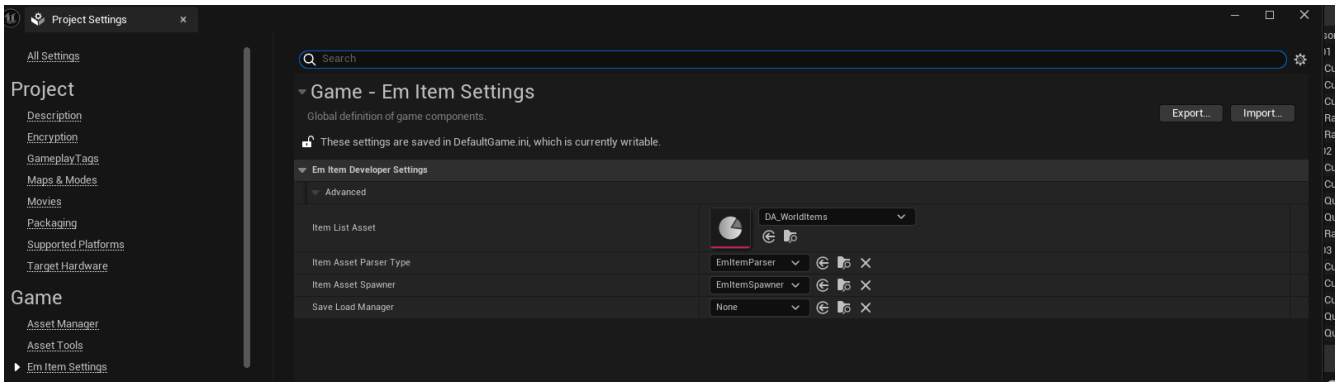
The Item World List requires a data asset to be created. This can be done by “right clicking” inside your folder viewer and typing in “data asset”. From the data asset viewer, type “em items in world asset”





Will come back to this after creating item definitions, but for now, we need to put this inside our project settings.

There are a few properties that must be set up prior to starting your game world. These properties can be found within the projects settings. Edit → Project Settings → Em Item Settings



From here, you must set up each property with some class (Save Load Manager is optional).

Item Life-Cycle States

```
/// Defines the state of the item. This state dictates collision, visibulity, and 3D scale.
UENUM(BlueprintType)
enum class EEmItemLifeCycleState : uint8
{
    /// The item is a physical actor in the world, visible and interactable.
    Dropped UMETA(DisplayName = "Dropped"),
    /// The item is in a container or in the player's inventory.
    InInventory UMETA(DisplayName = "In Inventory"),
    /// The item is attached to a character and being used.
    Equipped UMETA(DisplayName = "Equipped"),
    /// The item is attached to a character but not currently active (e.g., a weapon on the back).
    Holstered UMETA(DisplayName = "Holstered"),
    /// Up to another actor to handle collision,physics, and visuals
    Custom,
    Undefined
};
```

Items represented in the game can be represented by only 1 of the following states defined above. These states dictate there physics, collisions, visibility, size, location, and any other cosmetic information.

When deriving AEmItem the user should be aware of these states, as this could help speed along there development.

For items that are **InInventory**, **Equip**, and **Holstered**, they will be attached to the owner.

For items that are **Dropped**. They will exist with in the world, independently.

Custom – Requires implementation from the developer if special functionality is desired.

Note: Items that are **InInventory** may not have an instance object yet created. This is due to them not being physically represented. When the item does leave the inventory, then an item instance can be created.

Custom Asset Parsing

This plugin uses the Data driven techniques similar to that Lyra's plugin uses. However, the items and inventory containers gets created through a factory class called "**UEmItemParserBase**". There is an already derived type called "**UEmItemParser**" which parses the base types found inside this plugin.

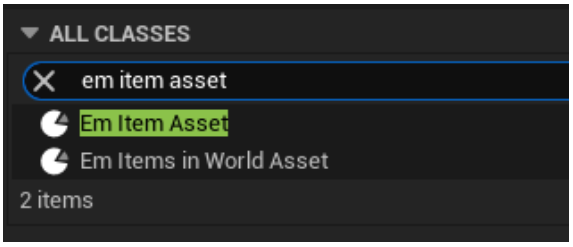
Note: Customizing the parser. Currently this must be done, within c++, as there are some low level details that get populated from source code.

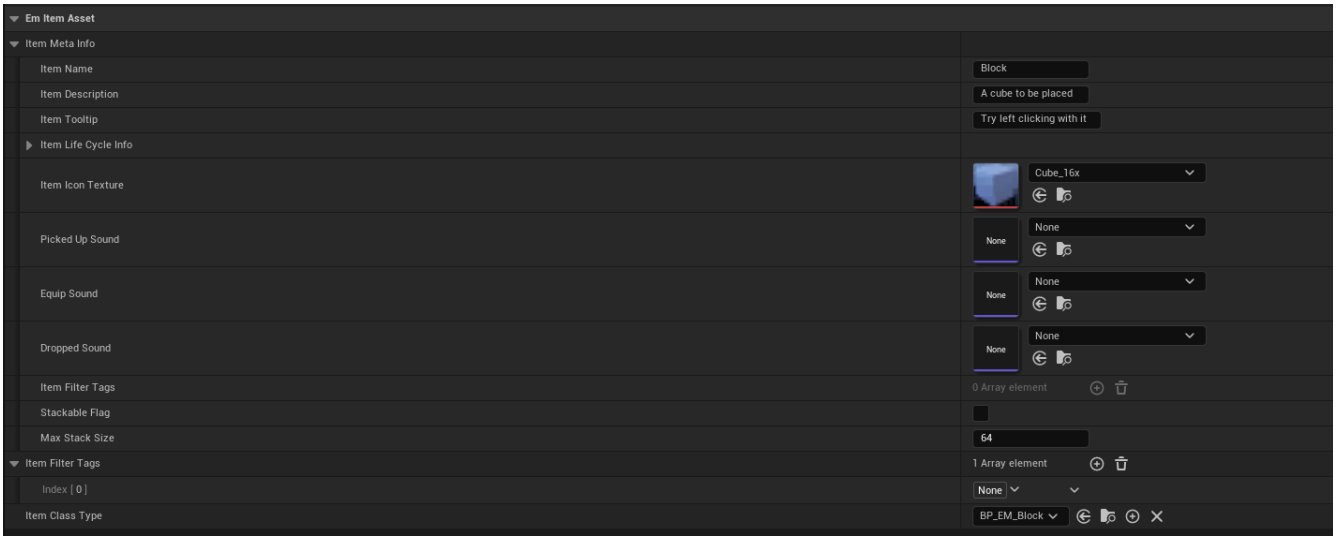
You should customize this parser when the following happens:
A: New **AEmItem** types are defined with their respective **UemItemAsset**.
B: New **UEmInventoryContainer** types are defined with their respective **UEmInventoryContainerAsset**

The Parse also contains the definition of asset IDs. This process needs to be deterministic, otherwise referencing certain objects will return unknown results.

Item Definitions

There are AemItems which is the class that will represents the item. However we can define different instances based on our **UemItemAsset.** To do this we can create Data assets within our folder.





Note: The base definition is kept thin to avoid bloat within the developer project. This plugin allows the control of additional properties to be define by deriving **UemItemAsset.** A simple example is modifying the mesh of the item.

Each attribute found within this editor contains tool-tips on there descriptions.

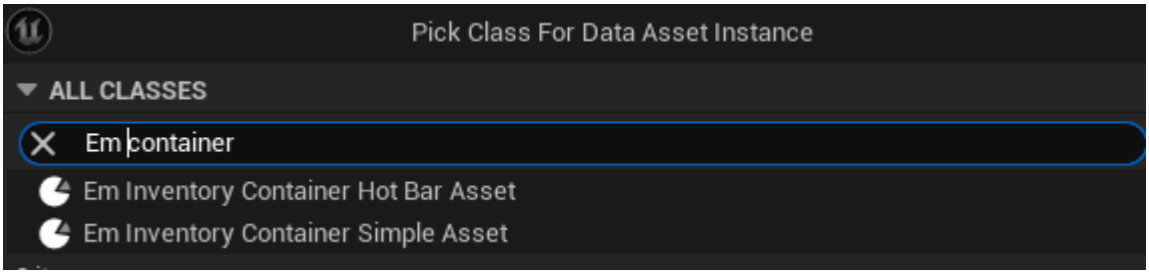
After completing a definition, you can apply it within the **“em items in world asset”** you created from earlier.

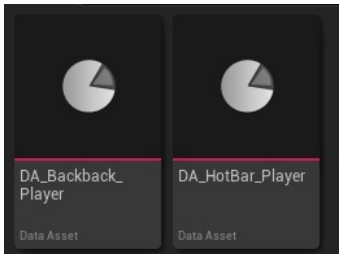
Inventory Definitions

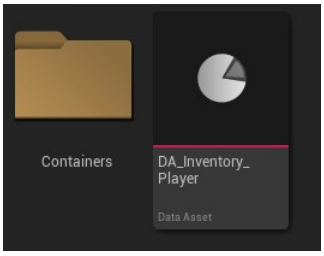
Inventory containers are the backbone on storing and interacting items within actors. Each container can define size, filters, and item lifecycles. These attributes can either be define by deriving from **“UEmInventoryContainer”** or by defining a Data Asset derived from the type “UemInventoryContainerAsset”. By default, there are 2 container assets that can be used

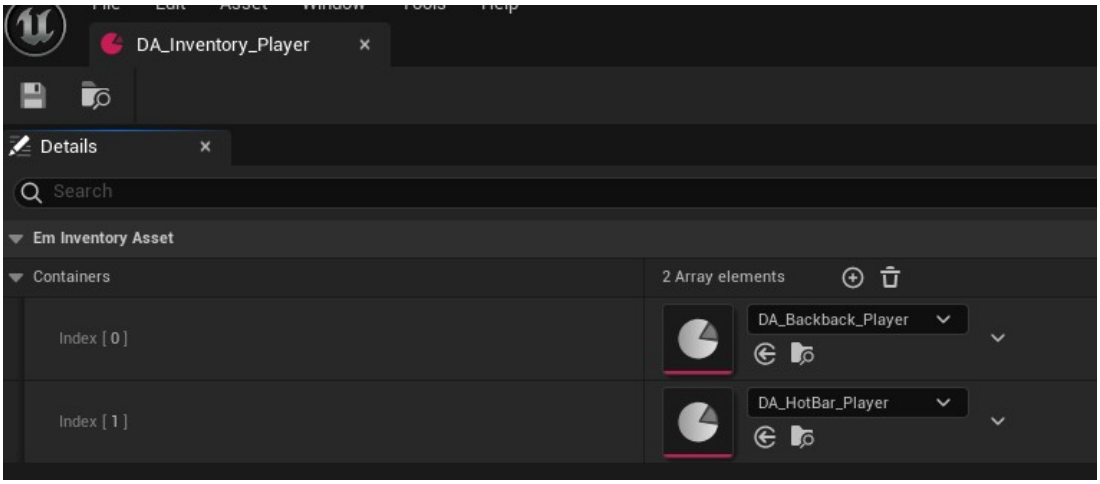
A: UEmInventoryContainerSimpleAsset
B: UEmInventoryContainerHotBarAsset

The simple container can be thought as just a backpack used in any sandbox game, or as armor/weapon slot, typically found in RPG’s.
The Hot Bar container is to its name. Think of this as the 9 slot hot bar found in games like Minecraft and ARK.
Lastly there is a “Inventory Asset”. This asset is a list of all desired containers. This asset needs to be placed within the **“UemInventoryComponent”.**









Note: New container types will need to derive their own data asset type from “ UemInventoryContainerAsset” and define it within the parser object.

Each property contains comments within the tool tips.

Customize Item Behavior

Custom Items can be derived from **“AEmItem”.** There are interfaces which can be define which allows for straight forward states. These states are define by the “EEmItemLifeCycleState”.

An important method to override for cosmetic updates is the **“OnLifeCycleChange”**.

Additionally, for games that would like to save Items, should derive “RetrieveItemModifiers” & “PopulateFromItemModifiers”. These methods are used to read and write from startup and startdown of game worlds. Examples on these modifiers can be found within the source code.

Customize Inventory Container

For example a simple inventory is good for storing items. However lets say we want a row of items to be used on the fly, but only 1 item can be equipped at a time. Well this is also possible to do while containing the simple inventory. The Hot Bar Container represents the row of items that can be equipped a single time.

This is just one example of many which can be created. For this plugin there are just the 2 stated above containers, however there are hooking points that can be derived from the base container type.

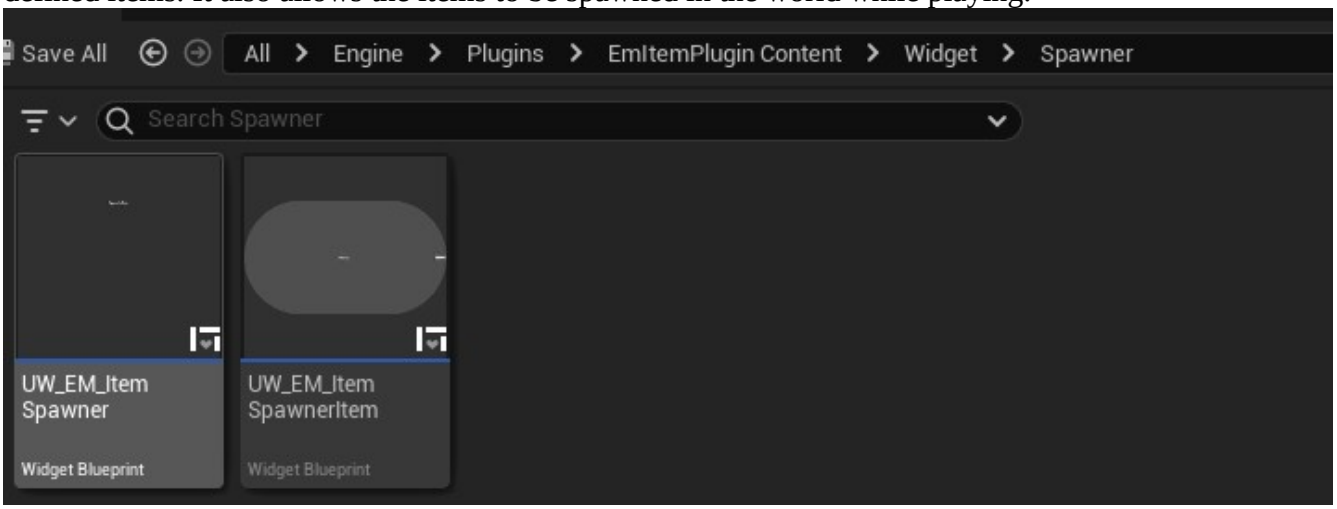
If custom containers are desired and derived. You might also want to create a component that manages it, as RPC (client to server) calls can’t be sent through the containers.

Spawning in Item in the Game World.

Spawning in items is straight forward with the Spawn Factor Object. It’s as simple as giving an Item Unique Id.

These “Item Unique Ids” can be found within the item themselves, the inventory information object, the data asset, or through the mapping definition found within the subsystem.

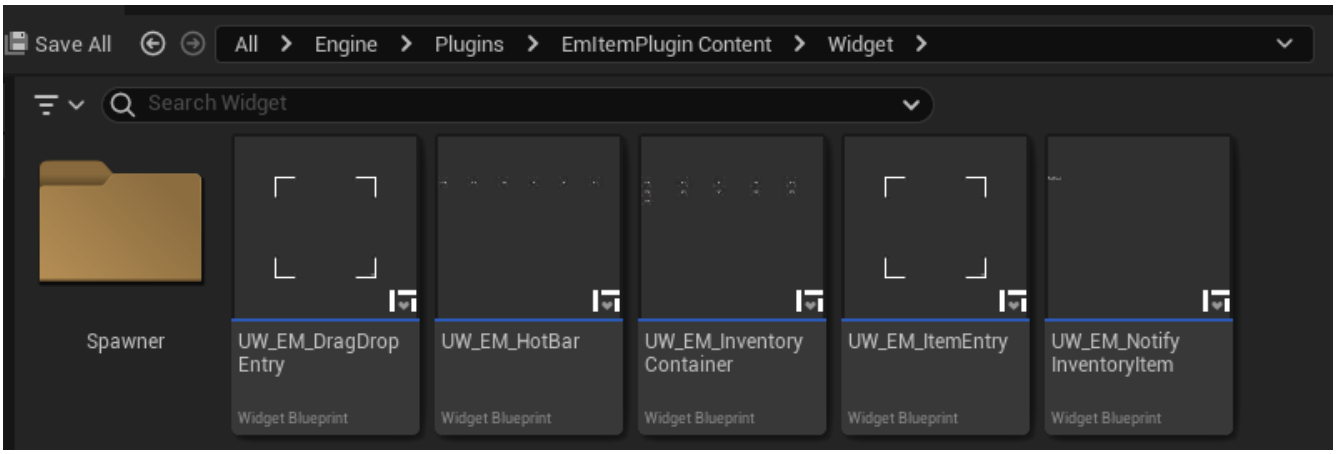
Tip: There is a build in widget called **“UW_EM_ItemSpawnerItem”**. This widget showcases all defined items. It also allows the items to be spawned in the world while playing.



UI

Classes – UemInventoryListWidget, UEmItemDragDropOperation, IemItemEntryWidgetInterface

Example on implementing them can be found within the plugin widget folder.



Save and Load Items.

This plugin also allows for Items and inventories to be saved and loaded between plays. There are predefined structures that contain the parsing needed to fully save and load the items and inventories. There is also example save classes that showcase on how to use the predefined structures.

Note: There is also a Save Load Manager, which can be derived and implement into any game. This feature is experimental and may not conflict with other save systems.

Items are straight forward to save, no required identify is needed to be saved with the current structures.
Inventory Components however takes some setup to be saved. These components require a **“SaveInstanceId”** which is used to identify its self within the save. Since there could be different chest and players in the world, this implementation is up to you, the developer.

Note: There is example code and objects that can be used, working outside the box. Look into **“UEmInventorySaveIdComponent”**.

Custom properties and attributes can be defined by **Modifiers.** Each class contains a **RetrieveItemModifiers** and **PopulateFromItemModifiers** interfaces.

Want to learn more?

Example Project: https://github.com/etymorph-studios/EmItemPluginExample