V1.0

Table of contents

Foreword

IMPORTANT

ADS Framework is a modular framework for Unreal Engine. It is not a complete game template.

A template usually provides a ready-to-use base with an already defined gameplay structure. This is useful for getting started quickly, but it can also limit the project to a specific logic, genre, or architecture.

A framework works differently. It provides reusable, configurable, and extensible tools designed to help you build your own game without imposing a specific type of gameplay. ADS Framework therefore does not replace the development that is specific to your project: it gives you a technical foundation to save time, structure your features, and avoid starting from scratch, while remaining reliable, secure, and performant. Each system is designed to integrate into all types of projects: action, adventure, RPG, sports, and more.

ADS Framework is developed and maintained with a commitment to continuous improvement. Despite the care given to development, documentation, and testing, it remains a technical product that may evolve based on user feedback and future versions of Unreal Engine.

If you encounter an issue or limitation, it is recommended that you contact support instead of leaving a negative review. Constructive feedback helps identify areas for improvement and allows the systems to evolve in the right direction.

ADS Framework Philosophy

ADS Framework is based on a simple idea: providing independent systems that can be used on their own, combined with each other, or combined with your own systems depending on your project needs.

Each Framework system can be integrated individually and does not require purchasing or using the other systems to work. You can therefore use it with your own architecture, your own Blueprints, your own interfaces, or other assets purchased on Fab.

If you choose to use several Framework systems, they share a common philosophy: modularity, clear logic, Blueprint integration, accessible configuration, and separation of responsibilities.

The systems are designed to be accessible to users who already have basic Unreal Engine and Blueprint knowledge without necessarily being advanced users. Some modules, while relatively simple to use, are still complex: it is therefore normal to allow a few days of acclimation to understand their logic, settings, and integration.

The included demos are intentionally lightweight. They use simple, readable logic to show how the systems work without unnecessarily increasing the project size. They are meant to demonstrate functionality, possibilities, and ease of use, without providing finalized visuals. This choice limits project weight and keeps the focus on the systems' logic.

C++ / Blueprints

ADS Framework systems are primarily developed in C++ to provide a stable, performant, structured foundation suitable for production use.

Each system, however, exposes a Blueprint layer designed to make integration into your projects easier. This approach lets you benefit from a C++ architecture while keeping usage accessible from the Unreal Engine editor.

Depending on the system, certain parts can be configured, called, or extended directly in Blueprints. Project-specific elements can therefore be created in Blueprints, or developed in C++ if you prefer to work with native logic.

The goal is to provide a balance between robustness, performance, accessibility, and freedom of adaptation.

Unreal Engine 6 Compatibility

ADS Framework is developed with a commitment to long-term maintenance and evolution.

Epic Games has presented Unreal Engine 6 as a major evolution of the Unreal Engine ecosystem, with a stronger role for Verse and the new Scene Graph gameplay framework. This evolution may gradually change how certain gameplay logic is developed in future versions of the engine.

With this in mind, ADS systems will continue to be maintained and adapted as much as possible to Unreal Engine evolutions. When the UE6 / Verse ecosystem is available, documented, and stable enough, the relevant Blueprint parts may gradually be adapted or replaced with solutions compatible with this new architecture.

The goal is to preserve the logic of ADS systems while supporting the evolution of the engine.

System Description

ADS Inventory System lets you create, organize, modify, and view inventories that can be used by a character or a container.

The system is designed to be flexible: it can be used for a shared global inventory and/or individual inventories, a slot-limited inventory, chest-style storage, or a more specific logic unique to your project.

Each item remains defined according to the needs of the game: consumable, resource, key item, currency, equipment, material, quest item, and so on. The plugin provides the general management structure, but your project remains responsible for the final rules: item usage, effects, UI, restrictions, pickup conditions, weight, rarity, equipment logic, or interactions with other systems.

ADS Inventory System can be used on its own, with your own architecture, or combined with other ADS Framework systems if your project requires it. It does not impose any combat, quest, merchant, equipment, or save system.

Installation

The plugin must be installed in the project. Do not use the version installed in the engine, because it will need to contain assets and configurations specific to the project.

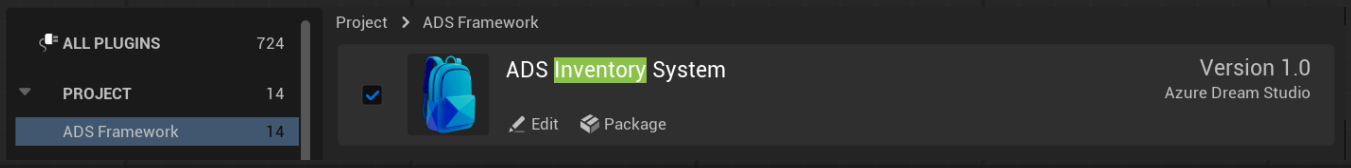
1- After installing the plugin in the engine through the Epic Games Launcher, close Unreal Engine and, from Windows Explorer, copy the plugin folder located in the engine plugins folder to the project Plugins folder.

"<UE installation folder>/Engine/Plugins/<ADS folder plugin>"

"<Project folder>/Plugins/ "

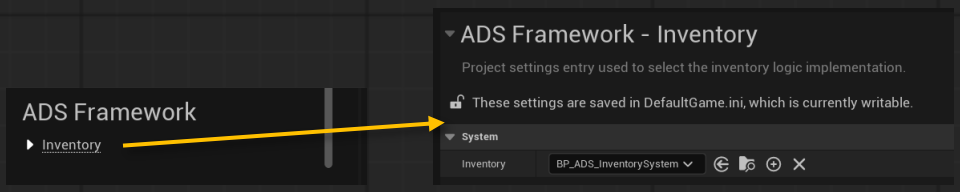
2- Open Unreal Engine.
(If Unreal Engine asks to compile the plugin on startup, accept.)

3- Enable the plugin located under "PROJECT/ADS Framework", then restart.



The plugin can be installed both in the engine and in the project. If this is the case, since there will be two versions of the same plugin, Unreal Engine will use the one located in the project. After copying the plugin into the project, you may delete it from the engine if you wish. In all cases, you must always use a version installed in the project and not in the engine.

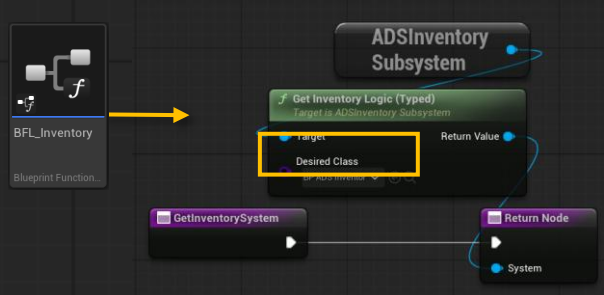
4- In "Project Settings", define the logic class that will be used by the project.



BP_ADS_InventorySystem should be assigned automatically. If it is not, assign it manually.

BP_ADS_InventorySystem is the recommended class and is sufficient for more than 95% of projects. However, it is possible to create a custom class. If you do so, remember to assign it here. (See the "Logic Class" section.)

5- Define the logic class in the Get Inventory System function located in BFL_Inventory.



You must assign the same class as the one defined in Project Settings.

How It Works

ADS Inventory System is based on central logic managed by a GameInstance Subsystem. This Subsystem creates and keeps the instance of the inventory system logic class used by the project. This central logic makes it possible to add items, retrieve inventory contents, manage global inventories, per-character inventories, storage, limits, and update notifications.

For actors that need their own inventory, the system uses an Inventory Actor Component that must be added to the character or actor concerned. This component registers the character with the central system and associates a main inventory and a storage inventory with it. Depending on the project configuration, the inventory can be shared globally by all characters or managed separately for each character.

Items are defined in a Data Table. Each row in this Data Table represents an item and serves as the main reference for retrieving its information: name, description, icon, type, rarity, maximum ownable quantity, stats, associated effects, and optional visual representation in the world. The inventory therefore does not duplicate the full item definition each time: it mainly stores information related to item ownership, such as owned quantity, acquisition date, the "new", "protected", "locked", or "pinned" state, then retrieves the fixed information from the Data Table.

Item effects are developed in dedicated Blueprints. One effect corresponds to one Blueprint. If an item must apply several effects, it references several effect Blueprints. For example, if an item must restore 50 HP and 10 MP, it can use one effect Blueprint to restore 50 HP and another effect Blueprint to restore 10 MP. This separation makes it easy to reuse the same effects across several different items without recreating the same logic each time.

Not all items need to physically exist in the world. An item can simply exist in the inventory without a visual representation. For items that need to be visible or pickupable in the level, the Data Table allows you to assign a dedicated visual Blueprint. The system also provides base Blueprints for representing an item in the world and handling its addition to the inventory when it is picked up.

The system distinguishes between the main inventory and storage. The main inventory can be limited by number of entries to create gameplay constraints, or it can be unlimited. Storage is intended to act as a reserve and can be used for chest logic, global storage, or character-specific storage. This makes it possible to adapt the system to several types of games: RPG-style shared inventory, individual per-character inventory, separate storage, shared chest, and so on.

The system also provides inventory sorting and view logic. Items can be prepared as data usable by a user interface, then sorted by different criteria such as name, acquisition date, type, rarity, or quantity. Interfaces remain specific to your project, but the system provides the data needed to build inventory menus, item lists, chests, or selection screens.

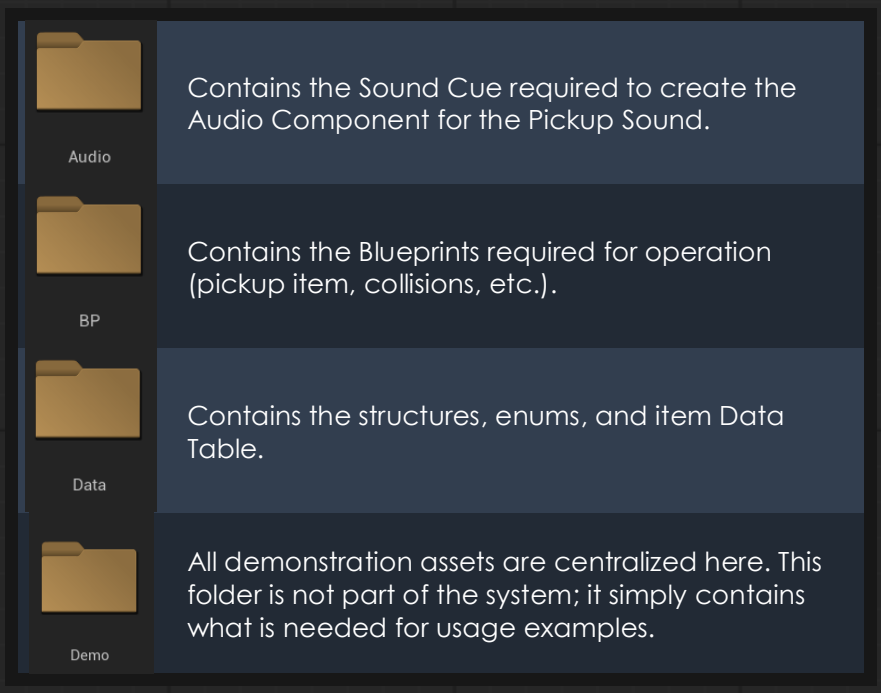
Event Dispatchers make it possible to react to important changes: item added, inventory updated, storage updated, limit reached, maximum quantity reached, new item obtained, and more. These events can be used to update an interface, play a sound, display a notification, trigger a save, or connect the inventory to another project system.

This architecture clearly separates responsibilities. The Data Table describes the items, the effect Blueprints describe what the items do, the visual Blueprints describe their optional appearance in the world, the Actor Component links a character to the system, and the Subsystem centralizes the global logic. This separation makes the system easier to maintain, easier to extend, and safer to adapt to each project's specific needs.

Plugin Contents

This documentation is intended to be a user guide. C++ classes are therefore not detailed in this documentation. They are only mentioned when they serve as a parent class or when they must be called in Blueprint. This system is developed in a hybrid C++/Blueprint way. This documentation focuses on integrating and using the system.

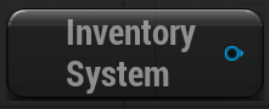
The plugin has its own 'Content' folder for Blueprint assets.



Detailed Usage

GameInstance Subsystem

It is created automatically by Unreal Engine with the GameInstance and must not be placed manually in the level. Its role is to create the central logic instance defined in Project Settings, then provide access to this instance for the entire lifetime of the GameInstance.



No asset needs to be created for the Subsystem. It can be retrieved through the Get Inventory System function provided by BFL_Inventory.

Parent Logic Class

ADSInventoryLogicBase serves as the parent class for inventory logic. BP_ADS_InventorySystem, which is included with the plugin, already inherits from it. You do not need to create another child class for a normal integration unless you want to create custom central logic. Otherwise, BP_ADS_InventorySystem is sufficient.

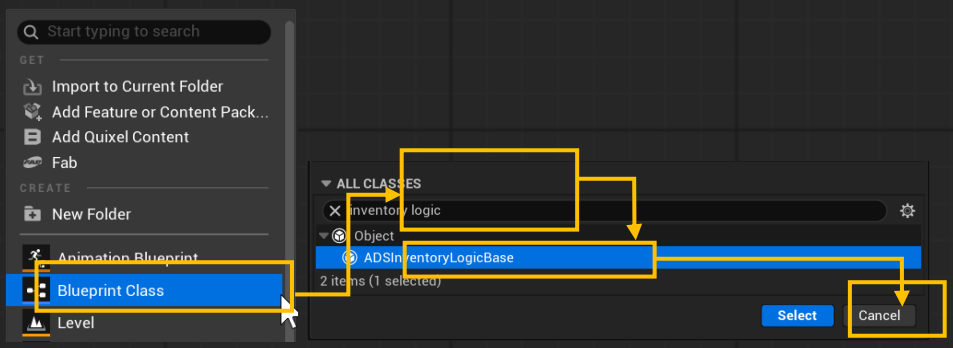
Do not forget to define the class that will be used in Project Settings.

Also do not forget to edit the Get Inventory System function in BFL_Inventory with the class being used.

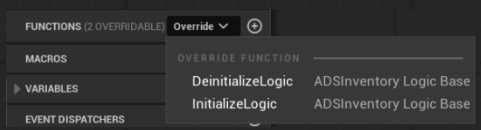
Creating a Logic Class

It is not necessarily required to create a logic class, because the included BP_ADS_InventorySystem is sufficient in most cases. That said, if you have project-specific needs, you can do it as follows:

In the Content Browser -> Right-click -> Blueprint Class -> ADSInventoryLogicBase



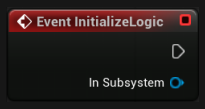
Once the asset is created, it must be developed. You have 2 functions to override that will be called automatically by the Subsystem during logic initialization and deinitialization.



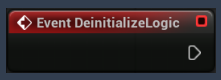
BP_ADS_InventorySystem contains all the inventory management logic. If you create your own class, you will then need to develop all the inventory management logic yourself. This will probably be a lot of work.

If you need to create your own class, it is recommended that you duplicate BP_ADS_InventorySystem and modify it for your project-specific needs instead of creating a class from scratch.

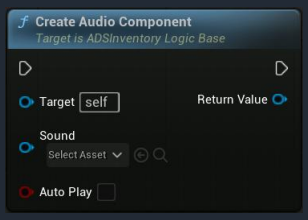
Base Logic Functions



Called automatically by the Subsystem after the logic is created. The In Subsystem input contains the reference to the Subsystem that owns this instance. Override only if custom logic needs to initialize data at startup.



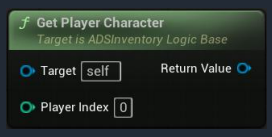
Called automatically when the logic is destroyed. Use this in custom logic to clean up references or stop persistent processing.



Creates a persistent Audio Component from a SoundBase. The Sound input defines the sound to use. The bAutoPlay input indicates whether the sound should be played immediately. The output returns the created Audio Component. Useful for integrating a Pickup Sound into the system.

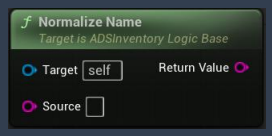
Spawns an Actor from a class and a Transform. The Owner and Instigator inputs can be provided if needed. The output returns the created Actor.

This function is equivalent to the native function of the same name. Since this class inherits from UObject, it does not access the native function, which is not available from a UObject. This function was therefore recreated in C++ in the parent logic class to allow spawning.

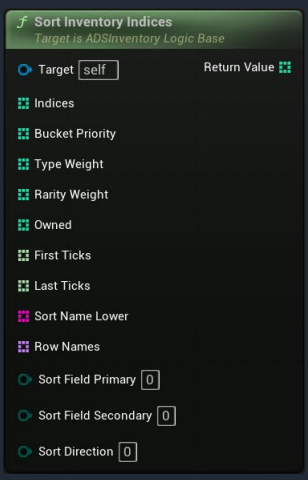


Returns the player Character corresponding to the PlayerIndex input. By default, PlayerIndex is 0.

This function is equivalent to the native function of the same name. Since this class inherits from UObject, it does not access the native function, which is not available from a UObject. This function was therefore recreated in C++ in the parent logic class so it can be used from the logic.



Returns a normalized version of text for alphabetical sorting: lowercase, removal of some leading articles, and accent normalization.



Advanced sorting utility function. It receives the sort key arrays, the primary and secondary sort fields, and the sort direction. It returns the sorted indices. In standard use, it is better to use SortInventory or GetInventoryView in BP_ADS_InventorySystem.

BP_ADS_InventorySystem

This class contains the central inventory management system. It was developed in BP so it can be easily modified if needed. In the intended use of this system, Blueprint logic is sufficient for most projects and has the advantage of remaining easy to modify from the editor. That said, for 95% of projects, it will not be necessary to edit it, except to configure the system. This section is useful only if you are using this logic and not fully custom logic.

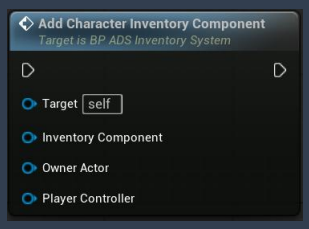
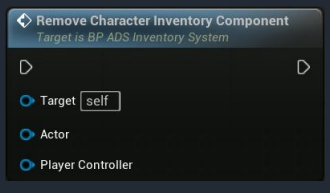
Configuring the System

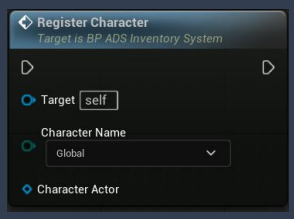
Some variables also serve as system parameters. They are organized under the "Parameters", "Private", and "Sort" categories.

MainInventoryMode	Defines the main inventory mode. Global means the main inventory is shared. By character means each registered character has its own main inventory.
StorageInventoryMode	Defines the storage mode. Global means storage is shared. By character means each registered character has its own storage.
GlobalSlotsLimit	Limits the number of entries in the global main inventory. The value -1 means no limit. This limit applies to the main inventory, not storage.
StorageAccessibility	Indicates whether storage is accessible. This value can be modified at runtime with SetStorageAccessibility.
PickupSound	Sound played on pickup when the add functions receive bPlayPickupSound set to True.
OrderTypes	Order used to sort item types. If E_ItemTypes is modified, this array must remain consistent with the enum values.
OrderRarities	Order used to sort rarities. If E_ItemRarity is modified, this array must remain consistent with the enum values.
MainInventoryGlobal	Global main inventory. Modify it through the system functions rather than directly, unless assigning a default inventory.
StorageInventoryGlobal	Global storage. Modify it through the system functions rather than directly, unless assigning a default inventory.

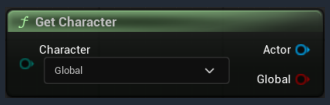
BP_ADS_InventorySystem Functions

Character Registration

	Registers an AC_Inventory in the central system. This function receives the character's logical name and the component reference. It is called automatically by AC_Inventory at BeginPlay. Call it manually only in a highly customized integration.
	Removes an AC_Inventory from the registry. It is called automatically by AC_Inventory at EndPlay. Call it manually only if you manage component registration yourself.



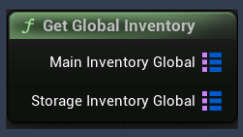
Prepares or registers a character in the system structures. This function is mainly used by the central logic and by AC_Inventory.



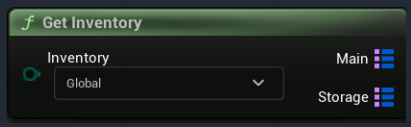
Retrieves the actor associated with a character name registered in the system. This function is useful when you need to find the character corresponding to a logical entry, for example PlayerCharacter or another name defined in the enum used by the project.

It does not return the character inventory. To retrieve the contents of a character inventory or storage, use GetCharacterInventory or GetCharacterStorage.

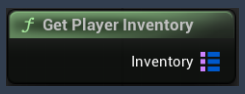
Retrieving Inventories and Storage



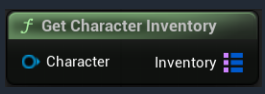
Returns the system's global data, meaning the global main inventory and global storage. Use this when inventory or storage modes are configured as Global, or when a UI needs to display shared data.



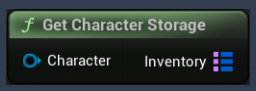
Returns the main inventory and storage Maps corresponding to the requested target, taking Main Inventory Mode and Storage Inventory Mode into account. This function serves as a general access point when a UI or gameplay logic needs to retrieve both data sets.



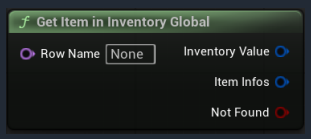
Retrieves the inventory linked to a player from their Player Controller or controlled pawn according to the system logic. Useful for projects centered on the PlayerCharacter.



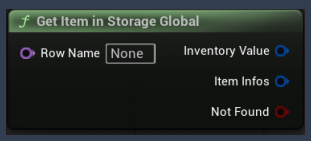
Returns the main inventory of a registered character. The main input is the character reference. The output is the item Map of the main inventory.



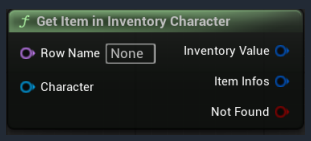
Returns the storage of a registered character. The main input is the character reference. The output is the item Map of the storage.



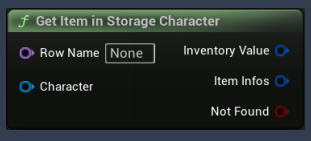
Searches for an item in the global main inventory. The function returns the item's inventory information, the item's fixed information from DT_Items, and a Not Found indication if the item was not found.



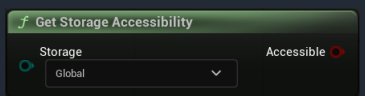
Searches for an item in global storage. The function returns the item's inventory information, the item's fixed information from DT_Items, and a Not Found indication if the item was not found.



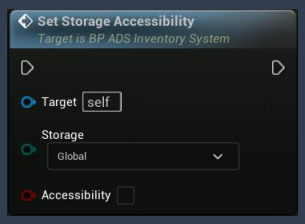
Searches for an item in the main inventory of a registered character. If the character is not in the list, the function cannot return a usable result.



Searches for an item in the storage of a registered character. If the character is not in the list, the function cannot return a usable result.

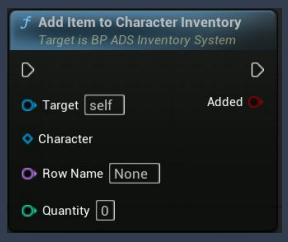


Returns the accessibility state of the specified storage.

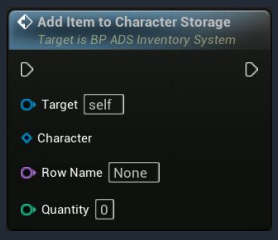


Changes the accessibility state of the specified storage. The Accessibility input defines the new value. The function triggers OnStorageAccessibilityChange with the new state to notify UIs or systems listening to this information.

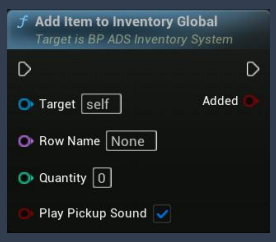
Adding Items



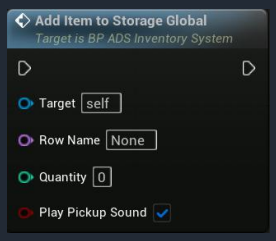
Adds a quantity of an item to a character's main inventory. It receives the Character reference, Row Name, and Quantity. The Added output indicates whether the add operation succeeded.



Adds a quantity of an item to a character's storage. Use this when storage is linked to a registered character. The function follows the same add logic as the main inventory, but targets storage.

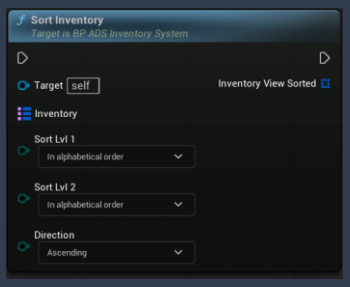


Adds a quantity of an item to the global main inventory. Use this when the main inventory is shared or when an external system needs to add directly to the global inventory. The Added output indicates whether the add operation succeeded.



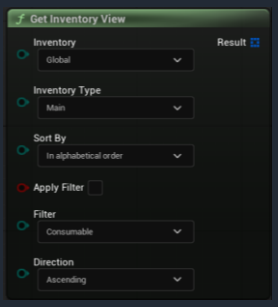
Adds a quantity of an item to global storage. Use this when storage is shared or when a global chest must be filled directly. The Added output indicates whether the add operation succeeded.

Sorting

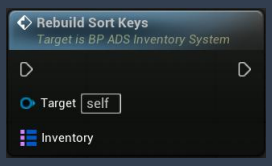


Sorts inventory data according to two sort levels and a direction. It receives Sort Lvl 1, Sort Lvl 2, and Direction. The result is a sorted list of S_ItemView.

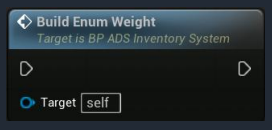
Generally useful for displaying a sorted inventory in a menu.



Prepares a list of items for a UI or menu. The function lets you choose the inventory type to display, the sorting criterion, and optionally a filter. It returns a list of S_ItemView.

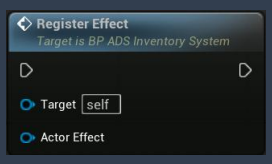


Rebuilds the internal data used by sorting. This function is mainly an internal preparation function. It may be useful if you significantly modify sorting data in a Blueprint child.

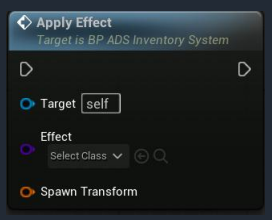


Rebuilds the weights used to sort item types and rarities. Use only if you customize the enums and the Order Types / Order Rarities arrays.

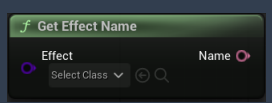
Effects and Sound



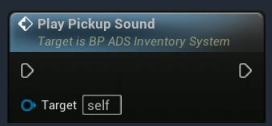
Registers an effect instance in the central logic. This function receives the Actor Effect. It is called by BP_ItemEffectParent so available effects can be found and applied.



Applies an item effect from an effect class and a spawn Transform. The logic finds the relevant effect, spawns it if necessary, then triggers its Apply event.

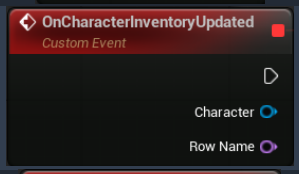
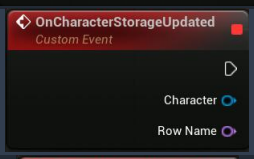
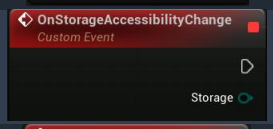
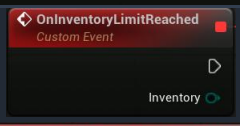


Retrieves the display name associated with an effect. This function relies on the registered effect information.



Plays the pickup sound configured in the Pickup Sound variable.

BP_ADS_InventorySystem Event Dispatchers

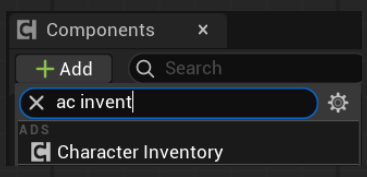
	Triggered when a new item is added to a main inventory.
	Triggered when an item reaches its maximum allowed quantity in a main inventory.
	Triggered when a new item is added to storage.
	Triggered when an item reaches its maximum allowed quantity in storage.
	Triggered when a character main inventory is modified.
	Triggered when the global main inventory is modified.
	Triggered when a character storage is modified.
	Triggered when global storage is modified.
	Triggered when storage accessibility changes. The new value is passed to the dispatcher.
	Triggered when the main inventory slot limit is reached.
	Triggered when adding an item fails because a limit prevents the add operation, especially when the maximum allowed quantity for the item has been reached.

Actor Component: AC_Inventory

AC_Inventory is the Actor Component to add to a character or actor that must have its own inventory. At BeginPlay, it retrieves the system with Get Inventory System, registers itself with BP_ADS_InventorySystem, registers the character if necessary, then adds the initial items configured in the component.

Adding the Component

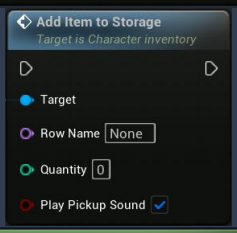
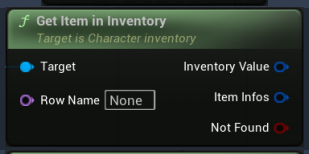
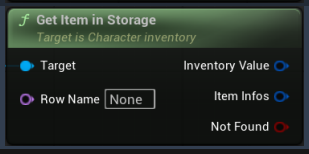
- Open the Blueprint of the actor that must have its own inventory.
- Add Character Inventory.



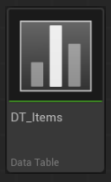
Configuring the Component

Character Inventory Initial	Defines an initial inventory for this character.
Storage Inventory Initial	Defines initial storage for this character.
Player Name	Defines an ID for this character. Select it from the list. This list (E_InventoriesNames) must be edited to match your project. Each character must have a different name.
Global Slots Limit	Maximum number of entries in the main inventory managed by the component. The value -1 means no limit. Storage remains unlimited in number of entries.

Component Functions

	Returns the component's two Maps: CharacterInventory and StorageInventory.
	Adds a quantity of an item to the component's main inventory. It receives Row Name, Quantity, and Play Pickup Sound. The Added output indicates whether the add operation succeeded.
	Adds a quantity of an item to the component's storage. It follows the same logic as AddItemToCharacter, but the add operation targets StorageInventory.
	Searches for an item in the component's main inventory. The function returns ownership information, the item's fixed information defined in DT_Items, and a Not Found indication if the item was not found.
	Searches for an item in the component's storage. The function returns ownership information, the item's fixed information from DT_Items, and a Not Found indication if the item was not found.

Data Table



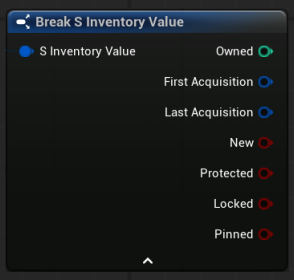
DT_Items is the main Data Table of the system. Each row represents an item. The row Row Name is used as a technical identifier: this Row Name is used by add, read, and pickup functions. It should be treated as an ItemID.

This is the Data Table that must be edited for the project's items.

Row Structure

Icon	Texture used to display the item in the interface.
Name	Name displayed to the user.
Description	Description displayed in the interface or menus.
Visual_BP	Actor added to the Child Actor Component of BP_Item to visually represent the item in the world. Not all items necessarily need to be placed in the world. It is therefore not necessary to create a Visual BP for every item.
Limit	Maximum ownable quantity for this item.
▶ ItemType	Item type defined by E_ItemTypes. E_ItemTypes must therefore be edited to include the types corresponding to the project.
Rarity	Item rarity defined by E_ItemRarity. E_ItemRarity must therefore be edited to include the rarities corresponding to the project.
▶ Effects	List of effect classes to apply when the item is used. One effect corresponds to one child Blueprint of BP_ItemEffectParent.
Statistics	List of item stats, if any. For example, a weapon may have stats while a potion may not. This Map uses an E_ItemStats key and a float value. E_ItemStats must therefore be edited to define all stat names corresponding to the project.
OptionalScript	Optional Map for custom item information. The key is a Name describing the information role, such as EquipmentSet, MenuAction, CraftingFamily, or QuestID. The value is free text read by your project Blueprints. The inventory system does not trigger this logic automatically. It only stores the data. To use it, a project Blueprint must read the Map and apply its own logic. Example: a helmet, armor, and sword can all use EquipmentSet = Knight. An equipment system can read this to detect a matching set and apply a bonus.

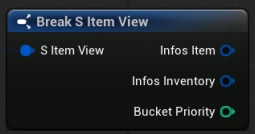
Inventory Structure



S_InventoryValue stores the state of an item in an inventory. This structure does not describe the item itself: it describes what the player owns and the state of the inventory entry.

Owned	Owned quantity.
FirstAcquisition	Date when the item was first acquired.
LastAcquisition	Date when the item was last acquired.
bNew	Indicates that the item is new. This value can be used by the UI to display an icon or marker until the item has been viewed.
bProtected	Indicates that the item must be protected against certain losses. Example: preventing the sale or deletion of a key item.
bLocked	Indicates that the item is locked, for example to prevent it from being used while keeping it visible.
bPinned	Indicates that the item is pinned, which can be used by the UI or sorting to highlight it.

Item View Structure



S_ItemView provides interfaces with already usable data. It combines fixed information from DT_Items and ownership information from the inventory.

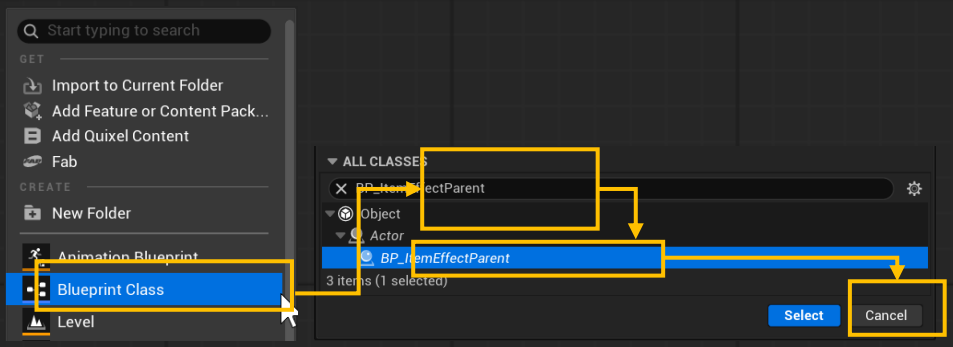
Item Info	S_Item structure containing the fixed item information from DT_Items.
Inventory Info	S_InventoryValue structure containing the item ownership state: quantity, dates, new, protected, locked, pinned.
Bucket Priority	Group priority used by sorting. 0 = Pinned, 1 = Equipped, 2 = New, 3 = Others.

Item Effect

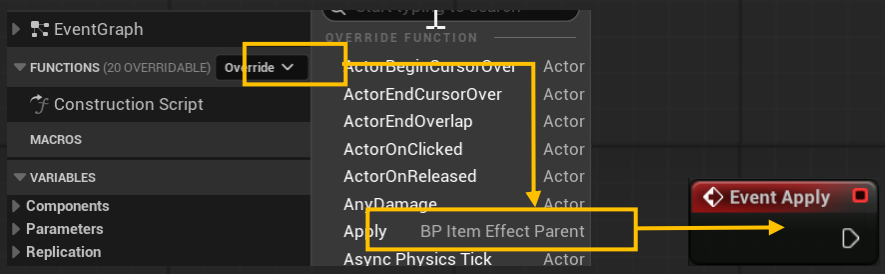
BP_ItemEffectParent is the parent to use for creating item effects. One effect corresponds to one Blueprint. If an item must produce several effects, it references several effect Blueprints in DT_Items.

Creating an Effect Blueprint

In the Content Browser -> Right-click -> Blueprint Class -> BP_ItemEffectParent

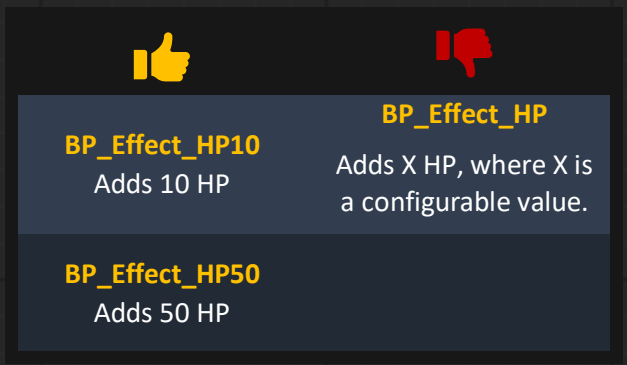


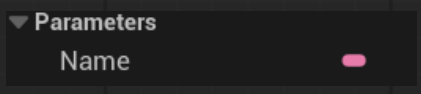
Then open the Blueprint and, in the Event Graph, override the "Apply" function.



The Apply function is the event to override to define the effect behavior. It will be called by the system when the effect must be applied. This is where you develop the effect logic, for example adding 50 HP to the character.

It is recommended to create one BP per effect and not a parameterized BP.





You can also assign a name to this effect through its Name variable so it can be displayed in a UI through the central logic's GetEffectName function.

Pickup Item

BP_Item is the Actor Blueprint intended for placing a pickup item in the world. It reads DT_Items, prepares the visual if a Visual_BP is defined, then adds the item to the target inventory or storage when its OnOverlap event is triggered.

Do not create a child of this Blueprint for each pickup item.

BP_Item is sufficient for all pickup items in the project because each instance can be configured. The visual is applied automatically based on what is defined in DT_Items.

BP_Item is therefore the actor to place in the world. Once placed in the world, simply configure the instance through its Details panel.

Destination	Indicates where the item should be added. Global targets the global inventory or storage. A character entry targets the corresponding character.
Inventory	Target inventory type: Main for the main inventory, Storage for storage.
Row Names	List of possible item Row Names. If the list contains a single entry, the pickup item will always give that item. If it contains multiple entries, the item can be chosen randomly. If it is empty, the selection can be made from all rows in DT_Items.
▶ Quantity	Quantity range. The final quantity is chosen randomly between X (minimum) and Y (maximum). For a fixed quantity, set the same value for X and Y.

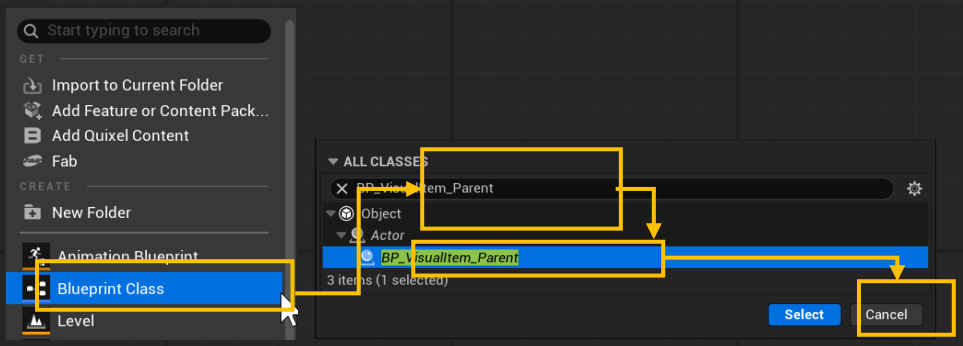
BP_Item does not manage collision directly. It relies on the collision defined in the visual representation Blueprint. Your representation BP must therefore also contain the collision required for pickup.

World Item Visual

BP_VisualItem_Parent is the parent intended for creating item visuals in the world. It does not contain the inventory logic itself. This separation keeps pickup logic in BP_Item while allowing each item to have its own visual. It therefore manages only its visual appearance in the world.

Creating a Visual Item Blueprint

Dans le Content Browser -> Clic-Droit -> Blueprint Class -> BP_VisualItem_Parent



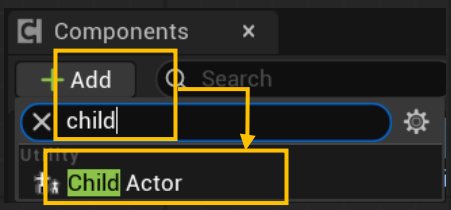
Add the required components (Mesh, particles, etc.).

For this item to be pickupable, collision must be added in this visual representation BP. BP_Item does not define the collision; the child of BP_VisualItem_Parent manages the collision so it matches the visual.

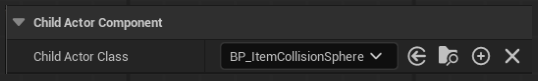
Collision

The BP_ItemCollisionBox, BP_ItemCollisionCapsule, and BP_ItemCollisionSphere Blueprints provide collision bases for detecting item pickup. The choice depends on the object shape and the desired precision.

To add collision, open the Blueprint of an item visual and add a Child Actor Component.



Then select this Child Actor Component to assign its class in the Details panel.



Choose from BP_ItemCollisionBox, BP_ItemCollisionCapsule, and BP_ItemCollisionSphere, then adjust its transform to match the item.

That is all. There is no need to manage overlap. The assigned collision handles it automatically. It communicates with the relevant BP_Item instance and automates the process. You only need to add the collision.

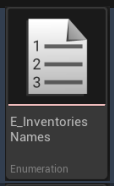
Blueprint Interface

BPI_Item standardizes the OnOverlap event for items. BP_Item already implements it. If you create your own pickup Actor without inheriting from BP_Item, it is recommended that you implement this interface to remain compatible with the provided collision logic.

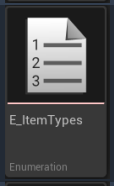
Enums to Edit

Enums are used to configure inventory names, inventory modes, item types, rarities, stats, and sorting. Some can be adapted to the project, but reserved entries must be kept and sorting-related variables must be updated when necessary.

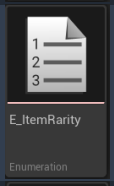
Here is the list of enums to edit so they match the project:



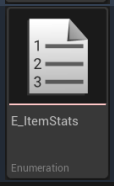
Contains the logical names of inventories or characters. The included values are PlayerCharacter, Character1_Demo, Character2_Demo, and Global. Global is a reserved entry that must not be removed.



Defines item types: Consumable, Key Item, Accessory, Armor, and Weapon. If this enum is modified, check Order Types in BP_ADS_InventorySystem.



Defines rarities: Common, Rare, and Legendary. If this enum is modified, check Order Rarities in BP_ADS_InventorySystem.



Defines possible item stats, including Physical Attack, Physical Defense, Fire Attack, Fire Defense, Ice Attack, Ice Defense, Lightning Attack, and Lightning Defense. The (Choose a stat) entry is marked as an entry that must not be removed.

Do not edit the other enums, because they are used by the plugin with those exact values. The enums listed above are used with project-specific values.

What the System Does Not Do

ADS Inventory System provides a complete architecture for managing inventories, storage, items, quantities, limits, effects, and data usable by a user interface. However, it does not automatically create all the gameplay specific to your project.

The system does not provide a finalized, publish-ready inventory with all menus, visuals, and game design rules already defined. User interfaces, menu animations, sounds, notifications, usage confirmations, error messages, and visual feedback must be adapted to your project.

The plugin does not decide for you how your items should be used. It provides the tools needed to define items in a Data Table, associate effects with them, add them to an inventory, store them, sort them, or retrieve them, but the precise rules remain specific to your game. For example, the system does not automatically decide whether an item can be used in combat, in exploration, in a menu, on a specific character, or only under certain conditions.

The system does not automatically create the gameplay effects of your items. Effects must be developed in dedicated Blueprints. For example, if a potion must restore health, if an object must increase a stat, or if a consumable must trigger a specific logic, that logic must be created in a suitable effect Blueprint. The system can then call these effects, but it cannot guess their behavior.

The OptionalScript field does not trigger any logic automatically. It only allows custom information to be added to an item as key/value pairs. This information can be read by your own Blueprints, for example to manage an equipment set, a crafting family, a special action in a menu, or quest logic. If no Blueprint in your project reads this data, it will have no effect.

The system does not replace a complete equipment system. It can store equipment items in an inventory and provide the data needed to identify them, but it does not automatically decide which slot to use, which mesh to equip, which stats to apply, or how to manage equipment restrictions. These rules must be defined in your own equipment system or in another dedicated system.

The system also does not replace a crafting, merchant, quest, save, or combat system. It can communicate with these systems and provide the required information, but it does not generate them automatically. For example, a crafting system can read owned items, a merchant system can add or remove items, and a quest system can check whether an object is owned, but the logic of these systems must still be developed or connected according to the needs of the project.

The plugin does not automatically define data persistence. It provides a usable structure for managing inventories during game execution, but saving and loading must be integrated into your own save system. It is up to your project to decide which data to save, when to save it, and how to restore it on load.

The system does not force a single way of designing an inventory. It can be used for a global inventory, a per-character inventory, storage, a chest, a system with limits, or a more flexible system. This flexibility means that some decisions must be made at the project level: maximum number of items, category organization, storage rules, behavior when the inventory is full, logic for rare items, protected items, locked items, and so on.

Finally, the assets included with the plugin should be considered a technical base and not a final art direction. Icons, meshes, interfaces, visual effects, sounds, and feedback must be replaced or adapted to match your game's visual identity and needs.

Troubleshooting

An item may sometimes fail to be added, an inventory may not update, an item may not trigger its effect, or an interface may not display the correct information. In most cases, the problem comes from missing configuration, an incorrect inventory name, an item missing from the Data Table, or Blueprint logic that does not call the correct function.

Accessing the Inventory System

To use the system from a Blueprint, retrieve the active instance with Get Inventory System.

Check the following:

- The Get Inventory System node is used to access the correct logic class (usually BP_ADS_InventorySystem).
- The called logic uses the instance returned by this node.
- The system is not retrieved from an empty reference or from an old uninitialized variable.
- The inventory logic class is properly configured in the project settings if the system uses a custom class.

If the system call does not work, start by displaying the value returned by Get Inventory System with an Is Valid. If the reference is not valid, the issue comes from system access before it comes from the inventory functions themselves.

The item is not added to the inventory.

If an item is not added, check the following points:

- The item exists in the item Data Table.
- The item name used in the function exactly matches the Data Table Row Name.
- The Data Table being used is the one expected by the system.
- The added quantity is valid.
- The targeted inventory exists.
- The logical inventory name being used corresponds to a registered inventory.
- The inventory has not reached its entry limit.
- The item has not reached its maximum allowed quantity.
- The called function targets the intended inventory: global inventory, global storage, character inventory, or character storage.

A common case is using an item name that looks correct visually but differs from the actual Row Name. The system must receive the exact row identifier, not only the item's display name.

A character inventory is not found

- If the system cannot find a character inventory, check the following points:
- The actor has an AC_Inventory.
- The component is present on the relevant character or actor.
- The logical inventory name defined in the component is correct.
- The character has been registered by the system.
- The called function uses the same logical name as the one configured on the component.
- The Blueprint calling the function does not execute too early, before the character is registered.

If a character inventory is requested before the character is registered, the system will not be able to find it. In this case, move the call later in initialization, or verify that the component had time to register itself.

Quick Checklist

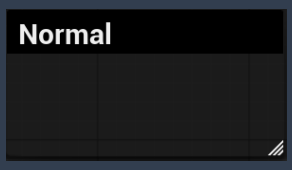
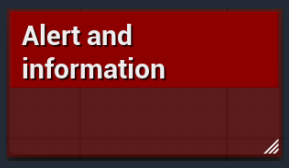
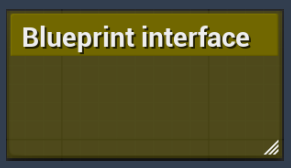
If there is an issue, check in this order:

1. Get Inventory System returns a valid reference.
2. The item exists in the Data Table.
3. The item Row Name is exactly correct.
4. The correct inventory is targeted.
5. The logical inventory name is correct.
6. The character has AC_Inventory if the inventory is linked to a character.
7. The inventory is not full.
8. The item has not reached its maximum quantity.
9. Effects are properly defined in Effects, not in OptionalScript.
10. The interface is refreshed after modification.
11. The Event Dispatchers are listened to from the instance returned by Get Inventory System.
12. Data is restored after system initialization when loading.

Conventions

Comments

This Framework uses a color code for Blueprint comments:

	Standard comment
	For warning and information messages. Most of the time, these comments will appear in demo BPs to warn that the logic is only an example and that real projects should implement something more complete.
	Blueprint Interface event

UI, Menu, and Widget

Most Unreal Engine users prefix all Blueprint Widgets with "WBP_" without distinguishing their role. This Framework uses a different rule:

- "UI_": For user interfaces with no direct interaction with the player. Example: UI_Health
- "MENU_": For user interfaces with which the player can directly interact. Example: MENU_Save
- "W_": Widget used as part of a UI or menu. Example: W_SaveSlot

Contact & Support

Azure Dream Studio is an independent structure managed by Benjamin, who handles development, documentation, training, and support.

If you have a request or comment about this system, you can contact me by email or through Discord. I will reply as soon as possible. Please take time zone differences into account. I do my best to respond as quickly as possible, but depending on your time zone, several hours may pass between the time you send your message and the time I am able to read it.

Please use only French or English for your messages. Requests sent in another language cannot be processed.

azure.dream.studio@gmail.com
Please use the following format in the email subject:
`"[ADS {System Name}]{Your subject}" as the email subject.`
Example: "[ADS Gameplay Ability System] Need information"

https://discord.gg/Vy3TvsjKpQ