Ultimate Inventory System

Table of Contents

Ultimate Inventory System

The Ultimate Inventory System will help developers create their own inventory system for their project. The system is designed to be a powerful, modular, and scalable inventory frame that can create a complex inventory structure as fast as possible.

To achieve this we created a robust and structured system where items are well organized using Item Categories and Item Definitions. Defining items has never been easier thanks to the Attribute System, which allows you to define properties that all items within a category should have. And all of this is done directly in the editor, no code required. Coupled with the multi-nested Item Categories structure and Attribute inheritance, modification and overrides, you can create complex item dependencies extremely fast.

In addition to item management the Ultimate Inventory System takes care of Item Actions, Currencies, Crafting, Item Shops, Equipping, Saving/Loading, and much more. It has all the features you would expect from a complete Inventory System package.

The minimum required version of Unity is 2019.3 due to the recent move from IMGUI to UIElements for editor scripting.

Getting Started

IMPORTANT NOTICE: The documentation is in the process of being updated for V1.2 of UIS. If you are using a previous version please refer to the pdf documentation within the asset folder. If you plan to update your asset, make sure to read the Update to Version 1.2 section of the documentation.

Importing

After the Ultimate Inventory System has been imported you will receive an error which states:

> A tree couldn't be loaded because the prefab is missing.

This error only appears once upon import and is harmless. This error is incorrect and Unity is aware of the bug. If you can upvote this bug that is highly appreciated!

In addition to the import error there will also be a few animation related warnings:

> File 'Idle' has animation import warnings. See Import Messages in Animation Import Settings for more details.

The reason this warning exists is because the animations were authored in Blender and Unity doesn't like the way Blender exported the animation. There's not a way around this but this is a one time warning and does not affect the animations when they are playing.

Demo Scenes

After the Ultimate Inventory System has imported it is recommended that you take a look at the included demo scene to see how everything works. The main demo scene is located at Opsive/UltimateInventorySystem/Demo/Demo.unity.

Some Additional demo scene focused on showcasing and explaining specific features are located in the Opsive/UltimateInventorySystem/Demo/_FeatureDemos folder.

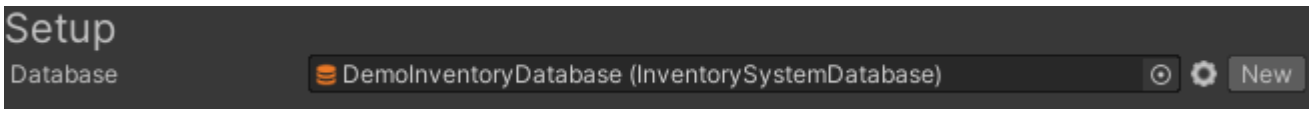
In order to reduce space the Demo folder can be removed and the inventory system will continue to function.

The database that the demo scene uses is called "DemoInventoryDatabase". The demo scenes are dependent on the structure of this database so modifying this database can break the demo.

New Inventory Database

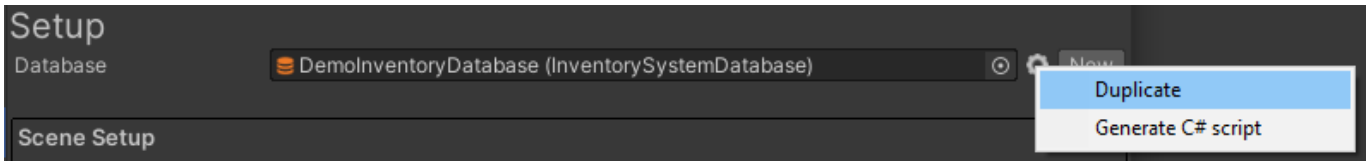
After you have gone through the demo scene it is recommend that you get started by duplicating the demo Inventory Database or by creating a new one. The database can be created from the Setup Manager located within the Inventory Manager editor. The

Inventory Manager can be accessed through the Tools -> Opsive -> Ultimate Inventory System -> Main Manager.



Note: All the demo prefabs, scriptable objects and scene will still reference the DemoInventoryDatabase. The assets in the project that references the demo database can be updated to use a new database. This can be done by right-clicking a folder, prefab, scene, and or scriptable object and going to Ultimate Inventory System -> Replace Database Objects. It is important to note that any assets created and or modified under the Opsive folder can and will be overwritten when the inventory system is imported again. As a result it is recommended to avoid using the original demo prefabs/scriptable objects in your personal projects.

Databases can be duplicated from the cog button:

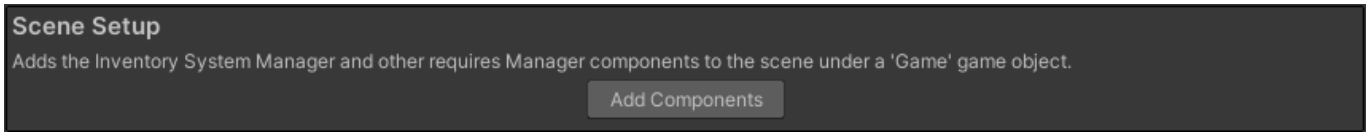


Important: Creating the parent folder of the database within the OS file explorer while creating a database may cause a “Missing Parent Folder” error. Simply create the parent folder first and then press the “New” or “Duplicate” button.

For tips on creating a new database from scratch go here.

Scene Setup

The Ultimate Inventory System requires a set of singleton components in order to function properly. These components must be added to the scene during edit time. These components can be added with the “Add Components” button under the Scene Setup section.
This will add a “Game” game object. One of the added components on that game object is the “Inventory System Manager” which will reference the Inventory Database. It will serve as a hub to get and create items and much much more.



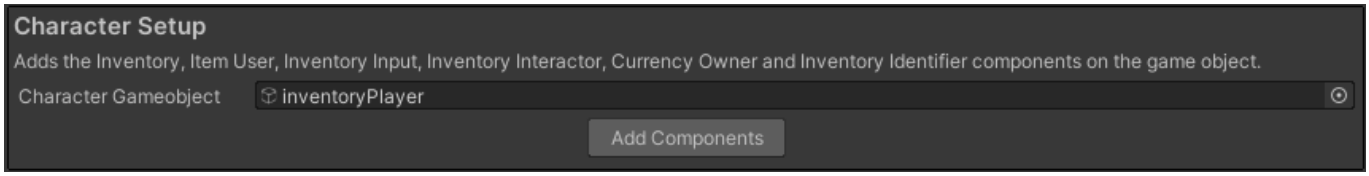
Character Setup

The Inventory component can be used for any game object not just the player. The player Inventory will most likely have a few additional components which are extremely useful:

- Inventory : The component which holds the items
- Currency Owner : holds the character currency

- Item User: the component that lets you use items in ItemActions
- Inventory Standard Input : The new input component that handles input for using items and interacting with the UI.
- Inventory Interactor: Allows components that are interacted with to get a reference to the inventory.
- Inventory Identifier: Allows you to register your inventory game object in the Inventory System Manager singleton such that it can be accessed from anywhere.

It is recommended to Tag the character as “Player” as some of the editor manager will use that tag to automatically fill in some fields.

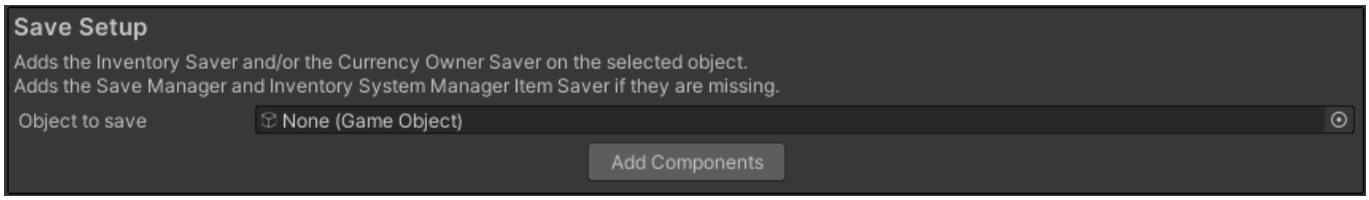


Save Setup

The Inventory comes with a built-in save system. It works by adding “Saver” components next to the components you would like to save. The Save System Manager can use those saver components to save and load data.

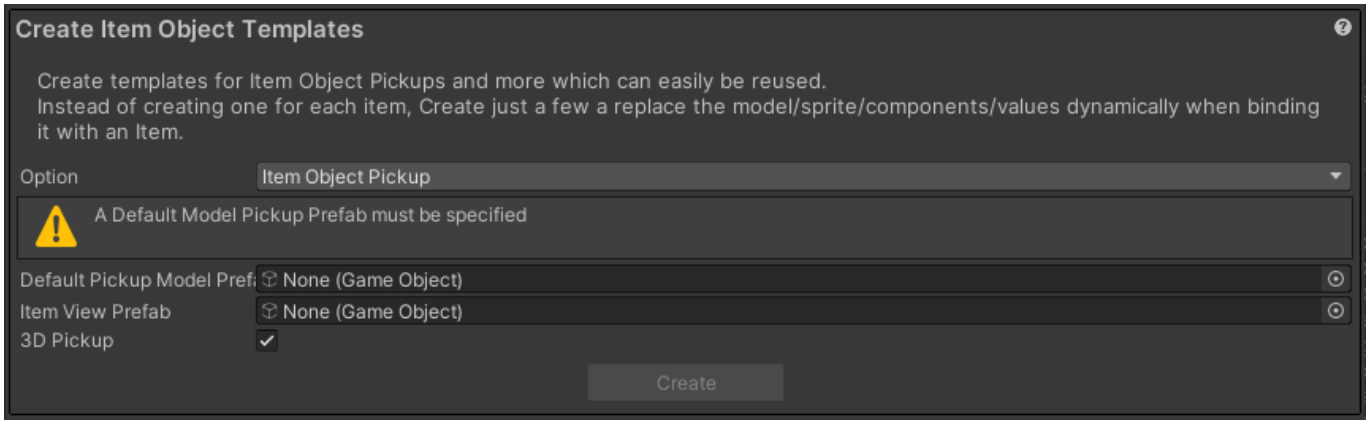
Therefore the save setup adds the the Save System Manager on the “Game” gameobject if it does not exist. It also adds the matching Saver components to the object specified in the field. For example if the object has the components:

- Inventory -> Inventory Saver
- Currency Owner -> Currency Owner Saver



Create Item Object Templates

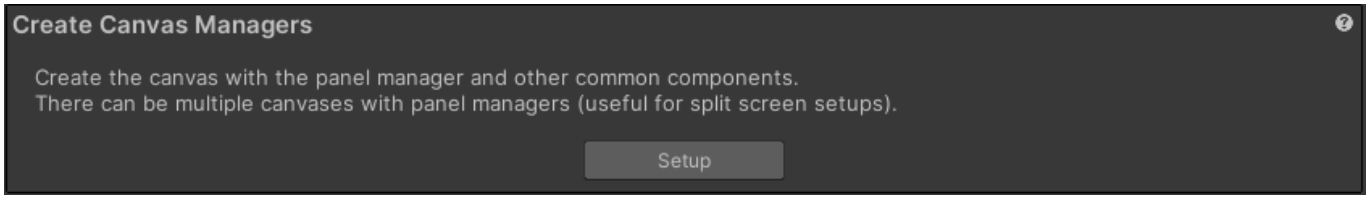
This section lets you create an Item Pickup Template, for 2D or 3D games. Once the template is created, edit it and save it as a prefab which you can reuse. Simply set the Item on the Item Object component at edit or runtime.
The item pickup can be pooled for reuse.



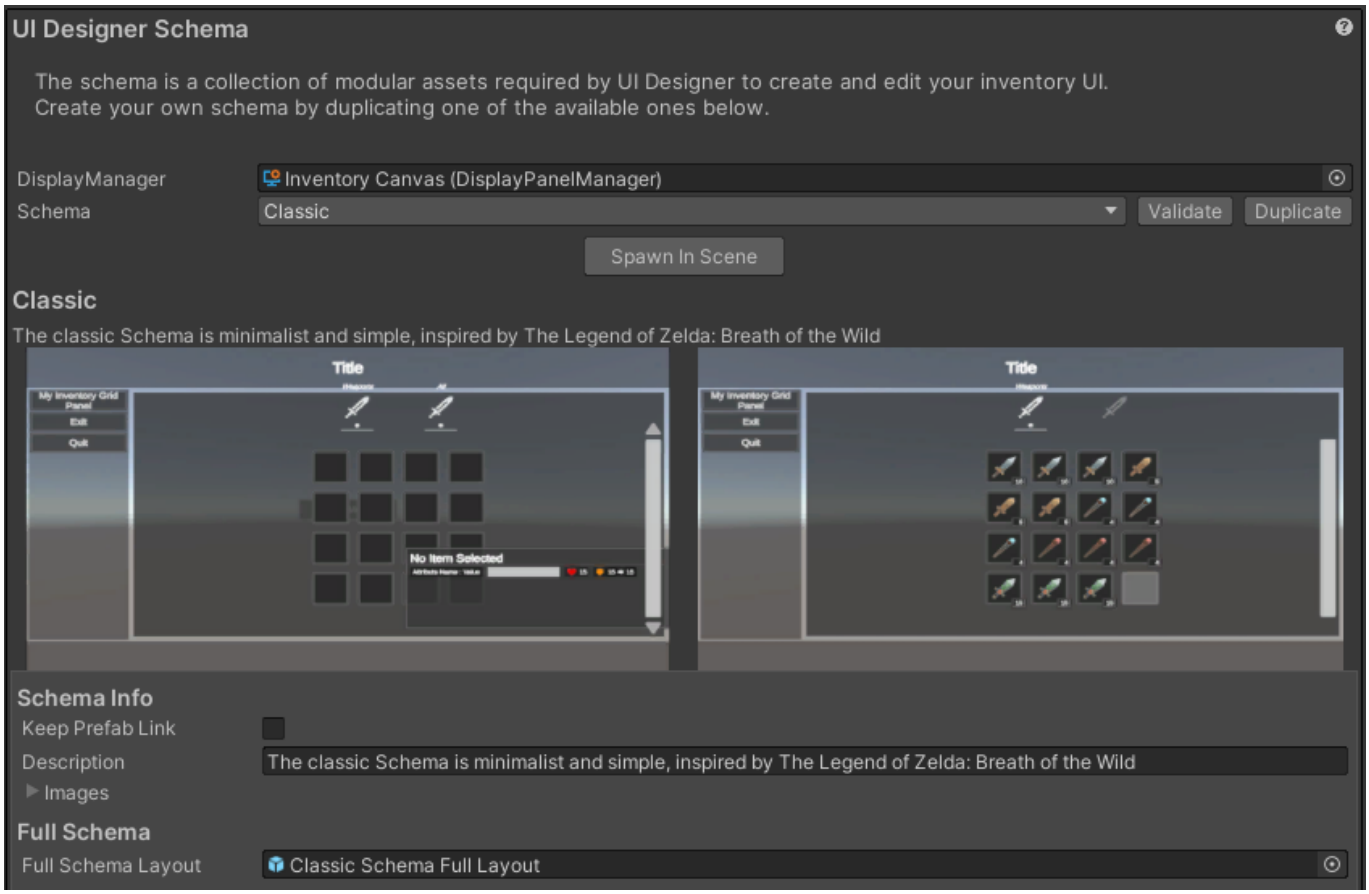
UI Setup with UI Designer

Open the UI Designer Manager to start adding UI to your scene.

In the Setup tab under the “Create Canvas Managers” Click “Setup”, This will spawn the canvas and Display Panel Manager required by the UI system.



Then under the UI Designer Schema section select a schema you like and duplicate it by pressing “Duplicate”. You won’t be able to use UI Designer until you duplicate the schema. This is to make sure that you will have your very own collection of prefabs in case you make changes.



Once you have duplicated the schema press the “Spawn In Scene” button to quickly setup a basic menu for the schema.

From there the other tabs will be available to you and you may use them to create, find and edit your UI.

Where to go from there

From that point you should have a completely working Inventory in your game. You may start by adding object to your Inventory Database, start small and simple. For tips on the Attributes to use for your Items head to the Attribute section of the documentation.

Once the Inventory database has some items, currencies and/or recipes. try them out in your scene by adding them to Inventories, Currency Owners, and Crafters respectively.

Make sure to read the Documentation thoroughly and watch the video tutorials. If you wish to know more or have questions or feature requests, please head to the forum where we will personally help you in any way we can.

Terminology

The Ultimate Inventory System is a very flexible system allow you to create any type of inventory. Usually the more a framework is flexible the harder it is to use and understand. The Ultimate Inventory System is structured in a way to make it as easy to understand as possible. To do this the system includes a workflow with some rules and streamlines the items should be created. There are many components in the system and they each play a key part:

Attributes

Attributes are used throughout the system. Attributes are allowed to override, inherit or modify the value of another attribute. Attributes are used to create variants of Item Definitions and speeds up iteration time.

The three variant types are:

- *Override:* Overrides the parent attribute value.
- *Inherit:* Inherits the parent attribute value.
- *Modify:* Uses an expression to compute a value that is dependent on the “parent” attribute or any other attribute in the same collection.
  Example myAttribute2 = “[myAttriute1] * 20 + <Inherit>”
  Modify is extremely powerful when it comes to speeding up iteration time. It also gives the ability for non-programmers to have simple logic on their items. Item Actions should be used for more complex equations.

Example:

- Parent -> child
- HealAmount: 5 -> HealAmount (override 10): 10
- HealAmount: 5 -> HealAmount (inherit): 5
- HealAmount: 5 -> HealAmount (modify “<Inherit>*3”): 15

Item Category

Item Categories are used to structure your items in a sensible way. All items of a certain category will have the same attributes, with varying values. This ensures that the same action can be performed on any items that are part of the same category. Categories can be nested and parented with multiple children and parents. Mutable categories will set all of the child items to mutable which mean that the attribute values can change at runtime.

Example:

- Equippable
- OneHanded
- Weapon : Equippable
- OneHandedWeapon : Weapon, OneHanded
- Dagger: OneHandedWeapon
- Sword: OneHandedWeapon

Item Definition

Item Definitions can be thought of as templates or molds for items. Item Definitions can have attributes that are consistent across all items created from it. It also contains a reference to a “Default Item” which is used as a base to create the other item instances.

Example:

- No Parent -> HealPotion (HealAmount: 10, CoolDown: 2)
- HealPotion -> SmallHealPotion (HealAmount (Override 6): 6, CoolDown (Inherit): 2)
- HealPotion -> BigHealPotion (HealAmount (Modify “<Inherit>*2.5”): 25, CoolDown (Override 3): 3)

Item

Items are created at runtime and contain attribute values that can be unique to that Item. Even though two Items have the same mold (Item Definition) they can be set to be slightly different. If the Item is part of a mutable Item Category it is even possible to change the values of the Item attributes at runtime.

Items do not have any “item specific logic” attached to it. It is meant to be a simple set of data. Using that data, which consists of the Item Category, Item Definition and Item attributes, the system knows what kind of actions can be performed on them.

Items can be mutable or immutable. Mutable items can be changed at runtime, whereas immutable items cannot be.

Item Action

Item Actions contain logic that can be performed on some Items of a certain category. A simple example is a “Consume” action which can be performed on any item of a “Consumable” category. Or an “Equip” action that could be performed on any “Equippable” items. These actions are game specific, with examples of use cases in the demo. The Item Actions are meant to be expanded upon in order to take full advantage of.

Item Action Set

Item Action Sets are Scriptable Objects that contain multiple Item Actions grouped by Item Category. They are extremely useful as they can be added in most UI that contains Items.

Category Item Action Set

Category Item Action Sets are Scriptable Objects that contain a list of Item Action Sets. They can be used to retrieve a subset of They are extremely useful as they can be added in most UI that contains Items.

Item Object

Item Object is a MonoBehaviour which binds an Item to a GameObject. This is useful for Items that can be equipped, picked up/ dropped or simply that need a presence in the scene.

Item Object Behaviour

Item Object Behaviours are components which allow attaching logic to an Item Object. For example using a “Sword” to attack or a “Gun” to fire, reload, etc. Having a presence in the scene lets you check for trigger/collision events or spawn projectiles from the point of the gun’s barrel.

Currency

Currency is an abstract object which can be used to exchange with other objects.

Crafting Recipe

A Crafting Recipe contains the input and outputs for combining one or more objects into a new item. The Crafting Recipe works with the Crafting Processor in order to perform the logic to covert one set of objects into another.

Inventory

An Inventory is a group of Item Collections. An Item Collection can have any restrictions on the items that go in or out. Splitting the inventory in different chunks is a good way to organize it. For example, having two Item Collections, one for the equipped Items and another for the non-equipped Items is a common organizational structure.

Item Collection

Item Collections contain a set of Items. Items are stored in a collection as a list of Item Stacks. The Item Collection sends events to the Inventory when Items are added, removed, etc…

Item Stack

An Item Stack is an object that stores an amount of an item amount. An example of an Item Stack is “5 apples”. Item Stacks should only exist within an Item Collection. It should not be confused with an Item Amount which can exist outside of the collection.

Item Amount

An Item Amount is a simple structure with an Item and an amount. It should not be confused with an Item Stack which requires an Item Collection.

Item Info

Item Info gives more specification about the items and where they came from. It has not only the Item Amount but also reference to an Item Stack and/or Item Collection.

For example imagine an Inventory with 2 Item Stacks:

- Stack 1:”5 apples”
- Stack 2:”10 apples”

Where apple is an immutable Item. If you wish to remove 3 apples from Stack 2 you can specify an Item Info as such. When adding Items to an Item Collection you may also specify where the Item came from and in which Item Stack it should be added to.

Item Slot Set

An Item Slot Set is a Scriptable object used to define the slots for an Item Slot Collection. Each slot is simply a name and an Item Category. The Item Slot Set and Item Slot Collection is often used for equipping items with an Equipper.

Item View

An Item View is used to display an item is the UI, it uses Item View Modules to customize exactly how the item shows up.

Item View Slot

An Item View Slot is used in the UI to detect interactions such as select, click, drag, drop and more. It usually contains an Item View.

Inventory System Database

This is an asset that is created in the editor and it is used to store all your Item Categories, Item Definitions, Currencies, Crafting Categories and Crafting Recipes for your project. These objects are referred to as Inventory System Objects. The database is simply a container of data and is loaded in the Inventory System Manager at the start of a game.

Inventory System Manager

The Inventory system Manager is very important as it manages every Item, Item Definition, Item Category, Currency, Crafting Category and Crafting Recipe in your project. When the project starts it will load the Inventory System Database and it will get references from any of the registered inventory system objects.

The Inventory System Manager ensures only valid objects are registered and also provides a convenient method to look up objects based on an ID or name. The Inventory System Manager is also used to create Items at runtime.

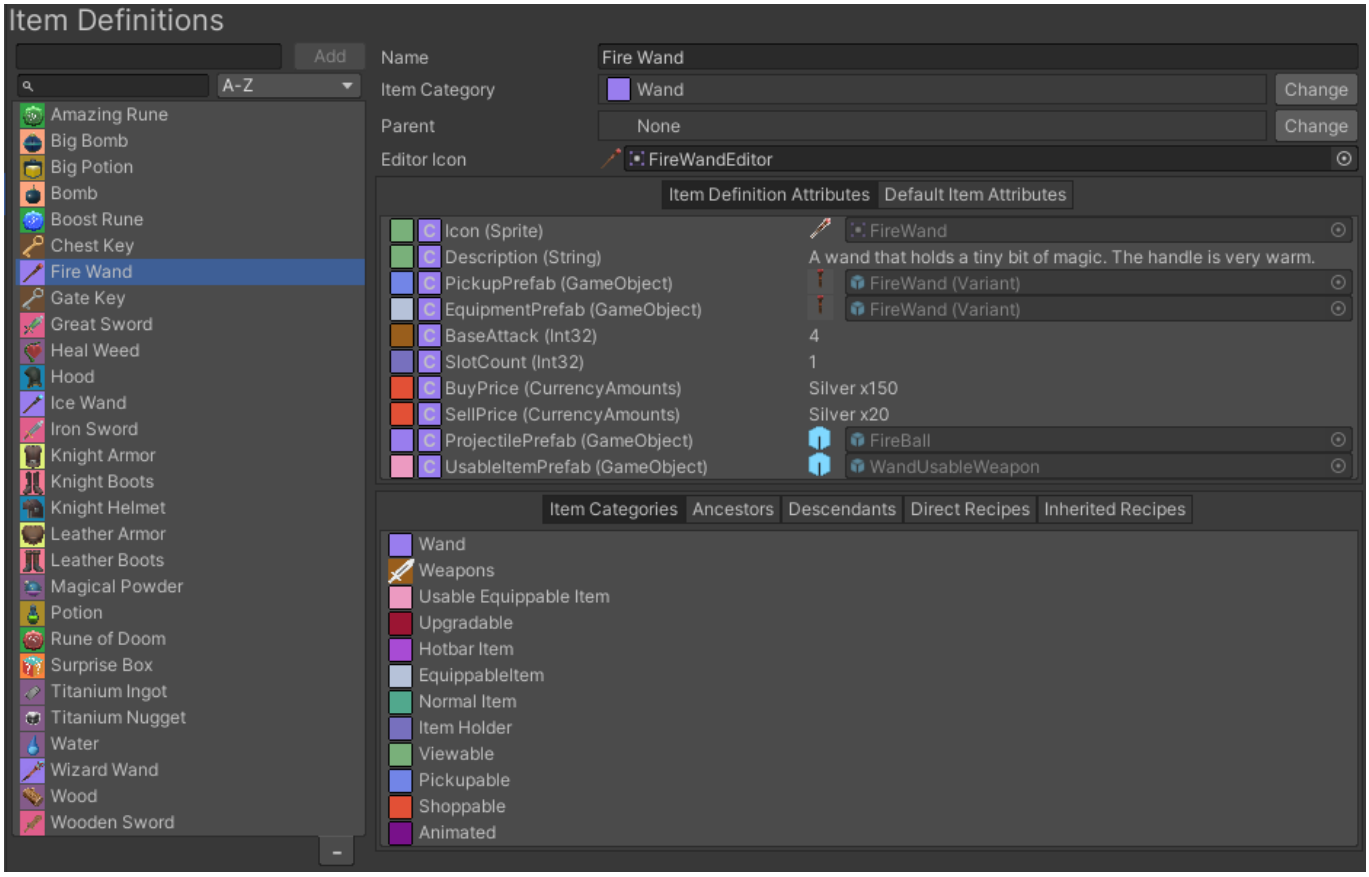
Demo Scene

When first getting started with the Ultimate Inventory System it is recommended that you play around the demo scene to get familiar with the system. The demo requires TextMesh Pro. If you do not have TextMesh Pro installed you can do so via the Unity Package Manager.

The demo scene is separated in two parts. The village and the forest. In the village the player character can take their time buying/selling, storing, upgrading and crafting items. In the forest the player is able to use their items to battle bandits and retrieve loot.

The demo character scripts are not part of the core UIS framework and should not be used outside of the demo scene. There are a few objects which are also demo specific such as the Character Stats, Item Upgrade Menu and the Gate.

You can edit the demo inventory database by going to Tools -> Opsive -> Ultimate Inventory System -> Main Menu. When getting started it is recommended that you create a few Item Definitions of your own to get a sense of how the items can be referenced in the scene.



The items can be added to the player character directly in the editor by setting them in the character's Inventory component. Select the "(Main) Main" Item Collection on the Inventory component of the "Player Character" to make the main Item Collection appear.

From here you can add your own Item Definitions. The player can start with an equipped item in hand by adding the weapon directly to the "Equipped" item collection.

All the menus are be found under the "Inventory System Canvas" GameObject. These menus are activated when the character interacts with objects in the scene. Most of the world objects, such as the shop and crafting stable, are found under World -> Static Interactables. The Main Menu and the Gameplay menu are slightly different as they are opened by other means. On the "Inventory System Canvas" GameObject you will find the Display Panel Manager which allows you to specify the game play and main menu.

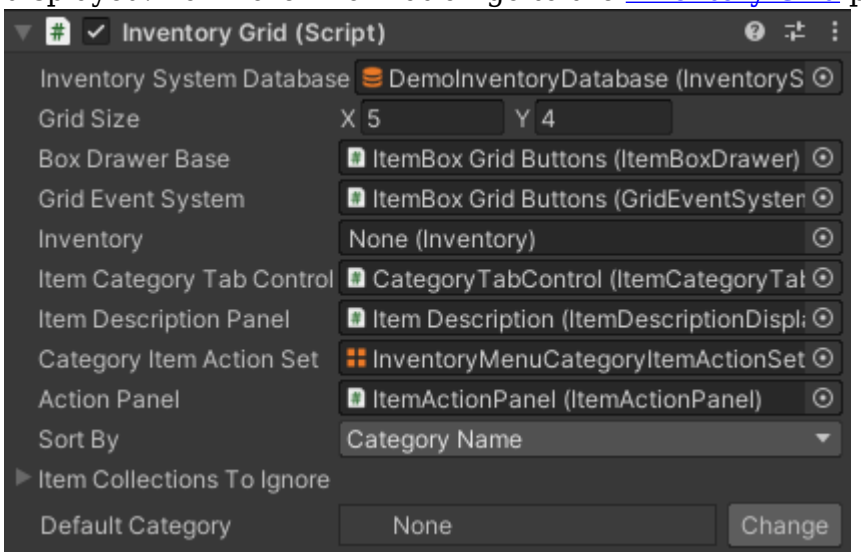
UI Customization

The inventory UI relies heavily on prefabs and Scriptable Objects in order to display the dynamic content. This makes it easy to switch out the UI with your own content as a lot of the workflow will be editing prefabs and Scriptable Objects.The best way to get started is to first add the UI to your scene with the UI Designer in the Editor.

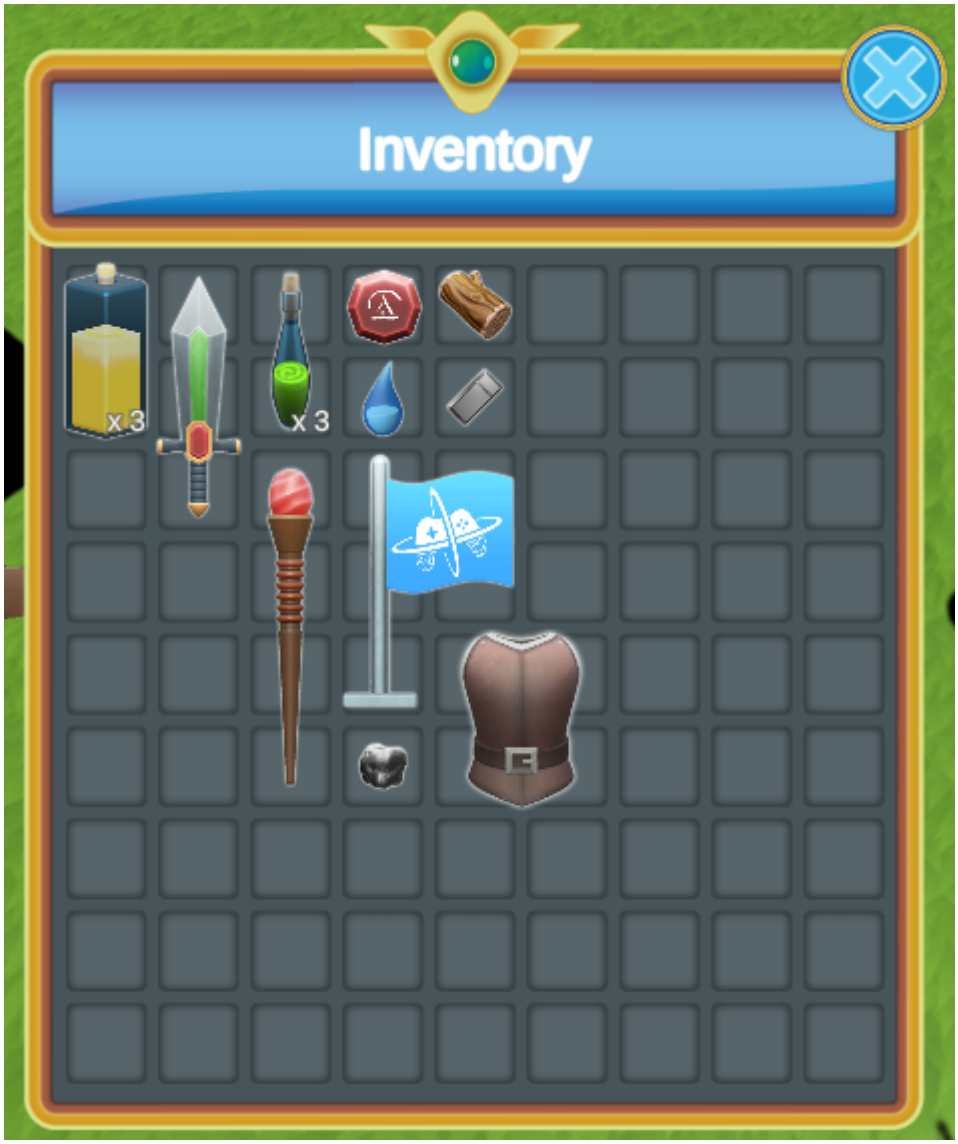
The visuals below show where you can modify each object type.

Inventory Grid & List

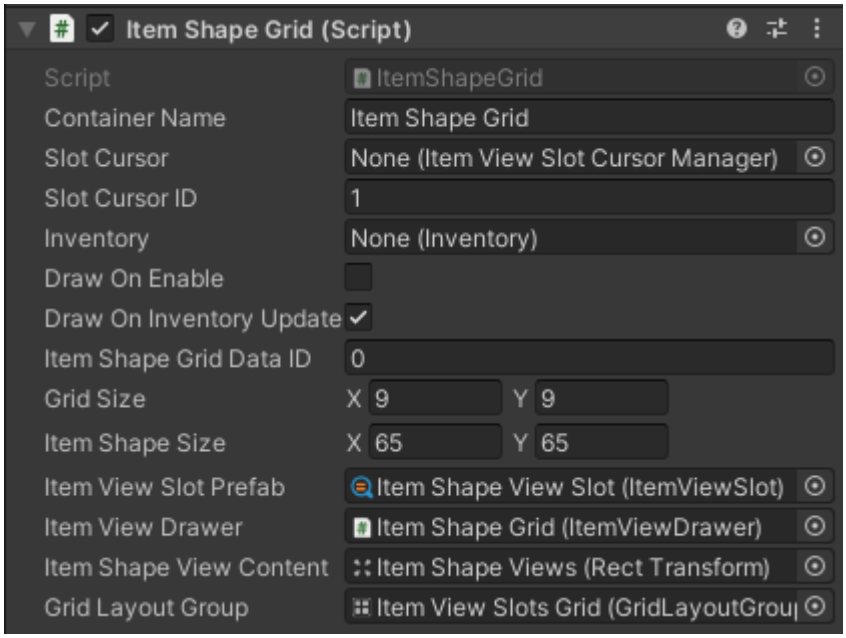
The Inventory Grid and List component allow you to customize how your items are displayed. For more information go to the Inventory Grid page.



Item Shape Grid

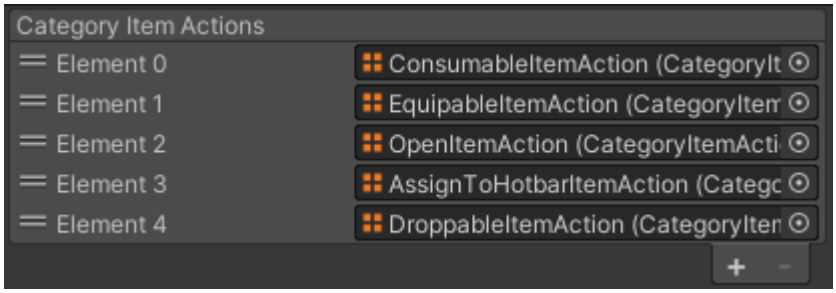


The Item Shape Grid component allow you to display your items such that each item can take multiple slots within a grid. For more information go to the Item Shape Grid page.



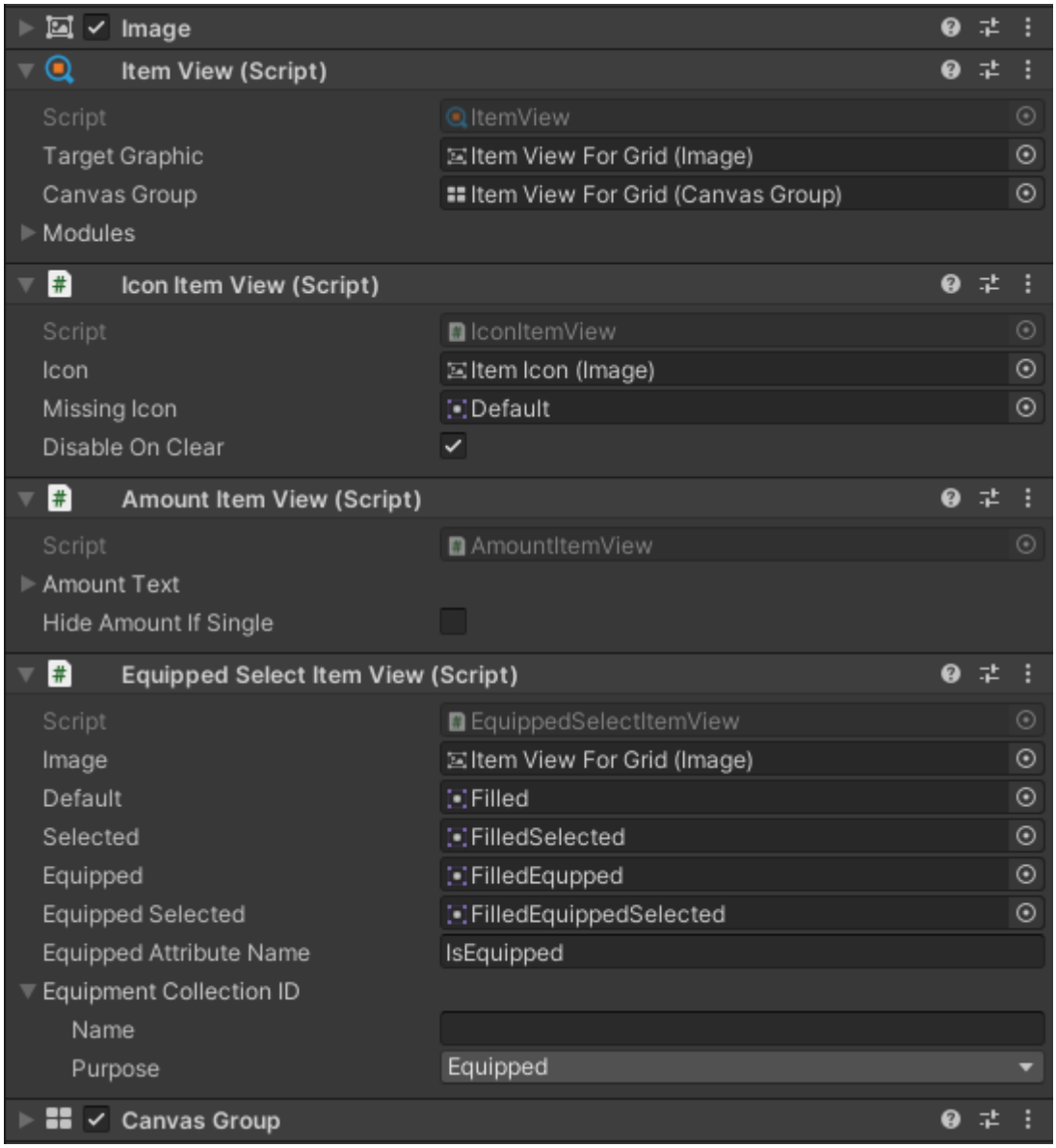
Item Action Set & Category Item Action Sets

The UI system allows you to customize the actions that can be performed on an item from the inventory UI. This is done by looking at the Item Category and getting the matching actions/prefabs. The Item Actions page has more details on how to setup these components.



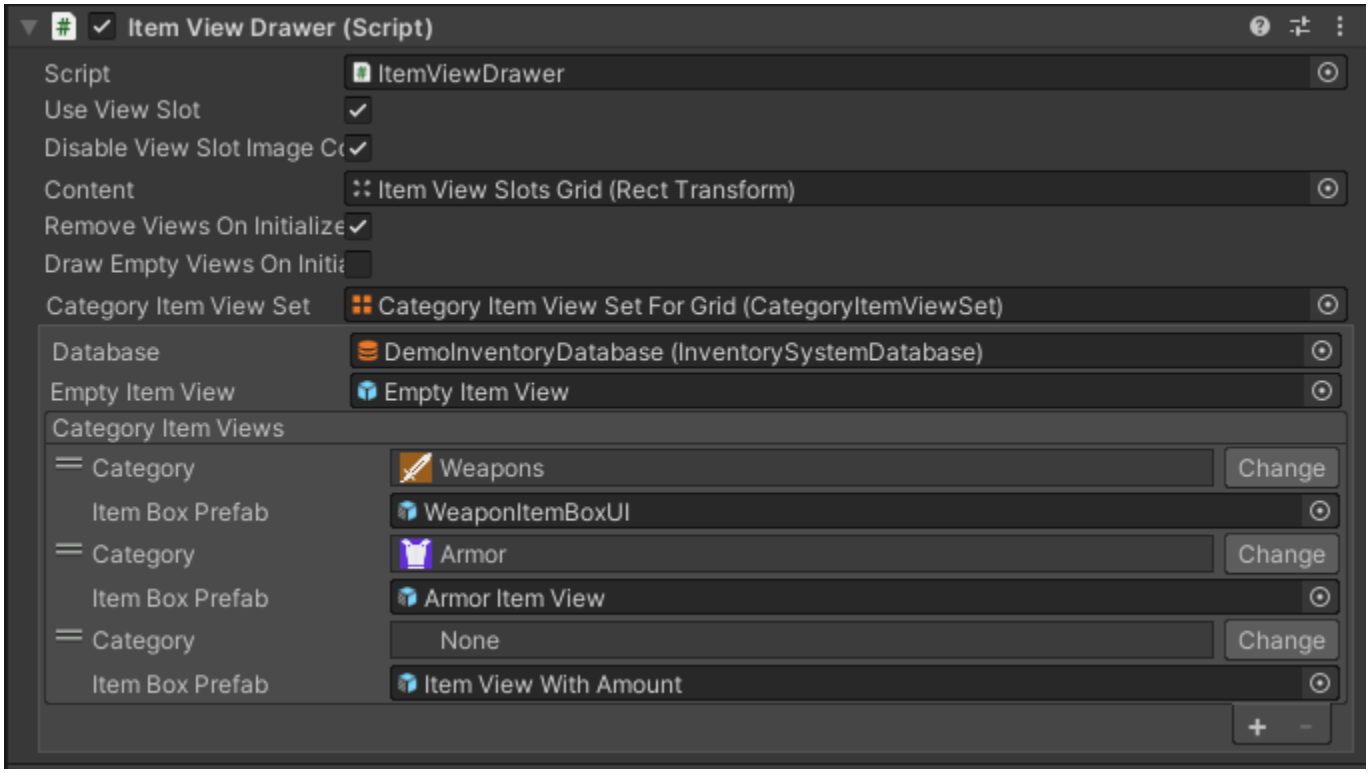
Item View (Modules) & Category Item View Sets

The Category Item View Set Scriptable Object can be used to modify how the items are displayed. New sets can be created within the project view menu under Create -> Ultimate Inventory System -> UI -> Category Item View Set. Learn more about it on the Item View page.

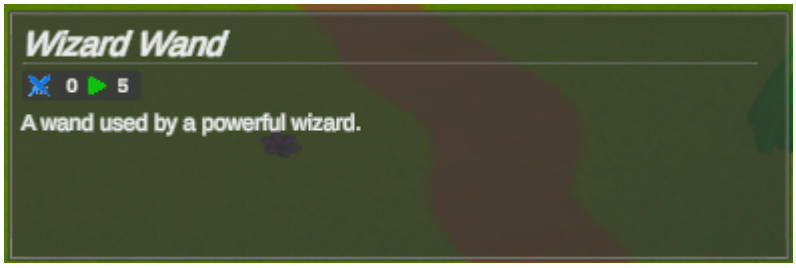


Item View Drawer

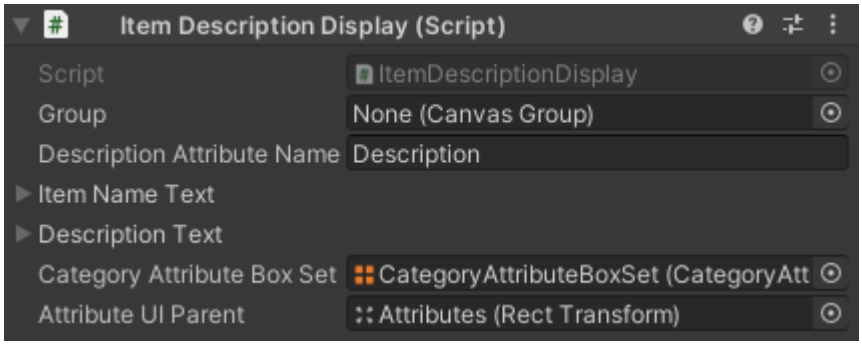
The Category Item View Set can be referenced from the Item View Drawer component which is used by the Inventory Grid/List component.



Item Description & Attribute View (Modules)

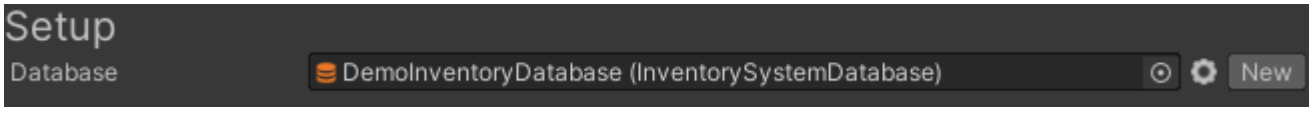


When selecting an item the inventory it can show a description with information about the item and its attributes. These can be customized similar to the Item Views. The Item Description and Attribute Views pages contain information on these components



New Database

A new Inventory System Database can be created from the Setup Manager located within the Inventory Manager editor. The Inventory Manager can be accessed through the Tools -> Opsive -> Ultimate Inventory System -> Main Manager.



> **Important:** Creating the parent folder of the database within the file explorer while creating a database may cause a “Missing Parent Folder” error. Simply create the parent folder first and then press the “New” button.

When creating a Inventory System Database from scratch it will come with an “All” Item Category with two Item Definition attributes “Icon” and “Description”. These two attributes are the most common attributes people will use. You may rename or delete this Item Category if you wish to.

It is recommend that you add a few Item Categories you believe you will need such as “Consumable”, “Equippable”, or “Droppable”. These can be considered as “functional” Item Categories, and is a good way to organize Items by their functionality. You may also have “Food”, “Weapon”, or “Material” which could inherit the “All” Item Category as well as a subset of the “functional” categories to further group your items. Detail categories such as “One-Hand Weapon”, “Sword” or “Knife” can be added to differentiate items which are within the same parent Category.

Any attribute defined in an Item Category will be inherited by all its Item Definitions. Therefore it is important to organize your Item Categories and set the attributes in the correct place.

> **Example:** Lets assume you want to make a game with different types of weapon. You may have the following Item Categories “All”, “Weapon”, “Droppable” and “Melee Weapon”, “Ranged Weapon”. “Weapon” could have “All” and “Droppable” as parent and “Melee Weapon” and “Ranged Weapon” as children. For the attributes you might have the following:
>
> - “Icon” and “Description” in the “All” category as Item Definition attribute
> - “DropPrefab” in the “Droppable” category as Item Definition attribute
> - “WeaponPrefab” and “Attack” in the “Weapon” category as ItemDefinition attribute
> - “Clip Size” in the “Range Weapon” as ItemDefinition attribute and “Clip Remaining” as Item attribute
>
> The above example would give you a good organization to seperate functionality depending on whether to the weapon is Ranged or Melee. In addition having the Droppable category seperate would allow you to drop other types of items like “Material” if you decide to add some later.
>
> These are just examples, you may organize your categories and attributes however you like and make sense for your game.
> For more examples on how to organize attributes look at this page.

Item Categories

The Item Categories are the most important inventory object as it structures your item

database, decides the attributes for all of the items, and is also used to separate the actions that can be performed on each item. Learn more about Item Categories here.

Item Definitions

Item Definitions can be thought of as templates for your items. The Item Definition will have attributes that are common across all of it's item instances. Each item instance may have a different attribute value for the same Item Definition.

As an example you may have a "Big Sword" Item Definition which has different attack values for each item instance. Learn more about Item Definitions here.

Items

Items are created at runtime from the Item Definition. The Default Attribute values of an Item can be defined in the Item Definition Editor. Learn more about Items here.

Attributes

Attributes hold the data on the Item Category, Item Definition and the Item. They can be any type. For example an "Icon" attribute can hold a Sprite and a "Description" attribute could hold a string. Some attributes are quite common such as "Description" and "Icon".

> **Important:**
>
> The most common attributes which are sometimes predefined in the UI Item View Modules or other components in the system can be found here.
>
> For more tips on creating good Attributes for your items take a look at this page.

For detailed information on the Editor functionality to create the Inventory Objects and Attributes go to the Editor Window section.

Defining Attributes

When creating a database it is important to organize your objects and attributes thoughtfully. The descriptions below give an idea for how you should setup your database.

> The most common attributes which are sometimes predefined in the UI Item View Modules or other components in the system can be found here.

Item Category

The item category will define what your item can and cannot do. Examples of categories could be *Equippable*, *Consumable*, or *One Handed*. The categories can be nested with

multiple parents the items that are part of a category will inherit all the properties from all the parent categories.

Mutability & Uniqueness

Categories also define if their direct items are mutable and unique. Mutability defines if an item can have its attribute values changed at runtime
Unique defines if the item will try to stack together when possible. Note that mutable items that are common (not unique) will keep the attribute values of the pre-existing item when stacked.

Examples for mutable/unique categories:

- *Immutable & Common:* Any items that do not change once created. The same item reference will be used for all items with the item definition as long as they have the same attribute values. Examples include a *Materials* category for items like *Wood*, *Rope* and *Steel*.
- *Mutable & Unique:* For items that should be considered unique. Unique in the sense that two items of with the same item definition won't have the same reference or ID. Since the items are mutable their attributes can be changed at runtime, changing the attribute of one item will not change attributes for another as each item is unique. Examples include a *Sword* category with items like *Iron Sword* and *Great Sword.*
- *Immutable & Unique:* Functions the same as *Immutable & Common* but items won't stack in the default Item Collection. Examples include a "Key Item" category for items such as *Map*, *Boss Key*, and *Chest Key*.
- *Mutable & Common:* Similar to *Mutable & Unique* except that when an item is added to a collection it will try to stack on top of an existing item with the same item definition. This is a good option when for equippable items that require mutability to be equipped yet does not need to be unique. Examples include a *Grenade* category for item like *Flash Grenade* and *Frag Grenade*.

Mutability and Uniqueness only affects items that are directly under that category.

Attributes

When defining your attributes it is important to place them in the correct collection. There are three options:

- *Item Category Attributes:* Attributes that relate the category directly. Examples include a *Category Icon*. All item definitions and items within a category will share that same attribute value.
- *Item Definition Attributes:* Attributes that will be shared by all items within the same item definition. Any attribute that will not change at runtime should be set here. Examples include a *Icon* such that each item with the same item definition can have their own icon.
- *Item Attributes:* Attributes that may change at runtime. These are specific to the item itself. Meaning two items with the same item definition could have different values for the same attribute. Examples include a *Durability* attribute that decreases for each strike of a weapon.

Item Definitions

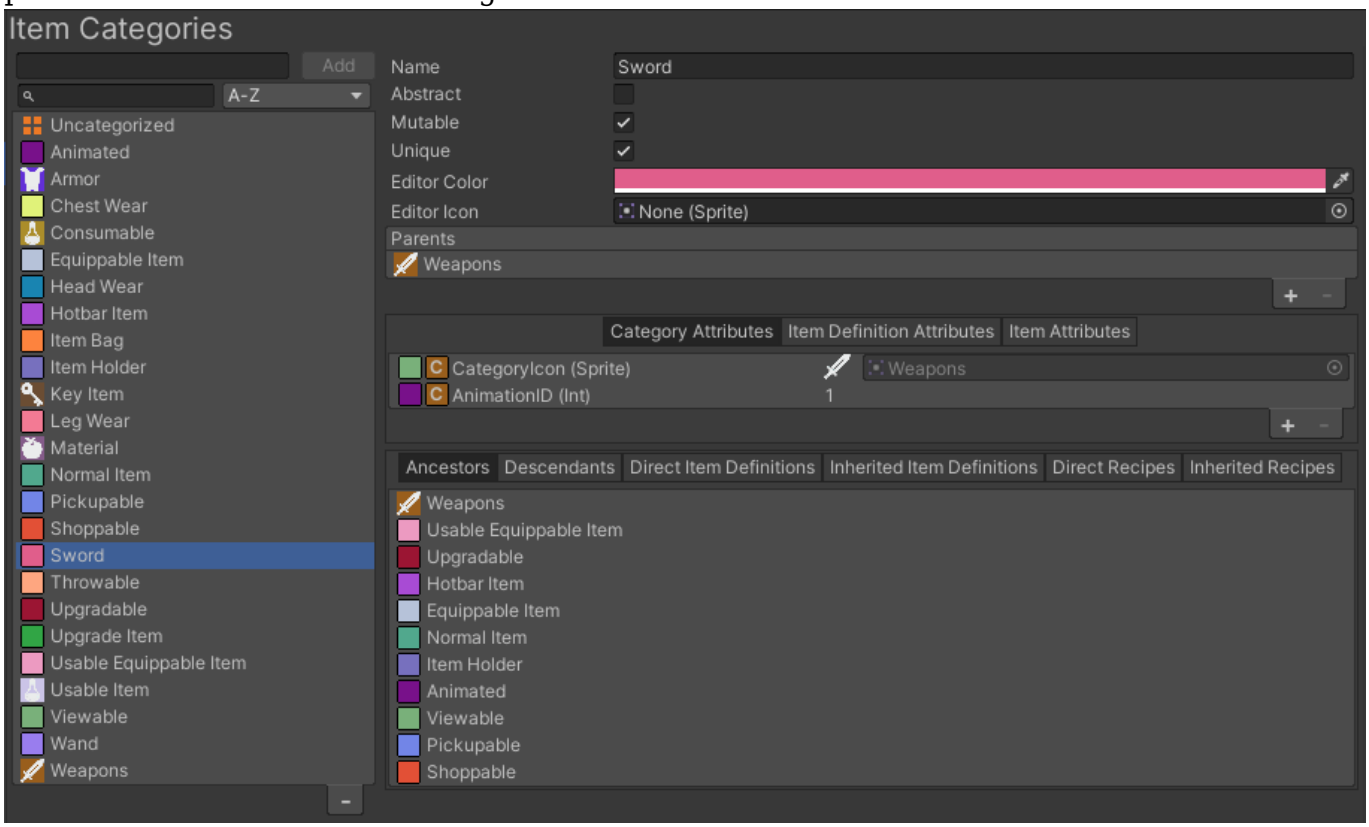
At runtime you may want to change the attribute value, such as to increase the attack value of the weapon. One solution may be to increase the attack attribute, but it may be better to replace the item by a new one.

As an example your database could have item definitions: *Sword*, *Sword +1*, *Sword +2*. When you upgrade a *Sword* it gets replaced by *Sword +1* item. This allows you to keep the item attribute values separated for each weapon behavior. Item definitions can be used to implement this structure cleanly with the parent system. With a parent structure of *Sword -> Sword +1 -> Sword +2*, all attribute values will be inherited by the child item definition.

Demo Examples

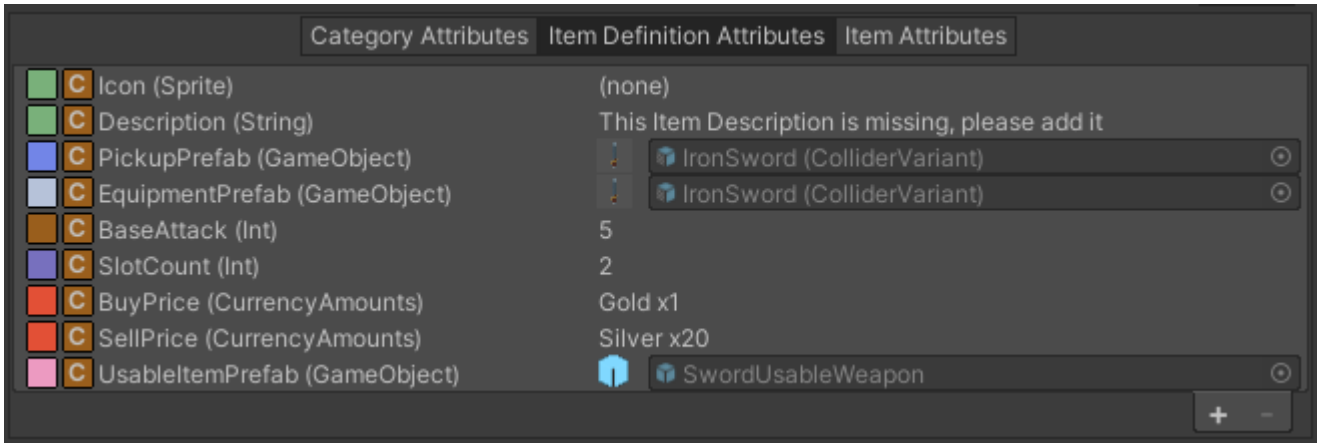
Sword Category

The *Sword* category is Mutable and Unique. In the demo a sword can be upgraded by attaching “upgrade” items to it. This means that each sword must be unique and mutable. Mutable is required so you can change the attack and “upgrade” items at runtime. Unique prevents the sword from stacking with the same item definition.

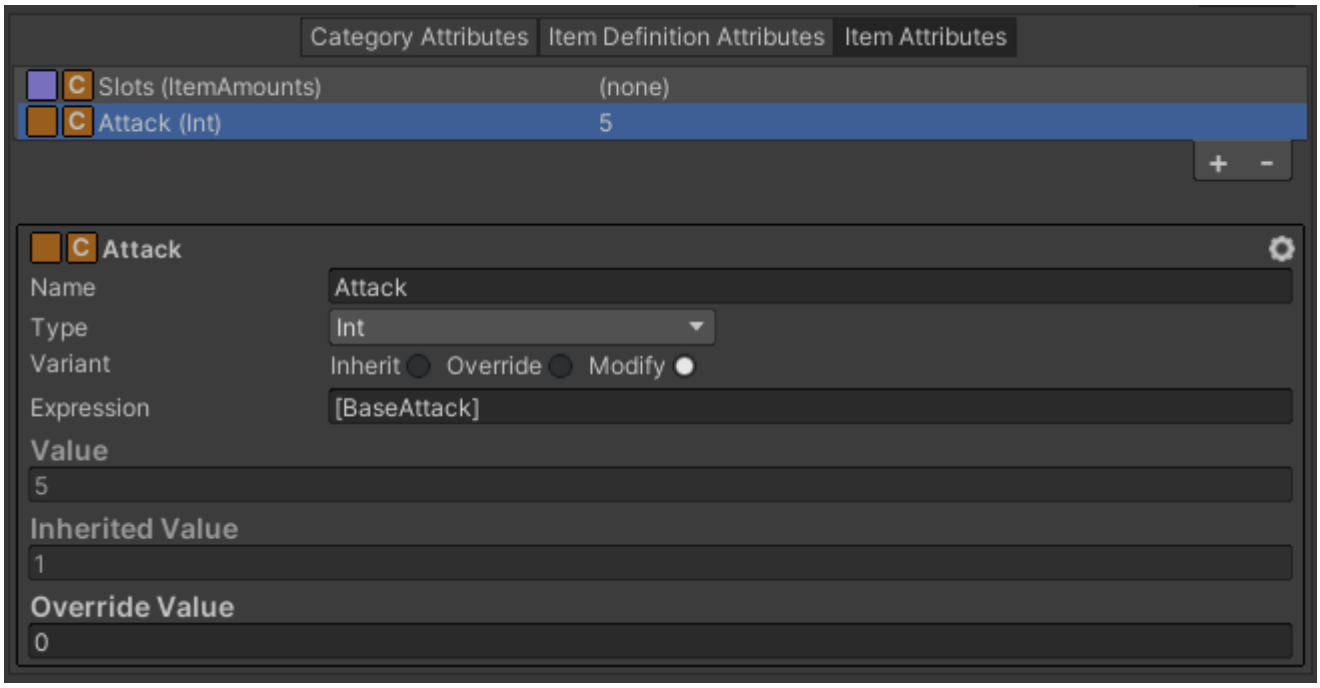


As shown in the screenshot the sword category has two category attributes:

- *CategoryIcon* is inherited from the *Viewable* category. This attribute defines the icon that should be shown in the editor.
- *AnimatedID* is inherited from the *Animated* category. This is used to know what animation to use when attack with a sword. All items in that category will animate the same way.



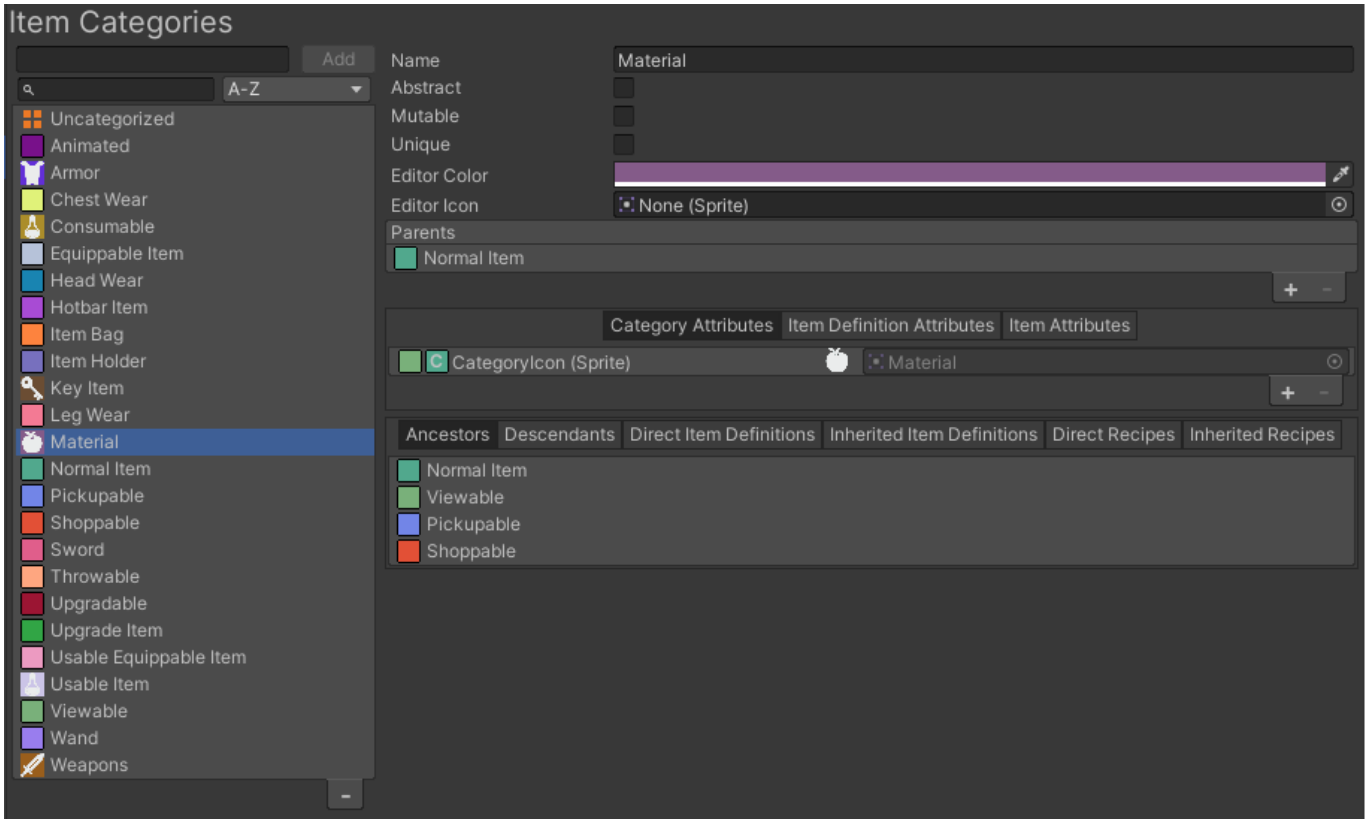
The item definition attribute are specific to each item definition, therefore each item definition in this category can have their own values for the defined attributes. Notice *BaseAttack* is part of the item definition. This value will not be able to change at runtime.



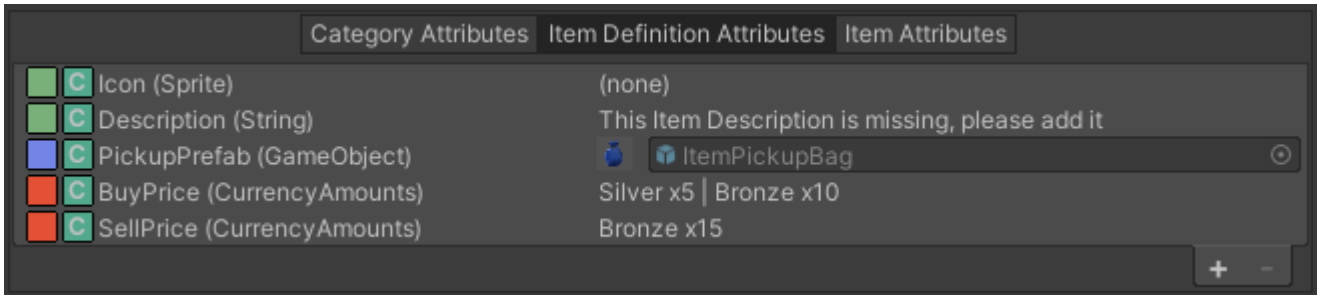
Only *Attack* and *Slots* will be able to change a runtime and therefore they are the only ones added as item attributes. *Attack* uses the modify variant to copy the *BaseAttack* value by default. This value can then be changed at runtime.

Material Category

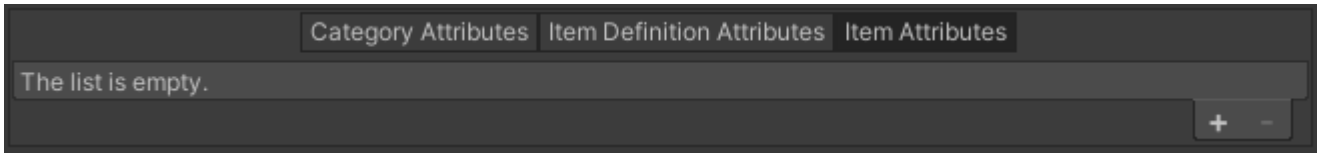
The *Material* category is Immutable and Common. It is Immutable because it has no attributes that need to change value at runtime. And Common because material items need to stack automatically.



The only category attribute is the *CategoryIcon*.



All the other attributes are set as item definition attributes.



There is no need for item attributes since all material item definitions will have a single item each.

Version 1.1 Update Guide

Version 1.1 comes with many improvements with a focus on the user interface. The goal was to make a modular and flexible system that allows users to mix and match features available to create there very own Inventory UI. Unfortunately this meant rewriting the entire UI system and all of its components. Therefore the UI from version 1.0.x is incompatible with the new system. As a result of these changes, there are few steps required in order to update to version 1.1 from a prior version.

If you believe some information is missing, please ask us in the forum and we will review the

upgrade steps.

Important: Make a backup of your project before updating!

Updating to version 1.1 can be accomplished by performing the following:

Remove version 1.0.x

Many scripts were renamed or removed and Unity Packages do not handle these changes very well. After version 1.0 has been removed version 1.1 can be imported.

Check your scene and prefabs for Missing Components

As mentioned previously many scripts were renamed, replaced or simply removed. Therefore some of you game objects or prefabs may need fixing.

Menus & Panels

If you have any menus or panels which was taken from the demo scene it is highly recommend it to remove it and rename it from scratch using the UI Designer. Unfortunately this is due to the amount of changes which were required to make the UI system more extensible. Learn more about UI Designer here.

Item Box -> Item View | Attribute Box -> Attribute View

Item Boxes, Attribute Boxes, etc. are now called Item Views, Attribute Views, etc. Views are used to display a certain object, it is usually, but not required, to be within a View Slot (i.e Item View Slot) which is a custom selectable that detects clicks, drag, select and more.

Components serializing Item Categories, Currency, and more

We now use some very handy structs to serialized objects from the database which allows the system to replace them if they are not part of the correct database. The structs are Dynamic Item Category, Dynamic Currency, Dynamic Crafting Recipe, etc.

Due to this change all categories which were set in the inspector must be re-assigned: Item Slot Set, Category Item Actions, Category Item View Set, etc.

Main inventory (used by the character)

A few changes have been made to the components that surround the main inventory.
A typical character Inventory also has the following components:

- *Inventory*: The component which holds the items.
- *Currency* Owner: holds the character currency.
- *Item User*: the component that lets you use items in Item Actions.
- *Inventory Standard Input*: The new input component that handles input for using items and interacting with the UI.
- *Inventory Interactor*: Allows components that are interacted with to get a reference to the inventory.

- *Inventory Identifier*: Allows you to register your inventory game object in the Inventory System Manager singleton such that it can be accessed from anywhere.

Category Item Actions -> Item Action Set

There was some confusion between Category Item Actions and Category Item Action Sets. Therefore Category Item Actions was renamed to Item Action Set.

Tip: Use Item Action Set as an attribute to an Item if you wish to have different Item Actions per Item(Definition) instead of per ItemCategory.

Equipper & Usable Equipped Items Handler

The Equipper no longer uses the items, it simply visually equips them.

The Usable Equipped Items Handler is the component that listens to the Inventory Input event to use Item Object Behaviors.
Simply add this component next to the Equipper and reference the Item User to get started.

Item Object Action -> Item Object Behaviour

A simple name change to reduce confusion with Item Actions, as you may wish to have an Item Object Action which inherits Item Action instead of MonoBehaviour.

Usable Item Object -> Item Object Behaviour Handler

The usable Item Object component was replace by the Item Object Behaviour Handler such that the inheritance of Item Object could be removed, and instead use a composite relationship.

Item Pickup Visual Listener -> Item Object View

The Item Pickup Visual Listener was used to change the model of the pickup object by getting a prefab from an Item Attribute.
The Item Object View does the same, but insteado of being dependent on the ItemPickup events, it is dependent on the Item Object events.
It also allows you to specify an Item View for World Canvas UI on the Item Object.

Take advantage of the new features

Version 1.1 comes with a ton of new features and API improvements. So make sure to read through the documentation to learn more about all the features.

Here are some of the new API which allows you to use the Inventory System Manager singleton to get very useful things like the Inventory and Display Panel Manager:

```
// Get your inventory, Item User and more from the Inventory
Identifier which can be fetched from anywhere in the code.
var inventory =
InventorySystemManager.GetInventoryIdentifier(1).Inventory;
```

```
// Add Items by name.
inventory.AddItem("potion", 3);

// Get the panel manager.
var displayPanelManager =
InventorySystemManager.GetDisplayPanelManager(1);

// Open a Display Panel by name.
displayPanelManager.OpenPanel("Main Menu");
```

Version 1.2 Update Guide

Version 1.2 comes with many improvements with a focus stability and feature completeness. The goal was to make a modular and flexible system that allows users to mix and match features available to create there very own Inventory UI.

Many of the changes made in V1.2 are related to feature requests by community members and better integration with other Opsive assets specifically the Ultimate Character Controller. Therefore some of the components from UIS 1.1.X might be incompatible with the new system. As a result of these changes, there are few steps required in order to update to version 1.2 from a prior version.

If you believe some information is missing, please ask us in the forum and we will review the upgrade steps.

> **Important**: Make a backup of your project before updating!

Updating to version 1.1 can be accomplished by performing the following:

Remove version 1.1.x

Many scripts were renamed or removed and Unity Packages do not handle these changes very well. After version 1.1 has been removed (including the Shared folder) version 1.2 can be imported.

If you own one of the character controllers make sure the reinstall it too as it contains additional shared scripts which are not within the Ultimate Inventory System.

Check your scene and prefabs for Missing Components

As mentioned previously some scripts were renamed, replaced, modified or simply removed. Therefore some of you GameObjects or prefabs may need fixing.

Major Changes

Feature Demo Scenes

We added minimalist demos scenes which explain certain features in detail. This was a

highly requested feature to help people understand how to use the asset with more examples.

CSV Import Export

You can now import and export inventory system databases to and from CSV files. There are some strict limitations to our solution but it should help people that like to visualize there items in a table format.

It is also useful when you transitioning from a previous inventory solution. where you already have a large list of items within a list.

Audio Manager

The Audio Manager has been rewritten from scratch. It is now a shared feature between the Ultimate Inventory System and the Character Controllers.

You will now be able to define how to play your audio clips with granular control. The Audio Config Scriptable Object lets you define a list of Audio Clips, Audio Mixer, volume, pitch and more which can be set to constant or random values.

You'll also be allowed to extend the Audio Manager Module to use custom your own Audio System or Third party Audio systems bypassing the Unity Audio System.

Save Meta Data

The save system now allows for save meta data. The save meta data can be extended to add context to the save file, such as play time or player progress. The save meta data is available in the Save Views to show this custom information.

Potential changes required in your projects

Here are a few changes you might be required to make in your project when upgrading:

Functions renamed

- AddItemCondition -> CanAddItem: Within the Inventory, Item Collection and Restriction classes.
- RemoveItemCondition -> CanRemoveItem: Within the Inventory, Item Collection and Restriction classes.

Scripts Removed/Renamed

- Audio Manager: The Audio Manager is now a shared file between the Ultimate Inventory System and the Character Controllers. It is much more versatile than the previous Audio Manager.
- FixedSizeItemCollection: This Item Collection was rarely used and had some issues. It is now removed, instead use Item Restriction Object to limit the number of items you wish in your Inventory to Item Collection.

Editor Window

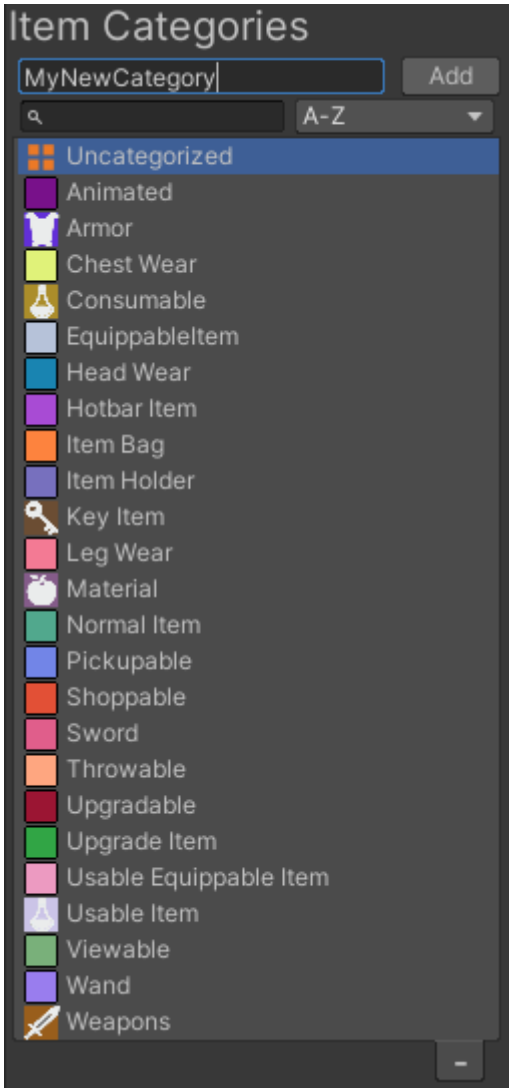
Organization

Before creating any inventory objects it is recommended that you first have a plan for how to organize the inventory. The first thing that you'll want to think about is the Item Categories. The Item Categories are the most important inventory object as it structures your item database, it decides the attributes for all of the items, and it is also used to separate that actions that can be performed on each item.

Once the categories are specified, the next step is to create the Item Definitions. Item Definitions can be thought of as templates for your items. The Item Definition will have attributes that are common across all of it's item instances. Each item instance may have a different attribute value for the same Item Definition. As an example you may have a "Big Sword" Item Definition but this sword has different attack values for each item instance.

Adding Objects

When a database is created it is given an Uncategorized Item Category and Crafting Category. These cannot be removed. Item Definitions require an Item Category to function properly and will be given an Uncategorized category when they are created in the editor. Similarly a Crafting Recipe will be assigned to the Uncategorized Crafting Category.



New inventory objects can be added by opening their respective editor and by entering the name within the field next to the “Add” button. The search bar and sorting options allow objects to quickly be found within the editor. Selecting the object from the list will allow you to edit it within the editor.

When an object is created in the editor it is saved as a ScriptableObject and a reference to that object is stored in the Database. Most objects that are related (i.e Item Definition and its Item Category) will keep a reference of each other.

Removing Objects

Removing objects can cause confusion as some objects are dependent on others. To make sure things are clear we specify here exactly what happens when one of the objects is deleted.

Removing Item Categories

When an Item Category is removed the connection to the children is broken. If the category has any parents these will be connected to the children. When the category is deleted the children will keep all the attributes that were “required” by the deleted category

All the Item Definitions that had a direct link to the deleted category are either given a category that was a child of the deleted category, or a parent. If neither exists it will be assigned to the Uncategorized category.

If the category is removed at runtime all Item Definitions and Items of that category are unregistered.

Remove Item Definitions

When an Item Definition is removed the connection to the children of the Item Definition being removed is broken. If the Item Definition has a parent then the child Item Definitions will be parented to their former grand parent Item Definition.

If the Item Definition is removed at runtime all of the Items belonging to that Item Definition are unregistered.

Removing Items

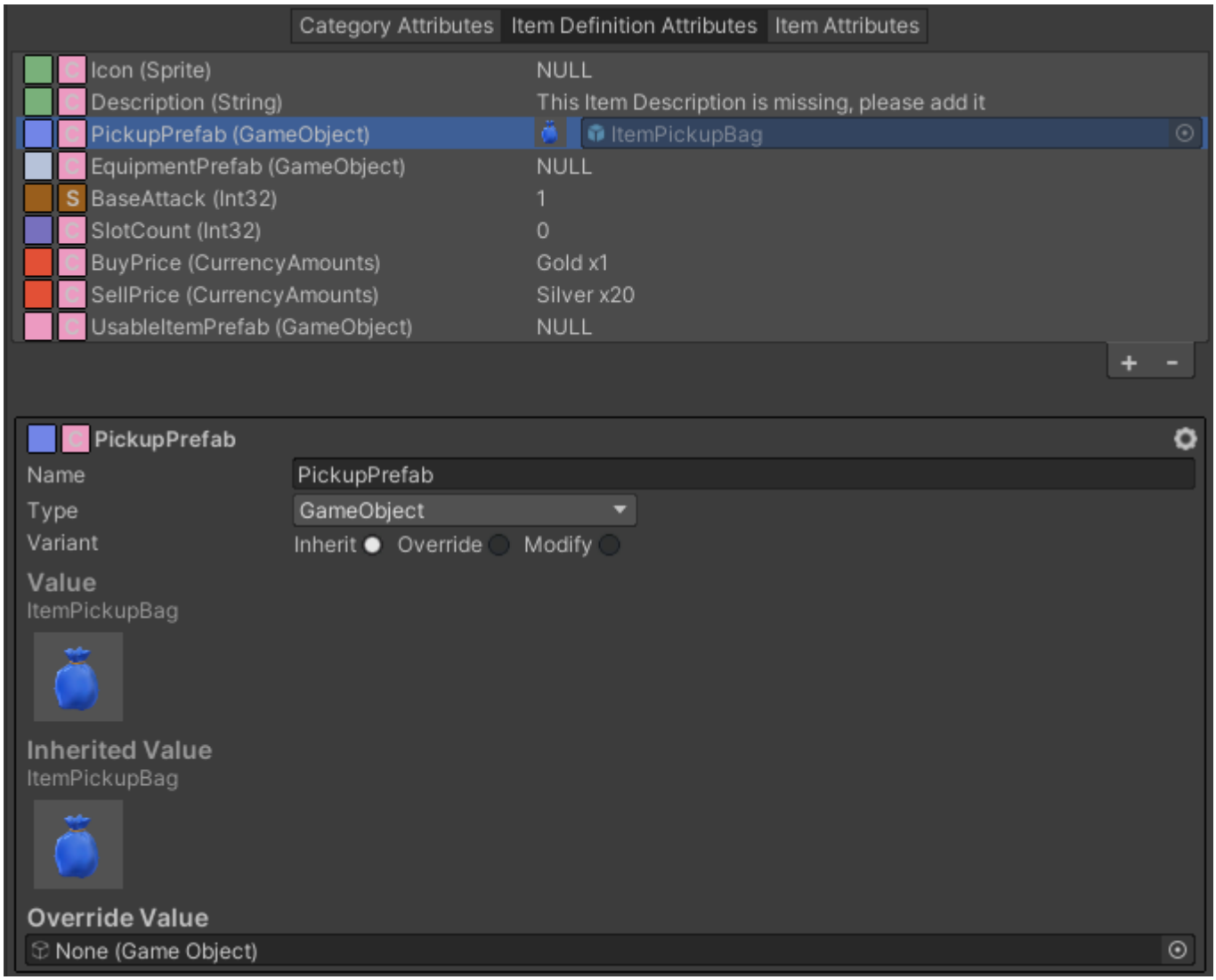
When an Item is removed no extra cleanup is required. The Item Categories and Item Definitions do not depend on the Item.

If a MonoBehavior is dependent on the Item an event function is triggered by the Item Object component just before the item is removed so extra cleanup logic can execute.

Attributes

Attributes are objects that define a value based on a unique name. Attribute values can be changed for Item Categories, Item Definitions, and Items. Attributes can only be created from the Item Category. Attributes created within the Item Category will be passed down to the Item Definitions and Items that belong to the category. Item Category attributes can be thought of as a variable declaration. Attributes within Item Definitions and Items can then be thought of as getting or setting that variable value.

Since attributes are defined from the Item Categories, there are a few options that are only available in the attribute editor within an Item Category. Compare the two screenshots below. The top screenshot was taken within the Item Category editor, and the bottom screenshot was taken within the Item Definition editor.



The list shows the name, type and the value of the attribute. Apart from models, prefabs, and sprites, the values are represented as text. There are two colored boxes next to the Attribute name. The box on the left represent the source of the attribute. The attribute source will always be an Item Category and the category can be accessed by clicking on the colored box. The right box shows where the attribute is being inherited from. The letter within the box shows the inherited object type:

- *C:* Item Category

- *S:* Source
- *I:* Default Item
- *D:* Item Definition

In the Item Category editor the following can be changed:

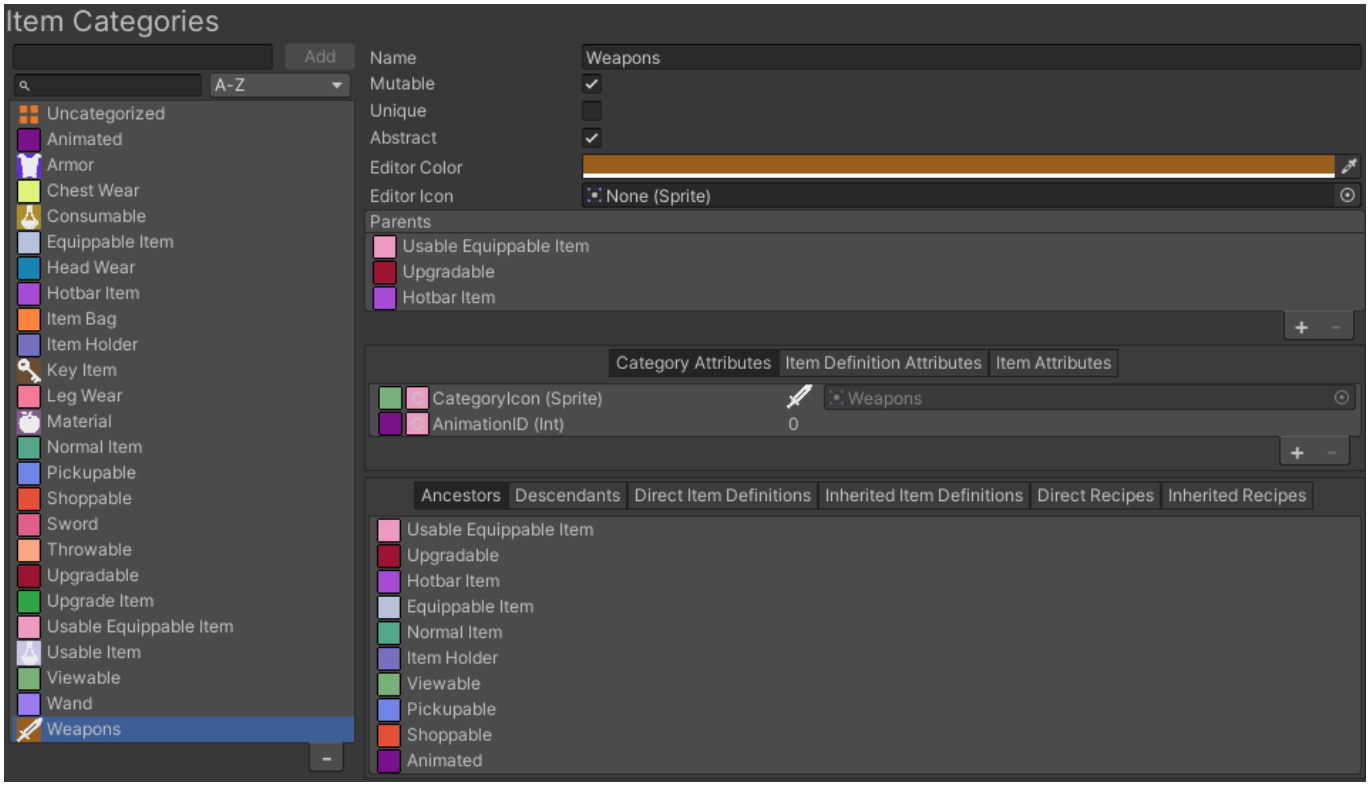
- *Name:* The name of the attribute. The attribute name must be unique for that Item Category and must start with a letter and include only letters, numbers and underscores.
- *Type:* The object type that the attribute represents. After the attribute has been created the type can be changed by clicking on the Type drop down. The selected object type will create a new attribute of generic type Attribute<T> where T is the object type. If a custom attribute type is selected then it will not be a generic type.

  > **Important:** New types can be added to the Types drop down. To do so go to Tools -> Opsive -> Unit Options and add your custom type. More details on this can be found within the Attributes page.

- *Variant:* The *Variant* option has three values:
  - *Inherit:* The attribute value will be determined by the parent attribute.
  - *Override:* The attribute value will be determined by the specified value.
  - *Modify:* The attribute value will be determined by a formula. See the Modify Variant section of the documentation for more details.
- *Pre-evaluate:* This toggle can be accessed from the cog button on the top right of the attribute view. If an attribute is pre-evaluated then the attribute value will be cached and will not be recomputed every time it is accessed. This is most apparent when the *Variant* value is set to Modify.
- *Move To:* This option can be access from the cog button on the top right of the attribute view. This option allows you to easily move an attribute between the three Attribute collection. By moving an attribute between the Item Definition and Item the attribute will keep all the values that were set for its children. If the attribute is moved from the Item Definition or Item to the Item Category then all the non-Item Category attribute values will be lost.

Item Category

Item Categories are used to organize the Item Definitions. The editor allows you to create and edit your existing Item Categories.



The Item Category editor is split in three sections: properties, attributes and relationships. The properties section shows the Item Category specific properties. The attributes section allow you to modify the Item Categories attributes. The relationships section is a useful view for observing how the Item Category relates to other objects within your inventory.

The search bar has some special characters for searching by

- “a:<AttributeName>” : Use the “a:” prefix in the search bar to filter all categories that have an attribute with that name.

Properties

The following Item Category properties can be modified:

- *Name:* The name of the Item Category. Item Categories are Scriptable Objects so the name will also represent the file name. The name must be unique.
- *Mutable:* A mutable Item Category sets its direct items as mutable. Mutable item can be have attribute values change at runtime. By setting the Item Category to be immutable we can assume that it will not change after the items creation. This improves memory usage by having a single item instance instead of many items.
- *Unique:* A unique Item Category sets its direct items as unique. Unique items will not stack in the default Item Collection. Common (not unique) items which are also mutable will merge with similar items when added to an Item Collection this can lose attribute values.
- *Abstract:* An abstract category will not show up as an option for Item Definitions category. Abstract categories have the purpose of organizing your Item Definitions in a clean way.
- *Editor Color:* The color that the category should appear in the editor. This is meant purely to visualize and organize the objects lists more efficiently.
- *Editor Icon:* The icon that should appear in the editor. If no *Editor Icon* is specified the attribute with the name “CategoryIcon” will be used.

- *Parents:* Item Categories can have multiple parents and multiple children. An item category cannot have a parent that is of it's own descendants as that would create a loop. Item Category parents must have non-ovelapping attributes. Attribute overlapping happen when two or more attributes have the same name but different type. If they have the same type then the first parent with that attribute will be used for the attribute inheritance.

Attributes

An Item Category has three attribute views: Item Category, Item Definition and Item. You can access each one individually by clicking on their respective tab.

- Item Categories attributes are inherited only by Item Categories. They are meant to be used for values that are specific the the Item Category or that will be shared by all items in the category.
- Item Definition attributes will be inherited by the the Item Categories and the Item Definitions. These are used to define attributes that will be specific to the Item Definition. This could be an item icon, description, etc.
- Item attributes will be inherited by Item Categories and Items. Item Definitions have a "Default Item" and it can be used to change values of item attributes in the editor. Item attributes are specific to the item it is on and no other. Two items with the same definition may have different item attributes. Examples include lifetime, durability, or item slots. Since only mutable items can change item attributes at runtime it is common to set item attributes only for mutable Item Categories.

The Item Category editor is the only place where you will be able to add and remove attributes. Adding an attribute in the different collections will affect the respective category children and/or the definition children. To add an attribute go in the attribute tab and press the '+' button. To remove an attribute, select it in the list and press '-'. You will be prompted to either remove the attribute for this Item Category and all its parents or remove it for all its and parents and children.

The Attributes page describe how to modify the Item Category attributes.

Relationships

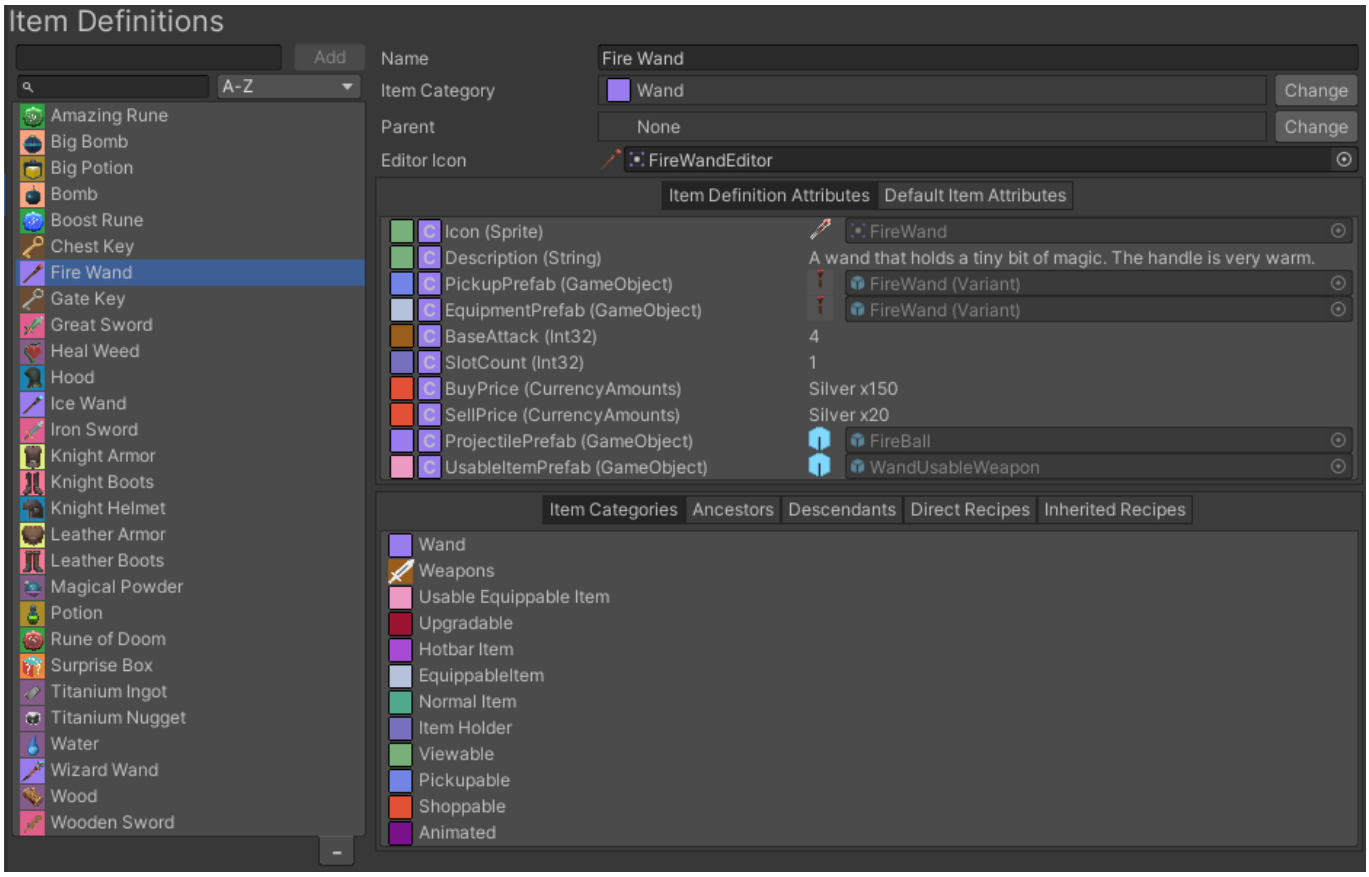
The relationships view shows a few lists which may help you locate the objects that relate to the current Item Category. Once you find the object that you want to view, press the colored box before the name to go to its page.

Relationship tabs include:

- Ancestors
- Descendants
- Direct Item Definitions
- Inherited Item Definitions
- Direct Recipes
- Inherited recipes

Item Definition

Item Definitions can be thought of as templates or molds for items. The Item Definition editor allows you to create and modify these definitions.



The editor is split in three sections: properties, attributes and relationships. The properties section show the Item Definition specific properties. The attributes section allow you to modify the Item Definition attributes. The relationships section is a useful view for observing how the Item Definition relates to other objects within your inventory.

The search bar has some special characters for searching by

- “a:<AttributeName>” : Use the “a:” prefix in the search bar to filter all definitions that have an attribute with that name.
- “c:<ItemCatgoryName>” : Use the “c:” prefix in the search bar to filter the item definitions by definition that inherit the category with that name.

You may right click on the Item Definition in the list to find the following actions:

- Dupplicate : Dupplicate the Item Definition, it is quite useful when making items which have the same Item Category and similar values
- Create Recipe: This action will create a Crafting Recipe with this Item as the Output.

Properties

The following Item Definition properties can be modified:

- *Name:* The name of the Item Definition. Item Definitions are Scriptable Objects so the

name will also represent the file name. The name must be unique.
- *Item Category:* The Item Category that the Item Definition belongs to. When creating a new Item Definition it will automatically be set as Uncategorized. An Item Definition must have a category to function properly and therefore it must never be null. Once an Item Category is set all the attributes will be created for the Item Definition and the default Item. Changing Item Categories will remove the previous attributes if they do not match with the new ones.
- *Parent:* The Item Definition that is a parent of the current Item Definition. A parent Item Definition must have the same Item Category. Once a parent is set, the attributes will be inheriting the attribute values of the parent instead of the Item Category. This allows for quick item variations as you may inherit most of the attributes and change the values of one or two.
- *Editor Icon:* The icon that should appear in the editor. If no *Editor Icon* is specified the attribute with the name “Icon” will be used.

Attributes

The attributes can be set for both the Item Definition and the Default Item. The Default Item is the item that will be used as the template for all other items of that definition. When creating an item at runtime the system will duplicate the default item if it is mutable.

The Attributes page describe how to modify the Item Definition attributes.

Relationships

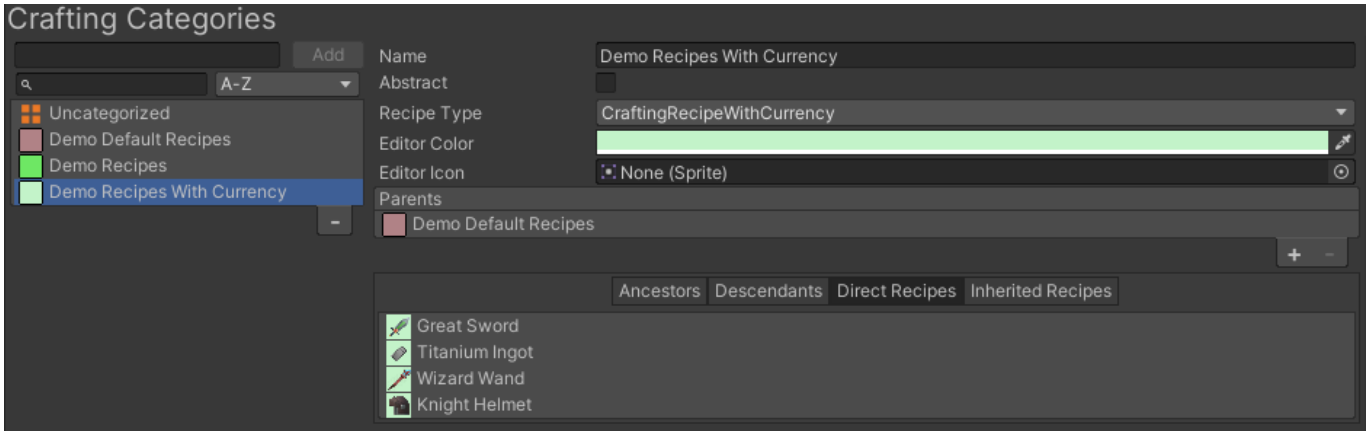
The relationships view shows a few lists which may help you locate the objects that relate to the current Item Definition. Once you find the object that you want to view, press the colored box before the name to go to its page.

Relationship tabs include:

- Item Categories
- Ancestors
- Descendants
- Direct Recipes
- Inherited Recipes

Crafting Category

The purpose of the Crafting Category is to organize recipes and to set the recipe type. Any recipe that is a direct child of the category must have the recipe type defined in the editor. That type must also inherit the base Crafting Recipe type.



The Crafting Categories is split into two sections: properties and relationships. The properties section shows the properties that are specific to the Crafting Categories. The relationships section is a useful view for observing how the Crafting Category relates to other objects within your inventory.

Properties

The following Item Category properties can be modified:

- *Name:* The name of the Crafting Category. Crafting Categories are Scriptable Objects so the name will also represent the file name. The name must be unique.
- *Abstract:* An abstract category will not show up as an option for Crafting Recipes category. Abstract categories have the purpose of organizing your recipes in a clean way.
- *Color:* The color is only used in the editor and is meant purely to visualize and organize the objects lists more efficiently.
- *Recipe Type:* The Crafting Recipe that should be used when processing the recipe. This allows you to define custom logic for determining the output of the crafting recipe.
- *Parents:* Crafting Categories can have multiple parents and multiple children. An item category cannot have a parent that is of it's own descendants as that would create a loop. Item Category parents must have non-ovelapping attributes. Attribute overlapping happen when two or more attributes have the same name but different type. If they have the same type then the first parent with that attribute will be used for the attribute inheritance.
- *Editor Color:* The color that the category should appear in the editor. This is meant purely to visualize and organize the objects lists more efficiently.
- *Editor Icon:* The icon that should appear in the editor. If no *Editor Icon* is specified the attribute with the name "CategoryIcon" will be used.

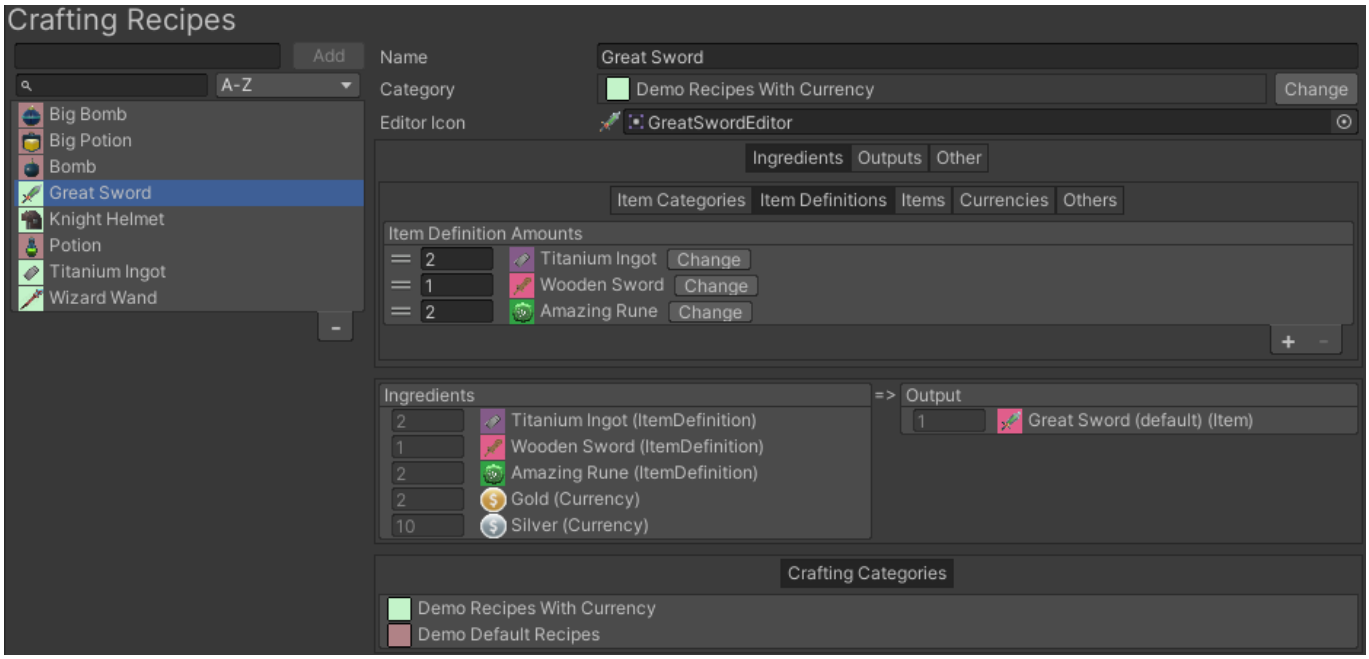
Relationships

For ease of navigation you may find the relationship box quite useful. It includes:

- Ancestors
- Descendants
- Direct Crafting Recipes
- Inherited Crafting Recipes

Crafting Recipe

Crafting Recipes define the input and the resulting output of a recipe. The Crafting Recipe editor allows you to quickly setup these recipes.



The view is divided in four parts: properties, recipe ingredients/outputs/other, visual representation of the recipe, and relationships. The properties section shows the properties that are specific to the Crafting Recipe. The recipe ingredients/outputs/other section defines the recipe. The visual representation allows you to easily see the result of the recipe. The relationship section is useful for observing how the Crafting Recipe relates to other categories.

The search bar has some special characters for searching by

- “c:<CraftingCatgoryName>” : Use the “c:” prefix in the search bar to filter the recipes by recipes that inherit the category with that name.

You may right click on the Item Definition in the list to find the following actions:

- Dupplicate : Dupplicate the Crafting Recipe

Properties

The following Crafting Recipe properties can be modified:

- *Name:* The name of the Crafting Recipe. Crafting Recipes are Scriptable Objects so the name will also represent the file name. The name must be unique.
- *Category:* The Crafting Category that the recipe belongs to. When a crafting category is selected the recipe will be converted to the new Recipe type. During that conversion previous values may be removed.
- *Editor Icon:* The icon that should appear in the editor.

Recipe Definition

The *Ingredients* tab lets you set the ingredients. Item Categories, Item Definitions and Items can be selected for the base recipe type. If you are using the basic crafting processor we recommend you use Item Definitions as ingredients. If you have selected another recipe type more options will show up, for example the recipe above shows a currencies tab.

The *Outputs* tab lets you set the recipe outputs. The output is usually one item, but for flexibility it can be set to a list of items. The options for the selected recipe will appear.

The *Other* tab is available to edit properties on a custom recipe type. All of the serialized fields will be drawn within this view.

Visual Diagram

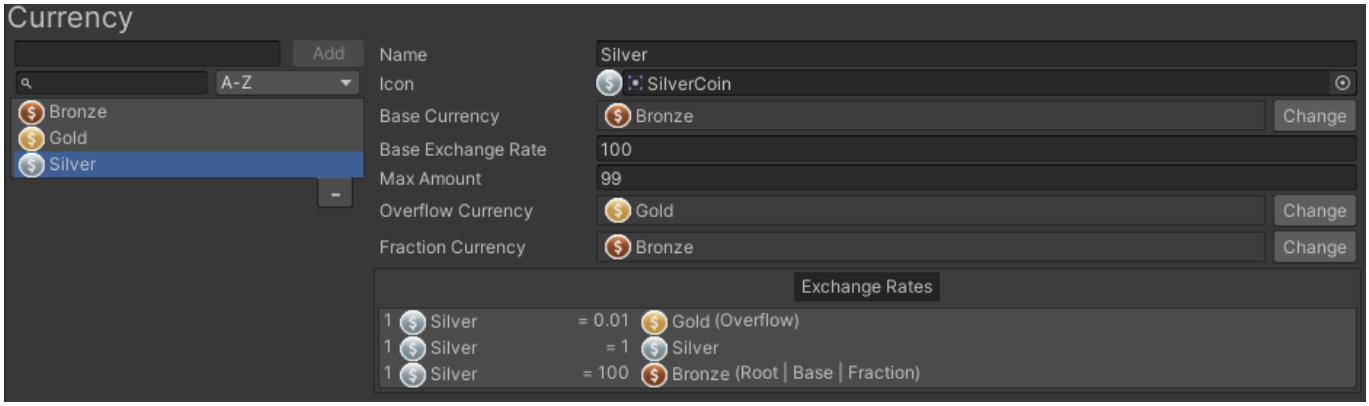
The resulting recipe will appear beneath the recipe definition section. This view will automatically update as the recipe is edited. This view allows you to see the recipe in its entirety instead of split up in multiple tabs

Relationships

The relationships view shows the Crafting Categories that are related to the recipe.

Currency

Currencies can be used to exchange with other objects. The currency editor allows you to define your currency. A currency on its own does not do much, but by nesting them and setting exchange rates you can easily create a multi-currency set up.



The editor contains the following currency properties:

- *Name:* The name of the currency. The name must be unique.
- *Icon:* The icon representing the currency. This is used both in the editor and at runtime.
- *Base Currency:* The currency that the selected currency is made up of. The base currency should always be less valuable than the selected currency.
- *Base Exchange Rate*: The amount of base currency that equates to this currency.
- *Max Amount:* The max amount of currency is used by the Currency Collection to limit the amount of currency it can hold

- *Overflow Currency:* The *Overflow Currency* must be more valuable than the selected currency. This field specifies the currency that will be used as conversion when the currency overflows past its max amount in a currency collection.
- *Fraction Currency:* The *Fraction Currency* must be less valuable than the selected currency. This field specifies the currency that will be used as conversion with fractional the currency amount in the currency collection.

The relationship box shows the relations between the currencies in the family. For currencies to share a family they must have the same source currency (the root currency base). You can click on the currency icon to jump to its page easily. The relation element shows the exchange rate for each currency. It also shows which currency is the Root, Base, Overflow and Fraction.

UI Designer

UI Designer is used to create all UI related to the Inventory System. Our goal is to allow our users to create any kind of inventory for their game. The User Interface is a big part of what makes an inventory unique so we came up with a new UI solution which should allow you to create very distinct inventory UIs in just a few clicks. That is possible thanks to our UI Designer schemas, which are Scriptable Objects that have a collection of prefabs and assets used to create your UI in the editor.

This page explains in more detail about the UI Designer schemas.

Our UI solution is extremely modular, flexible and extensible. But that also comes with a lot of small components which could have been slightly hidden. That's why we made UI Designer, to have all tools necessary to create your very own UI directly in the editor with shortcuts to create, find and edit your Inventory UI.

We understand that some of you will want to create your own UI, that's why you can pick and choose the components you want to use. The inventory system is independent from the UI so you may choose to not use it at all.

Getting Started

Important : Before Using the UI Designer you should set up the Inventory Database you plan to use and reference it in the Main manager Setup tab.

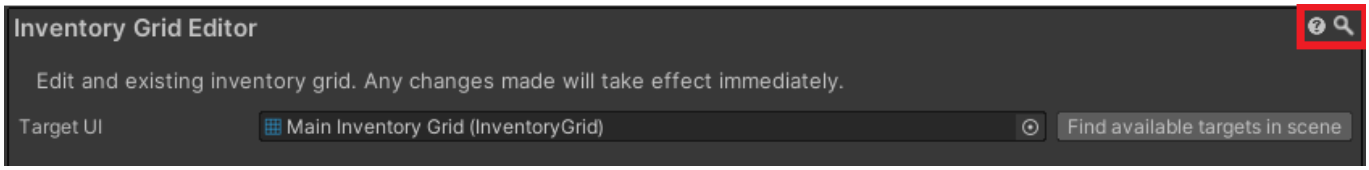
The first step to use the UI Designer is to create the Canvas Managers from the Tools -> Opsive -> Ultimate Inventory System -> UI Designer menu option. In the Setup tab under the "Create Canvas Managers" Click "Setup", This will spawn the canvas and manager scripts required by the UI system.

Then under the UI Designer Schema section select a schema you like and duplicate it by pressing "Duplicate". You won't be able to use UI Designer until you duplicate the schema. This is to make sure that you will have your very own collection of prefabs in case you make changes.

Once you have duplicated the schema press the big "Spawn In Scene" button to quickly

setup a basic menu for the schema. From there the other tabs will be available to you and you may use them to create, find and edit your UI.

Important: Note the little question mark and the magnifying glass in the top right of most of the UI Designer sub menus. These are used to direct you to the relevant documentation, and select the relevant component in the scene hierarchy.



Most tabs are separated in two parts: Create and Edit.
In the Create section it is required to specify where the UI should be spawned by selecting a Parent Transform. Most times the creator comes with some options that will determine the prefabs/assets in the schema used to create the object.

The Edit section has a very handy “Find Available Targets in Scene” button. You may also select component by dragging it in the field.
Once an object is targeted the edit options will appear, giving you information about your UI and allowing you to edit it.

Schemas

UI Designer uses UI Designer schemas to create the UI in the editor. UI Designer schemas are a collection of prefabs and assets which can be assembled together like Lego pieces to build your UI. We offer the schemas bellow and we may add more in the future:

- **Classic:** A minimalist UI with big icons and buttons, a great starting point for any Inventory UI
- **RPG:** A UI with a cartoony adventure look, which focuses on floating panels and Item Shape Grids.

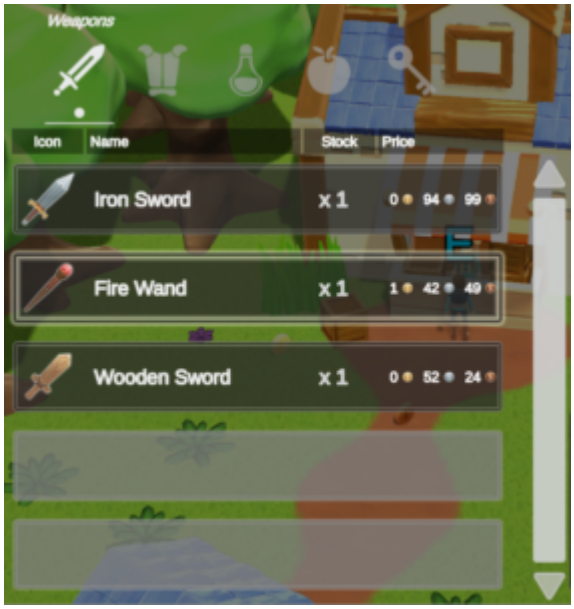
You can learn more about each schemas in the documentation sections below. You may even create a share your own UI Designer schema if you wish. All that is required is to follow the constraints for each prefab in the schema.

Classic

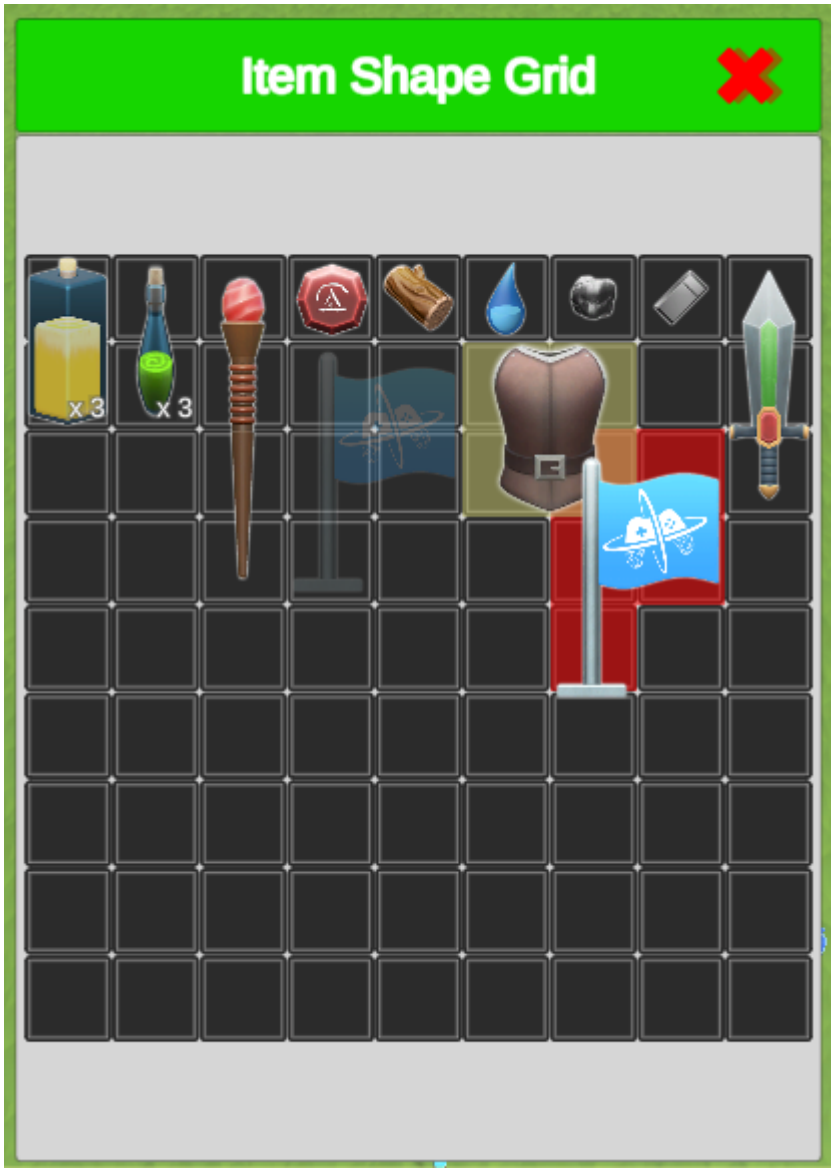
The classic schema is a minimalist UI with big buttons and icons inspired by the game “The Legend of Zelda: Breath of the Wild”. It is perfect to get started quickly and and the classic schema is used for the demo. Here are a few images of what you may expect from this schema:

Inventory Grid

Inventory List

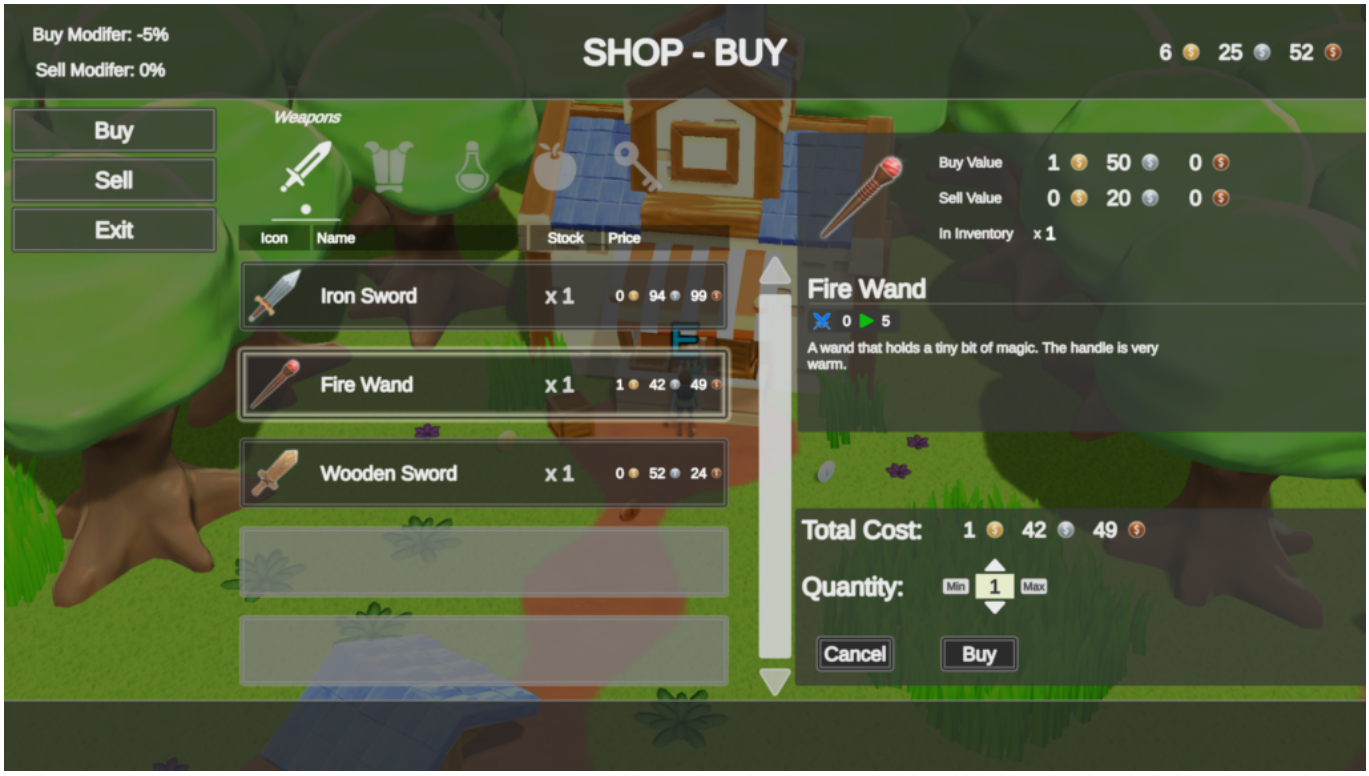


Inventory Grid Item Shape



Item Hotbar

Shop Menu

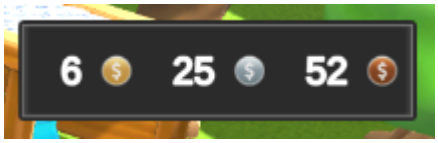


Crafting Menu

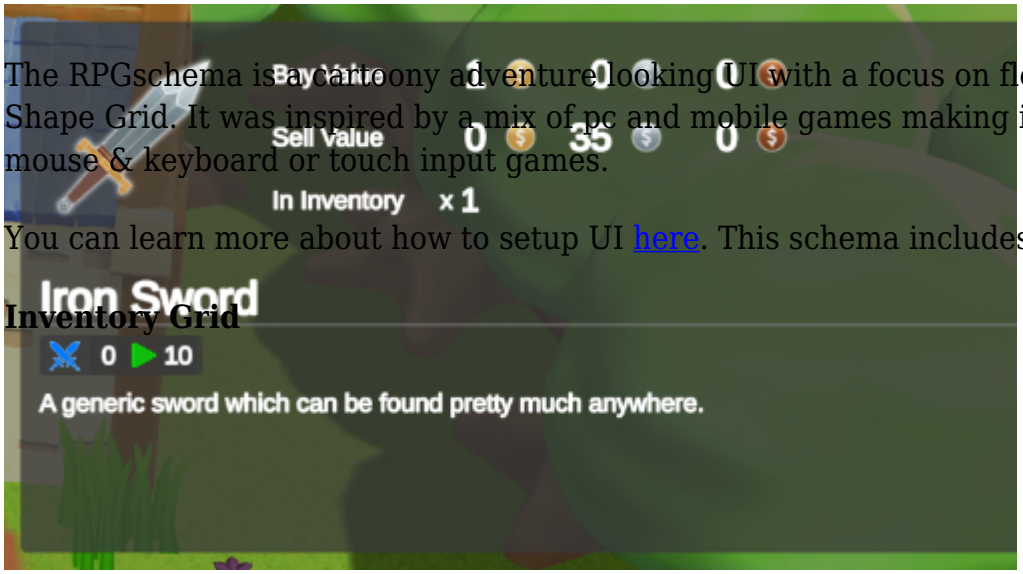
Save Menu

Item Description

Multi Currency View



RPG

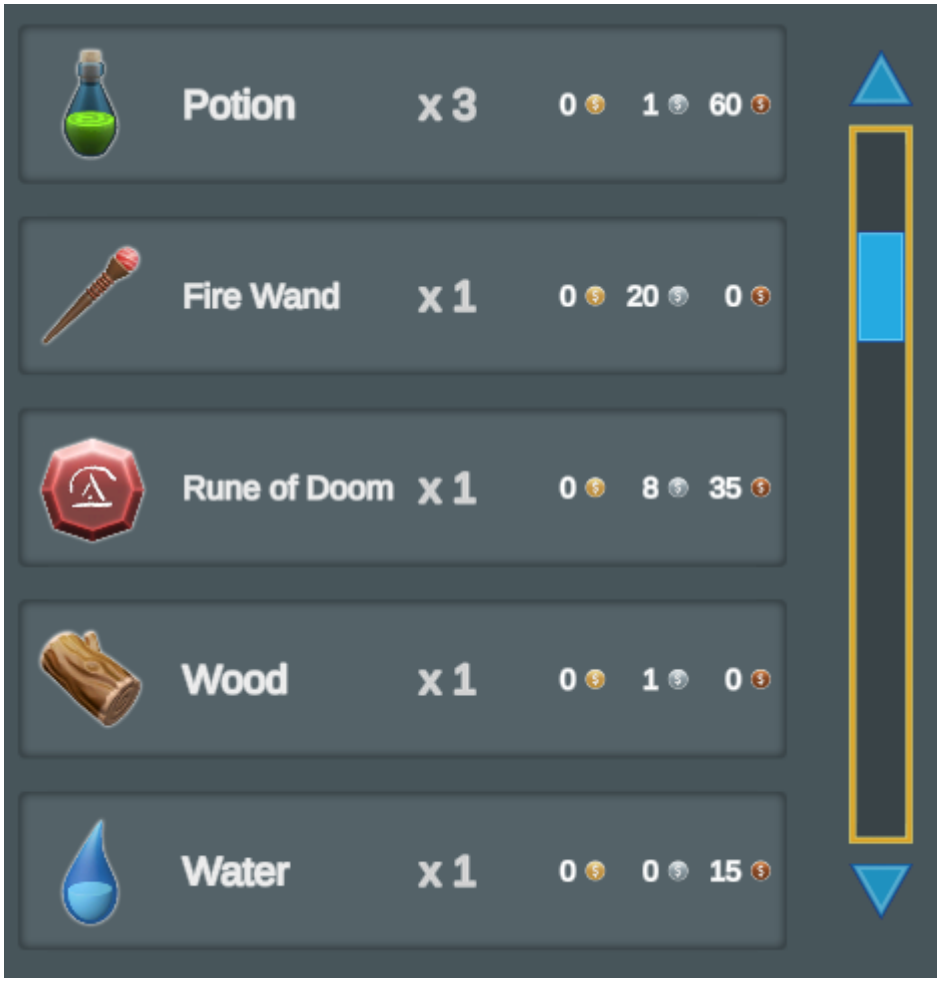


The RPGschema is a cartoony adventure looking UI with a focus on floating panels and Item Shape Grid. It was inspired by a mix of pc and mobile games making it mostly suitable for mouse & keyboard or touch input games.

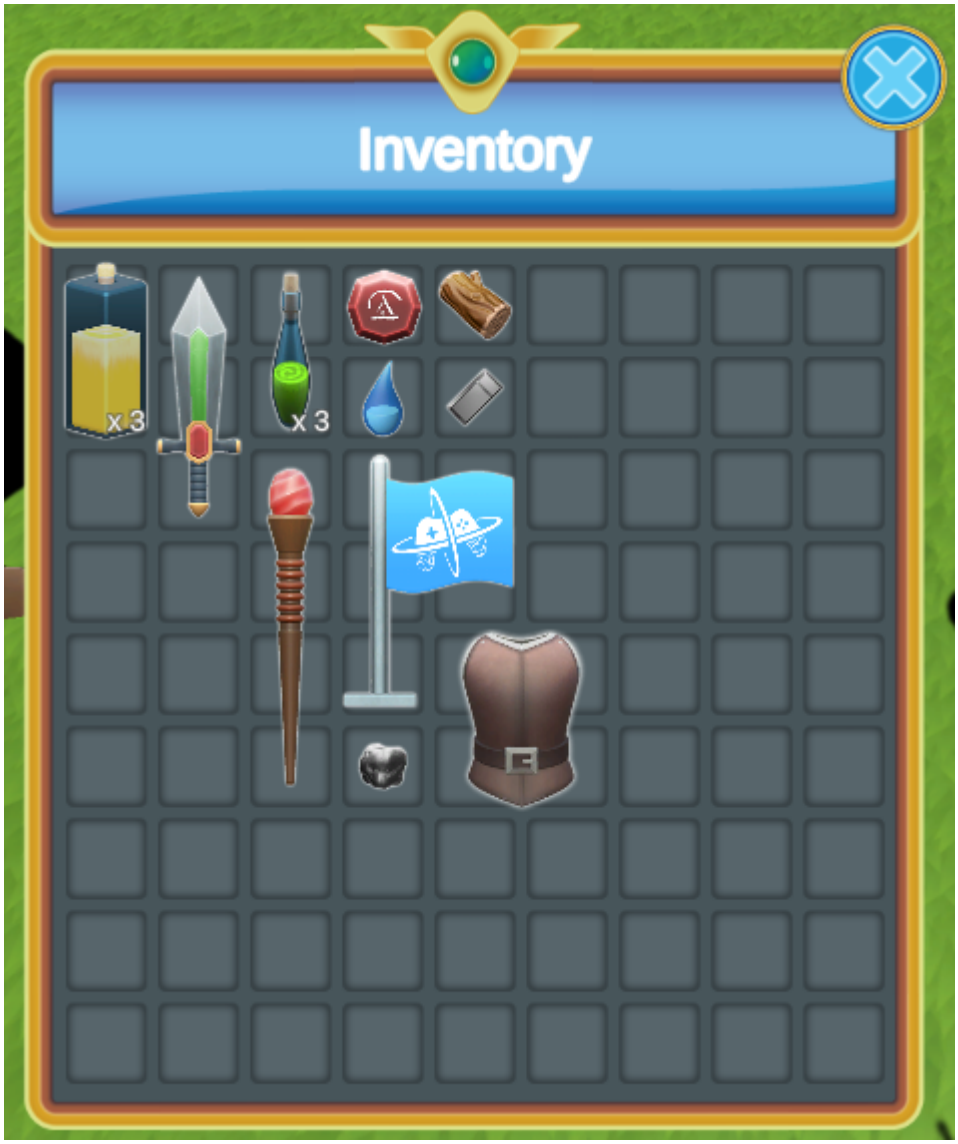
You can learn more about how to setup UI here. This schema includes:

Inventory Grid

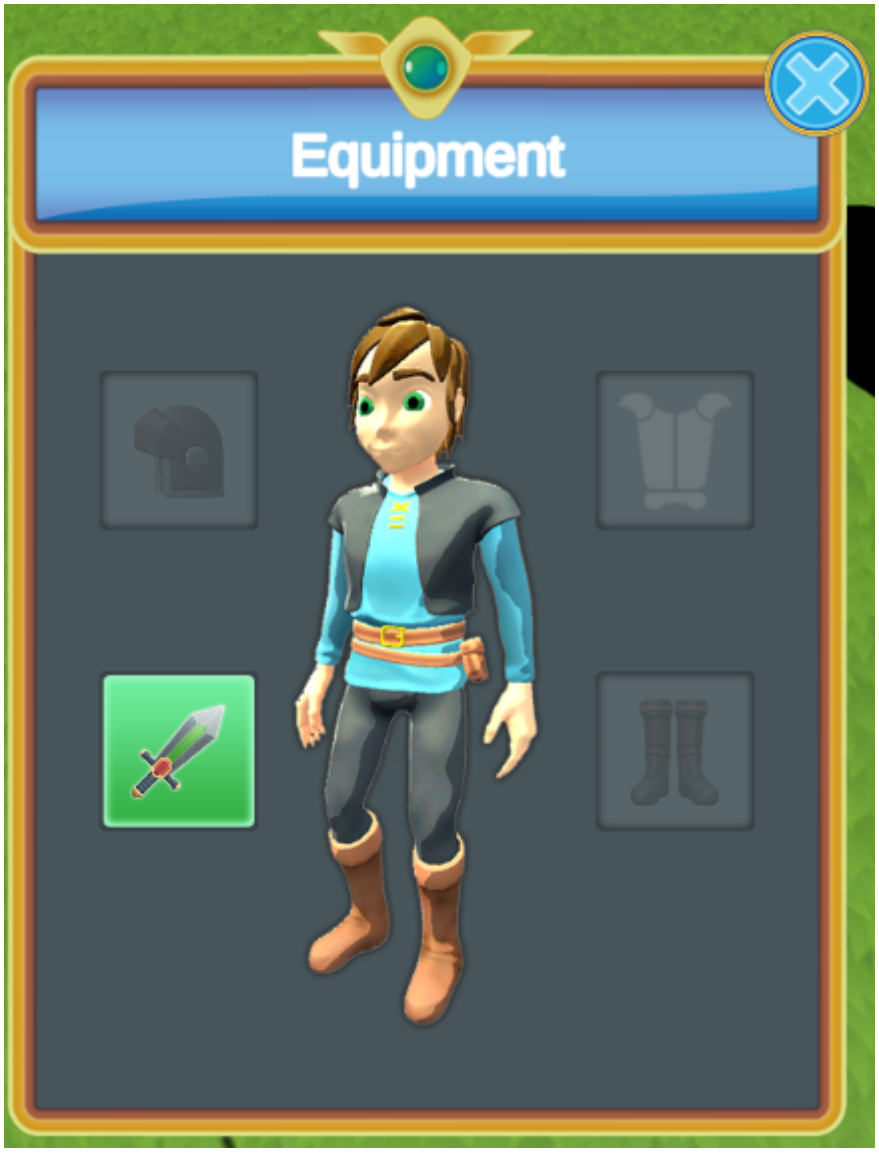
Inventory List



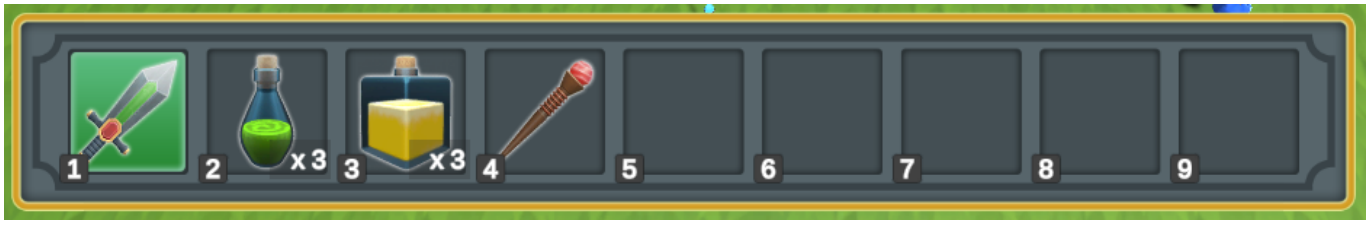
Inventory Grid Item Shape



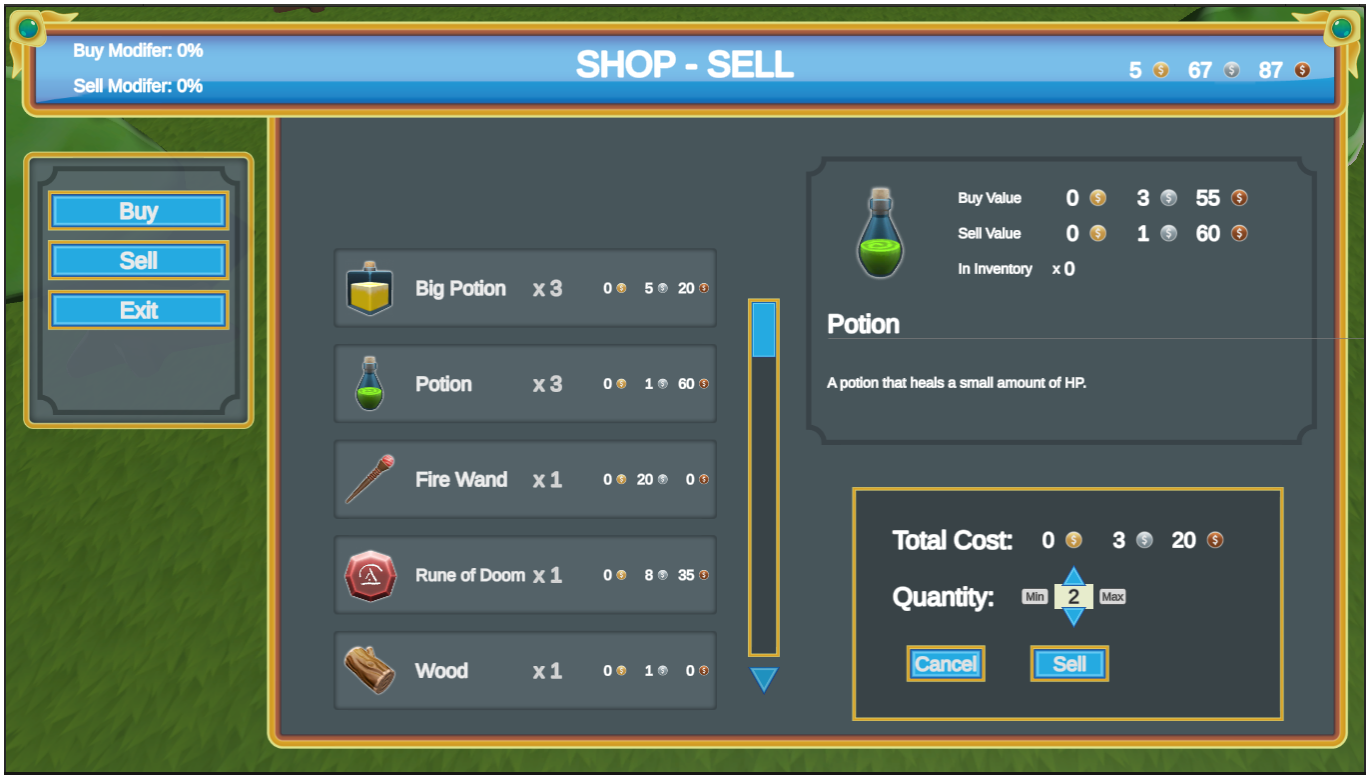
Equipment (Item Slot Collection View)



Item Hotbar

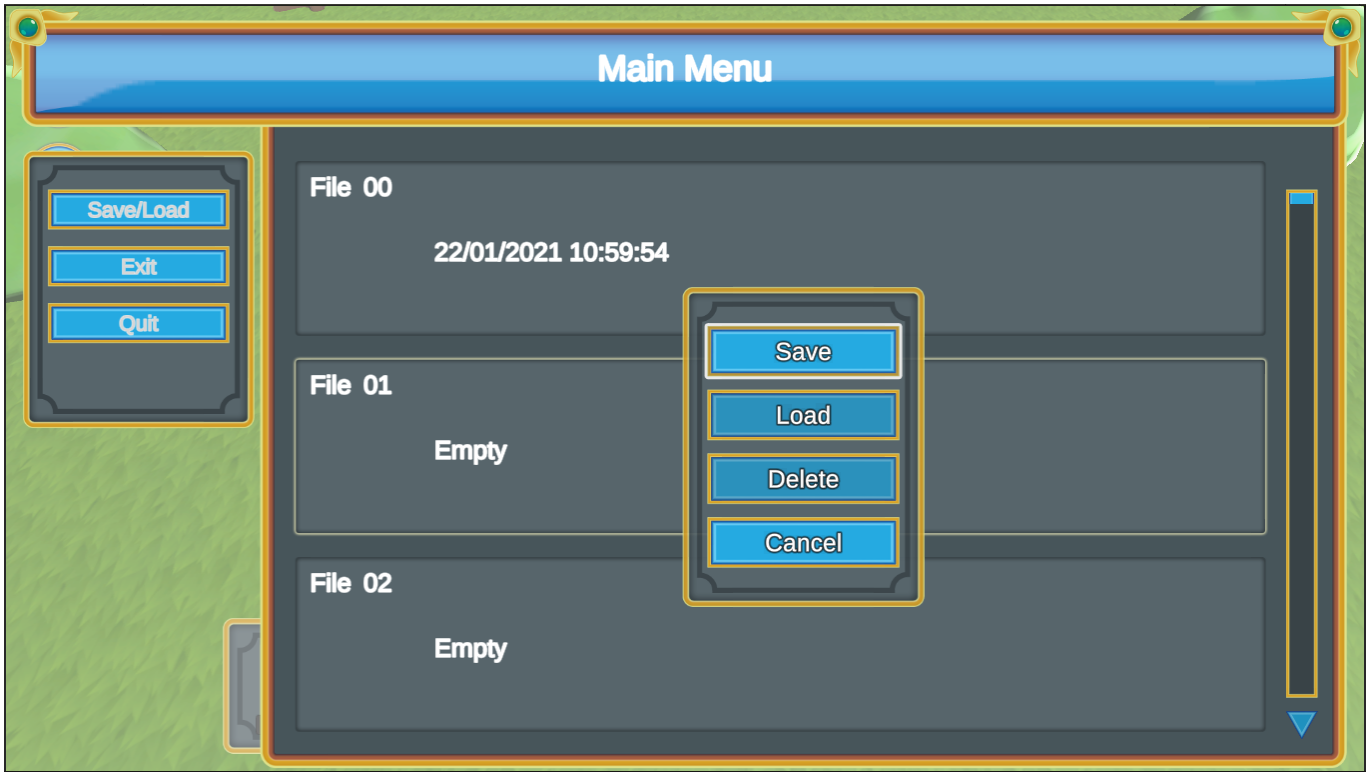


Shop Menu

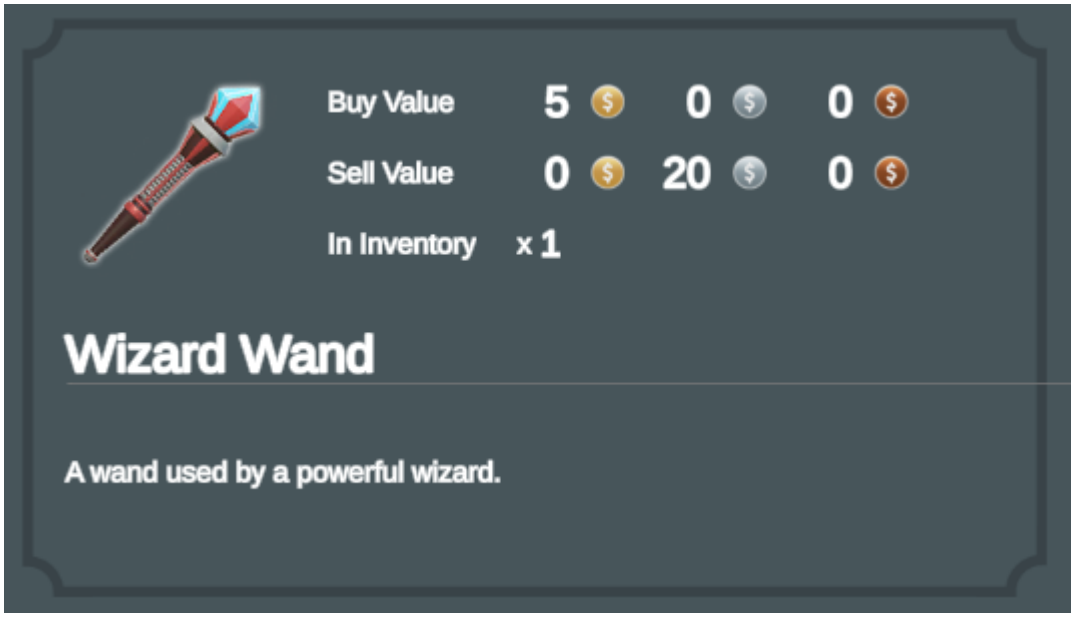


Crafting Menu

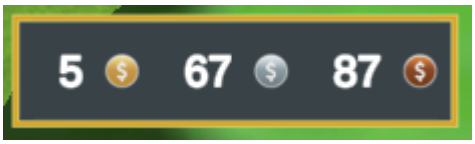
Save Menu



Item Description



Multi Currency View



Main Menu

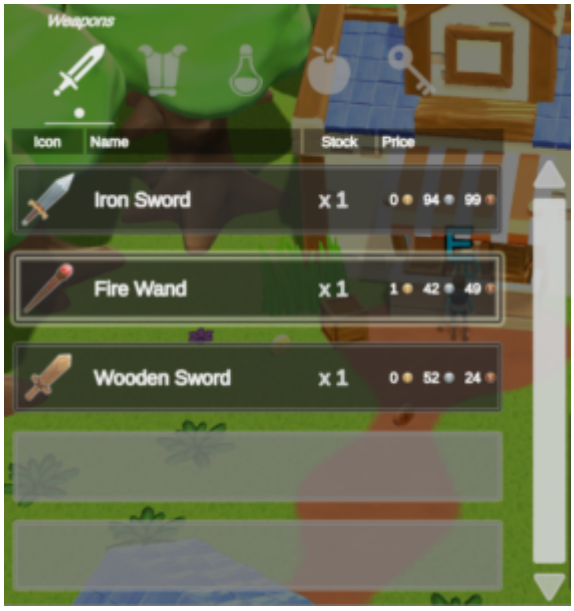
The Main Menu is a component which can have sub panels. It is referenced by the Display Panel Manager such that you may easily open and close it.

Learn more about the Main Menu component here. The Edit sections allows you to find the main menus sub panels and the buttons that opens them. To Add panels to the Main Menu simply select the “Main Menu” option in the Panel Options when creating one of the

available panels (Inventory Grid, Save Menu, Crafting Menu, etc.)

Inventory Grid

The Inventory Grid is used to display the content of an inventory. The Inventory grid can display lists and grids of items. To set these up efficiently it is highly recommended to use the Inventory Grid Creator in the UI Designer.



This page contains more information about the Inventory Grid component. The component is an Item View Slot Container and the documentation for the common edit options can be found on this page.

Create Options

When creating an Inventory Grid, a few options are available to choose from.

- Parent Transform: Choose the Rect Transform where the Inventory Grid will be

spawned under.
- Panel Option : A few prefabs are available as Panels, the panel will be the parent of the inventory grid
    - Basic : Just a simple Rect Transform with a Panel Component
    - Simple: The panel has a header with a title
    - Floating: The panel has a header with a title, the header has a drag handler component which allows the player to move the panel with the mouse
    - Main Menu: The panel is specially made to fit inside the Main Menu. The parent Transform must be the main menu panel content transform
- Panel Name : The name of the panel. The Display Panel Manager can be used to get panels by name
- Inventory : You may set the inventory that the Inventory Grid will monitor
- Grid Options : There are a few types of Inventory Grids
    - Grid : A simple Inventory Grid
    - List : Makes an Inventory Grid will a single column and a scrollbar
- Inventory Grid Name : The name of the Inventory Grid can be useful in a few use cases, the most obvious one being drag&drop conditions

Grid Size & Layout Group

The grid size should be set from UI Designer such that the item view slots can be set correctly set to preview the grid size. The default Unity Layout groups are used to space out the item view slots. It is important to set it to know the directional axis of the grid.

The Layout Group Navigation is used to to easily set up explicit selectable navigation by checking only neighboring components. It uses the Layout Group to have information about the order in which the item view slots should be connected.

Grid Navigation

The Grid Navigator, not to be confused with layout group navigation, is used to show more items than the grid size can show. You may scroll through, page navigate, tab navigate, etc. It works both with mouse and with keyboard/controller.

Filters & Sorters

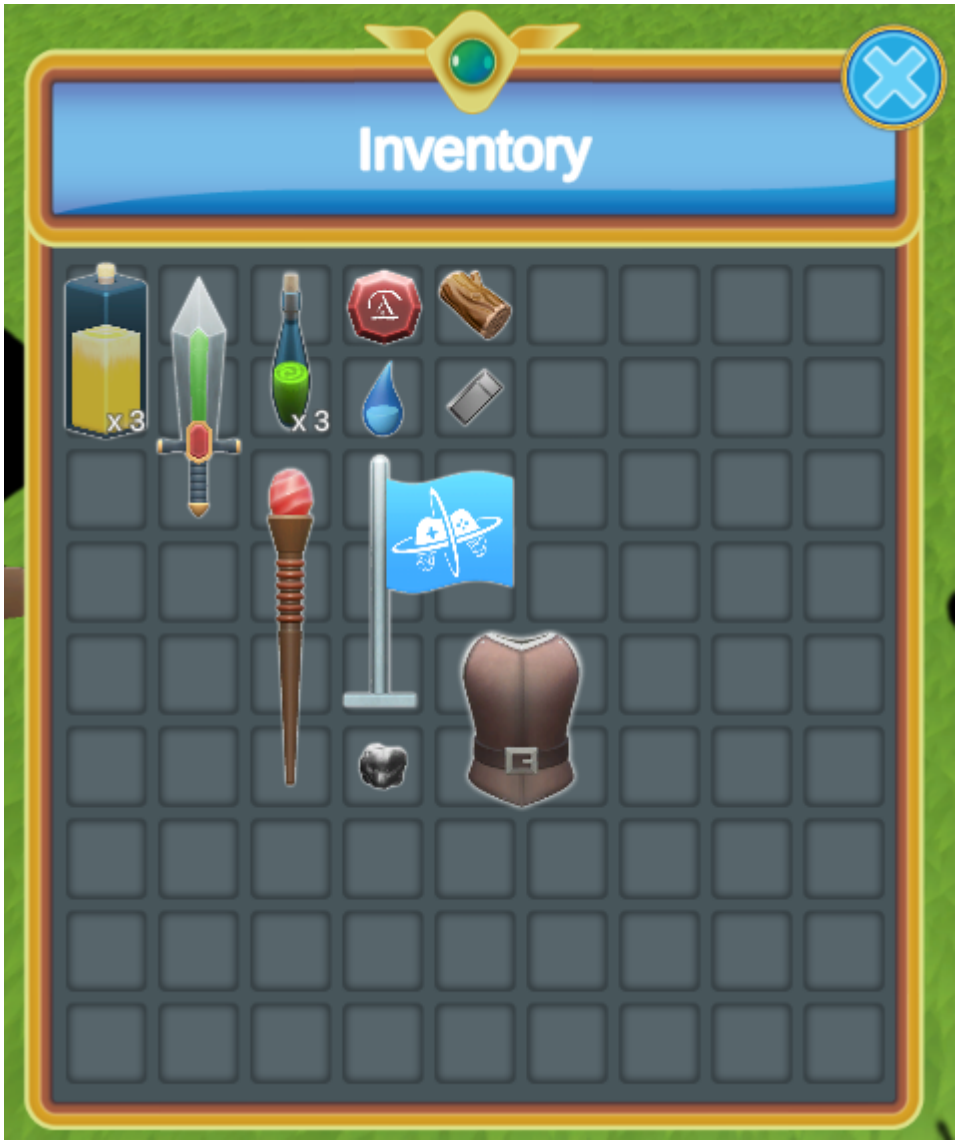
Filters and sorters can be added on the grid directly or swapped dynamically at runtime. Use the Default filters to set a default sorter/filter for the items displayed in the grid. Use the search bar to create a search bar that filters the grid whenever a user starts typing in it. Use the sort dropdown to sort the items within the grid at runtime.

Grid Tabs

Grid tabs are controlled by a Tab Controller and Tab Toggles. In the case of Inventory Grids, each type will have an Inventory Grid Tab Data which has a reference to the filter to use when that tab is active and more.

Item Shape Grid

The Item Shape Grid is used to display the content of an inventory that is organized by an Item Shape Grid Controller & Data. This allows items to take multiple slot within a finite grid. To set it up efficiently it is highly recommended to use the Item Shape Grid Creator in the UI Designer. The “RPG Schema” is focused on Item Shape Grids. The Item Shape Grid should not be confused with an Inventory Grid as they function quite differently.



This page contains more information about the Item Shape Grid component. The component is an Item View Slot Container and the documentation for the common edit options can be found on this page.

Create Options

When creating an Item Shape Grid, a few options are available to choose from.

- *Parent Transform:* Choose the Rect Transform where the Inventory Grid will be spawned under.
- *Panel Option:* A few prefabs are available as Panels, the panel will be the parent of the inventory grid.
    - Basic: Just a simple Rect Transform with a Panel Component.

    - Simple: The panel has a header with a title.
    - Floating: The panel has a header with a title, the header has a drag handler component which allows the player to move the panel with the mouse.
    - Main Menu: The panel is specially made to fit inside the Main Menu. The parent Transform must be the main menu panel content transform.
- *Panel Name:* The name of the panel. The Display Panel Manager can be used to get panels by name.
- *Inventory:* The Inventory must be assigned as some components will be automatically added (Item Shape Grid Controller & Item Shape Grid Data).
- Grid Name: The name of the Grid can be useful in a few use cases, the most obvious one being drag&drop conditions.
- *Grid Size:* The number of cells within the grid.
- *Item Shape Size:* The size of each cell within the grid (in pixels).
- *Item Collection:* The Item Collection that the Item Shape Grid Data should monitor. Setting it to non will monitor the entire inventory.

Edit Options

The edit options allow you to see if there is anything wrong with your Item Shape Grid or the relevant inventory. There are also options to edit the grid size and Item shape size.

Since the Item Shape Grid is an Item View Slot Collection, all the common options will also be available.
Note that since Item Shape Grids are different from the usual Item View Slot Collection, the Item Views must have some specific Item View Modules to work correctly, please refer to the Item Shape Grid page to learn more.

Equipment

The Equipment panel is made using an Item Slot Collection View. It allows you to create an Item View Slot Container that has an Item View Slot for each Item Slot within an Item Slot Set. You can set the Item View Slot Restrictions to allow only items with a specific Item Category to be added in a slot.

Learn more about Item Slot Collection View here. The component is an Item View Slot Container and the documentation for the common edit options can be found on this page.

Create

An Item Slot Set must be assigned to create the correct number of Item View Slot and set their Item Category restrictions.

Equipment Options

The Equipment Options is defined by the Item Slot Set. The editor gives you easy access to select your Item View Slot in the hierarchy as well as adding more Item View Slot Restrictions.

Item Hotbar

The Item Hotbar displays items to the player in a way that allows for easy access to them, whether it is to use item actions using input or drag and dropping item view slots.

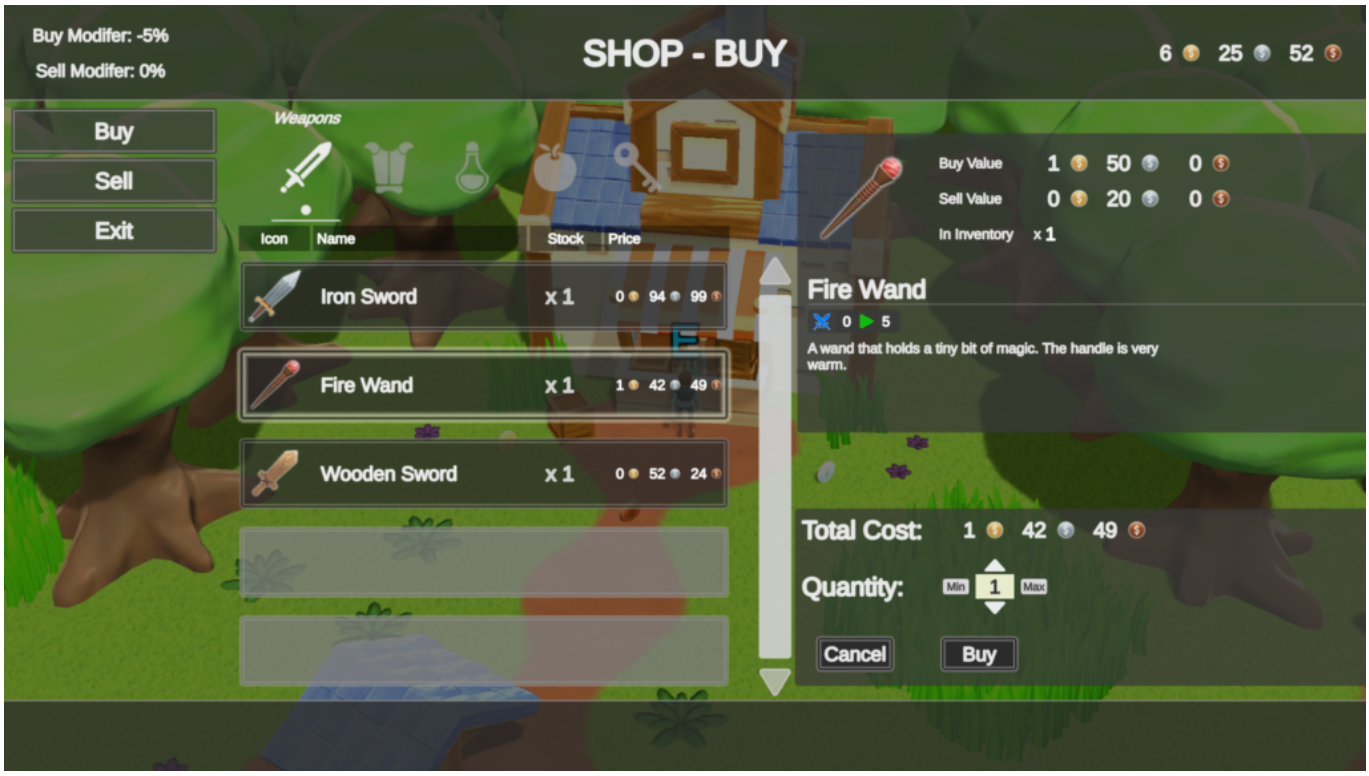
Learn more about the Item Hotbar on this page. The component is an Item View Slot Container and the documentation for the common edit options can be found on this page.

Hotbar Options

Find the Item View Slots for your hotbar, add, remove and edit them.

Shop

The Shop Menu is used to display a user interface where the player can buy and sell items through the Shop component.



To learn more about the Shop Menu go to this page. The Shop Menu may be created as a basic menu or be set within the Main Menu.

The UI Designer allows the following changes to an existing Shop Menu:

- Add a Shop component directly on the Shop Menu.
- Add the Inventory Grid which displays the shop items.
- Add the Multi Currency View used to show the full price.
- Add a Quantity Picker to select the amount to buy/sell.

Crafting

The Crafting Menu is used as an interface to craft items using the Crafter component.

To learn more about the Crafting Menu component see this page.

Important: This Crafting Menu is one solution for crafting, but it is highly recommended to extended the crafter or the crafting processing to create custom crafting solutions. This would then mean creating a custom Crafting Menu. Head to the forum to discuss possible solutions for our use case and share your experience with other users.

The Crafting Designer allow you to create a Crafting Menu with a basic panel or as a Main Menu sub panel. Editing an existing Crafting Menu allows you to:

- Add a Crafter directly on the Crafting Menu.
- View and Edit the Recipe Panel (used to display the recipe selected to craft).
- Edit the Grid Size and Layout of the Recipe Grid.
- Edit the Grid Navigation (similar to Inventory Grid Navigation).
- Add/Remove Grid Tabs.

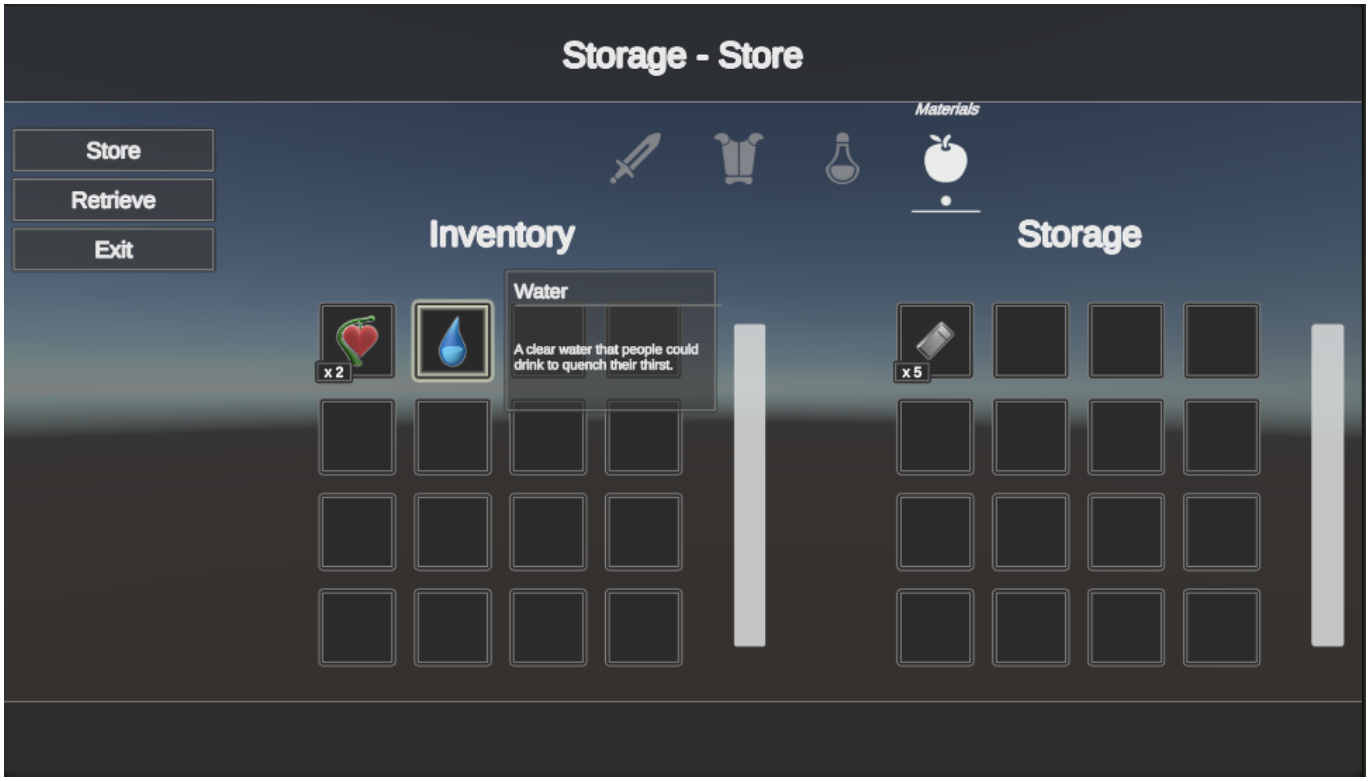
Save

The Save Menu is used to save and load data for your game. Since most projects will use a custom save solution our save menu is very basic and simple.

To learn more about the Save Menu component see this page. The Save Designer tab allows you to create the save menu in a basic panel or as a Main Menu sub panel. The UI Designer allows you to find all the Saver components in your scene to know what is currently monitored to be saved and loaded. You may also edit the grid size, layout and navigation.

Storage

The Storage Menu is used to exchange items from one Inventory to another. This is usually between the “player character” Inventory and a “storage” Inventory.



When using a Storage Menu created using UI Designer you may receive a few warnings:

> Some of the categories referenced on the item info category filter do not reference the right database.

Simply click on each warning to find the tab filter which causes the warning and replace the category referenced by the filter.

The rest of the Editor gives easy access to the different components of the Storage Menu. It also contains short cuts to start editing the Inventory Grid for the player and for the storage.

Learn more about the Storage Menu here.

Chest

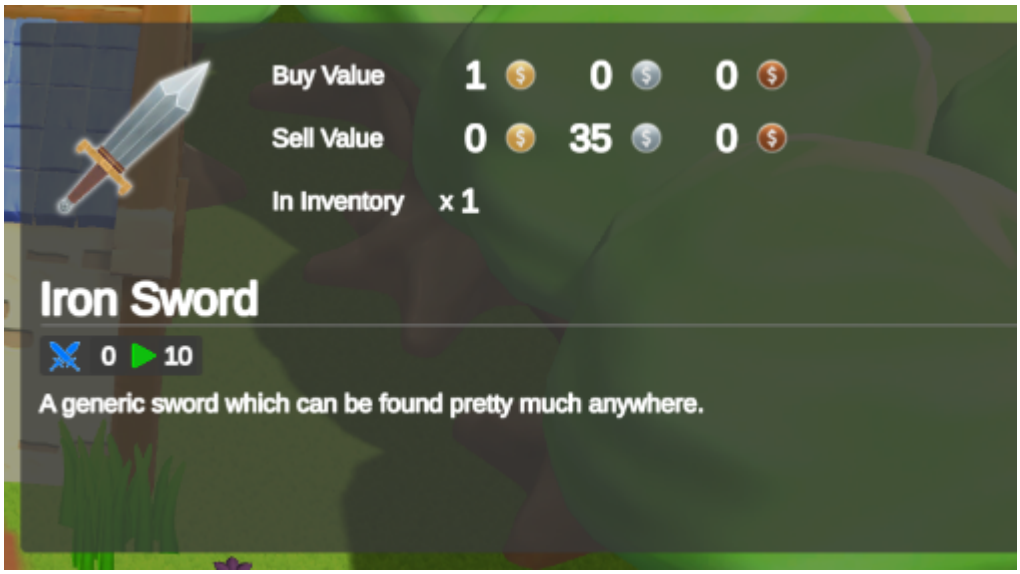
The UI Designer Chest components in the scene will automatically find the Chest Menu and assign themselves to it when opened. The easiest way to get started with the Chest Menu is to simply spawn a “Chest” prefab from the Demo folder in your scene. When the player interacts with the chest through the Inventory Interactor and Interactable components, the Chest Menu will be opened and bound the the chest the player interacted with.

Learn more about Chest Menu here.

Important: This Chest Menu is one solution for making chest in a game, but it might not suit your style of game. Feel free to create your own kind of Chest Menu using advantage of the Inventory Grid (or a custom Item View Slot Container) and other UI component. You may also want to create a custom Chest component if you decide to create a custom Chest Menu.

Item Description

The Item Description component is a special Item View that is specifically designer to be used to display the Item Info description.

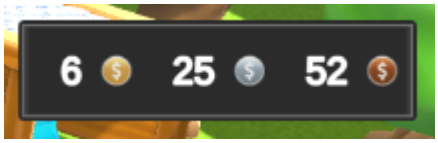


To learn more about Item Description see this page.

The Item Description Designer tab has a few description templates to get started quickly.nWhen editing an Item Description you may add/remove Item View Modules. You may also bind the Item Description to an Item View Slots Container, which displays the description of the selected Item.

Currency

Currencies are displayed using a Multi Currency View. It maps a Currency to a Currency View. Once a Currency Amount or a Currency Collection is set the Multi Currency View will display the amount of each currency specified.



The Currency Designer tab allows you to create Multi View Currencies, View Currencies and Currency Owner Monitors from templates defined in the schema.

To learn more about those components see this page. The editor may also find Multi Currency Views that are anywhere in the scene.

Inventory Monitor

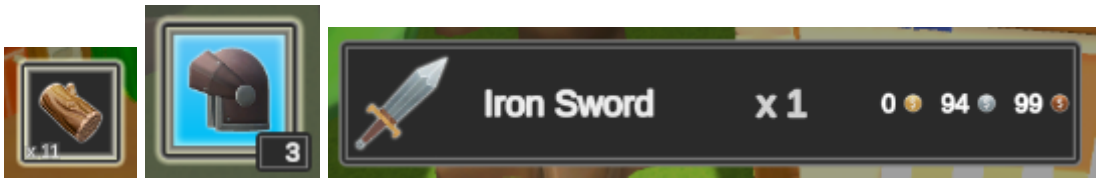
The Inventory Monitor is used to pop up Item Views on the screen whenever items are

added to an inventory. To learn more about the Inventory Monitor see this page.

The component is very simple therefore UI Designer simply creates a Inventory Monitor component.

Item View

Item Views are used to display an item in the UI, it works with Item View Modules which listen to changes on the Item View.



This page explains more about the Item View (Modules) component.

When using the UI Designer multiple Item Views are available as templates. Once a new Item View is created they may be edited to add/remove Item View Modules.

Item View Module Types will appear automatically in the drop down when creating a new module. To make sure custom Item View Modules appear in the list make your custom script inherit the Item View Modules class or the base View Module class.

Example of Item View Modules are:

- *Name Item View*: Writes the name of the item in a Text/Text Mesh Pro component.
- *Icon Item View*: Displays the Sprite "Icon" attribute of the item in an Image component.

Attribute View

Attribute Views are used to display an attribute value in the UI, it works with Attribute View Modules which listen to changes on the Attribute View.

For more information on the Attribute View component see this page.

Once a new Attribute View is created with UI Designer they may be edited to add/Remove

Attribute View Modules.

Attribute View Module Types will appear automatically in the drop down when creating a new module. To make sure custom Attribute View Modules appear in the list make your custom script inherit the Attribute View Modules class or the base View Module class.

Example of Attribute View Modules are:

- *Value Attribute View*: Writes the attribute value as a string in a Text/Text Mesh Pro component.
- *Float Value Attribute View*: Writes the float attribute value with the customizable decimal place in a Text/Text Mesh Pro component.

Item View Slots Container

Item View Slots Container is used by the InventoryGrid, Item Hotbar and Equipment panel. They all share the same base logic of having an array of Item View Slots. Therefore all the UI Designer tabs for those components share some of the same functionality: Item View Drawer, Item Actions, Item Moving (drag & drop), Item Description, etc.

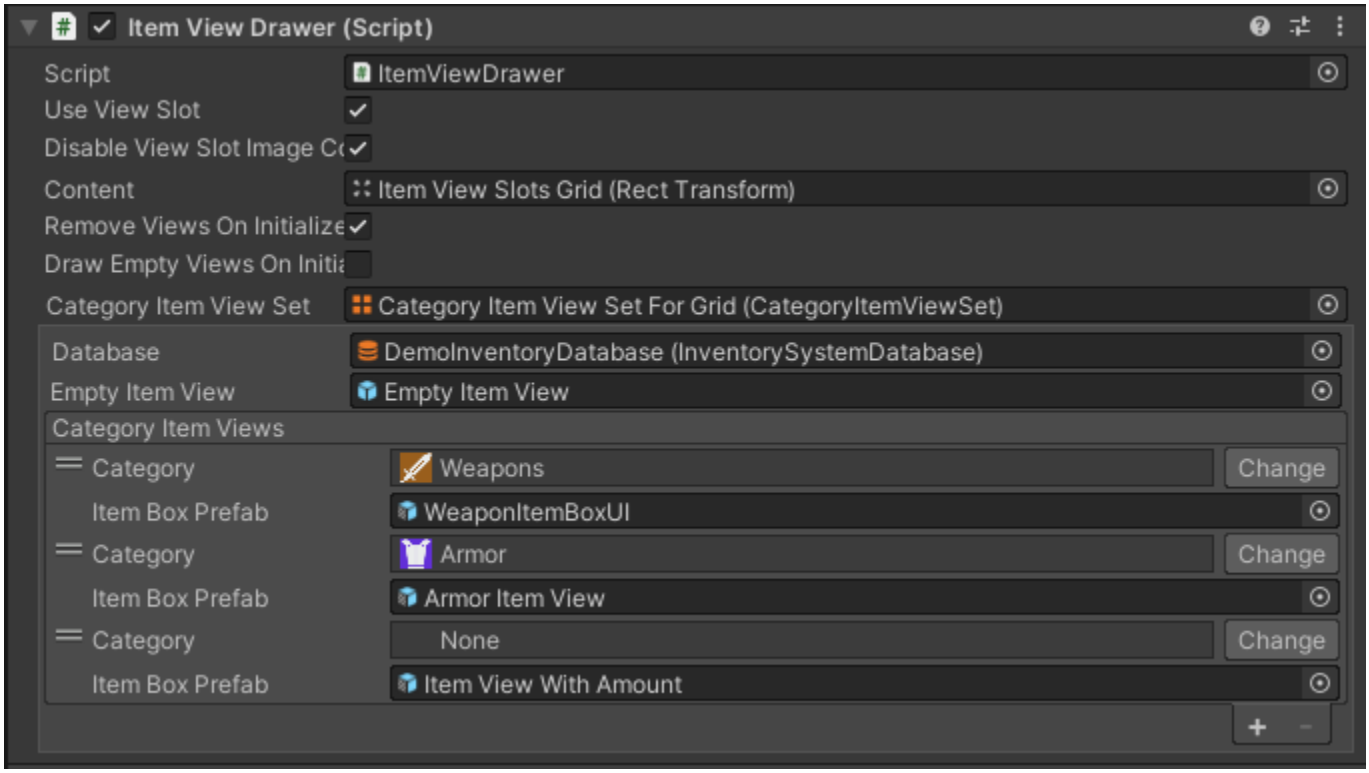
For more details on the View Slots Container see this page.

Item View Drawer

The Item View Drawer is used to spawn the appropriate item views for the item to display. The Item Views can be set as prefabs and are organized by Item Categories. The order and inheritance are important to find the best match for the item to display.

When using Item View Slots it is required to toggle on the “Use View Slots” option. Most of the time Item View Slots have images to preview the item views positions, you may disable that image such that the Item View has all the visual. It is important to the the Content transform to the parent of your Item View slot components.

The Category Item View Set is the Scriptable Object that will match the Item Categories to Item Views.



Item Actions

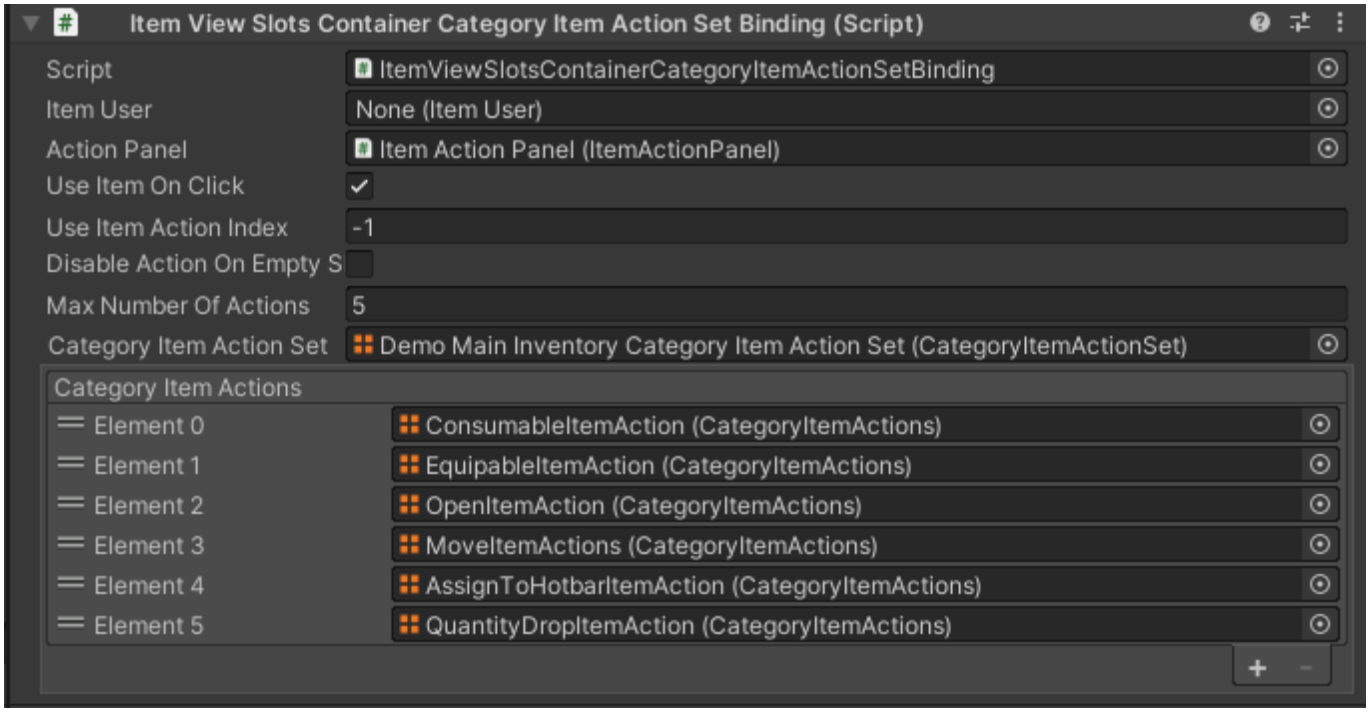
You may easily add Item Actions to an Item View Slot Container to be triggered on the selected item when clicked or through code.

There are two option Use Category Item Actions or Category Item Action Set:

Category Item Action Set

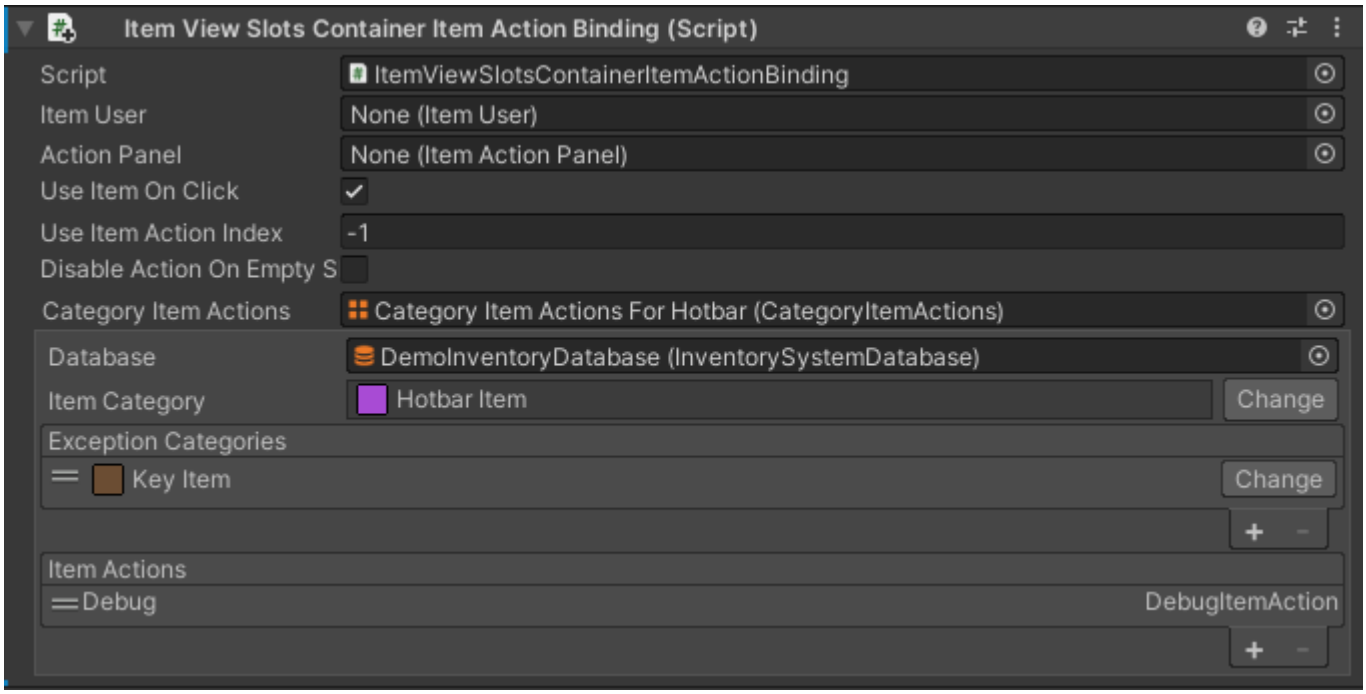
The component contains the following properties:

- *Item User*: The Item User is used to know who will use the item. If an inventory is bound to the Item View Slots Container, the Item User will be found next to the Inventory.
- *Action Panel*: The Action Panel is optional it allows you to open a panel with all the relevant Item Actions your item is allowed to call, if null the item actions will be used directly.
- *Use Item On Click*: Choose whether or not the item action can be used via click
- *Use Item Action Index*: The Category Item Actions are lists of actions. The index here will find the item action within the actions list. If -1 all the possible actions will be triggered
- *Disable Action On Empty Slots*: This will prevent actions from being called of empty item view slots
- *Max Number Of Actions*: The array size for the item actions.
- *Category Item Action Set*



Category Item Actions

The same as the Category Item Action Set except using a single Category Item Actions Scriptable Object instead of a list.



Item Description

The Item View Slot Container Description Binding is an extremely easy way to bind an Item Description to the selected Item View Slot in the Container. Item Descriptions are special Item Views and you can find out more about them on this page.

Moving Items

The Item Moving System for drag & drop and item action moving is split up in multiple components.

Cursor Manager

The Cursor manager is set next to the Canvas manager components. It dictates how the item is shown while moving on the screen by spawning an Item View of choice with the item in front of everything. This component is required for any of the drag/drop/move components to work.

Drop Handler

The Item View Drop Handler defines the actions possible once dropping the item on another Item View Slot. It is used for both drag & drop and Item Action Move Cursor. With the information of the source and destination, Item View Slot Container, and Item View Slot. The Item View Slot Drop Action Set can be used to specify what happens when the item is dropped.

Item View Slot Drop Action Set

The Item View Slot Drop Action Set has a list of conditions/actions that defines what action will be performed. Some examples of conditions are: Item View Slot Container Name, Is Item View Slot Empty, Can the container or slot contain the source item.

Some examples are Moving the item index within the same Item View Slot Container, Exchanging between the Item View Slot Containers, or setting the Item View Slot.

Drag Handler

The Item View Slot Drag handler makes sure the events of dragging an Item View Slot are set correctly to keep track of its origin and more.

Move Cursor

The move cursor allows you to move items without a mouse, it is used by the ‘Move’ Item Action. It is recommended to add the Item Action Binding components to the *Unbind while Moving* field such that clicking to place the item won’t trigger the Item Action.

Import & Export

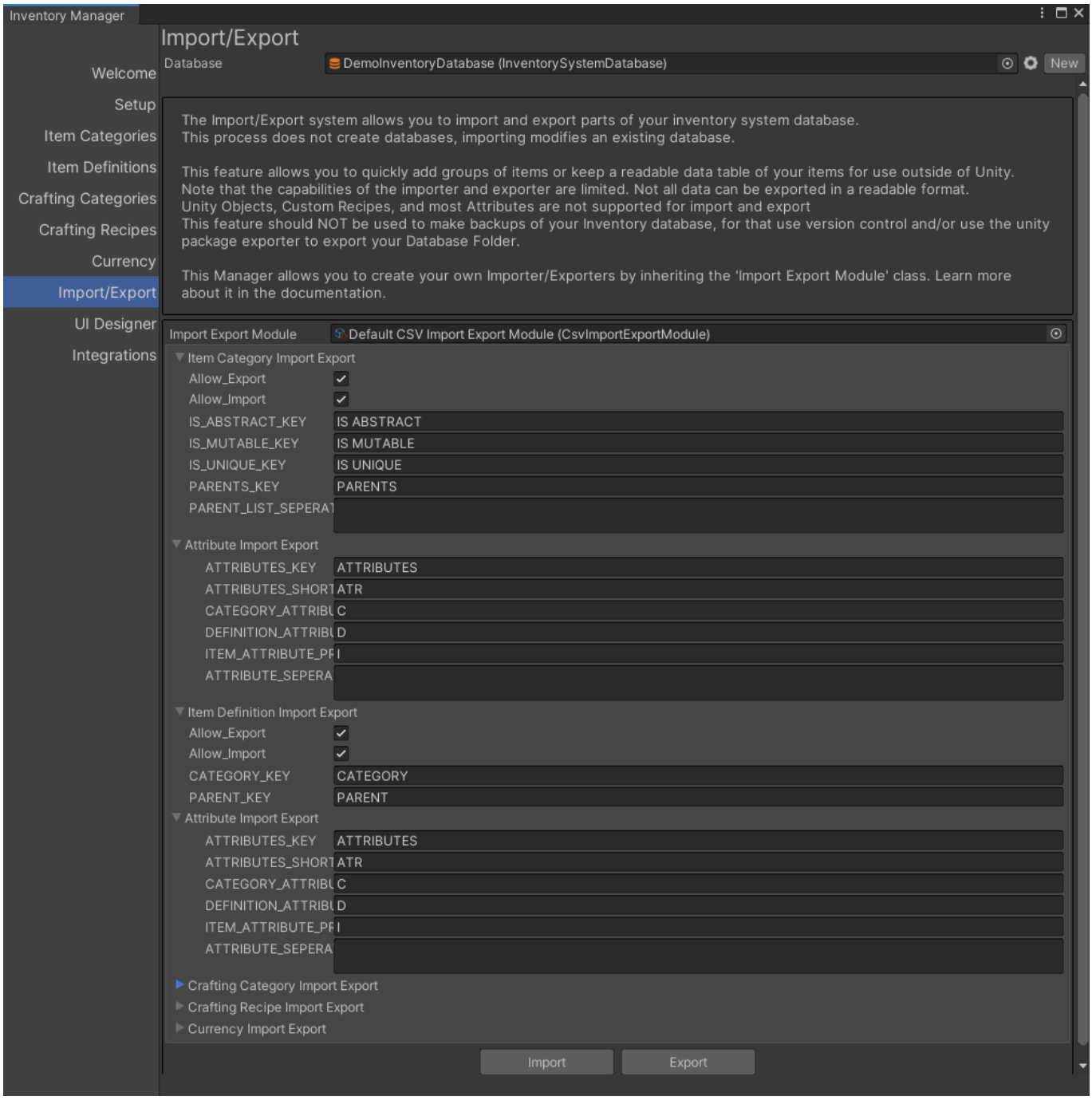
The Import/Export system allows you to import and export parts of your inventory system database.

This process does not create databases but rather it modifies an existing database. This feature allows you to quickly add groups of items or keep a readable data table of your items for use outside of Unity.

> Note that the capabilities of the importer and exporter are limited. Not all data can be exported in a readable format. Unity Objects, Custom Recipes, and most Attributes are not supported for import and export.
>
> This feature should **NOT** be used to make backups of your inventory database.

We recommend using version control or a proper backup solution for database backups.

By default the Import Export system uses a CSV Import Export Module that imports and exports parts of the Inventory System Database



With the CSV Import Export Module you may choose the keys for the columns and other parameters to customize the format of the exported and imported data.

The Import Export Module is a Scriptable Object so any changes made to it will be saved. It is recommend that you make a duplicate if you plan to use this feature often to prevent values from being overwritten when updating the asset.

The exported data will look something similar to:

Item Categories

ITEM CATEGORY	NAME	IS ABSTRACT	IS MUTABLE	IS UNIQUE	PARENTS	ATTRIBUTES	[ATR:C]CategoryIcon<UnityEngine.Sprite>	[ATR:D]Description<System.String>	[ATR:D]Icon<Ur
3423580008	Viewable	TRUE	TRUE	TRUE		[O]CategoryIcon [O]Description [O]Icon [O]Shape [O]ShapeIcon		This Item Description is missing, please add it	
1922247342	Pickupable	TRUE	TRUE	TRUE		[O]PickupPrefab			

Item Definition

ITEM DEFINITION	NAME	CATEGORY	PARENT	ATTRIBUTES	[ATR:D]Icon<UnityEngine.Sprite>	[ATR:D]Description<System.String>	[ATR:D]PickupPrefab<UnityEngine.GameOb
4023030452	Potion	Consumable		[O]Icon [O]Description [I]PickupPrefab [O]HealAmount [O]BuyPrice [O]SellPrice [O]Shape [O]ShapeIcon	Potion (UnityEngine.Sprite)	A potion that heals a small amount of H	ItemPickupBag (UnityEngine.GameObject)
351262454	Wood	Material		[O]Icon [O]Description [I]PickupPrefab [O]BuyPrice [O]SellPrice [I]Shape [I]ShapeIcon	Wood (UnityEngine.Sprite)	Wood found on the floor. Nothing speci	ItemPickupBag (UnityEngine.GameObject)

Crafting Category

CRAFTING CATEGORY	NAME	IS ABSTRACT	PARENTS	RECIPE TYPE
2377458452	Uncategorized	FALSE		Opsive.UltimateInventorySystem.Crafting.CraftingRecipe
2505128647	Recipes With Cu	FALSE	All Recipes	Opsive.UltimateInventorySystem.Crafting.RecipeTypes.CraftingRecipeWithCurrency

Crafting Recipe

CRAFTING RECIPE	NAME	CATEGORY	CATEGORY INC	DEFINITION IN	ITEM INGREDIENTS	OUTPUTS
625086989	Big Potion	Default Recipes		Potion x1 Heal Weed x1 Water x1		Big Potion x1
231541197	Potion	Default Recipes		Heal Weed x1 Water x1		Potion x1

Currency

CURRENCY	NAME	EXCHANGE RA	PARENT	MAX AMOUNT	OVERFLOW CURRENCY	FRACTION CURRENCY
529954384	Bronze	100		99	Silver	
3230874826	Silver	100	Bronze	99	Gold	Bronze
974222359	Gold	100	Silver	99		Silver

This feature is particularity useful for importing large sets of Item Definitions or Recipes. All keys other then the main key (“ITEM CATEGORY”, “ITEMDEFINITION”) and the name key are optional when importing data. Therefore if you wish to change the attribute value of many item in one go this can be a good solution (the attribute value must be of a supported type: int, string, float, etc.).

> Note that the existing objects in the database will never be deleted on import. They will either be added if they do not yet exist or they will be modified if the data in the imported csv is different.

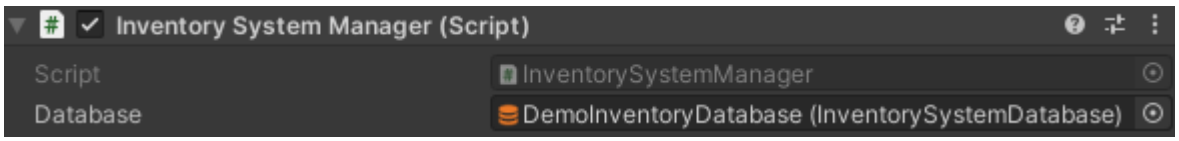
This Manager allows you to create your own Import Export Module by inheriting the ‘Import Export Module’ class.

```
/// <summary>
/// The Import Export Module is a scriptable object used to import and
export data from a database.
/// </summary>
public abstract class ImportExportModule : ScriptableObject
{
    /// <summary>
    /// Export the inventory system database.
    /// </summary>
    /// <param name="database">The database to export.</param>
    public abstract void Export(InventorySystemDatabase database);
    /// <summary>
    /// Import the inventory system database.
    /// </summary>
    /// <param name="database">The database to import.</param>
    public abstract void Import(InventorySystemDatabase database);
}
```

Refer to the CSV Import Export Module source code for examples on how objects can be imported/exported.

Inventory System Manager

The Inventory System Manager is a singleton component which organizes manages all the objects within the inventory. The Inventory System Manager needs to be added to every scene and can be added through the Scene Setup of the Inventory System Manager.



When the scene starts the manager will load the database and organize it. The Inventory System Manager class can be used to get a reference to an object:

```
InventorySystemManager.GetItemCategory("Weapons");
InventorySystemManager.GetItemDefinition("Big Potion");
InventorySystemManager.GetCurrency("Gold");
```

The Inventory System Manager can also create items at runtime:

```
// Create an item using its name
InventorySystemManager.CreateItem("ItemName");

// Create an item with its Item Definition
InventorySystemManager.CreateItem(itemDefinition);

// Specify the ID you wish your item should have (useful when saving
or using a server/client)
InventorySystemManager.CreateItem(itemDefinition, itemID);
```

```
// Create a copy of an Item
InventorySystemManager.CreateItem(item);
```

As the main singleton of the package the Inventory System Manager can be used to register and get a lot more objects:

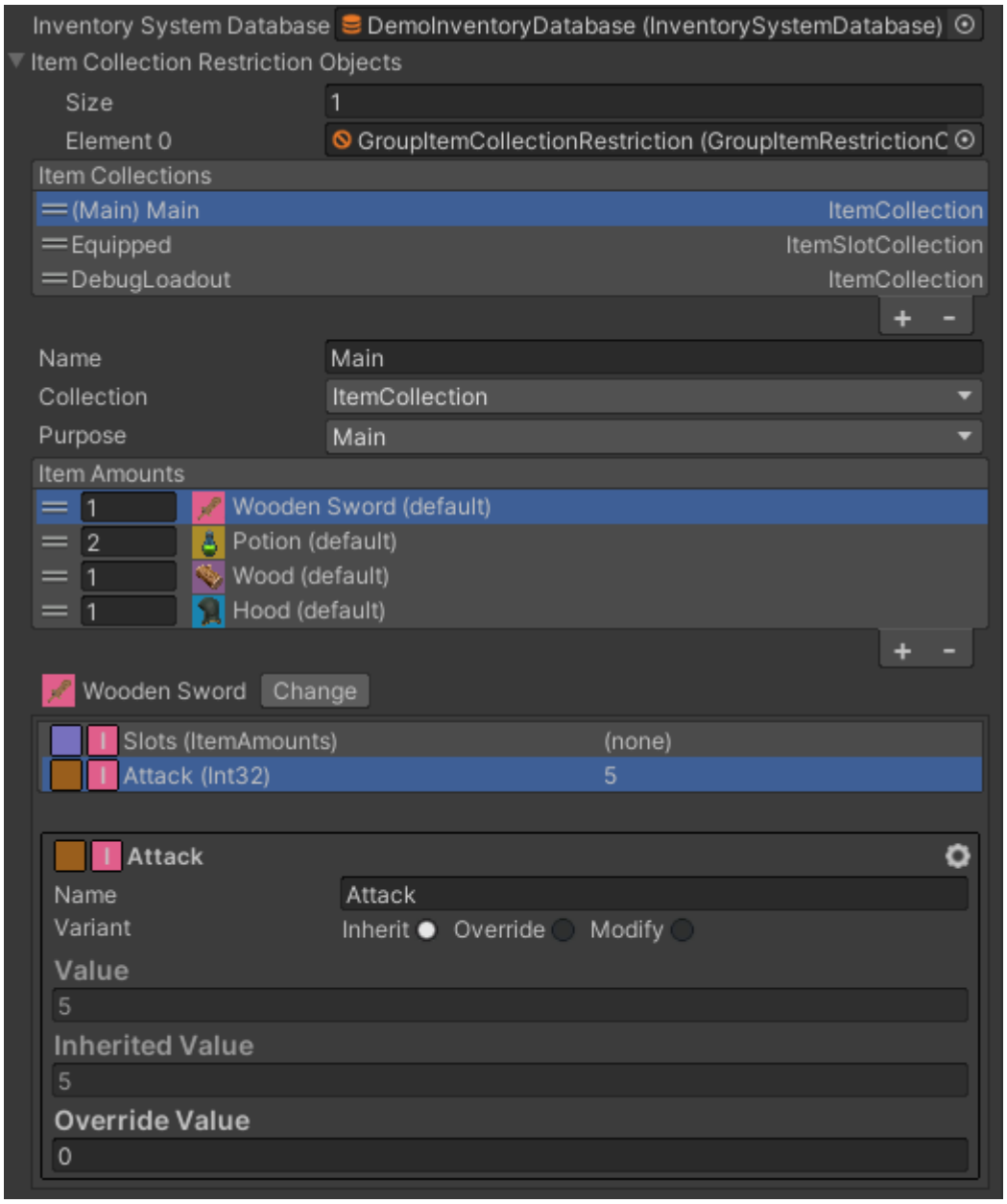
```
// Get a Display Panel Manager
var panelManager = InventorySystemManager.GetDisplayPanelManager(id);

// Get an Inventory Identifier by ID (a component which sits next to
an Inventory)
var inventoryIdentifier =
InventorySystemManager.GetInventoryIdentifier(id);
var inventory = inventoryIdentifier.Inventory;

// Get an Item View Slot Cursor Manager, which is used for drag & drop
var itemViewSlotCursorManager =
InventorySystemManager.GetItemViewSlotCursorManager(id);

// Get Item View Slot Containers using their panel names and the panel
manager id
var itemHotbar =
InventorySystemManager.GetItemViewSlotContainer<ItemHotbar>(id, "Item
Hotbar");
var inventoryGrid =
InventorySystemManager.GetItemViewSlotContainer<InventoryGrid>(id,"Inv
entory Grid");

// Get & Set whatever objects you want using their types and an ID as
key.
InventorySystemManager.SetGlobal<Transform>(myTransform, id);
var transform = InventorySystemManager.GetGlobal<Transform>(id);

var itemObjectSpawner =
InventorySystemManager.GetGlobal<ItemObjectSpawner>(id);
```

Inventory

Inventories are a set of Item Collection objects. This makes more sense with an example. Let's say that a character can have items in their bag, in their pocket, or even equipped with their hand. In this situation there would be three Item Collections: Bag, Pocket, and Equipped. When you create an Inventory you can set as many Item Collections as you want and you can name and can give them a purpose by flagging them. The Item Collection names need to be unique and an Inventory must have at least one Item Collection.

In the inspector you can create new Item Collections and change their type. The Item Collection page contains more details about how Item Collection objects work. When an Item Collection is selected in the Inventory a list containing the Item Amounts within the

collection will be shown. Selecting an Item Amount allows you to edit it. Press the “Change” button to select a new item. You can edit the Item Attributes directly in the Inventory inspector.



When selecting an item in the list field it is useful to know about the search and sort options. The list has a search and sort fields which allows you to quickly find the item you are looking for. There are even special tokens you can write in the search field to filter by category or attribute name:

Example

- *“a:MyAttribute”:* Fetch all items which have the “MyAttribute” attribute.
- *“c:MyCategory”:* Fetch all items that inherit the category “MyCategory”.
- *“str c:Weapon a:Attack”:* Fetch all items that contain the substring “str” in their name, inherit the “Weapon” category and have the “Attack” attribute.

The Inventory will show you the items in your Item Collections in real time. This makes debugging easier as you can quickly check where items are located. You can even change item attribute values (no event is sent when an attribute is changed in the inspector).

The Inventory class is a MonoBehaviors so you can add it on any GameObject. It is used in many different cases:

- Character inventory.
- Enemy inventory.
- Shop inventory.
- Chest inventory.
- Storage Inventory.
- Pickup inventory.
- And many more.

We recommend that you use the Inventory component for any use case that requires a collection of items.

Inventory API

Once you have an Inventory you may add/remove and retrieve the items stored in it. When using inventories we use item infos a lot as they contain more information about the item such as its amount, its item stack, item collection inventory, etc...

For Events on the Inventory component check out the EventNames.cs file. To learn how to use those events please refer to the events page.

Get All Items

```
// Get all the items in the inventory (includes all the item
collections except the ignored ones 'Hide' and 'Loadout')
var allItems = inventory.AllItemInfos;
```

Add and Remove by name

```
// Add an item to an inventory by name.
m_Inventory.AddItem("potion", 5);
// Remove an item to an inventory by name.
m_Inventory.RemoveItem("potion", 3);
```

Check if the inventory has an item

```
// Check if the inventory has an item. by using an item definition
amount
m_Inventory.HasItem( (1, itemDefinition) );
```

Create items from item definitions

```
//Find your item definition.
var myItemDefinition =
InventorySystemManager.GetItemDefinition("MyItemDefinition");
//Create an item from that definition.
var myItem = InventorySystemManager.CreateItem(myItemDefinition);
```

```
//Or you can directly create an item from the item definition name.
var myOtherItem =
InventorySystemManager.CreateItem("MyOtherItemDefinition");
```

Make an ItemInfo from an item

The commands above a very simple, but they are also quite limited. Some times more control is necessary. Especially when using unique items and or Multi Stack Item Collections.

```
//Make your item Info, Note double (( )) the (1,myItem) is an
itemAmount.
//Note: ItemInfos can have a reference to the item stack and the
itemcollection/inventory where the item comes from.
var myItemInfo = new ItemInfo( (1,myItem) );
inventory.AddItem(myItemInfo);

//Alternatively you can directly cast the item amount to an item info
both are structs not classes.
//Note: It returns the item info added, this allows you to know the
amount of item added and in which stack it was added to.
var itemInfoActuallyAdded = inventory.AddItem((ItemInfo) (1, myItem));
```

Add and Remove using ItemInfo

```
//The line above is equivalent to adding an item directly to the main
item collection.
inventory.MainItemCollection.AddItem((ItemInfo) (1, myItem));
```

Get an Item Collection

It is recommended to define item collections by unique names. You can use the name to get the item collection from the inventory.

```
// Get an Item Collection by name.
var itemCollection = m_Inventory.GetItemCollection("Item Collection
Name");
// if your item collection is of a specific type simply cast it
var itemSlotCollection = m_Inventory.GetItemCollection("Equipment") as
ItemSlotCollection;
```

Check if an item can be added

When using Item Restrictions you can restrict items and prevent them from being added to the inventory.

The return nullable Item Info can either be null if the amount cannot be added, or it will have the amount that can be added. In some cases the amount may be different thant the item to add since some of the amount might fit but not all of it.

```
//create the Item Info you plan to add.
var itemInfoToAdd = (ItemInfo)(5, myItem);
var itemCollectionToAddTheItemTo =
inventory.GetItemCollection(ItemCollectionPurpose.Equipped);

// Check if the item can be added, the return type is a ItemInfo? (The
? means it is NULLABLE).
var canAddItemResult =
itemCollectionToAddTheItemTo.CanAddItem(itemInfoToAdd);
if(canAddItemResult.HasValue){
        if(canAddItemResult.Value.Amount == itemInfoToAdd.Amount ){
                // The Item Amount can be fully added.
        }else{
                // The Item can be added but only partially.
        }
}else{
        // The item cannot be added.
}
```

Get an Item Info from the inventory

```
//If you wish to add an item to another item collection you can get it
index, purpose or name.
inventory.GetItemCollection(ItemCollectionPurpose.Equipped).AddItem((I
temInfo) (1, myItem));
```

Get an Item Info from the inventory

```
//You can get the first matching item stack to an item.
var retrievedItemInfo = inventory.GetItemInfo(myItem);
if (retrievedItemInfo.HasValue == false) {
    //No item stack with that description was found in the inventory.
}
```

When removing items get the reference to the Item Stack where the item was removed from.

```
//We can remove items from an inventory and it returns the item info
that was actually removed.
var itemInfoRemoved = inventory.RemoveItem((ItemInfo)(1,myItem));

//Specifying an item info that has a reference to an item stack will
remove from that stack first, if possible.
inventory.RemoveItem(retrievedItemInfo.Value);

//We could have chosen to remove a different amount, than the amount
retrieved.
inventory.RemoveItem( (1,retrievedItemInfo.Value) );
```

Get a list of items within the inventory using a filter

```
//You can get multiple items from your item collections
//You'll need an array to store your item infos.
var array = new ItemInfo[0];
//You can use a pooled array instead of creating a new array each
time. IMPORTANT: make sure to return the pooled array once you
finished using it.
var pooledArray = GenericObjectPool.Get<ItemInfo[]>();

//Set your filter and sort parameters, for this example we'll filter
by category.
var filterParameter =
InventorySystemManager.GetItemCategory("MyCategory");

//Item info list slice will have all the itemInfos in the inventory
that contain inherits the category.
//You can set whatever filter parameter you want.
var itemInfoListSlice = inventory.GetItemInfos(ref pooledArray,
filterParameter,
    //Specify your filter
    (candidateItemInfo, category) => { return
category.InherentlyContains(candidateItemInfo.Item); }
);

//IMPORTANT: Don't forget to return the pooled array once you finished
using the array.
GenericObjectPool.Return(pooledArray);
```

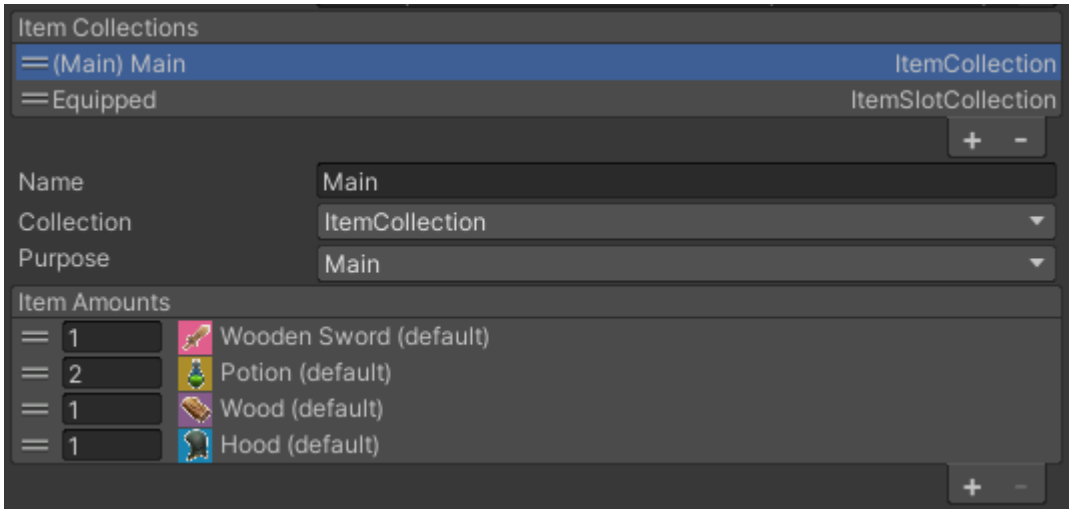
You can filter and sort the items retrieved with the GetItemInfos function:

```
ListSlice<ItemInfo> GetItemInfos<T>(ref ItemInfo[] itemAmounts, T
filterSortParam,
    Func<ItemInfo, T, bool> filterFunc, Func<T, Comparer<ItemInfo>>
sortComparer, int startIndex = 0)
```

This function is generic so you can set the exact filter parameters you want no matter the type.

Item Collections

An Item Collection is a grouping of items. Items are stored in a list of Item Stacks where an Item Stack is a class with an item and an amount (not to be confused with an Item Amount which is a struct). Unique items will always have an amount of 1 by default because. Common items can have any positive amount.



The purpose of the Item Collection can be specified within the inspector. This makes it easy to filter for a particular Item Collection based on the purpose of the Item Collection. If you plan to use many Item Collections it is recommended to set all the Purposes to None (aka Other) and use only names to differentiate Item Collections to avoid confusion.

- *None:* No specified purpose.
- *Main:* Default collection in an inventory.
- *Secondary:* Allows for custom use.
- *Equipped:* The included items have been equipped.
- *Loadout:* The collection is used for a loadout. The items won't show in the UI.
- *Hide:* Hides collections won't show in the UI.
- *Drop:* Used to drop items.

When you add and remove items it is recommended that you use the Item Info struct. An Item Info does not only have an Item Amount but it also has a reference to an Item Collection and/or an Item Stack. This allows you to specify where an item came from before you add it. This is particularly useful combined with restrictions as it allows you to reject an item and give it back where it came from. This can also be used by the Inventory Monitor to ignore items that come from a loadout or from Item Collections contained within the same inventory.

You can also get all the items which are of a certain category or definition. For the most flexibility you can get exactly the Item Info that you want by defining your own filter and comparer when calling the GetItemInfos(…) function.

Example: Returning all of the ItemInfos that have mutable items and ordered by Item Definition name:

```
var filteredAndSortedItemInfos = itemCollection.GetItemInfos(ref
itemInfos,
x => x.Item.IsMutable,
Comparer<ItemStack>.Create((i1, i2) =>
i1.Item.ItemDefinition.Name.CompareTo(i2.Item.ItemDefinition.Name)) );
```

The base Item Collection class has the following features:

- item amounts must be positive, negative amounts is not allowed.

- Unique items can only have an amount of 1. If the same unique item is added twice a new duplicate item will be created with a new ID.
- There is only one Item Stack per Item (Use an Multi Stack Item Collection or a custom Item Collection to have multiple stacks of a single item).
- Any amount of item and item stack can be added (Use Item Restrictions to limit the amount of item that can be added, or use an Item Slot Collection if you wish to have a constant set of item slots).

You can create your own Item Collections and add restriction for further customization. For example there are restriction that allow only X items can be stored or only items of a certain category, etc.

Out of the box we offer a few generic Item Collections which can be setup for your own needs:

- Item Slot Collection: An Item Slot Collection is used to have a constant set of slots for your items. It is particularly useful to define your character equipment, hotbar, pouch, etc…
- Item Transaction Collection: This collection is often used as the Main Item Collection. It serves as a middle ground for sending item to the correct item collection whenever a new item is added to the inventory. It is paritcuarly useful when your Inventory is organized in many item collections
- Multi Stack Item Collection: This Item Collection allows multiple stacks of the same Item. Each Item Stack has a limit defined by a default value or by an attribute on the item. Once that limit is past the amount overflows into another Item Stack within the collection.

The Inventory component allows you to use Item Restrictions which apply restrictions on multiple Item Collections at the same time.

Item Collection API

Create an Item from an Item Definition

```
//Find your item definition.
var myItemDefinition =
InventorySystemManager.GetItemDefinition("MyItemDefinition");
//Create an item from that definition.
var myItem = InventorySystemManager.CreateItem(myItemDefinition);

//Or you can directly create an item from the item definition name.
var myOtherItem =
InventorySystemManager.CreateItem("MyOtherItemDefinition");
```

Add an item using an Item Info

```
//Make your item Info, Note double (( )) the (1,myItem) is an
itemAmount.
//Note: ItemInfos can have a reference to the item stack and the
itemcollection/inventory where the item comes from.
```

```
var myItemInfo = new ItemInfo( (1,myItem) );
itemCollection.AddItem(myItemInfo);

//Alternatively you can directly cast the item amount to an item info
both are structs not classes.
//Note: It returns the item info added, this allows you to know the
amount of item added and in which stack it was added to.
var itemInfoActuallyAdded = itemCollection.AddItem((ItemInfo) (1,
myItem));
```

Check if an item can be added (Restrictions can prevent items from being added)

The returned nullable Item Info has tells us if the item can be added and how much of the amoun can actually be added.

```
//create the Item Info you plan to add.
var itemInfoToAdd = (ItemInfo)(5, myItem);

// Check if the item can be added, the return type is a ItemInfo? (The
? means it is NULLABLE).
var canAddItemResult = itemCollection.CanAddItem(itemInfoToAdd);
if(canAddItemResult.HasValue){
        if(canAddItemResult.Value.Amount == itemInfoToAdd.Amount ){
                // The Item Amount can be fully added.
        }else{
                // The Item can be added but only partially.
        }
}else{
        // The item cannot be added.
}
```

Get an item Info from the collection

```
//You can get the first matching item stack to an item.
var retrievedItemInfo = itemCollection.GetItemInfo(myItem);
if (retrievedItemInfo.HasValue == false) { //No item stack with that
description was found in the inventory. }
```

Get the amount of an item the collection has

```
//If you are interested in counting the amount of multiple stacks for
matching description use GetItemAmount.
var amount = itemCollection.GetItemAmount(myItem);

//The boolean parameter lets us choose whether to check inherently or
not.
var amountDefinition =
itemCollection.GetItemAmount(myItemDefinition,true);
```

Remove an item and get a reference of the item stack where the item was removed from

```
//We can remove items from an inventory and it returns the item info
that was actually removed.
var itemInfoRemoved = itemCollection.RemoveItem((ItemInfo)(1,myItem));

//Specifying an item info that has a reference to an item stack will
remove from that stack first, if
possible.itemCollection.RemoveItem(retrievedItemInfo.Value);
//We could have chosen to remove a different amount, than the amount.
retrieved.itemCollection.RemoveItem( (1,retrievedItemInfo.Value) );
```

Get a list of items within the item collection using sorted and filters.

```
//You can get multiple items from your item collection.
//You'll need an array to store your item infos.
var array = new ItemInfo[0];
//You can use a pooled array instead of creating a new array each
time.
IMPORTANT: make sure to return the pooled array once you finished
using it.
var pooledArray = GenericObjectPool.Get<ItemInfo[]>();

//Set your filter and sort parameters, for this example we'll filter
by category.
var filterParameter =
InventorySystemManager.GetItemCategory("MyCategory");

//Item info list slice will have all the itemInfos in the inventory
that contain inherits the category.
//You can set whatever filter parameter you want.
var itemInfoListSlice = itemCollection.GetItemInfos(ref pooledArray,
filterParameter,
    //Specify your filter.
    (candidateItemInfo, category) => { return
category.InherentlyContains(candidateItemInfo.Item); });

//IMPORTANT: Don't forget to return the pooled array once you finished
using the array.
GenericObjectPool.Return(pooledArray);
```

Item Slot Collection

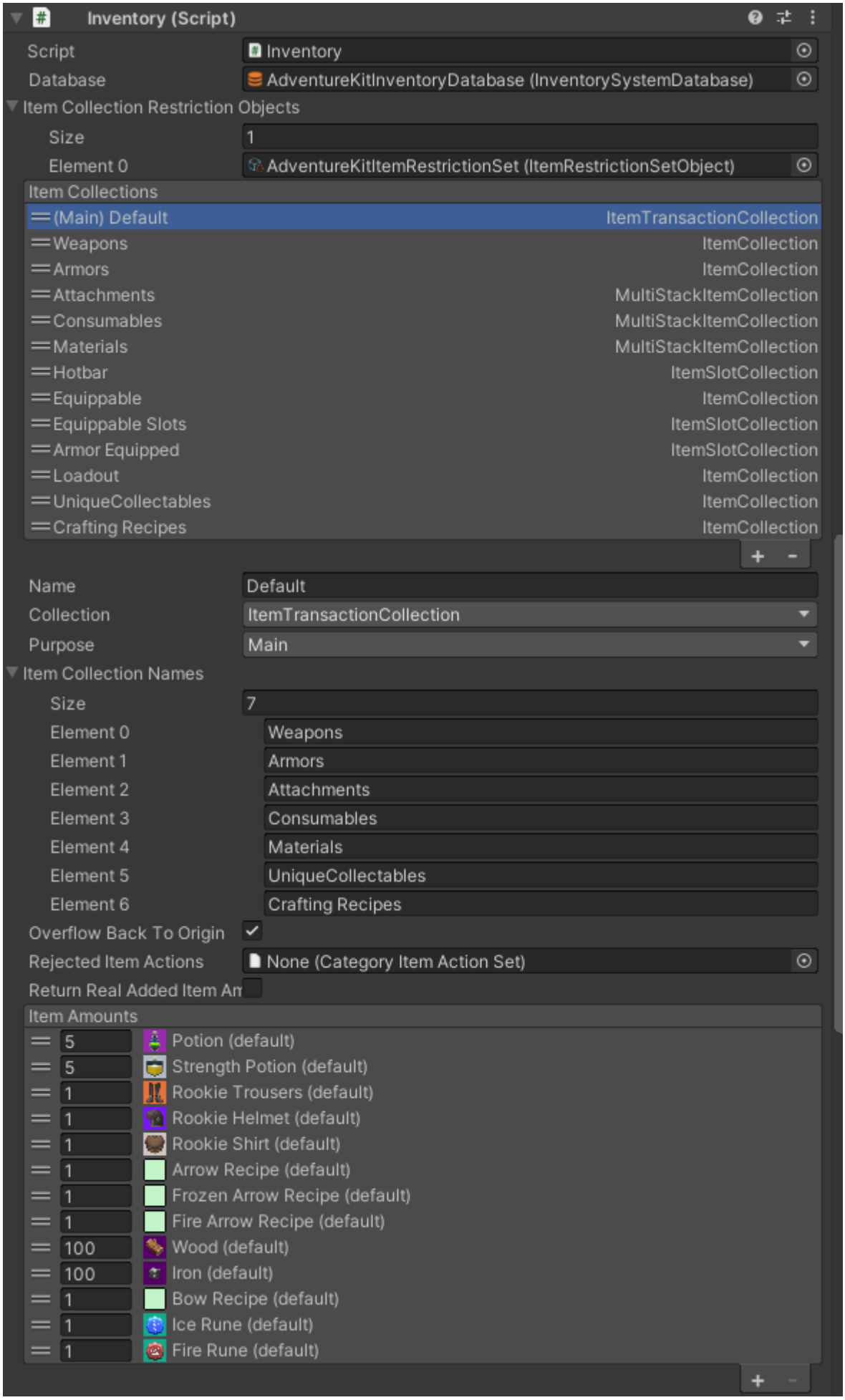
Item Slot Collection is a custom Item Collection which only allows you to add items within specific slots defined by an Item Slot Set scriptable object. This collection type is often used for storing equipped items.

Item Slot Collection API

```
// Get the Item Slot Collection within the inventory using Get Item
Collection and cast it to its type.
var equipmentItemCollection =
m_Inventory.GetItemCollection(m_EquipmentItemCollectionID) as
ItemSlotCollection;

//Use the slot Index to add, get and remove.
var itemAdded = equipmentItemCollection.AddItem(itemInfo, slotIndex);
var itemInSlot = equipmentItemCollection.GetItemInfoAtSlot(slotIndex);
var itemRemoved = equipmentItemCollection.RemoveItem(slotIndex,
amountToRemove);

//Or use the slot name.
itemAdded = equipmentItemCollection.AddItem(itemInfo, slotName);
itemInSlot = equipmentItemCollection.GetItemInfoAtSlot(slotName);
itemRemoved = equipmentItemCollection.RemoveItem(slotName,
amountToRemove);

var item = equipmentItemCollection.GetAllItemStacks()[0];

//Might not be equal to.
var otherItem = equipmentItemCollection.GetItemInfoAtSlot(0);
Item myItem = InventorySystemManager.CreateItem("myItem");

//Gets the slot where the item actually is.
var realSlotIndex = equipmentItemCollection.GetItemSlotIndex(myItem);

//Get the index where the item could potentially fit if it was added.
var potentialSlotIndex =
equipmentItemCollection.GetTargetSlotIndex(myItem);
```

Item Transaction Collection

The Item Transaction Collection is a special Item Collection which does not hold more than one Item at a time. It is usually used as the Main Item Collection to move items to the Item Collection where they belong.



Item Collection Names : The names of the Item Collections that the Transaction Collection should check when an item is added.

Overflow Back To Origin : When an item overflow (aka is rejected) should it be returned where it came from? This only works if the origin of the Item is available from the Item Info.
Rejected Item Actions : The Item Actions on this Category Item Action will be invoked on the item that was rejected.
Return Real Added Item Amount : Return the item amount that was actually added in the right item collection or return also the amount that was rejected (in case it was added externally)?

The Item Transaction Collection works by checking the linked ItemCollection in order and checking if the item added fits in any of them. To know if an item fits in an Item Collection Item Restrictions are used. If the item does fit in an item collection it is added where it belongs. If not an event is triggered saying that the item was rejected.

API

```
//Listen to the On Rejected item event.
EventHandler.RegisterEvent<ItemInfo>(m_Inventory,
EventNames.c_Inventory_OnRejected_ItemInfo, HandleItemInfoRejected);
```

Multi Stack Item Collection

The Multi Stack Item Collection allows multiple Item Stack per item.

- *Default Stack Size Limit*: The stack limit for a stack of a Common item.
- *Stack Size Limit Attribute Name*: The attribute name pointing to an integer, used for limiting the stack size for a specific Common items

Item Restrictions

In most games Inventories have limitations, such as space or stack size, etc… By default ItemCollections have no limitations in the Inventory System. You are allowed to set your own restrictions as you see fit.

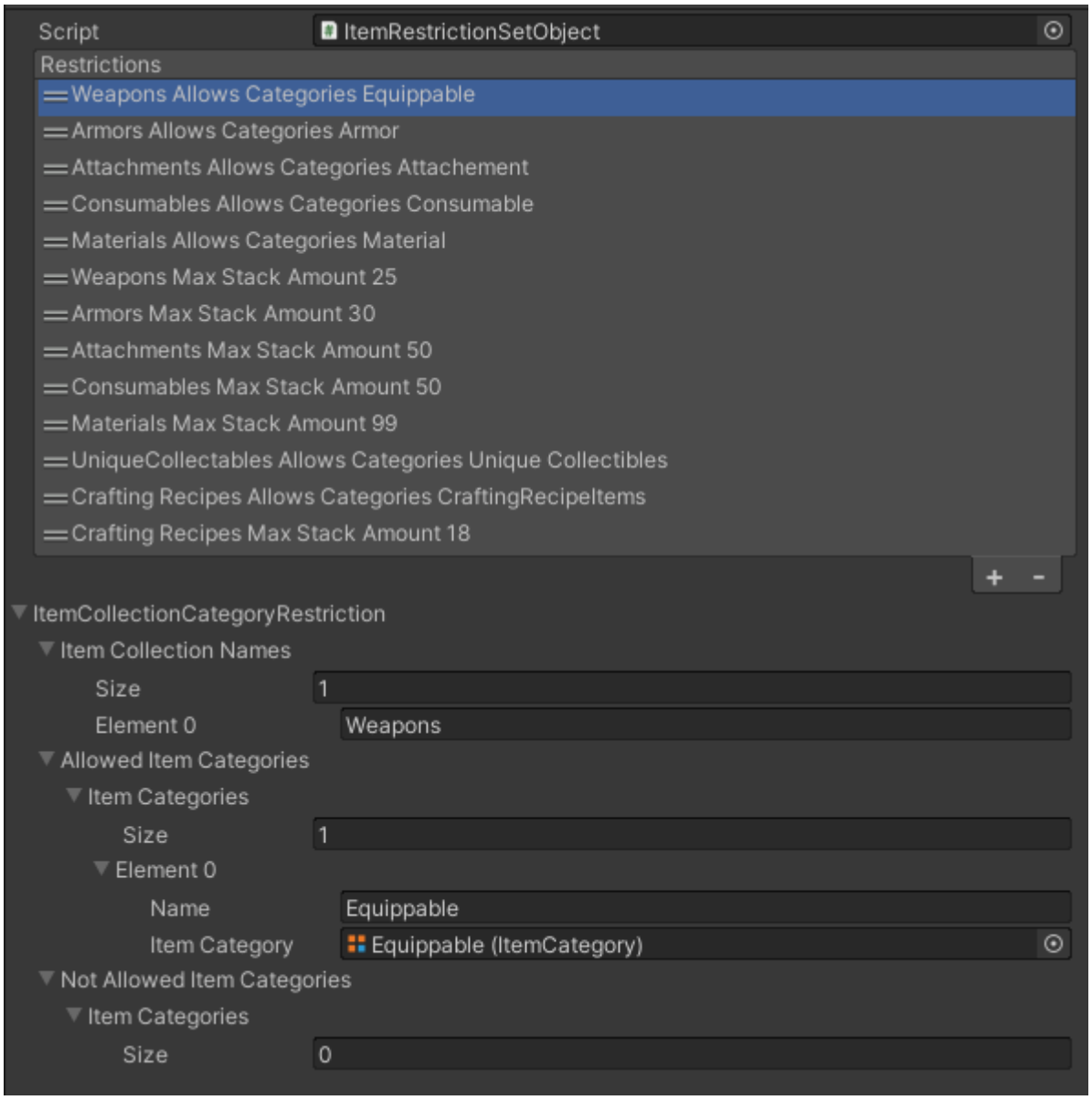
There are three main ways of setting restrictions on your inventory:

1. Using an “Item Restriction Set Object”.
2. Using a custom component which inherits the IItemRestriction interface.
3. Using custom Item Collections

All of these can be combined together for complex inventory restriction setups. In some cases where it is important to know if an Item has been rejected, it is recommended to use an “Item Transaction Collection” as your main item collection. A rejection event handler can be set from the Item Transaction Collection to deal with those cases.

Item Restriction Set Object

The Item Restriction Set Object is a Scriptable Object that can be created by right-clicking in the project view and select Create -> Ultimate Inventory System -> Inventory -> Item Restriction Set.

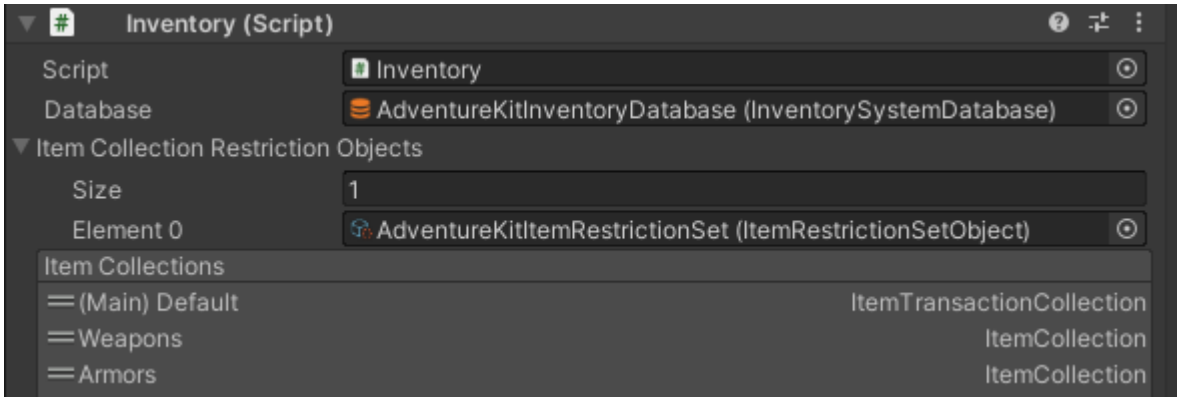


Many restrictions of different types can be added to the Item Restriction Set. Any script inheriting the “IItem Restriction” interface will appear as an option that can be added to the Item Restriction Set. The options that are the most common are:

- **Item Collection Category Restriction**: Restricts some item collections to only allow certain category of items to be added to them
- **Item Collection Stack Amount Restriction** : Allow only a certain amount of item stacks in an inventory, putting a limit on how many items can be hold.

To create your own custom Restriction and use it in the Item Restriction Set, follow the instruction below and take inspiration from the built-in restrictions.

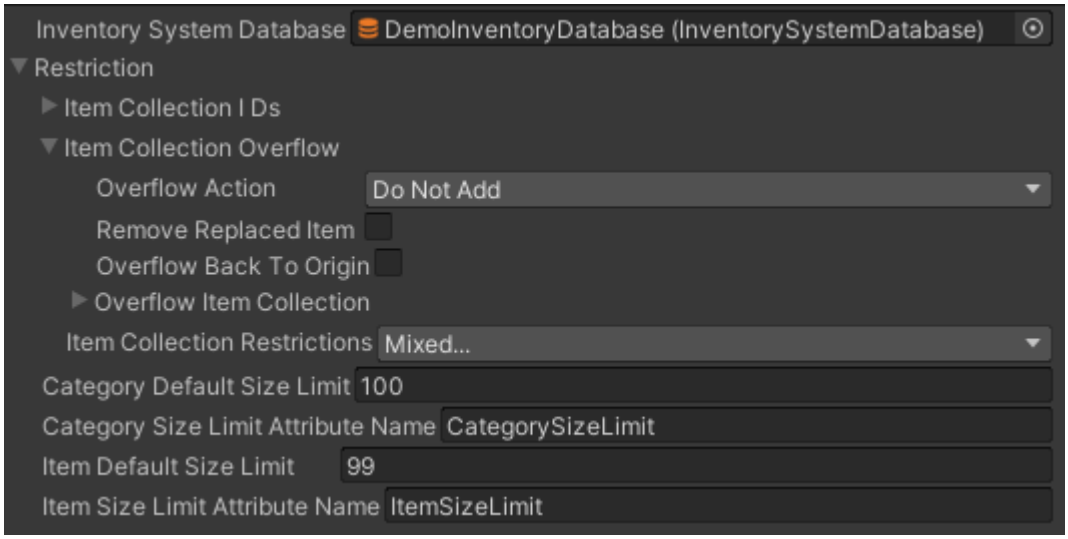
Once created the Restrition Object may be assigned to an Inventory in the inspector.



Group Item Restriction Object

> **Note**: The Group Item Restriction Object can still be used even though it is recommended to use the new Item Restriction Object instead which gives more flexibility.

The Group Item Restriction Object is a Scriptable Object that can be created by right-clicking in the project view and select Create -> Ultimate Inventory System -> Inventory -> Group Item Restriction.



There are a few settings which can be configured:

- *Item Collection IDs:* The subset of Item Collections that will be affected by the restriction.
- *Item Collection Overflow:* Specifies what happens if an item overflows the restriction. For example you may choose to reject the item or replace an old one instead. You can even choose to move items that where forcibly removed to another Item Collection or even apply item actions to them. A common use case for this is to add a drop action when an item is forcibly removed.
- *Item Collection Restriction:* A mask of restriction types. Depending on the options selected some additional settings will appear. You can choose to restrict the amount of item which are part of a particular Item Definition or Item Category by setting a

default stack size. For more control you can set the stack size on the Item Categories, Item Definitions or Items directly using the attribute system. If the attributes name match the one specified it will overwrite the default stack size for that restriction.

Custom Item Restrictions

To create your own restrictions is very simple, simply create a component or class which inherits the interface “IItemRestriction”. And set it next to the Inventory component or use it inside an Item Restriction Object.
During the Initialization phase the Inventory will find all the restriction components on its gameobject and combine them with the ones set in the inspector.

When the Inventory “Can Add Item” function is called it will check all restrictions to see if it passes all the conditions before the item can be added or removed.

```
/// <summary>
/// Interface for item collection restrictions.
/// </summary>
public interface IItemRestriction
{
    /// <summary>
    /// Initialize the item collection.
    /// </summary>
    /// <param name="inventory">The inventory.</param>
    /// <param name="force">Force to initialize.</param>
    void Initialize(IInventory inventory, bool force);
    /// <summary>
    /// Can the item be added to the item
    /// </summary>
    /// <param name="itemInfo">The item info to add.</param>
    /// <param name="receivingCollection">The receiving item
Collection.</param>
    /// <returns>The itemInfo that can be added (can be
null).</returns>
    ItemInfo? AddCondition(ItemInfo itemInfo, ItemCollection
receivingCollection);
    /// <summary>
    /// Can the item be removed from the itemCollection.
    /// </summary>
    /// <param name="itemInfo">The item info to remove (contains the
itemCollection).</param>
    /// <returns>The item info that can be removed (can be
null).</returns>
    ItemInfo? RemoveCondition(ItemInfo itemInfo);
}
```

Events

When an item is added but it does not fit because of restrictions an event is executed. That event is called

“c_Inventory_OnAddItemRejected_ItemInfoToAdd_ItemInfoAdded_ItemInfoRejected”. To learn more about events go the the Events page.

- The Item Info To Add is the original Item Info that you tried to add to the Item Collection. It might contain the Item Collection where the item originally came from.
- The Item Info Added is the Item Info that was added to the Item Collection if the item amount was partially added. That happens when only some of the item but not all amount could fit in the collection.
- The Item Info Rejected has the amount of items that could not be added.

Use this combination of information to deal with item rejection.

```
private Start(){
        EventHandler.RegisterEvent<ItemInfo, ItemInfo,
ItemInfo>(m_Inventory,
EventNames.c_Inventory_OnAddItemRejected_ItemInfoToAdd_ItemInfoAdded_I
temInfoRejected, HandleItemRejection);
}

private HandleItemRejection(ItemInfo itemInfoToAdd, ItemInfo
itemInfoAdded, ItemInfo itemInfoRejected){
        // Handle the rejection by dropping the item, or printing a
message.
}
```

Item

The Item class is the core of the inventory system. Items are simple objects (not a MonoBehavior or Unity Object) which can be unique or shared. The Terminology section explains the differences between the different item objects.

Items are created at runtime and contain attribute values that can be unique to that Item. Even though two Items have the same mold (Item Definition) they can be set to be slightly different. If the Item is part of a mutable Item Category it is even possible to change the values of the Item attributes at runtime.

Items do not have any “item specific logic” attached to it. It is meant to be a simple set of data. Using that data, which consists of the Item Category, Item Definition and Item attributes, the system knows what kind of actions can be performed on them.

Items can be mutable or immutable. Mutable items can be changed at runtime, whereas immutable items cannot be.

There is also a difference between Unique and Common items. Common Items will stack whenever possible while Unique items will always have an amount of one when added to an Item Collection.

Item API

Create Item by Name

```
// Create an Item from the Inventory System Manager by name.
var potion = InventorySystemManager.CreateItem("Potion");
```

Create a Unique Mutable Item with a predefined ID

```
// If for some reason the item must have a predefined ID set it when
you create the item.
var armor = InventorySystemManager.CreateItem("Armor", 777);
// For items to have different IDs they must be either Unique, Mutable
or both. This is defined on the Item Category.
var armorIsUnique = armor.IsUnique;
var armorIsMutable = armor.IsMutable;
```

Create Item from an Item Definition

```
// Create an Item from the Inventory System Manager by Item
Definition.
var swordDefinition =
InventorySystemManager.GetItemDefinition("Sword");
var sword = InventorySystemManager.CreateItem(swordDefinition);
```

Get an Attribute on an Item to get or set its value

```
// Change an attribute value on the sword item
var attackAttribute = sword.GetAttribute<Attribute<int>>("Attack");
// Get the attribute value
var attack = attackAttribute.GetValue();
// Setting the override value at runtime is only allowed on Mutable
Items.
attackAttribute.SetOverrideValue(attack + 5);
```

Make an Item copy to keep the same runtime values

```
// Make a copy of an existing item, copies the attribute values.
var swordCopy = InventorySystemManager.CreateItem(sword);
```

Check if an Item is part of an Item Category or Item Definition

```
// Check if the item is part of a category.
var weapon = InventorySystemManager.GetItemCategory("Weapon");
var swordIsWeapon = weapon.InherentlyContains(sword);
// Similar function are available for Item Definition or even between
categories.
var weaponIsWeapon = weapon.InherentlyContains(weapon);
var swordDefinitionIsWeapon =
weapon.InherentlyContains(swordDefinition);
```

Add an Item to an Inventory

```
// Add an item to an inventory by name.
m_Inventory.AddItem("potion", 5);
// Add an item to an inventory by item definition.
m_Inventory.AddItem(swordDefinition, 5);
// Add an item to an inventory or for more control by item Info and
item stack.
var itemInfoToAdd = new ItemInfo(sword, 1);
ItemStack itemStackDestination = null;
m_Inventory.AddItem(itemInfoToAdd, itemStackDestination);
```

Add an Item to an Item Collection

```
// Get an item collection by name.
var itemCollection = m_Inventory.GetItemCollection("Item Collection
Name");
// Add an item to the item collection
itemCollection.AddItem("potion", 3);
// Add an item to an item collection by item definition.
itemCollection.AddItem(swordDefinition, 5);
// Add an item to an item collection or for more control by item Info
and item stack.
itemCollection.AddItem(itemInfoToAdd, itemStackDestination);
```

Get an Item from an Item Collection

```
// Get an item from the item collection, it returns a nullable Item
Info.
var itemInfoResult = itemCollection.GetItemInfo(sword);
// If the item does not exist in the item collection the result will
be null.
if (itemInfoResult.HasValue == false) {
    // The item does not exist.
} else {
    // The item was found in the Item Collection.
    var swordItemInfo = itemInfoResult.Value;
    // The Item Info has the amount and reference to the item stack.
}
```

Check if an Item can be added to an Item Collection

```
// Check if the item can be added, the return type is a ItemInfo? (The
? means it is NULLABLE).
var canAddItemResult = itemCollection.CanAddItem(itemInfoToAdd);
if(canAddItemResult.HasValue){
    if(canAddItemResult.Value.Amount == itemInfoToAdd.Amount ){
        // The Item Amount can be fully added.
    }else{
        // The Item can be added but only partially.
```

```
    }
}else{
    // The item cannot be added.
}
```

Remove an Item from an Item Collection

```
// A simple remove.
itemCollection.RemoveItem("potion", 3);
// But most time you'll want more control.
// Get the Item Info to remove
var retrievedItemInfoResult = itemCollection.GetItemInfo(sword);
if (retrievedItemInfoResult.HasValue == false) {
    // The item does not exist in the inventory.
    return;
}
// Use the retrieved item info to make a new item info with the amount
of item you wish to remove.
var itemInfoToRemove = (ItemInfo)(1, retrievedItemInfoResult.Value);
// We can remove items from an inventory and it returns the item info
that was actually removed.
var itemInfoRemoved = itemCollection.RemoveItem(itemInfoToRemove);
// Check the amount of the item Info removed to make sure the item was
removed.
if (itemInfoRemoved.Amount == itemInfoToRemove.Amount) {
    // The Item was removed successfully.
}
```

Get all the Item, Item Definition and Item Categories in the system

```
// Get all ItemCategories, Item Definitions and Items in the game.
var allItemCategories =
InventorySystemManager.ItemCategoryRegister.GetAll();
var allItemDefinitions =
InventorySystemManager.ItemDefinitionRegister.GetAll();
var allItems = InventorySystemManager.ItemRegister.GetAll();
```

Get all the Item Definition that are part of Item Category

```
//Get all ItemDefinitions of an Item Category
var allMyCategoryItemDefinitions = new ItemDefinition[0];
myCategory.GetAllChildrenElements(ref allMyCategoryItemDefinitions,
true);

//Get all items loaded in the system that are part of the ItemCategory
(this includes every single instances of items)
foreach (var item in allItems ) {
    if (myCategory.InherentlyContains(item)) {
        //item is part of category
```

```
    }
}
```

Item Upgrades

There are many ways to upgrade an item:

Parent/Child Item Definitions

This is the recommended approach if you have a simple upgrading system where upgradable items have a finite and linear upgrade path. By finite and linear path means items which upgrade with the same result each time. Example : Sword -> Sword +1 -> Sword +2 -> Sword +3. With this approach you can have one Item Definition for each sword: Sword, Sword +1, Sword +2, Sword +3.

The Item Definitions can then be linked with a parent/child relationship. This allows you to inherit attribute values of the parent which can then be overridden or modified for each definition. This is a great way to say “Sword +2 must always have 20% more attack than Sword +1”

A new Item Definition attribute called *Upgrade* of type Item Definition should then be created. This will allow you to link the Item Definition upgrades in order. If needed a *UpgradeRequirements* attribute of a custom type could be created. This attribute would decide when you are allowed to upgrade your item.

A custom script would then be necessary in order to get the upgrade requirements and check if the upgrade is valid. If it is valid the item can then be replaced by the new one using the *Upgrade* Item Definition attribute.

This approach is fairly simple and offers a good solution for those who want an item upgrade system that is easily manageable. Note that it is a trade off with scalability, as this approach does not scale well once you reach hundreds of upgradable items with many upgrades each.

Item Definitions and Item Attributes

This approach is recommended if you have an simple item upgrade dependencies and few upgradable items.

This approach uses Item attributes to define the Item Definition array/list that the item can be upgraded to. As an example imagine that you have a Sword with 3 attack and a Sword+1 which has 5 attack. The Item Definition “Sword” could be created with the string[] attribute *UpgradeName* and int[] attribute *UpgradeAttack*. The item should then have two int attributes called *Attack* and *Level*.

Within a new script you can check the item level to know if the item can be upgraded. If the item can be upgraded the item’s *Level* attribute should be increased. The the Item Definition’s *UpgradeName* and *UpgradeAttack* can then be fetched to replace the item’s name and attack values.

Upgrade Items and Upgradable Items with Slots

This is the approach that is used in the demo. That does not mean it is the best approach for all cases. This approach consists of two parts. Upgradable items and Upgrade items. Upgradeable items are items which can hold other items in an array called *Slots.* These slots can be populated with items categorized as Upgrade items. This allows you to create one upgrade item that can be used to upgrade any upgradable item.

A base value needs to be added to the Item Definition and this value is recomputed each time the item is upgraded. A new attribute called *Slots* with type ItemAmounts should then be added to the Item Definition on the upgradeable item. This stores the items that the item can be upgraded to. A new script should then be created which knows how each upgrade will affect an upgrable item of the Item Category. When the item is upgraded the correct upgradable item can be selected based on the base value on the ItemDefinition and the upgrade items that are contained within the *Slots* attribute.

In the demo scene the “Boost upgrade” upgrades “Weapons” and “Armors” items differently. It will increase teh *Attack* attribute on the Weapon items and increase the *Defense* attribute on the Armors items.

This approach is very flexible and scalable though it requires a more complex script compared to the two previous solutions.

Crafting

This approach is different compared to the others. It uses Crafting Recipes as a base for upgrades. With the crafting system you create a list of generic ingredients and have a custom Crafting Processor which follows different logic depending on the ingredients provided. You could have a recipe called “Weapon Upgrade” which is defined using Item Categories as 1 Weapon + 5 Materials. Your Crafting Processor would take items that fit that description (for example, 1 Sword + 2 Iron + 3 Bronze). It could then compute the swords upgraded attack value using the rarity and compatibility of the materials as parameter. The same Crafting Recipe and Crafting Processor can be used to upgrade any weapons

This approach is most appropriate for very complex systems and as it requires more code than the other solutions. It is recommended for intermediate level programmers.

Which workflow is the best?

There is no workflow that is better than the other. They all have their advantages and disadvantages. The workflow that you should choose should depend on the desired structure of your inventory. You can also mix and match the approaches to take advantage of their strengths.

Item Skills

Due to the incredible flexibility of the Attribute system it is often asked how could an Item be used as a Skill.

For our purposes a skill is an Item which has a functionality which can change between items within the same category. This is different from the classic Category Item Action Set system used in the UI where any Item within a category will run the same Item Action.

There are no right or wrong answers on how to implement Item Skills. It all dependents on what you wish to achieve and what are the existing systems in your game. Here are some the most common options but note they are not the only ones possible.

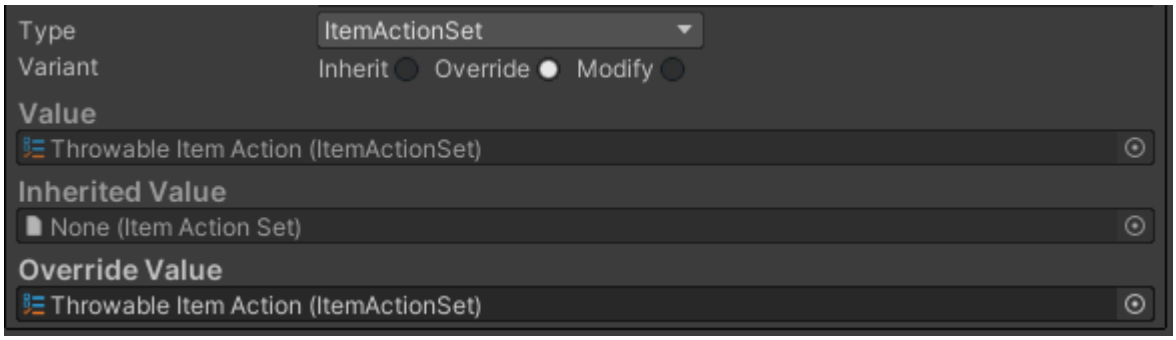
Option 1: Use a Scriptable Object as an Attribute

The first option is to use a Scriptable Object Attribute on the Item. This way anywhere in code you may get the attribute of your item and call the function on it, passing the Item and other information as parameter.

The type of Scriptable Object can be custom or you may simply use an “Item Action Set”.

Example

Lets set an Item Action Set attribute. It Could be set on any of the Attribute Collections, although most likely you will wish to add it as an Item Definition Attribute.

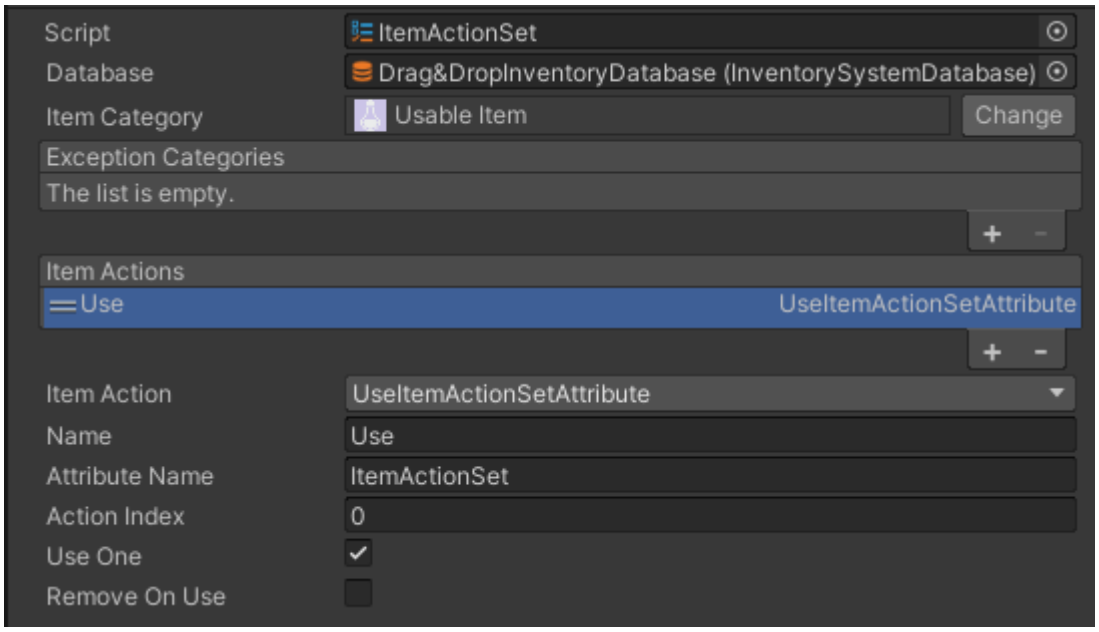


Through code you may then get that Scriptable Object and invoke the Item Action on it:

```
/// <summary>
/// Check if the action can be invoked.
/// </summary>
/// <param name="itemInfo">The item.</param>
/// <param name="itemUser">The item user (can be null).</param>
/// <returns>True if the action can be invoked.</returns>
protected bool TryUseItem(ItemInfo itemInfo, ItemUser itemUser){
         //Cannot use the item if null.
    if (itemInfo.Item == null) { return false; }

         //Get the Attribute with your scriptable object
    var attribute =
itemInfo.Item.GetAttribute<Attribute<ItemActionSet>>(m_AttributeName);

         //Make sure the attribute exists
    if (attribute == null) { return false;}

         // Get the scriptable object (in this case the Item Action
```

```
Set)
    var actionSet = attribute.GetValue();
    if (actionSet == null) { return false; }

    //Since the Item Action has a list of items we need to choose
which one we will use by specifying the index (or we could have
executed all the actions).
    if (m_ActionIndex < 0 || actionSet.ItemActionCollection.Count >=
m_ActionIndex) { return false; }

    // Use the item action on a single item, or the entire amount of
item?
         if (m_UseOne) { itemInfo = (1, itemInfo); }

    //Call the function on the scriptable Object.
actionSet.ItemActionCollection[m_ActionIndex].InvokeAction(itemInfo,it
emUser);

         // You may wish to remove the item once used.
    if (m_RemoveOnUse) {
        itemInfo.ItemCollection?.RemoveItem(itemInfo);
         }

         return true;
}
```

In the case of Item Action Set as an attribute you may use the “Use Item Action Set Attribute” Item Action to use the items as skills within the UI with breeze. It essentially does the same as the code example above.



And from there you may now add a different Scriptable Object (with the same base type as the one set as attribute) to each item.

Tip: Make good use of the Item User parameter to get any Component on its game object. It

is very useful to get the transform of the Character to spawn things for example.

Option 2: Add Components next to the Item Object

This option is only relevant if the item you plan to use as skills are spawned as Item Objects within the game world, in the scene.

The idea is to change functionality of the Item Object by editing some of the components on its game object depending on the Item bound to the Item Object.

There are multiple ways to do this. One option is to use Item Object Behaviours Handler with many Item Object Behaviours and change the index of the Behaviours in the array of Behaviours.

Option 3: Combine Option 1 & 2

The most versatile option is this one. Use a Scriptable Object on the Item to get different functionality for each Item.
If that functionality requires additional information/functionality if may get it components in the scene through the Item User. For example the Item Action may simply call a Restore Health function on the character.

Tip: Use the itemUser.gameobject.GetCachedComponent<MyComponentType>() function to be more efficient when getting the same component multiple times.

Item Stats

In many games Items can be equipped and/or used to affect the character stats and/or the weapon the character is carrying. Using the attribute system you may add any data on an Item, how this data is used depends completely on the type of game one is making.

Here are a few solutions/ideas to inspire you, they are not the only options available so do not feel constrained by them.

Character stats (Equipper)

The most obvious way to affect the character stats is when an item is equipped. The Equipper component is a good start for visually equipping items, it also has a very convenient event that is executed when an Item is equipped or unequipped. Listen to the "EventNames.c_Equipper_OnChange" event to know when the character stats must be updated. A very similar solution to the one below is actually used in the demo scene.

```
/// <summary>
/// Example of character stats.
/// </summary>
public class ExampleCharacterStats
{
    protected IEquipper m_Equipper;
    protected int m_BaseMaxHp;
```

```
    protected int m_BaseAttack;
    protected int m_BaseDefense;
    protected int m_MaxHp;
    protected int m_Attack;
    protected int m_Defense;
    public int BaseMaxHp => m_BaseMaxHp;
    public int BaseAttack => m_BaseAttack;
    public int BaseDefense => m_BaseDefense;
    public int MaxHp => m_MaxHp;
    public int Attack => m_Attack;
    public int Defense => m_Defense;
    /// <summary>
    /// Constructor.
    /// </summary>
    /// <param name="baseStats">The base stats.</param>
    /// <param name="equipper">The equipper.</param>
    public ExampleCharacterStats(int maxHp, int attack, int defense,
IEquipper equipper)
    {
        m_BaseMaxHp = maxHp;
        m_BaseAttack = attack;
        m_BaseDefense = defense;
        m_Equipper = equipper;
        if (m_Equipper != null) {
            EventHandler.RegisterEvent(m_Equipper,
EventNames.c_Equipper_OnChange, UpdateStats);
        }
        UpdateStats();
    }
    /// <summary>
    /// Update the character stats.
    /// </summary>
    public void UpdateStats()
    {
        m_MaxHp = BaseMaxHp + m_Equipper.GetEquipmentStat("MaxHp");
        m_Attack = BaseAttack + m_Equipper.GetEquipmentStat("Attack");
        m_Defense = BaseDefense +
m_Equipper.GetEquipmentStat("Defense");
    }
}
```

The Equipper comes with a useful GetEquipmentStat(“AttributeName”) function that finds the attribute on all the equipped items and adds them together in one value.
It is important to recompute the stat each time as adding and removing float values over and over game may start deviating from the correct value due to floating precision errors. This is why it is advised to keep the base stats separate from the weapon stats.

Of course instead of using the GetEqupimentStat(“AttributeName”) function you may get the attributes manually to compute the stats however you wish, whether is a percentage,

special effect (poison, fire, etc..), or anything really. Learn more about attributes here.

You may learn more about the Equipped component and how to equip weapons here.

Note that you are free to create your own Equipper script from scratch, you are not restricted to the solution we provide.

Weapon stats (Item Binding)

Some times instead of changing the character stats you may be interesting in changing the Weapon stats. Depending on your game the object dealing damage can either be the character or the weapon.

The easiest option to affect weapon stats is to use an Item Binding object. When spawning/equipping a weapon game object it is highly recommended that the weapon has an Item Object component which can be used to bind the game object to an Item. When the binding happens a "EventNames.c_ItemObject_OnItemChanged" is executed. You may listen to this event to deal with a new item being bound to the equipped item. You may use this event to change model, vfx, sound, ect… and of course it can also be used to change stats.

For convenience we added the Item Binding component which lets you bind attribute values from the bound item to any component on the Weapon. And this can be done entirely in the inspector without a single line of code! To learn more about the Item Binding object go here

Consumable stats (Item Action)

There are times where a non-equippable item can be used to affect stats. A good example is food which can be used to heal the character or give with bonuses for X seconds. For this it is recommend to use a combination of custom Item Actions and Monobehaviors. The Item Action can be used to call a function on the custom Monobehavior. For example an "Heal Item Action" which calls the function "Heal" on a custom "CharacterHealth" component.

To learn more about Item Actions go here and to learn more about how to use items as skills go here

Other/Custom

There are many other ways items can be used to affect the player character stats and/or the weapons the character equips. For example equipping a Bow and swapping the Arrow to give different effects (normal, fire, poison, etc..).

These more advanced use case require a bit more custom functionality but the Item and Attribute system makes it easier to scale than implementing such a system from scratch.

Item Actions

Item Actions are used to perform actions on an item from the Inventory. It should not be confused with item object behaviors which are meant to use items within the game world, having a reference to a GameObject (such as swinging a sword).

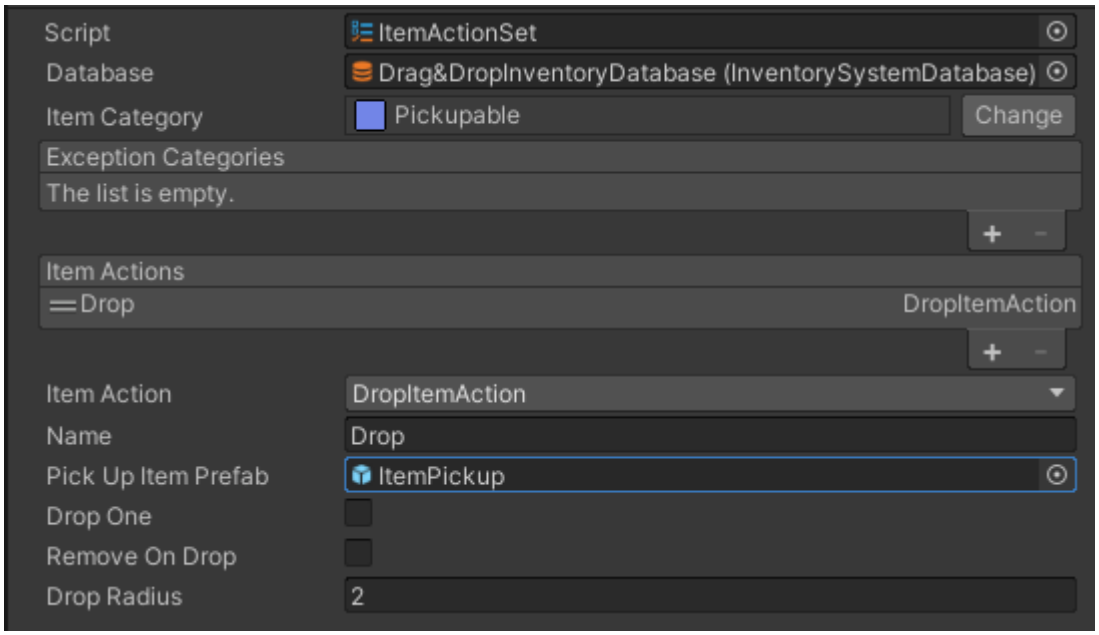
The Item Drop action is a good example of an Item Action. It is used to drop an item from the inventory.

Item Actions have two main methods. CanInvoke(ItemInfo, ItemUser) and Invoke(ItemInfo, ItemUser). Using the Item Info you can create your own logic that lets you choose when an action can or cannot be performed. If the action can be performed then you can access the item attributes to invoke an action specific to the item used. The Item User allows you to pass in a reference to an Item User object, which is usually located on the Player Character. This is useful if you wish get references to components on a character.

Item Actions can be used anywhere at any time but they are mostly used in Item View Slots Containers like the Inventory Grid or Item Hotbar. With the Item Action Set and Category Item Action Set you can map Item Categories to different Item Actions.

Item Action Set

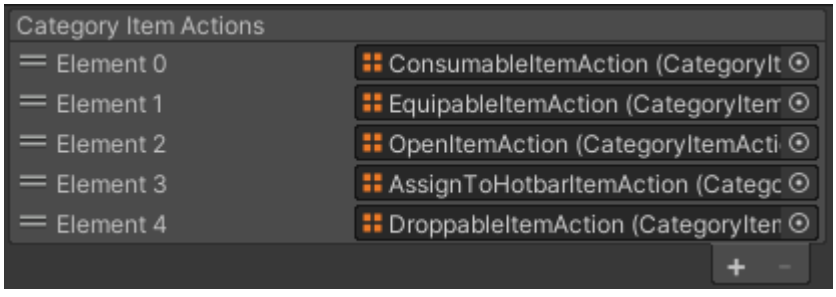
Item Action Set is a Scriptable Object that can be created from Create -> Ultimate Inventory System -> Item Actions -> Item Action Set. Using the same path you'll be able to create the Category Item Action Set.



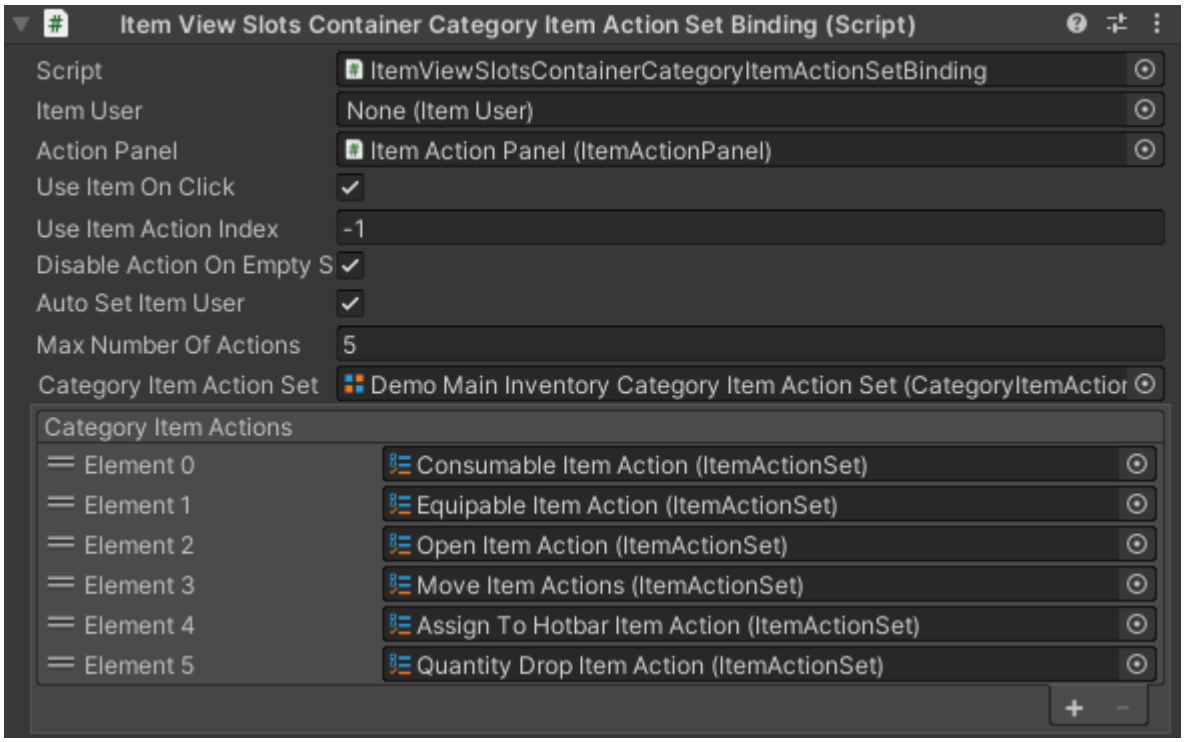
Item Action Sets let you set a list of Item Actions that are relevant to the Item Category specified. For example a "Pickupable" Item Category could have a "Drop" action.

You may also add categories to the exclude categories. These categories will be used by the Category Item Action Set to find all Item Actions which matches with an Item.

Category Item Action Set



The Category Item Action Set is a list of Item Action Sets and it can be used by the “Item View Slots Container Category Item Action Set Binding” component to match item actions for a selected Item. This allows the UI to know which actions can be used by an item and will behave accordingly. For example an item “Sword” could be part of an “Equippable”, “Pickupable” and “Hotbar Item” Item Category. Which mean the Item Actions available for the “Sword” will be “Equip”, “Drop” and “Assign” each from a different Item Action Set object. The Inventory Grid inspector the list of item action will allow you to show a list of possible action when clicking on an item.



Item Action API

Creating your own Item Action is very simple, you simply need to create a new class that inherits Item Action and override the CanInvokeInternal and InvokeActionInternal function.

```
[System.Serializable]
public class MyItemAction : ItemAction
{
    /// <summary>
    /// Can the item action be invoked.
    /// </summary>
    /// <param name="itemInfo">The item info.</param>
    /// <param name="itemUser">The item user (can be null).</param>
    /// <returns>True if it can be invoked.</returns>
    protected override bool CanInvokeInternal(ItemInfo itemInfo,
ItemUser itemUser)
    {
        // Return true if the item can be invoked or false if it
cannot.
        return true;
    }
```

```
    /// <summary>
    /// Consume the item.
    /// </summary>
    /// <param name="itemInfo">The item info.</param>
    /// <param name="itemUser">The item user (can be null).</param>
    protected override void InvokeActionInternal(ItemInfo itemInfo,
ItemUser itemUser)
    {
        // Invoke the item action
    }
}
```

There are a few special Interfaces and abstract Item Actions which can be particularly useful when creating a custom Item Action. Here is a list:

- *IActionWithPanel*: Allows you to set the Panel which called the Item Action. Example : Move Item Action
- *ItemActionWithQuantityPickerPanel*: Spawns a Quantity Picker Panel from prefab and uses async/await to get a quantity before calling the “real” Item Action. Example: Quantity Drop Item Action.
- *ItemViewSlotsContainerItemAction*: An Item Action with a reference to an Item View Slots Container. Example : Open Other Item View Slots Container Item Action.
- *ItemActionWithAsyncFuncActionPanel<T>*: Spawns an Async Func Action Panel which can be used to show a list of options. Example: Assign Hotbar Item Action
- *ItemObjectAction* : Lets you reference an Item Object which may be useful when the Item User has no way to identify the relevant Item Object from the Item Info.

Once your custom Item Action is created it will appear in the Item Action type drop down of the Category Item Actions object.

Built-in Item Actions

There are a few built-in Item Actions which can be used out of the box. Some of them require a specific setup to work as expected.

Simple Item Actions

Simple Item Actions are actions which trigger as soon as the action is clicked in the inventory menu.

Debug Item Action

This is default Item Action. It prints a log statement in the console with the values of all the attributes on an item. It can be useful when debugging your items.

Debug Item Object Action

Similar to Debug Item Action but it prints logs for both the Item Object and the Item Info, very useful for debugging.

Drop Item Action

The drop Item Action has a reference to an Item Pickup Prefab. That prefab requires Item Pickup component.
The item is dropped around the item user game object, or if it does not exist, it will be dropped around the inventory game object.

The Opsive Character Controller integration includes a Item Drop Item Action should be used instead of the default Drop Item Action.

Duplicate Item Action

Duplicates the Item and adds it to the Inventor of the Item Info.

Move To Collection Item Action

This Item Action is used to move an item from one item collection to another within the same inventory. This can be used to equip and unequip items by moving them from the main item collection to the equipped item collection.

The Opsive Character Controller integration includes a Equip Unequip Item Action which should be used instead of the default Move To Collection Item Action

Remove Item Action

Remove an Item from the inventory (does not drop, simply deletes the item from the Inventory of the Item Info).

Use Item Action Set Attribute

Use Item Action Set Attribute is an item action that calls an item action on an Item Action Set that was set on an Item as an Attribute. It also allows you to set a cooldown automatically to reuse that item, which can be used with a Cooldown Item View to show the cooldown on the item UI.

Multi Item Action

Allows you to set multiple item action into one, combining simple item action to create complex ones without writing code.

Item Action With Panels

These are Item Actions which use the abstract Item Actions with panels above.

Quantity Drop Item Action

This Item Action function similarly to the Drop Item Action except it shows a Quantity Picker Panel for the user to choose the amount of items to drop.

The Opsive Character Controller integration includes a Quantity Item Drop Item Action which should be used instead of this one.

Assign Hotbar Item Action

This item action allows the player to assign the selected item to a slot on the item hot bar. The Async Func Action Panel Prefab must have a Async Func Action Panel Int component or any component that inherits the Async Func Action Panel <int> class.

To invoke the action, the Item User must either inherit the Item Hotbar Owner interface or have a neighboring component which inherits the Item Hotbar Owner interface.

In the demo the PlayerCharacter component is both an Item User and an Item Hotbar Owner.

The UltimateInventorySystemBridge component in the Opsive Character Controller integration is the Item User and the Item Hotbar Owner component should be added as a neighboring component.

Item Action With Confirmation Pop Up

This item action allows you to nest another Item Action and ask the player to confirm his choice before executing the nested Item Action.

Move Item Action

Uses the Item View Slot move Cursor component on the Item View Slots Container to move an Item form one Item View Slot to another.

Open Other Item View Slot Container Item Action

This item Action can be used to open another Panel with an Item View Slot Container and listens to the Item Selected. This Item Action can be used to search and select an Item from an inventory for example.

Abstract Item Actions

The are abstract classes which are inherited by complex item actions. These are usually Item Actions which wait for the input from the player.

Item Action With Quantity Picker Panel

This abstract Item Action allows to invoke an item action after waiting for the player to choose an amount in a quantity picker panel.
This Item Action requires a Quantity Picker Panel Prefab, which is pooled and spawned under the Action Panel. The method InvokeWithAwaitedValue can be overridden to perform the action:

```
/// <summary>
```

```
/// Invoke with the action with the awaited value.
/// </summary>
/// <param name="itemInfo">The itemInfo.</param>
/// <param name="itemUser">The item user.</param>
/// <param name="awaitedValue">The value that was waited for.</param>
protected abstract void InvokeWithAwaitedValue(ItemInfo itemInfo,
ItemUser itemUser, int awaitedValue);
```

Item Action With Async Func Action Panel

This abstract generic Item Action allows to invoke an item action after waiting for the player to choose value of type 'T' in a list of options.
This Item Action requires a Async Func Action Panel Prefab, which is pooled and spawned under the Action Panel. he method InvokeWithAwaitedValue can be overridden to perform the action:

```
/// <summary>
/// Invoke with the action with the awaited value.
/// </summary>
/// <param name="itemInfo">The itemInfo.</param>
/// <param name="itemUser">The item user.</param>
/// <param name="awaitedValue">The value that was waited for.</param>
protected abstract void InvokeWithAwaitedValue(ItemInfo itemInfo,
ItemUser itemUser, T awaitedValue);
```

Item Object Action

Lets you set an Item Object using "SetItemObject" function.

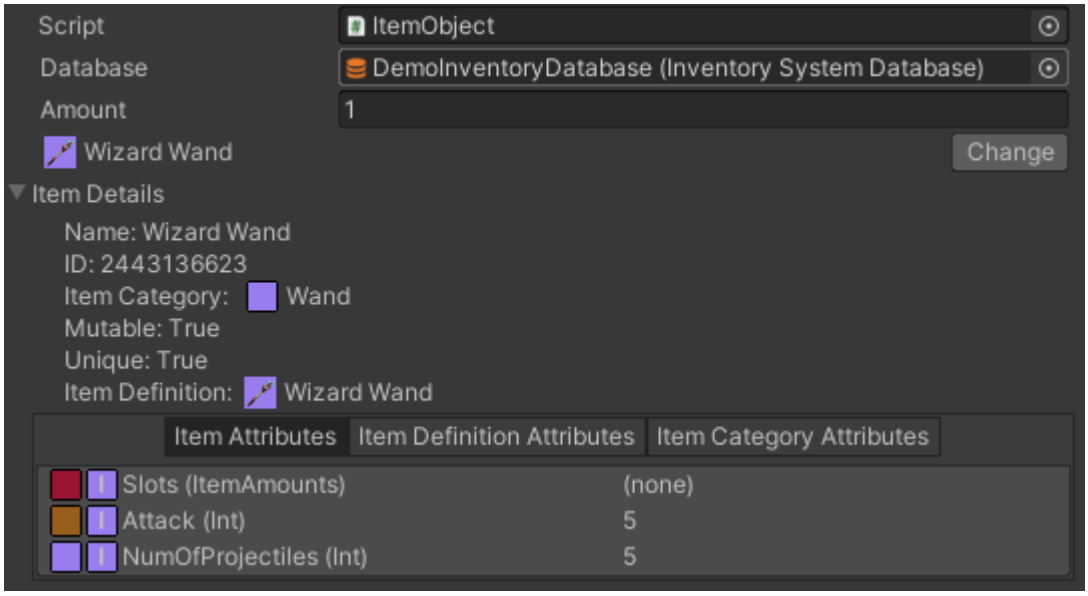
Item View Slots Container Item Action

lets you reference the Display Panel and the Item View Slots Container where the Item Action was called from. This is particularly useful when the item index in the Item View Slots Container is important for the Item Action.

Item Objects

The Item Object component links a particular Item to the GameObject. Any item that is visible within the game world should have an Item Object component. For example, item pickups and equipped items are both ways that items can take shape in the game world.

The ItemObject component is quite flexible as it allows you to bind/unbind an item to it. Therefore you may easily pool some game objects with an ItemObject and reuse them to spawn any item in your scene.



An event is triggered when an item is attached or detached from an Item Object. Some components listen to that event to change how the ItemObject looks in the game world.

Item Object API

Register to the event

```
// Register to the event of the Item changing on the Item Object
EventHandler.RegisterEvent(m_ItemObject,
EventNames.c_ItemObject_OnItemChanged, HandleItemChanged);

private void HandleItemChanged(){
        //The item changed
        var itemInfo = m_ItemObject.ItemInfo;
}
```

Attach an Item to an Item Object

```
// Set the item to the Item Object
var newItem = InventorySystemManager.CreateItem("MyItemName");

// By default it will add 1 unit of the item.
m_ItemObject.SetItem(newItem);

// You may change the amount at any time.
m_ItemObject.SetAmount(5);

// Alternatively you may set an Item Info.
var itemInfo = (ItemInfo)(5, newItem);
m_ItemObject.SetItem(itemInfo);
```

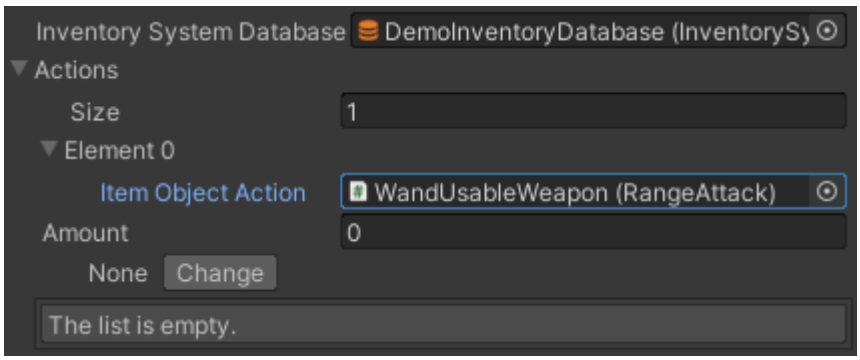
Get the Item Object from the Item

```
//If your Item is Unique you may get the Item Object from the Item
var itemObject = newItem.GetLastItemObject();

// You may easily check if an item is bound to an Item Object.
if(newItem.IsBoundToItemObject(itemObject)){
        //The item is bound to that item object.
}

// If it is not unique the item may have multiple Item Objects:
// Loop over all the item object
var count = newItem.GetItemObjectCount();
for(int i =0; i < count; i++){
        var itemObjectAtIndex = newItem.GetItemObjectAt(i);
}
```

Item Object Behaviour Handler

Item Object Behaviours Handler is a component which sits next to an Item Object and is used by the Equipper to use equipped items. It allows you to easily trigger Item Object Behaviours through different inputs. A Item Object Behaviour Handler can have multiple Item Object Behaviours linked to it. For example if you have a “Gun” category you can specify an action to fire, reload, aim, etc.

Item Object Behaviours are components that you can add on your Item GameObject. You can add as many as you wish. In the demo scene the character attacks with the “Fire1” button (left button click by default) when a sword or a staff is equipped. Looking at the Usable Weapon prefabs you’ll see that the action does not have an input assigned to it.



The Item Object Behaviours can be triggered through code, this is usually done by the Equipper component which listens to the Inventory Input component.

The Item Object Behaviour components are game specific so you will likely need to implement your own. The base class Item Object Behaviour has two methods: CanUse and Use. By default CanUse will be true if Time.time >= m_NextUseTime. This means you can setup a cooldown by setting m_NextUseTime = Time.time + coolDown when the action is used. If you wish to have more control on when an item can be used, you can override the CanUse method.

```
/// <summary>
/// An abstract class used to create actions for item objects.
/// </summary>
public abstract class ItemObjectBehaviour : MonoBehaviour
{
    protected float m_NextUseTime;
    /// <summary>
    /// Can the item object be used.
    /// </summary>
    public virtual bool CanUse => Time.time >= m_NextUseTime;
    /// <summary>
    /// Use the item object.
    /// </summary>
    /// <param name="itemObject">The item object.</param>
    /// <param name="itemUser">The item user.</param>
    public abstract void Use(ItemObject itemObject, ItemUser
itemUser);
}
```

If you choose to override the Use method you will be given two parameters, the Item Object being used and the Item User (usually on the character). Using these you can get the item attributes (or the Item User's stats) to customize what happens when the item is used.

In the demo we use the character stats to modify the damage when attacking. We also modify the number of projectiles coming out of the character's staff and change the projectile prefab.

Tip: Item Object Behaviours go really well with Item Bindings, Item Bindings are used to bind item attributes to properties on any component.

Equipping Items

Equipping armor and weapons is a must when talking about inventory and items. There are two types of equipment, skinned and not skinned. Skinned equipment is any object that gets deformed as the player animates, such as the player's clothes. Skinned equipment must be imported in Unity with the same rig as the Skinned Mesh Renderer it will skin to. Non-skinned equipment are the other type, such as a sword.

It is particularly hard to create an equipment system that works for all games. The system is customizable and for even more freedom it is possible to replace it with your own. The equipment system is a module which no other system depends on.

Note that the Equipment system does not take care of stat changes. Stat changes are not part of the Inventory System, they are only part of the demo. The demo shows a way of how stats can be implemented for the character. The Character Stats class takes in an Equipper

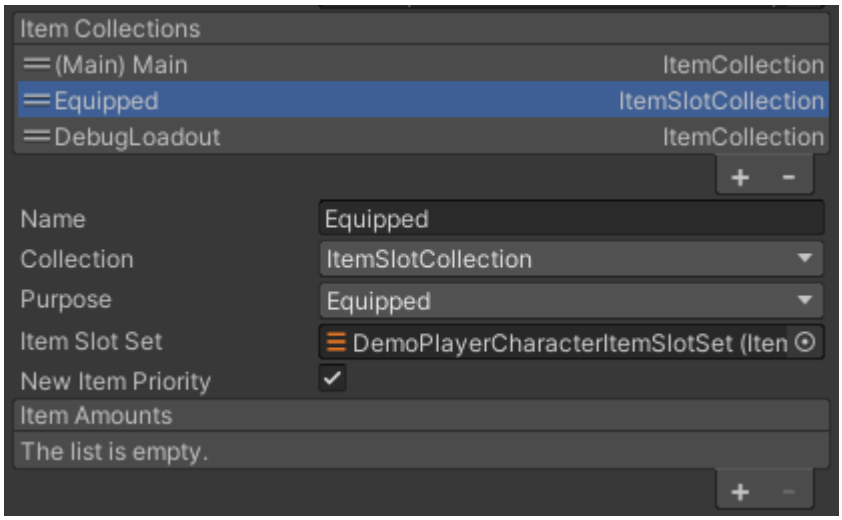
in its constructor which is then used to compute the character stats at any given time. You are free to follow a similar approach.

The equipping system consists of three components: the Item Slot Collection, the Item Slot Set, and the Equipper.

Item Slot Collection

The inventory can have many Item Collections. These collections can be of any type, which means Item Collections can act differently within an Inventory. The Item Slot Collection is a subclass of the Item Collection class which allows us to specify the Item Slot Set. The Item Slot Set restricts what item and how many items can be added to it. In most cases the Item Slot Collection contains the items that are equipped within the inventory so we give it the name “Equipped” and set its purpose to “Equipped”. It is important to set a unique name and purpose within the Item Collections contained in inventory. This is because the Equipper component will access the Item Collection by the name or purpose.

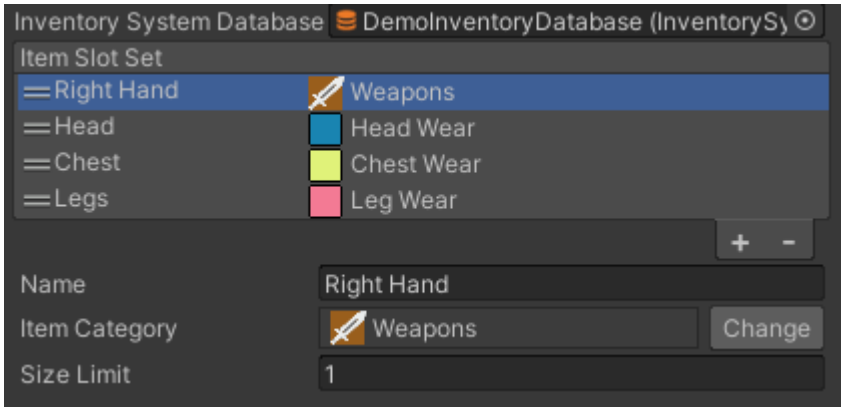
When an item is added to a Item Slot Collection it is first checked to ensure the item can be added to a slot. If it can then that slot is checked to determine if it is empty or occupied. If it is empty then the item can be added without a problem. If the slot is occupied then the existing item is replaced by the newly added item. The removed item will then be placed in the Main Item Collection of the Inventory.



Item Slot Set

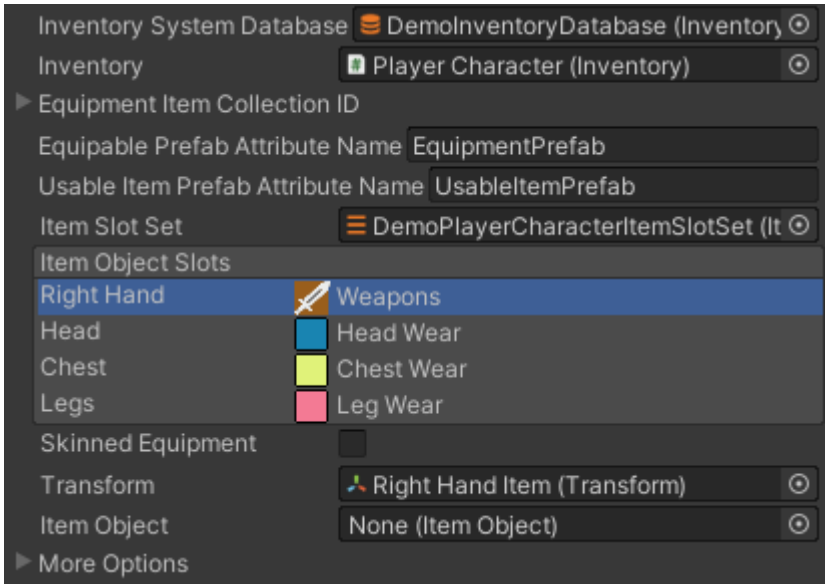
The Item Slot Set is a Scriptable Object used to define a set of item slots. A lot of RPGs have item slots for the head, right hand, left hand, chest and leg. But not every project is the same and you may want more or less slots. You can set the slots from the inspector. Item Slots are defined as a Name, an Item Category and a size limit. You can have as many or as little as you need.

As an example consider the Right Hand slot which is defined with a Weapon Item Category and a size limit of 1. In this scenario you are allowed to equip one weapon in your right hand.



Equipper

The Equipper is used to spawn and visually equip the items within the “Equipped” Item Collection. When specifying the Item Slot set, the Equipper will allow you to specify if an Item Object slot is skinned or not, and if not where the equipped item should be spawned. You can also set the GameObjects that should be hidden when something is equipped in that slot



Two prefab attributes can be spawned by the equipper when equipping an item:

- Equipment Prefab: The Equipment prefab is the prefab that is usually set as an Item Definition Attribute. It will be spawned when the item is equipped. It should have a visual model, either a child Skinned Mesh Renderer or a simple Mesh renderer. Normally this prefab does not have any scripts attached to it, although it can.
- Usable Item Prefab: (Optional) This prefab is usually set as an Item Category Attribute. That’s where you would set the scripts for using your item. For example a script to use a melee attack, or shoot projectiles. This prefab should not have any models just scripts. When used the Equipment prefab will be spawned as a child of this prefab.

The reason we chose to split these in two attributes is to give the option to seperate logic from the model. Allowing you to have a single prefab to define how to do a melee attack or shoot projectile, shared between many items and simply have the model being swapped. Of course an Item Binding component can be used to switch the property values on the Usable Item to change values such as “Attack”, “Number of Projectiles”, etc…

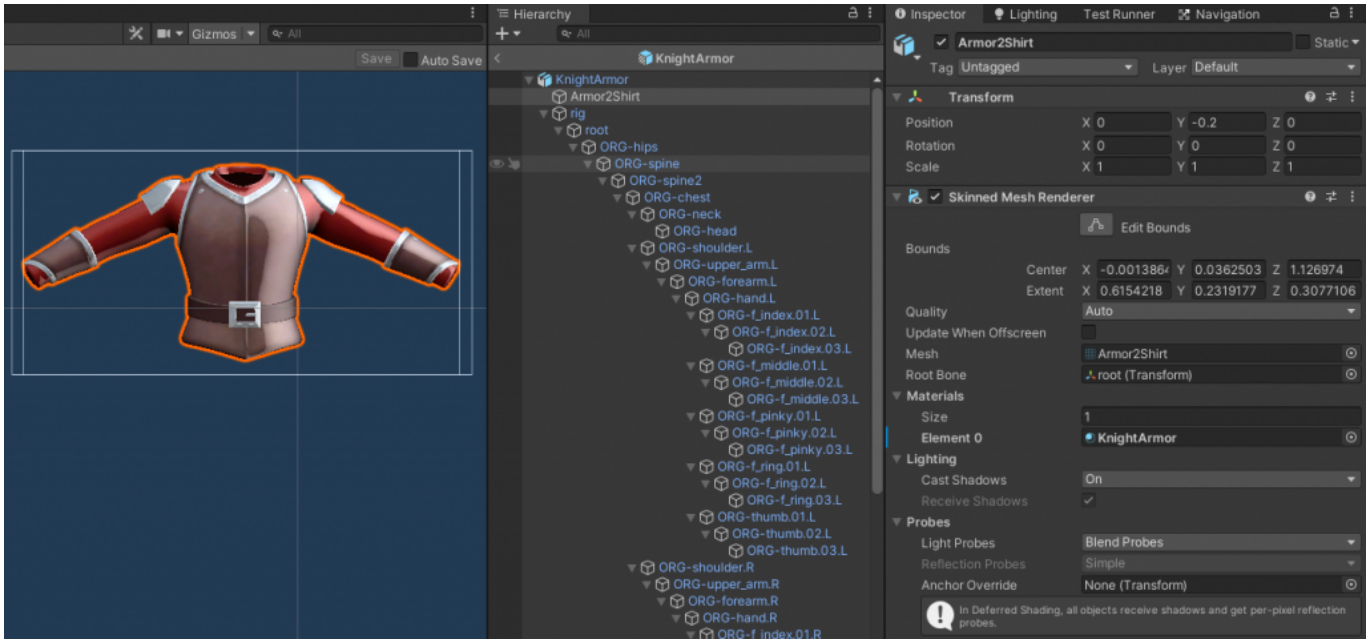
Skinned Mesh Equipment

Skinned Mesh Equipment prefabs must be set such that the skinned mesh can be stripped from the prefab and attached to the character bones when spawned.
The skinned mesh equipment prefab would be set as the “Equipment Prefab” attribute. In most use cases the “Usable Item Prefab” attribute would either not exist or be null.

It is recommended that the top level of the prefab is simply a transform containing two children: the Skinned Mesh and the full character rig.

> **IMPORTANT**: Without the Character rig the item won’t be able to be equipped as it won’t be able to cross reference the used bones with the character bones. Of course the character rig hierarchy on the item must match the character rig it will be equipped to.

Note that only the game object with the skinned mesh will be spawned on the character, the character rig will be discarded to clean up the character hierarchy. For that reason items are pooled per character and not shared between all characters.



As you can see the equipment system is quite simple and it is independent from the Usable Item Objects or character stats. This makes it very easy to extend or completely swap out for something that fits your game requirements.

Item Binding

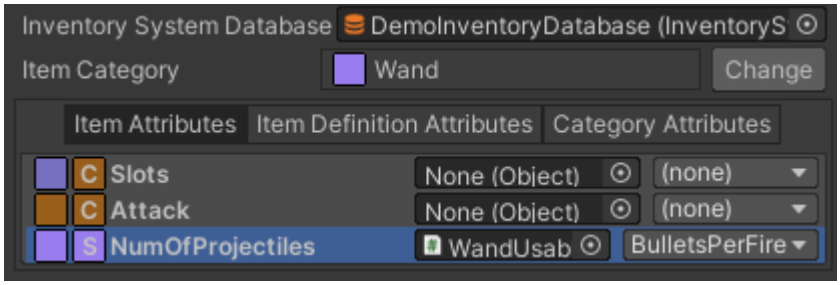
The Item Binding component allows you to bind an item attribute to any property of the same type. This means that when the attribute value is retrieved, it will use the property to get the value. Similarly, when an attribute value is set, it will use the property to set the value. This makes synchronizing data extremely easy. The category of the item must be specified ahead of time. This is because the Item Category determines what attributes the

item contains.

New bindings can be setup by performing the following:

- Select the GameObject that you'd like to add the binding to. This GameObject should be the visible representation of your item.
- Add the Item Object (or Usable Item Object) component to your GameObject.
- Add the Item Binding component to your GameObject.
- Select the database and the category for the attribute that you'd like to bind.
- Specify the component that contains the property that you'd like to bind.
- Select the property that you'd like to bind to the attribute.

In the example below we setup a binding between the NumOfProjectiles attribute with the BulletsPerFire property.



The Item Binding component works on properties, and not field. In the example above the BulletsPerFire property looks like:

```
// This is a field.
[SerializeField] protected int m_BulletsPerFire;

// This is a property.
public int BulletsPerFire
{
    get => m_BulletsPerFire;
    set => m_BulletsPerFire = value;
}
```

You can now set the ItemObjects Item at any time in the game and your RangeAttack "BulletsPerFire" property will always match the items "NumOfProjectiles" attribute value.

Item Pickups

Item pickups are components used to "pickup" items using an Interactable (the Item Pickup) and an Interactor (the character). There are multiple types of pickups:

- Item Pickup
- Inventory Pickup
- Currency Pickup
- Random Inventory Pickup

The Pickup Base class inherits the Interactable Behavior class. This class is used to add

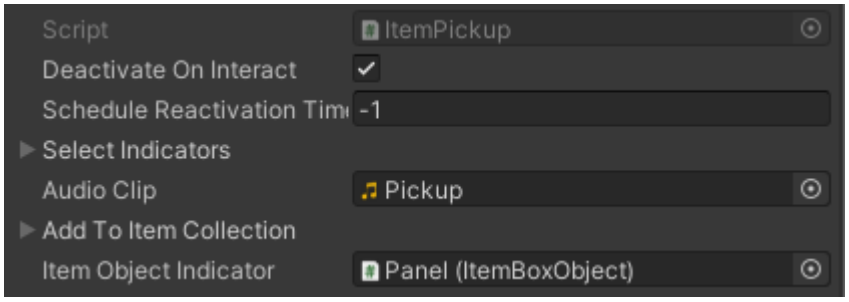
interactions to a GameObject and can be interacted with by any Interactor.

Note that the Item Pickup component is not the Interactable, it is an Interactable Behavior. For the Item Pickup component to work an Interactable component needs to be added. It works for both 2D and 3D trigger events.

There are three actions that can be performed on a Interactable component: select, deselect and interact. Select and deselect can be used to show an indicator when an Interactable is in range. With Item Pickups we use select and deselect to display the name of the item when the character is close enough to interact with it (pick it up).

Item Pickup

The Item Pickup component uses the Item Object component to decide which item is picked up. In some cases you may want the Item Pickup to display differently depending on the item attached to the Item Object. In the demo scene the Item Object Visualizer component listens to the item being set to the Item Object. This indicates when the mesh/sprite of the Item Pickup needs to be updated.



With this approach you can have a single Item Pickup prefab that works for all your items and you can customize how it looks from the inventory system editor.

The other types of pickups work in similar ways:

Currency Pickup

Currency Pickup works using a Currency Collection. When picked up it will give the Interactor the amount specified on the Currency Collection.

Inventory Pickup

The Inventory Pickup allows the Interactor to pick up multiple items. All the items in the inventory will be copied, and the copies will be given to the Interactor.

Random Inventory Pickup

The Random Inventory pickup is similar to the inventory pickup with the difference that instead of picking up the totality of the Inventory, the component will use the Item Amounts as a probability table. This can be used for loot boxes. You can set up a range for how many items will be picked up and the component will give the Interactor any random item within that change using the Inventory as a probability table.

Item Pickup Requirements

Character GameObject Requirements:

- *Rigidbody*: A Rigidbody/Rigidbody2D component and at least one Collider/Collider2D component.
- *Inventory*: An Inventory component.
- *Inventory Interactor*: A component inheriting the “Interactor With Inventory” interface. It allows you to interact with objects, like pickups, and let them know that you have an Inventory, allowing pickup items to be added to that inventory. Without an “Interactor With Inventory”, the character won’t be able to pickup items.
- Note the layer layer in which the character is set as it is used by the interactable component.

Item Pickup GameObject:

- *Item Pickup*: An Item Pickup component (or any of the other types of pickups).
- *Interactable*: An Interactable component, Rigidbody/Rigidbody2D, and at least one Collider/Collider2D component.
- *Trigger Collider*: The collider must be set as trigger.
- On the Interactable component the 2D or 3D trigger events should be selected, as well as the layer of the character.
- *Item Visualizer*: The Item Object Visualizer or Inventory Item Visualizer can be used to change how the item pickup looks like at runtime.

Item and Currency Droppers

There are many cases where you’ll want to drop items or currency at runtime. This may be when the enemy dies, a puzzle is solved or a tree was cut down.

There are three ways to drop Items: Item Object Spawner, Drop Item Action and Dropper components

Item Object Spawner

The item Object Spawner is probably the easiest one to use anywhere in code. All you need is to add this component somewhere in you project (it is recommended to add it next to you InventorySystemManager).

Make sure to assign it an ID (default 1), you will use this id in your code to find it. And assign it a prefab for your Item Pickup.

```
// Get the Item Object Spawner anywhere in your code.
var itemObjectSpawner =
InventorySystemManager.GetGlobal<ItemObjectSpawner>(itemObjectSpawnerI
D);

// Spawn an Item using an item Info and a position.
itemObjectSpawner.Spawn(itemInfo, spawnPosition);
```

```
// There are options to spawn with a delay, an auto destroy or even
both.
itemObjectSpawner.SpawnWithDelay(itemInfo, spawnPosition, delay);
itemObjectSpawner.SpawnWithAutoDestroy(itemInfo, spawnPosition,
autoDestroyTimer);
itemObjectSpawner.Spawn(itemInfo, spawnPosition, delay,
autoDestroyTimer);
```

Drop Item Action & Quantity Drop Item Action

The drop Item Actions can easily be added to an item Action Set to be used in the UI. There are options for the drop offset and radius, etc…

Droppers

The most designer friendly way to do spawn pickups is to use Droppers. This not only works for Items but also for inventories and currencies.

Simply add one of the dropper component on a gameobject (like an enemy). The dropper components are:

1. Item Dropper
2. Currency Dropper
3. Random Item Dropper
4. Random Currency Dropper

The Item and Random Item Droppers work with both Item and Inventory Pickups.

The Random Item Dropper is a good component to spawn pickups when an enemy dies. This component allows you to specify an inventory where the amounts for each item will be used to create a probability table. The higher the amount, the higher the probability it will get dropped. A drop radius can then be set, along with the item pickup prefab.

The Random Currency Dropper works in a similar manner except the random amount of currency is decided by a normal probability with a minimum and maximum range. This range is specified by a delta percentage offset from a base currency amount of a Currency Owner.

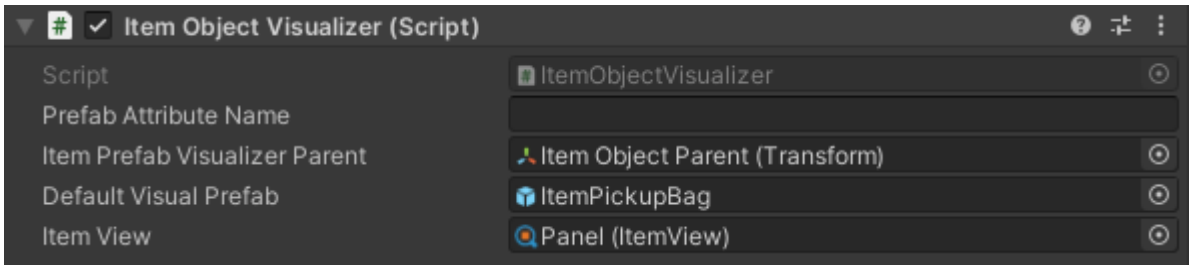
Once your droppers are set in the inspector, to drop the objects you can get a reference to the component and call the “Drop()” method. If you wish to drop a specific item you can call “DropItemPickup (item)” or “DropInventoryPickup(itemList)” or change the contents of the Inventory attached to the drop component.

Item Object Visualizer

The Item Object Visualizer is meant for the Item Object to take shape in the game world.

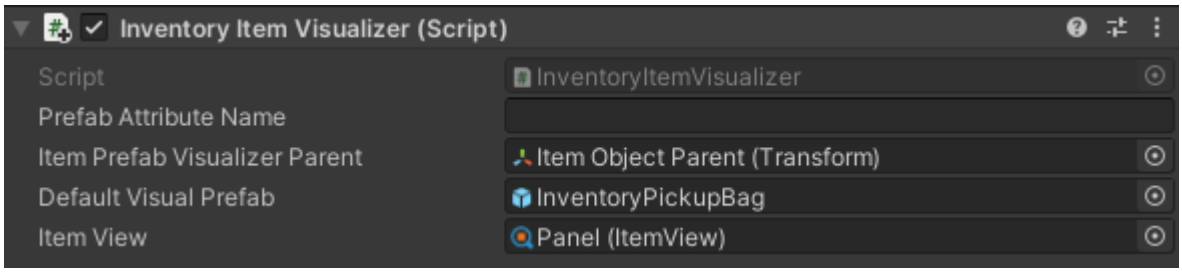
This is done by swapping the model/sprite of the Item Object when a new Item is bound to it. The model/sprite prefab can be set as an attribute on the Item. This allows us to pool the main Item Object prefab (Item Pickup, Weapon Item Object, etc.) and reuse it efficiently by

simply swapping the model/sprite which can also be pooled.



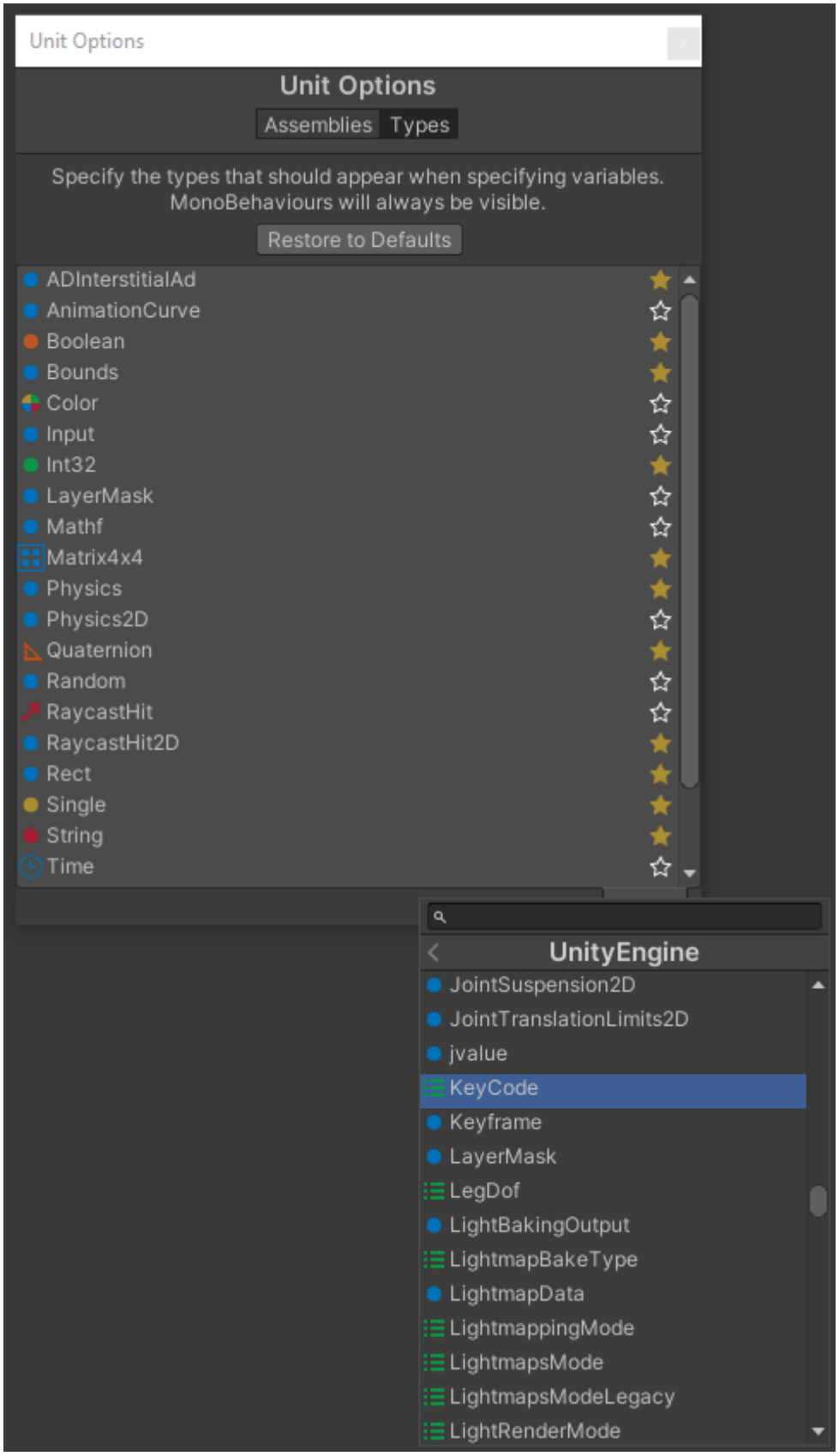
An Item View may also be referenced for world space UI.

A similar component called the Inventory Item Visualizer does the same thing but for the first Item found in an Inventory instead of an Item Object.



Attributes

Attributes are objects that have a name and a value. The value can be of any type. By default all of the common types (int, float, string, Vector3, etc) can be selected within the attribute dropdown box. New types can be added with the Unit Options window. The Unit Options window can be accessed from go to Tools -> Opsive -> Unit Options.



The Attribute class can also be extended in order to change the expression parsing logic. After you have created a new class make sure you add that class to the Unit Options editor. The editor will automatically detect classes that inherit the Attribute<T> class and you'll be able to choose it like any other type in the dropdown.

Attributes can get their value using three different variants:

- *Inherit:* The attribute will get its value from its parent attribute. The parent attribute is defined based on the object attached to it. This table shows the inheritance relations:

Child Attribute Attached To	Parent Attribute Attached To
Item	Default Item
Default Item	Default Item of Item Definition parent or Item Category (Item Attributes)
Item Definition	Item Definition parent or Item Category (Item Definition attributes)
Item Categories	Item Category (first parent containing attribute) or Null (or type default)

- *Override:* The attribute gets its value directly from itself.
- *Modify:* The attribute evaluates the modify expression and gets a resulting value.

Modify Variant

If your attribute variant is set to Modify then a *Modify Expression* field will appear. This expression works with the following attribute types:

- Int
- Float
- String

It is impossible to make a modify expression evaluator that works for all types. Therefore you must create your own Attribute Type if you wish to add a custom modify expression parser and evaluator. For consistency we recommend that you create your own modified expression parsers using the same syntax as we do.

Modify Expression Syntax

The Modify Expression is very powerful tool as it lets you grab values of other attributes as well as perform logic when retrieving the attribute value.

Important: Let's consider the case where you have Attribute A1 which inherits from Attribute A2. A2 is set to the "Modify" variant type. In this situation the attributes that are part of the expression will be picked from the perspective of the attribute that first requested the value, in this case A1. If the values should be picked from the perspective of the attribute holding the expression, in this case A2, one can add the '$' character in front of the attribute name.

As the expression is a string you can create your very own syntax by writing your own parser. But for consistency it is recommended to use the syntax below to get other attribute values.

- *[myOtherAttribute]:* Gets the value of an attribute named "myOtherAttribute".
- *<Override>:* Returns the override value.
- *<Inherited>:* Returns the inherited value.
- *$:* The dollar symbol can be added in front of any "get value". It will get the value of the attribute from the perspective of the attribute holding the expression.

Example: “$[myOtherAttribute]” or “$<Inherited>”.

Using Attributes on Item Objects

Most people will use Item attributes to affect Item Objects in certain ways. For example affecting damage amount using a float “Damage” attribute.
The easiest way to achieve this is to add an Item Binding component next to the Item Object on your game object prefab. It allows you to bind any attribute on your item to any component property in your prefab.
When needing more control over what happens when an Item is bound to an Item Object, you may listen to the event. This can be useful to spawn the correct Model/Sprite, play an audio clip, etc… which could all be defined as attributes on the Item, Item Definition or Item Category.

Get/Set Attribute API

Attributes are found in Item Categories, Item Definitions and Items. The methods used to get and set attribute values is similar for each object type. Here are examples on how to get attributes on an item:

```
// Retrieves an attack attribute value by getting its attribute.
var attackAttribute = item.GetAttribute<Attribute<int>>("Attack");
if (attackAttribute != null) {
    // The real attack value.
    var myAttack = attackAttribute.GetValue();
    // The override attack value.
    var myAttackOverride = attackAttribute.OverrideValue;
    // The inherited attack value.
    var myAttackInherited = attackAttribute.GetInheritedValue();
}

// Retrieves an icon attribute value.
if (item.TryGetAttributeValue("Icon", out Sprite myIcon)) {
    // myIcon exists.
}

// Loops through the Item attributes.
var indludeItemDefinitionAttributes = true;
var includeItemCategoryAttributes = true;
var attributeCount =
item.GetAttributeCount(indludeItemDefinitionAttributes,includeItemCate
goryAttributes);
for (int i = 0; i < attributeCount; i++) {
    var attribute = item.GetAttributeAt(i,
indludeItemDefinitionAttributes,includeItemCategoryAttributes);
    // We don't know the type of the attributes but we can get their
values as objects.
    Debug.Log(attribute.GetValueAsObject());
}
```

You are not allowed to create new attributes on the item as the Item Category will define which attributes the item must have.
Important: You are allowed to set attribute values at runtime ONLY when the item is mutable. You can set an item as mutable from its Item Category.

```
if (!item.IsMutable) {
    // Immutable items cannot have their attribute set.
}

// Retrieves an attack attribute.
var attackAttribute = item.GetAttribute<Attribute<int>>("Attack");
if (attackAttribute != null) {
    // The variant type can be changed.
    attackAttribute.SetVariantType(VariantType.Inherit);
    // Sets a new attack of 10. This will automatically set the
Variant Type to Override.
    attackAttribute.SetOverrideValue(10);
    // Sets a new modify expression. This will automatically set the
Variant Type to Modify and mark it as pre-evaluated.
    attackAttribute.SetModifyExpression("<Inherited> + 5",true);
}

// Be careful when using classes.
var itemSlotsAttribute =
item.GetAttribute<Attribute<ItemAmounts>>("Slots");
if (itemSlotsAttribute != null) {
    var itemSlots = itemSlotsAttribute.GetValue();
    // Do not change the itemSlots directly as it might be inherited!
    // The overriding value should be checked first.
    if (itemSlots == itemSlotsAttribute.OverrideValue) {
        // The itemSlots can be modified.
        itemSlots.Add(newItem);
    } else {
        // The itemSlots is inherited form another object. The content
should first be copied.
        itemSlots = new ItemAmounts(itemSlots);
        itemSlots.Add(newItem);
        // Do not forget to set the new itemSlots value to the
attribute.
        itemSlotsAttribute.SetOverrideValue(itemSlots);
    }
}
```

Common Attributes

Common Names

Description <string>

The “Description” Attribute is usually defined as an Item Definition Attribute. The Description attribute is used in in the “Item Description” component to display the description text.

Icon <Sprite>

The “Icon” Attribute is usually defined as an Item Definition Attribute. The Icon attribute is used in multiple places such as the “Icon Item View” component or even in the Editor to preview the Icon in the little colored box (Only if Editor Icon is null)

CategoryIcon <Sprite>

The “CategoryIcon” Attribute is usually defined as an Item Category Attribute. The CategoryIcon attribute is used in the Editor to preview the Item Category Icon in the little colored box (Only if Editor Icon is null)

Price, SellPrice, BuyPrice <CurrencyAmounts>

The Attributes “Price”, “SellPrice” and “BuyPrice” are usually defined as an Item Definition Attributes. Either “Price” or “SellPrice”+”BuyPrice” is used.
Some prefabs such as “Big Item Description” use “BuyPrice” and “SellPrice” attributes by default, this can be changed in the inspector.

PickupPrefab <GameObject>

The “PickupPrefab” Attribute is usually defined as an Item Definition Attribute. By default the “Item Object View” component will use this attribute to spawn the visual prefab when an Item is bound to an Item Object (example when dropping an Item Pickup).

UsableItemPrefab<GameObject>

The “UsableItemPrefab” Attribute is used by the Equipper component. It is spawned when the Item equips a none-SkinMesh. The prefab must have an ItemObject component and may also have usually an “Item Object Behavior Handler” component to use the equipped Item. The “UsableItemPrefab” Attribute value is meant to be used to define the weapon functionality. The “EquipmentPrefab” Attribute is the attribute used to show the Weapon/Cloth Model.

EquipmentPrefab<GameObject>

The “EquipmentPrefab” Attribute is used by the Equipper component. It is spawned when the Item equips an Item. It is the visual part of the Item. It is spawned under the “UsableItemPrefab” which contains the functionality.

ItemActionSet <ItemActionSet>

The “ItemActionSet” Attribute is used by the “Use Item Action Set Attribute” Item Action. It is really useful to give different Item Action depending on the Item. For example magic scrolls which can be cast but all do something completely different.

Shape<ItemShape>

The “Shape” Attribute is used by the Item Shape Inventory Grid system, both in the Grid Data components and the relevant Item View Modules. In all cases the attribute name can be set in the inspector if “Shape” is unavailable

Slots <ItemAmounts>

The “Slots” Attribute is used in the “Item Slots Item View” and is used to show the Icons of the Items that are slotted unto another item.

IsEquipped <Bool>

The “IsEquipped” Attribute is used in the “Equipped Item View” and can be used to defined if an Item is equipped or not. This is optional as usually the system can tell an Item is equipped simply by checking if the Item is contained in the Equipped Item Collection.
If it is used the “IsEquipped” Attribute should be set as Item Attribute and the Item should be Mutable such that it can be changed at runtime.

StackSizeLimit <Int>

The “StackSizeLimit” Attribute is used by the “MultiStack Item Collection” to limit Item Stacks max size.

CategorySizeLimit<Int>

The “CategorySizeLimit” Attribute is used by the “GroupItemRestriction” object. It is completely optional. It is used to limit the number of Items with that Item Category the Inventory can hold.

DefinitionSizeLimit<Int>

The “DefinitionSizeLimit” Attribute is used by the “GroupItemRestriction” object. It is completely optional. It is used to limit the number of Items with that Item Definition the Inventory can hold.

ItemSizeLimit<Int>

The “ItemSizeLimit” Attribute is used by the “GroupItemRestriction” object. It is completely optional. It is used to limit the number of particular Items the Inventory can hold.

Common Types

Attribute value types can be set to any type in your project. While attributes can be of any type, the types below are the more commonly used attribute types.

Int

Integers are used to define full numeric values. They also work with the modify expression feature.

Float

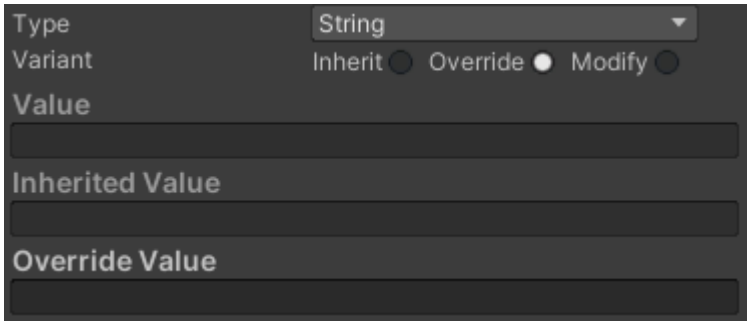
Floats, also known as Singles, are used to define decimal values. They also work with the modify expression feature.

Bool

Bools are used to define if a value is on or off.

String

Strings are used to define names, descriptions, etc. They also work with the modify expression feature.



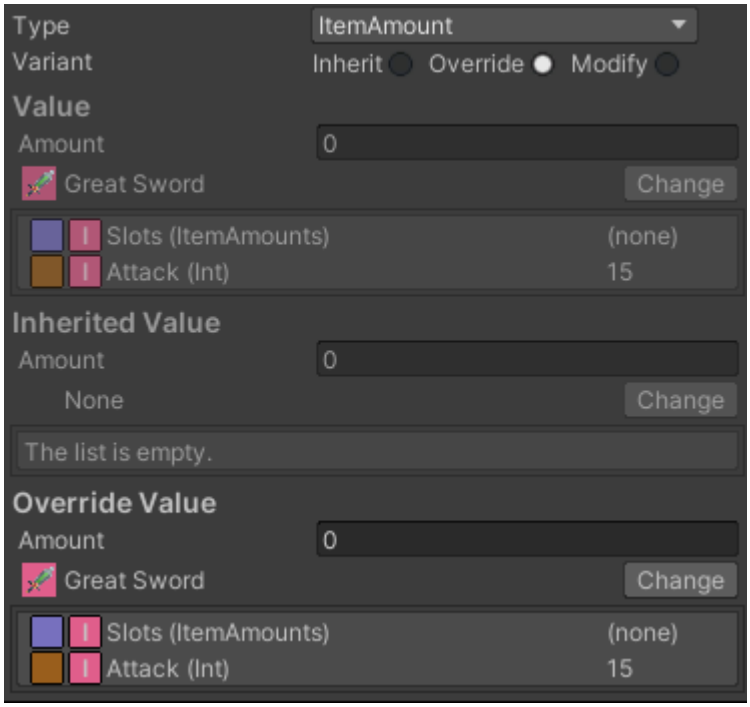
Unity Objects (GameObject, Sprite, Material, etc.)

Any Unity object can be used as an attribute and it can be used for referencing prefabs,

icons, sounds, etc.

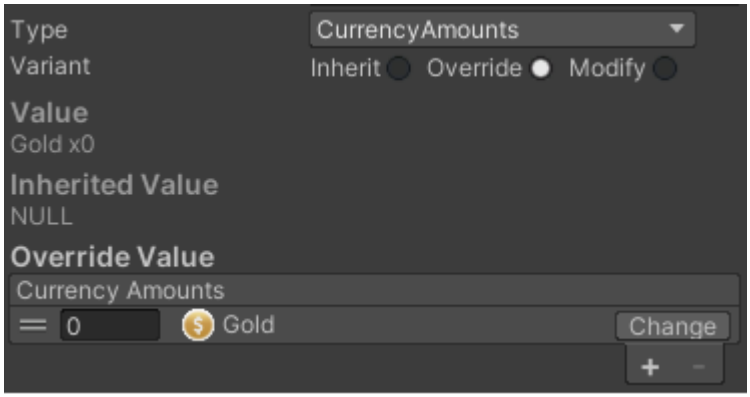
Item Amount

Item Amounts are used for item slots and attachments. It allows for item nesting and takes care of serializing/deserializing nested item correctly, which enables the saving and loading of nested items.



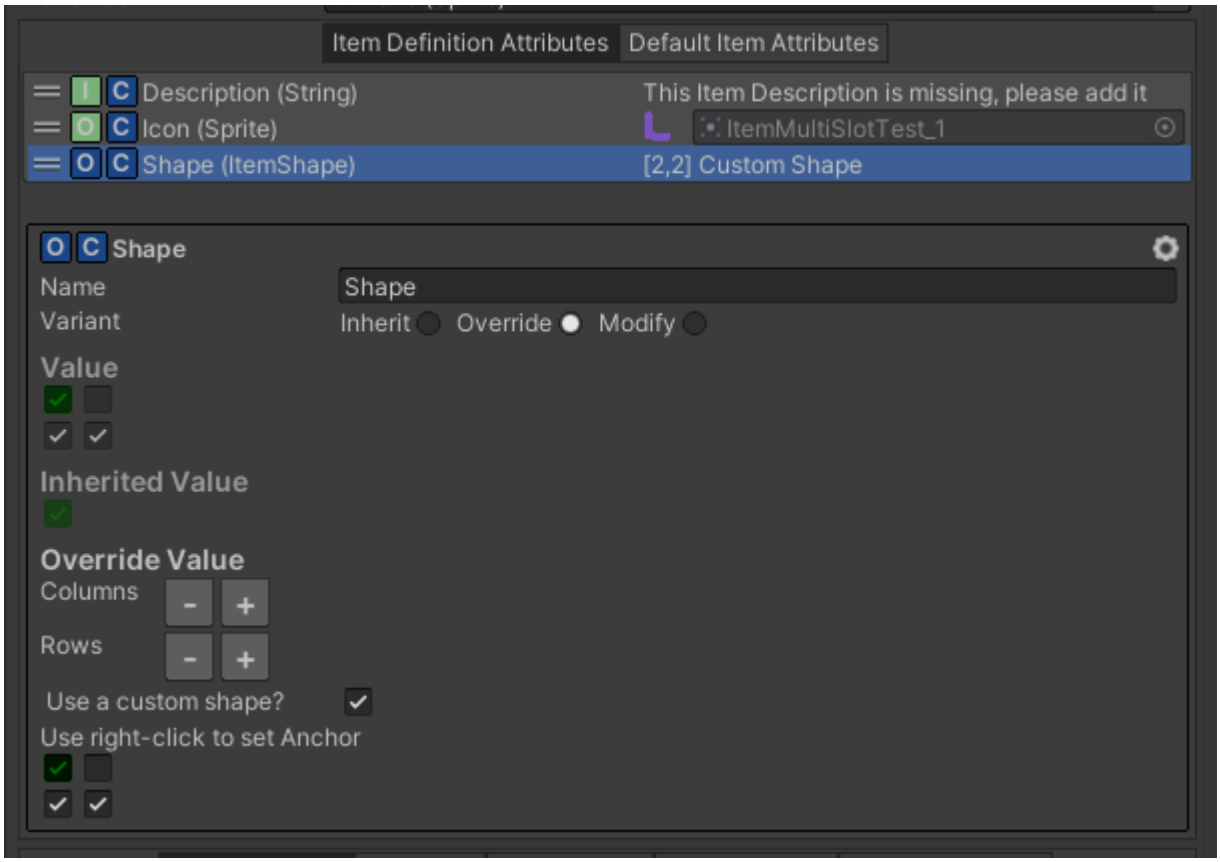
Currency Amounts

Currency Amounts are used to define the prices of items. It is usually defined as an Item Definition attribute as it does not usually change over time.



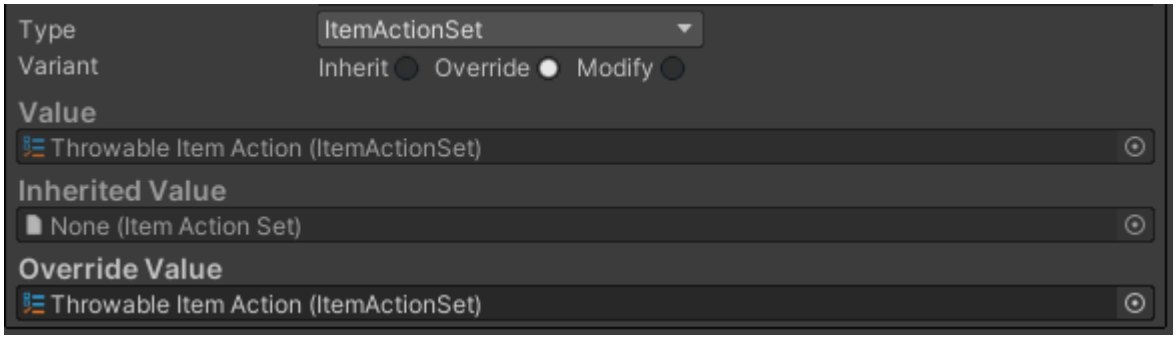
Item Shape

The Item Shape is used to define the shape of an Item in an Item Shape Inventory Grid. It let's set the shape with a boolean grid and an anchor.



Item Action Set

You may add scriptable objects as Attributes to add functionality to your items. A good way to use Item Actions which are item specific instead of Category specific is to use an Item Action Set as attribute.



Currency

The currency feature built into the inventory system allows for shops to buy, sell and exchange items from one inventory to another. Each project can have their own take on what a currency is and how it functions. The currency implementation has been abstracted out so that you can implement your unique currencies in case you need to. With that being said, the currency implementation is very generic and should accommodate the majority of use cases.

Currency

Currency is an object that contains data on how it converts to another currency. It is that simple. For example let's say that you want to implement a classic bronze, silver, and gold currency:

Bronze:

- *Max Value:* 99
- *Overflow Currency:* Silver

Silver:

- *Base Currency:* Bronze
- *Base Exchange Rate:* 1 Silver = 100 Bronze
- *Max Value:* 99
- *Overflows Currency:* Gold
- *Fraction Currency:* Bronze

Gold:

- *Base Currency:* Silver
- *Base Exchange Rate:* 1 Gold = 100 Silver
- *Max Value:* 99
- *Fraction Currency:* Silver

And just like that we have set up currencies. This data doesn't do anything by itself but when coupled with a Currency Collection they are used as constraints to know when and how to convert one currency to another.

You can set up as many currencies as necessary. In the example above there is only one root currency and that is Bronze. You can have many roots, but keep in mind that currencies that do not share the same root cannot be converted to one another. The conversion rate is computed by using the root base currency as reference.

```
//Get the dollar currency from the database.
var dollars = InventorySystemManager.GetCurrency("Dollars");

//create a variable holding 50$.
var fiftyDollars1 = new CurrencyAmount(50, dollars);
```

```
//You can use this shortcut if you find it easier to read.
var fiftyDollars2 = (50, dollars);
```

Currency Collection

A Currency Collection contains the collection of currencies, but it also contains the logic for adding, subtracting and dividing set of currencies. Using the previous example we could do the following:

- 1 Silver 99 Bronze + 1 Gold 1 Silver 1 Bronze = 1 Gold 2 Silver 0 Bronze
- 1 Silver – 99 Bronze = 1 Bronze
- 3 Silver / 40 Bronze = quotient: 7, remainder: 20 Bronze

When dealing with shops its also useful to have price drops or price inflation. Therefore you can also do:

- 1.5 * (2 Silver 1 Bronze) = 3 Silver 1 Bronze
- 50 * (3 Silver 10 Bronze) = 1 Gold 55 Silver 0 Bronze

Note that the Currency Collection may lose precision in some equations because it always keeps currency amounts to a valid state. Using the same example we cannot have amount lower than 1 Bronze or higher than 99 Gold 99 Silver 99 Bronze (the 99 was defined the currency specification).

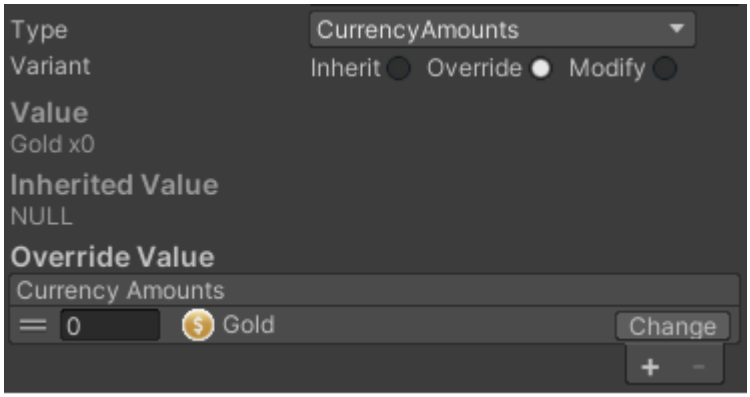
```
var dollars = InventorySystemManager.GetCurrency("Dollar");
var euro = InventorySystemManager.GetCurrency("Euro");

if (dollars.TryGetExchangeRateTo(euro, out var dollarsToEuro)) {
//The conversion is possible.
//Convert fifty dollars in euros.
var fiftyDollarsInEuros = (50*dollarsToEuro,euro);
}

// Debug the amount of currency owned.
Debug.Log(currencyOwner.CurrencyAmount);

// Add 50 Dollars to the Currency Owner.
currencyOwner.CurrencyAmount.AddCurrency(dollars,50);
Debug.Log(currencyOwner.CurrencyAmount); // 50 Dollars

// Check if the currency owner own at least 30 Dollars.
Debug.Log(currencyOwner.CurrencyAmount.HasCurrency(dollars,30));//True

// Check if the currency owner own at least 70 Dollars.
Debug.Log(currencyOwner.CurrencyAmount.HasCurrency(dollars,70));//Fals
e

// Remove 20 Dollars.
currencyOwner.CurrencyAmount.RemoveCurrency(dollars,20);
Debug.Log(currencyOwner.CurrencyAmount); // 30 Dollars
```

Currency Owner

The Currency Owner is essentially an inventory for your currency. If your character can use currency then the Currency Owner should be added to them. Currency can be anything in an abstract sense. It can be an object, a value, a concept, etc. The currency system does not impose restrictions on the type of currency used. If you'd like to implement your own currency logic you can implement the CurrencyOwner<CurrencyT> class with your own currency type. The Shop also has a Shop<CurrencyType> class that can be implemented.

Pricing items

Item can be given a price as attributes. It is recommend to use the *Currency Amounts* attribute type which lets choose an array of amounts of currency. In the shop you are able to set a buy and sell price using an attribute name.



Shop

Shops are used to buy and sell items using currency. The currency could be anything. By default the shop can be used with Currency Collection currency. But you can create a new shop with different types of currency.

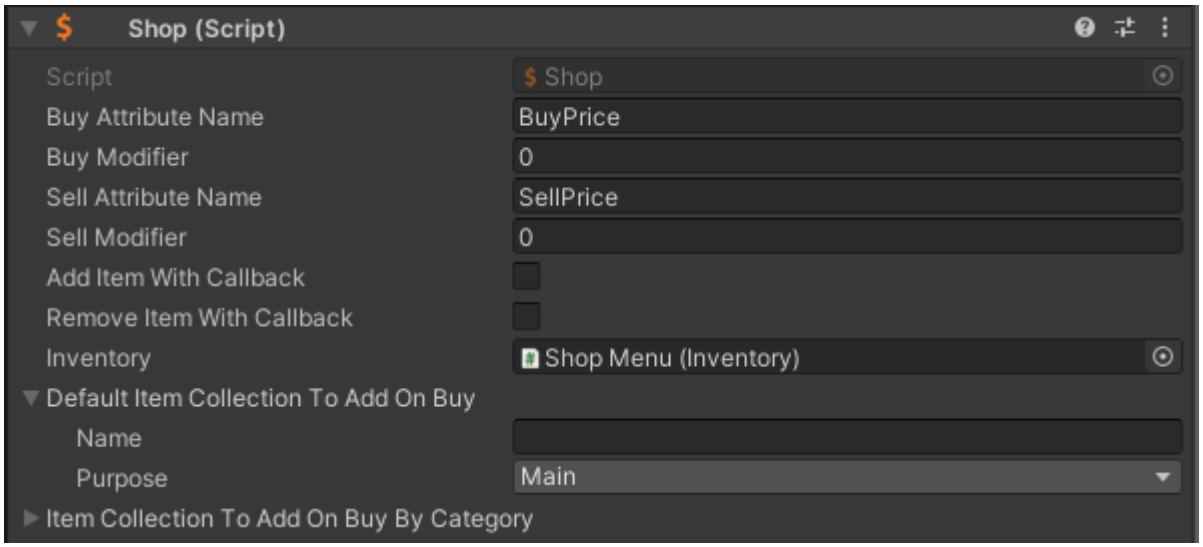
Buying and selling items from/to a shop is as simple as calling the following functions:

```
/// <summary>
/// Buy an item from the shop.
/// </summary>
/// <param name="buyerInventory">The inventory of the buyer.</param>
/// <param name="currencyOwnerBase">The currency owner of the
buyer.</param>
/// <param name="item">The item to buy.</param>
/// <param name="amount">The amount to buy.</param>
/// <returns>The process can fail returning false.</returns>
BuyItem(Inventory buyerInventory, ICurrencyOwner currencyOwnerBase,
Item item, int amount)

/// <summary>
/// Sell an item to the shop.
/// </summary>
/// <param name="sellerInventory">The inventory of the seller.</param>
```

```
/// <param name="currencyOwnerBase">The currency owner of the
seller.</param>
/// <param name="item">The item being sold.</param>
/// <param name="amount">The amount being sold.</param>
/// <returns>The process can fail returning false.</returns>
SellItem(Inventory sellerInventory, ICurrencyOwner currencyOwnerBase,
Item item, int amount)
```

The item prices are defined directly in the editor using attributes. It is recommended to define the buy price and sell price as Item Definition attributes with the name “BuyPrice” and “SellPrice” respectively. If you do not want to define a sell price you can go the the Shop Currency Collection component and change the sell price attribute name to “BuyPrice”. This will allow both the buy and sell price values will be fetched from the “BuyPrice” attribute.

The Shop component allows you to set a price modifier for buying and selling. This allows you to easily create a shop which sell prices are 20% lower than the sell price by setting sell modifier to “-0.2”.



The shop is designed to be extremely flexbile so you can customize the items being sold, the sell and buy prices, and more. It can even be extended to have a custom price depending on who is buying or selling items

There are many ways a shop could work. By default the shopkeeper has an Inventory that contains the items that can be sold. When an item is sold it is duplicated so the shop has a never ending supply of items. The functionality can be changed so that the are not duplicated and therefore sets a limit on the number of items that can be bought. You could also make it dynamic in the sense that you could buy back items that you just sold. These are ideas that are not implemented by default but can be by extending the Shop Currency Collection class.

Crafting

Crafting is often used when working with a lot of items. There are so many types of crafting systems that it would be impossible to create a system that fits all. The included crafting

system is very generic and scalable while at the same time can do very simple crafting by default.

There are a few key components when dealing with crafting:

1. *Crafting Category:* Used to organize recipes, similar to Item Category for organizing items.
2. *Crafting Recipe:* The data used to specify the ingredients and default output.
3. *Crafting Ingredients:* A collection of objects used as inputs for a recipe.
4. *Crafting Output:* A collection of items used as output for a recipe.
5. *Crafting Processor:* The logic that crafts items using a recipe as a guideline.
6. *Crafting Result:* The result from the crafting process containing a crafting output (the craft items) and additional data such as whether the crafting process was successful or not.
7. *Crafter* : The component which has the Crafting processor and a list of the recipes which can be crafted.

These components can be extended to design an extremely complex crafting system. For example the crafting processor could take the player data as input to know whether they can craft an item, a recipe could give different outputs depending on the actual ingredient used, etc.

The crafting system will not do exactly what you want out of the box. The following will showcase the default functionality which allows for a slightly complex crafting scenario.

Crafting Recipe, Ingredients, and Output

As specified before, crafting recipes are guidelines for the crafting processor. Therefore they are quite abstract. Out of the box the recipe has ingredients and an output.

Ingredients contain a list of amounts of Item Categories, Item Definitions and Items. In simple crafting systems you would use Item Definitions as ingredients, such that any item of the Item Definitions specified could be used as ingredients in the crafting process.

Output is a list of item amounts. Out of the box you can define a simple recipe such as

2 Heal weed + 1 spring water => 1 heal potion

But you can also make a more specific recipe:

1 heal potion (with quality == good as an item attribute) + 1 magic orb => 1 health magic orb

Or something more abstract

1 sword + 5 material => 1 great sword

where material is a category and can be any item of that category. For example:

1 sword + (2 iron ingot + 1 cotton + 2 rocks) => 1 great sword

With a little creativity the out of the box solution can be used in interesting ways. The extensibility and scalability makes it easy to create your very own crafting system.

Crafting Processor

Recipes can be bent a little to your liking since they are guidelines and not rules. This means that two crafting processors which use the same recipe can have different results.

The processor takes in a recipe and a list of selected items and then tells you if it can craft or not. If it can, then you can craft the item, which will remove the selected items from the inventory and add the crafted item.

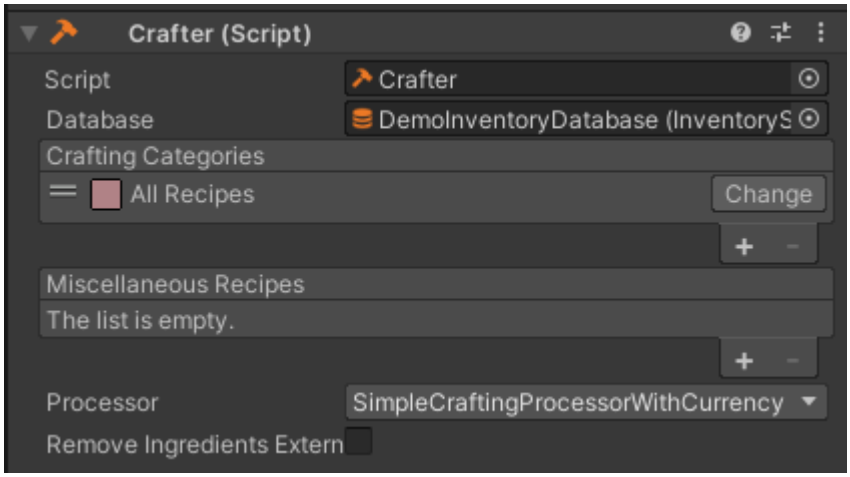
```
/// <summary>
/// Craft the items.
/// </summary>
/// <param name="recipe">The recipe.</param>
/// <param name="inventory">The inventory containing the
items.</param>
/// <param name="selectedIngredients">The items selected to craft the
recipe.</param>
/// <param name="quantity">The quantity to craft.</param>
/// <returns>Returns a crafting result.</returns>
public CraftingResult Craft(CraftingRecipe recipe, IInventory
inventory,
    ListSlice<ItemInfo> selectedIngredients, int quantity = 1)
```

For convenience the processor can also auto select valid items to craft directly from the inventory. It will do so by default if you do not specify a list of items.

The Crafting Processor is quite basic, and it is meant to be easily extendable to suit your needs. You may decide to extend the crafting processor to allow crafting only when your character has a certain crafting level, or change the result of the crafting processing depending on the quality of the ingredients used, etc…

Crafter

The crafter is the component which has the Crafting Processor and the list of Item Categories which can be crafted. By default it uses a “Simple Crafting Processor With Currency” which may be changed in the inspector using the dropdown. Your custom Processor should automatically appear as an option once created.



- *Remove Ingredients Exernally*: In some cases you may want to remove the ingredient item differently than normal, if true the event "EventNames.c_InventoryGameObject_OnCraftRemoveItem_CraftingProecessor_ItemInfoListSlice_ActionBoolSucces" will be used to remove the ingredients.
- *Crafting Categories*: All the recipes in that category will be added in the list of craftable recipes.
- *Miscellaneous Recipes*: A list of craftable recipes.

```
// Edit the list of recipes which can be crafter by adding/removing
recipes from the list
List<CraftingRecipes> recipes = m_Crafter.CraftinRecipes;

// Change the Crafting Processor at anytime
m_Crafter.Processor = newProcessor;
```

The Crafter is used by the Crafting Menu. Learn more about it here

Custom Crafting Processors

The crafting system comes with two crafting processors out of the box: Simple Crafting Processor and Simple Crafting Processor with Currency. These two processors will cover many use cases but can be expanded upon to fit any use case. The examples below explain how the system can be extended.

Crafting level constraint

In some games, recipes can only be crafted if the character has a certain level. This can be implemented in one of two different ways:

1. Split the recipes by crafting categories and map the character level to crafting categories using a custom component, Scriptable Object or directly on the crafting processor.
2. Create a custom Crafting Recipe type and add a field for a character level.

Both of these options can be implemented by creating a new processor and overriding the Can Craft function.

```
public class CustomCraftingProcessor : SimpleCraftingProcessor
{
    /// <summary>
    /// Check if the parameters are valid to craft an item.
    /// </summary>
    /// <param name="recipe">The recipe.</param>
    /// <param name="inventory">The inventory containing the
items.</param>
    /// <param name="quantity">The quantity to craft.</param>
    /// <param name="selectedIngredients">The item infos
selected.</param>
    /// <returns>True if you can craft.</returns>
    protected override bool CanCraftInternal(CraftingRecipe recipe,
IInventory inventory, int quantity, ListSlice<ItemInfo>
selectedIngredients)
    {
        // Get the character level from your custom character script.
        var myCharacterLevel =
inventory.gameObject.GetComponent<MyCharacter>().level;
        // Find the level required by the recipe.
        var levelRequired = 0;
        // Option 1: create the GetLevelFor function that return the
level required for crafting recipes in that category.
        levelRequired = GetLevelFor(recipe.Category);
        // Option 2: create a custom recipe with a level field and use
that field as the constraint.
        if (recipe is MyCustomRecipe customRecipe) {
            levelRequired = customRecipe.level;
        }

        if (myCharacterLevel < levelRequired) { return false; }

        return base.CanCraftInternal(recipe, inventory, quantity,
selectedIngredients);
    }
}
```

Change processor output with external sources

In some games the crafting result can change depending on external sources, such as a mini game score, the player stats or the items quantity/quality. This can be implemented with one of the following methods:

1. Use the recipe item amounts output to make a list of different possible result. This method is limited as you would only be able to return a single output per craft.
2. Make the crafted Item mutable such that its attribute values can be changed when it is

crafted.
3. Create a custom recipe type with multiple output possibilities.

These options can be implemented by creating a new processor and overriding the Craft Internal method:

```
public class CustomCraftingProcessor : SimpleCraftingProcessor
{
    // Your custom code that returns a crafting quality. Examples
include a character or a mini game.
    [SerializeField] protected MyCraftringQualityProvider
m_CraftingQualityProvider;

    // Optional value for testing.
    [SerializeField] protected int m_Option;
    /// <summary>
    /// Craft the items.
    /// </summary>
    /// <param name="recipe">The recipe.</param>
    /// <param name="inventory">The inventory containing the
items.</param>
    /// <param name="quantity">The quantity to craft.</param>
    /// <param name="selectedIngredients">The item infos
selected.</param>
    /// <returns>True if the character can craft.</returns>
    protected override CraftingResult CraftInternal(CraftingRecipe
recipe, IInventory inventory, int quantity,
        ListSlice<ItemInfo> selectedIngredients)
    {
        // Check that the recipe can be crafted.
        if (CanCraftInternal(recipe, inventory, quantity,
selectedIngredients) == false) {
            return new CraftingResult(null, false);
        }

        // Remove all the ingredient items from the inventory.
        if (RemoveIngredients(inventory, selectedIngredients) ==
false) {
            return new CraftingResult(null, false);
        }
        // Get the crafting quality from an external source.
        var craftingQuality =
m_CraftingQualityProvider.CraftingQuality;

        // Get the result of the craft.
        ItemAmount[] resultItemAmounts;

        // Option 1 : choose one of the outputs in the list of output.
This solution is limited to one output.
```

```
        if (m_Option == 1) {
            resultItemAmounts = new ItemAmount[1];

            var selectedIndex = 0;
            if (recipe.DefaultOutput.ItemAmounts.Count <
craftingQuality) {
                selectedIndex = recipe.DefaultOutput.ItemAmounts.Count
- 1;
            }
            var itemAmount =
recipe.DefaultOutput.ItemAmounts[selectedIndex];
            var craftedItemAmount = new
ItemAmount(InventorySystemManager.CreateItem(itemAmount.Item),
itemAmount.Amount * quantity);
            resultItemAmounts[0] = craftedItemAmount;
        }

        // Option 2 : Set the quality of the mutable item directly by
changing the attribute value.
        else if (m_Option == 2) {
            resultItemAmounts = new
ItemAmount[recipe.DefaultOutput.ItemAmounts.Count];
            for (int i = 0; i < resultItemAmounts.Length; i++) {
                var itemAmount = recipe.DefaultOutput.ItemAmounts[i];
                var craftedItem =
InventorySystemManager.CreateItem(itemAmount.Item);

                // The attribute can be any type and does not need to
be an integer.
                var qualityAttribute =
craftedItem.GetAttribute<Attribute<int>>("Quality");
                if (qualityAttribute != null) {
qualityAttribute.SetOverrideValue(craftingQuality);
                }
                var craftedItemAmount = new ItemAmount(craftedItem,
itemAmount.Amount * quantity);
                resultItemAmounts[i] = craftedItemAmount;
            }
        }
        // Option 3 : create a custom recipe type with multiple
crafting output possibilities.
        else {
            if (!(recipe is MyCustomQualityRecipe qualityRecipe)) {
return new CraftingResult(null, false); }

            ItemAmounts itemOutputAmounts =
qualityRecipe.GetOutputForQuality(craftingQuality);

            resultItemAmounts = new
```

```
ItemAmount[itemOutputAmounts.Count];
            for (int i = 0; i < resultItemAmounts.Length; i++) {
                var itemAmount = itemOutputAmounts[i];
                var craftedItem =
InventorySystemManager.CreateItem(itemAmount.Item);
                var craftedItemAmount = new ItemAmount(craftedItem,
itemAmount.Amount * quantity);
                resultItemAmounts[i] = craftedItemAmount;
            }
        }

        // Add the crafted items in the inventory.
        for (int i = 0; i < resultItemAmounts.Length; i++) {
            inventory.AddItem((ItemInfo)resultItemAmounts[i]);
        }
        // Convert the crafted items into an crafting output.
        var output = new CraftingOutput(resultItemAmounts);

        // Return a success crafting result.
        return new CraftingResult(output, true);
    }
}
```

Input

The built-in input uses the Input Manager for its input which has mouse, keyboard, and limited controller support. You may install the required Inputs for the Ultimate Inventory System by going to the Setup Manager in the Editor.

If your project makes use of controllers it is highly recommend that you use an asset dedicated to controller support such as Rewired, InControl or the new Unity Input System. The Ultimate Inventory System is integrated with each of these assets and the integration can be downloaded here.

The Player Input component is base component that the Unity Input and integration components derive from. By using the Player Input component any new input can be added by just swapping out the component and the rest of the code does not need to change.

Events

The “OnEnableGameplayInput” event can be sent if you’d like to disable (or enable) character input. As an example the following will disable the input on the character specified by m_Character:

```
EventHandler.ExecuteEvent(m_Character, "OnEnableGameplayInput",
false);
```

The Opsive.Shared.Events namespace must be included for the EventHandler class to be found.

```
using UnityEngine;
using Opsive.Shared.Events;

public class MyObject : MonoBehaviour
{
    [Tooltip("A reference to the Ultimate Character Controller
character.")]
    [SerializeField] private GameObject m_Character;

    /// <summary>
    /// Disable the input.
    /// </summary>
    private void Start()
    {
        EventHandler.ExecuteEvent(m_Character,
"OnEnableGameplayInput", false);
    }
}
```

API

The PlayerInput API follows the same standard as Unity's input class with GetButton, GetButtonDown, GetButtonUp, and GetAxis methods. This component also allows for double press or long press detection with GetDoublePress and GetLongPress.

As an example the following code will check if the "Jump" button is down:

```
using UnityEngine;
using Opsive.UltimateCharacterController.Input;

public class MyObject : MonoBehaviour
{
    [Tooltip("A reference to the Ultimate Character Controller
character.")]
    [SerializeField] private GameObject m_Character;

    /// <summary>
    /// Initialize a reference to PlayerInput.
    /// </summary>
    private void Awake()
    {
        m_PlayerInput = m_Character.GetComponent<PlayerInput>();
    }

    /// <summary>
    /// Check for the jump button press.
    /// </summary>
    private void Update()
    {
```

```
        if (m_PlayerInput.GetButtonDown("Jump")) {
            // Do Jump.
        }
    }
}
```

Inspected Fields

Horizontal Look Input Name

The name of the horizontal camera input mapping.

Vertical Look Input Name

The name of the vertical camera input mapping.

Look Vector Mode

Specifies how the look vector is assigned. If a mouse is being used then the look vector will almost always be based on the mouse movement. If a controller is used then the controller will supply the look input.

- *Smoothed*: Apply a smoothing to the look vector. This smoothing interpolates the input over several frames to reduce jerky player input.
- *Unity Smoothed*: Uses the input's direct value.
- *Raw*: Use the input's direct (raw) value.
- *Manual*: The look vector is assigned manually. This is useful for VR head movement.

Look Sensitivity

If using look smoothing, specifies how sensitive the mouse is. The higher the value to more sensitive.

Look Sensitivity Multiplier

If using look smoothing, specifies a multiplier to apply to the LookSensitivity value.

Smooth Look Steps

If using look smoothing, the amount of history to store of previous look values.

Smooth Look Weight

If using look smoothing, specifies how much weight each element should have on the total smoothed value (range 0-1). Reducing the weight to 0 is similar to setting the Look Vector Mode to Raw. A value of 1 will cause the result to be a simple average of all of the recently sampled smooth steps (and will feel very laggy).

Smooth Exponent

If using look smoothing, specifies an exponent to give a smoother feel with smaller inputs.

Look Acceleration Threshold

If using look smoothing, specifies a maximum acceleration value of the smoothed look value (0 to disable).

Controller Connected Check Rate

The rate (in seconds) the component checks to determine if a controller is connected. The only way to check if a controller is connected in Unity generates garbage so this value should be set to 0 if you do not plan on supporting controllers.

Connected Controller State

The state that should be activated when the controller is connected.

Force Input

Specifies if any input type should be forced. By default the controller will enable the virtual buttons for a mobile platform but this field allows you to force standalone input on a mobile platform. Virtual input can also be forced for a platform that normally does not use virtual controls.

Disable Cursor

Should the cursor be disabled?

Enable Cursor with Escape

Should the cursor be enabled when the escape key is pressed?

Prevent Look Vector Changes

If the cursor is enabled with escape should the look vector be prevented from updating?

Handlers

The handler components use input to trigger the Ultimate Inventory System API. The handles use the PlayerInput component which adds an abstraction level that allows to swap out the input system without needing to rewrite any code.

Display Panel Manager Handler

Used to open and close panels.

Usable Equipped Item Handler

Allows you to setup the input to use the equipped Usable Item Object when using the default Ultimate Inventory System Equipping System.

Inventory Interactor

The Inventory Interactor has the option to be triggered using a input.

Hotbar Handler

Has inputs to use an item in a specific slot.

Item View Slot Container Item Action Handler

Allows you to map an input to an Item Action and trigger it for the selected Item View Slot.

Grid Base

The base class of the Item Info Grid and other gird components. It deals with the input to go to the next or previous tab.

Split Screen Co-op UI

The Inventory System allows for co-op with split screen UI. Below is shown an example for a co-op setting where players are on one screen and each have their own independent UI.

Bother players have their own canvas with their Inventory

If the option to set time scale to zero when a panel is opened is set to false on both Display Panel Managers, a player can run around while the other does some shopping.

Menus like the Shop, Crafting, etc… can be opened independently for each player. Both players are allowed to use the same Shop and Crafter at the same time.

There are some restrictions to split screen UI:

- Both characters must have their own UI canvas.
- You must use the new Input System from Unity or Rewired from the asset store.
- When using this method with the new Input System from Unity only gamepads can be used, keyboard and mouse won't work in the UI at all (a current limitation of the new Unity Input System, which Unity plans to fix in future versions).

Here are the steps to turn the demo into a Split Screen UI scene.

Import one of the advanced Input System & Integration

New Input System From Unity

The following steps should be performed in order to setup the Unity input system with your character:

1. Import the new Input System package from the package manager. Once you import the Input System package you must also import the integration it can be found here.
2. On the Player Character GameObject replace the "Unity Input" component with the "Unity Input System" component from the integration.
3. Make sure your Player Character GameObject also has a Player Input component from the new Input System package. It must reference a Input Action Asset. You may create one from scratch, but for convenience it is recommended to use the one provided in the integration: Character Input.
4. Test the demo to make sure everything still works.

Rewired

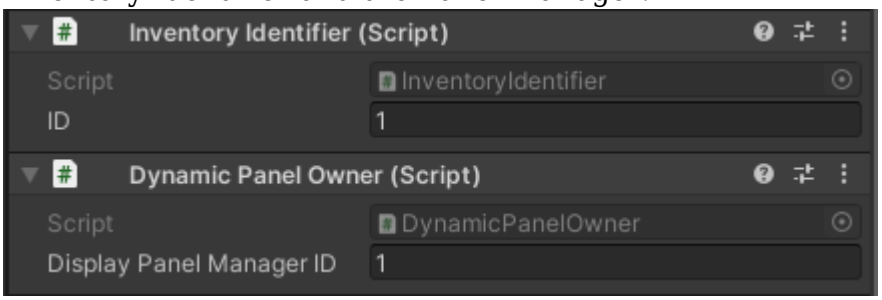
The following steps should be performed in order to setup the Rewired input system with your character:

1. Import the Rewired package from the asset store or the package manager. Once you import the Rewired package you must also import the integration it can be found here.
2. On the Player Character GameObject replace the “Unity Input” component with the “Rewired Input” component from the integration.
3. Add the “Rewired Input Manager Co-op” prefab from the integration package into your scene. This prefab has the inputs pre-setup for co-op. “Player1” with ID 0 and “Player2” with ID 1. This ID must be set on the Rewired Input component.
4. Test the demo to make sure everything still works.

Duplicate the player & UI

The second character and UI should now be added to the scene. This can be done by performing the following steps:

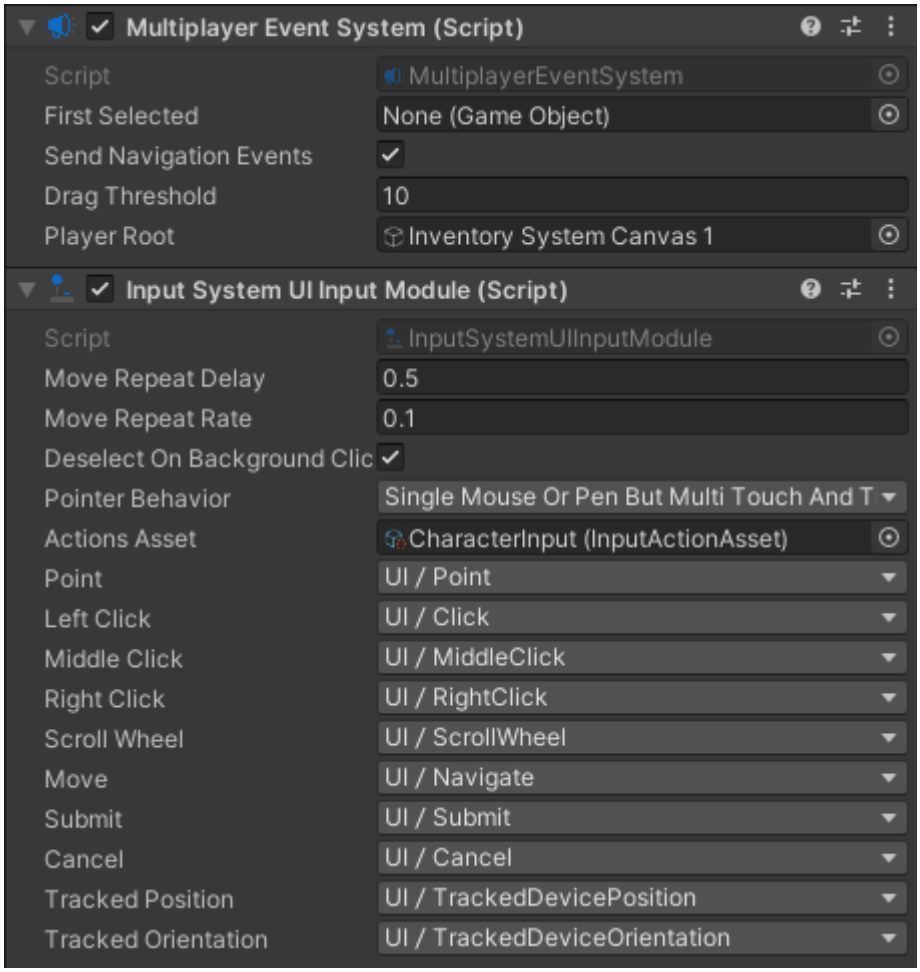
1. Duplicate the player GameObject and the Inventory Canvas.
2. Resize the top level panel in the canvas such that it fits half the screen. The easiest way is to scale it by half.
3. On the players make sure to set a different Inventory Identifier ID (recommended 1 and 2). If you are using a Dynamic Panel Owner make sure to use the same ID for the Inventory Identifier and the Panel Manager.



4. Remove the Event System Game Object. Replace it by with two Game Objects called Event System 1 and 2.

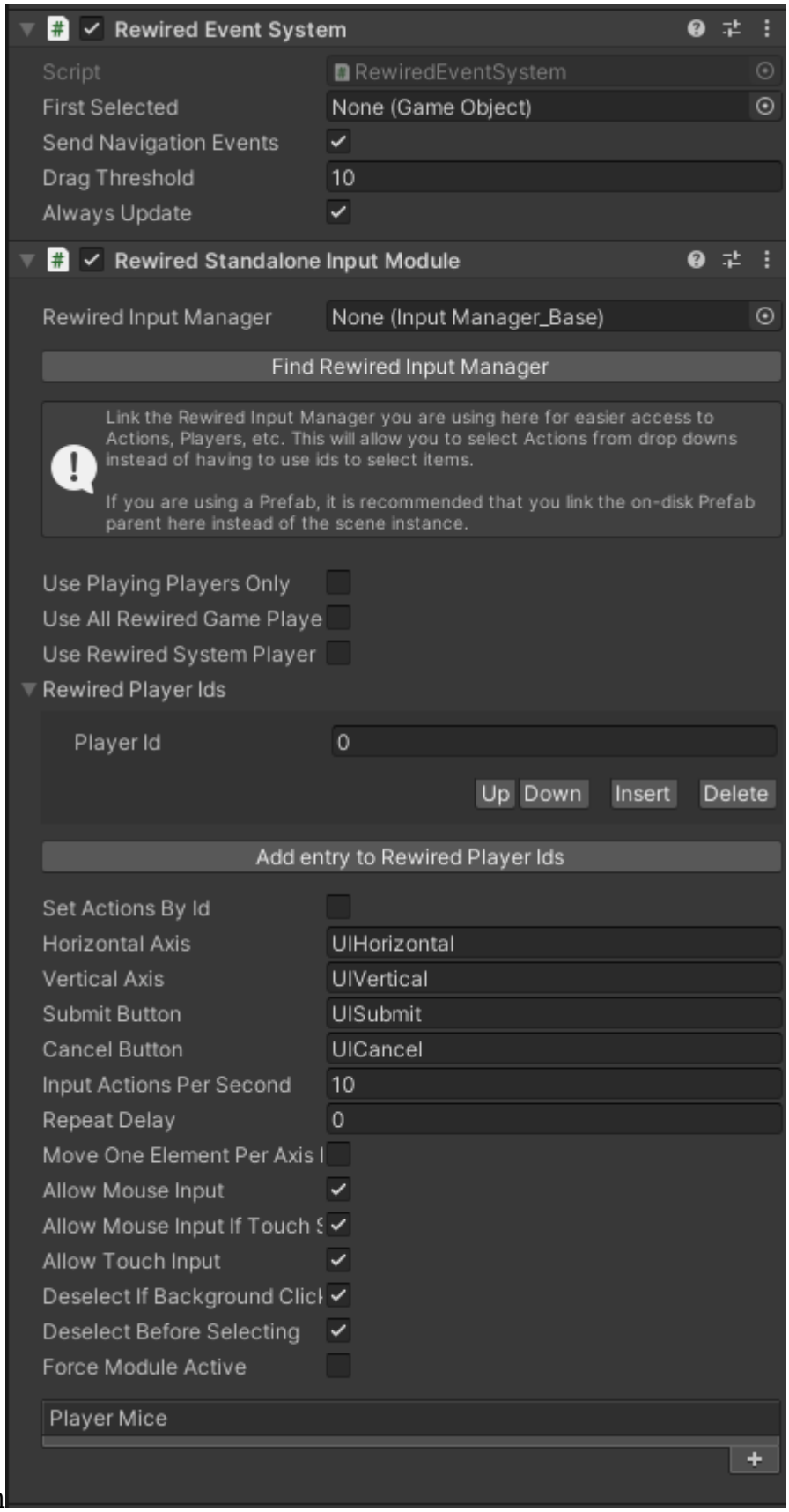
Unity Input System

1. On these Event System Game Objects add two components: Multiplayer Event System and Input System UI Input Module. Both are part of the Unity Input System.
2. In the “Player Root” field of the Multiplayer Input System make sure to assign the appropriate Canvas.
3. By default the Input System UI Input Module will use a UI Action Asset. This should be removed. The Input System UI Input Module within the Player Input on your character should be referenced instead. Make sure that each player points to a different Event System GameObject.
4. On the Player Input component set the *Default Scheme* from <Any> to Gamepad. Keyboard input does not work for multiplayer event systems.



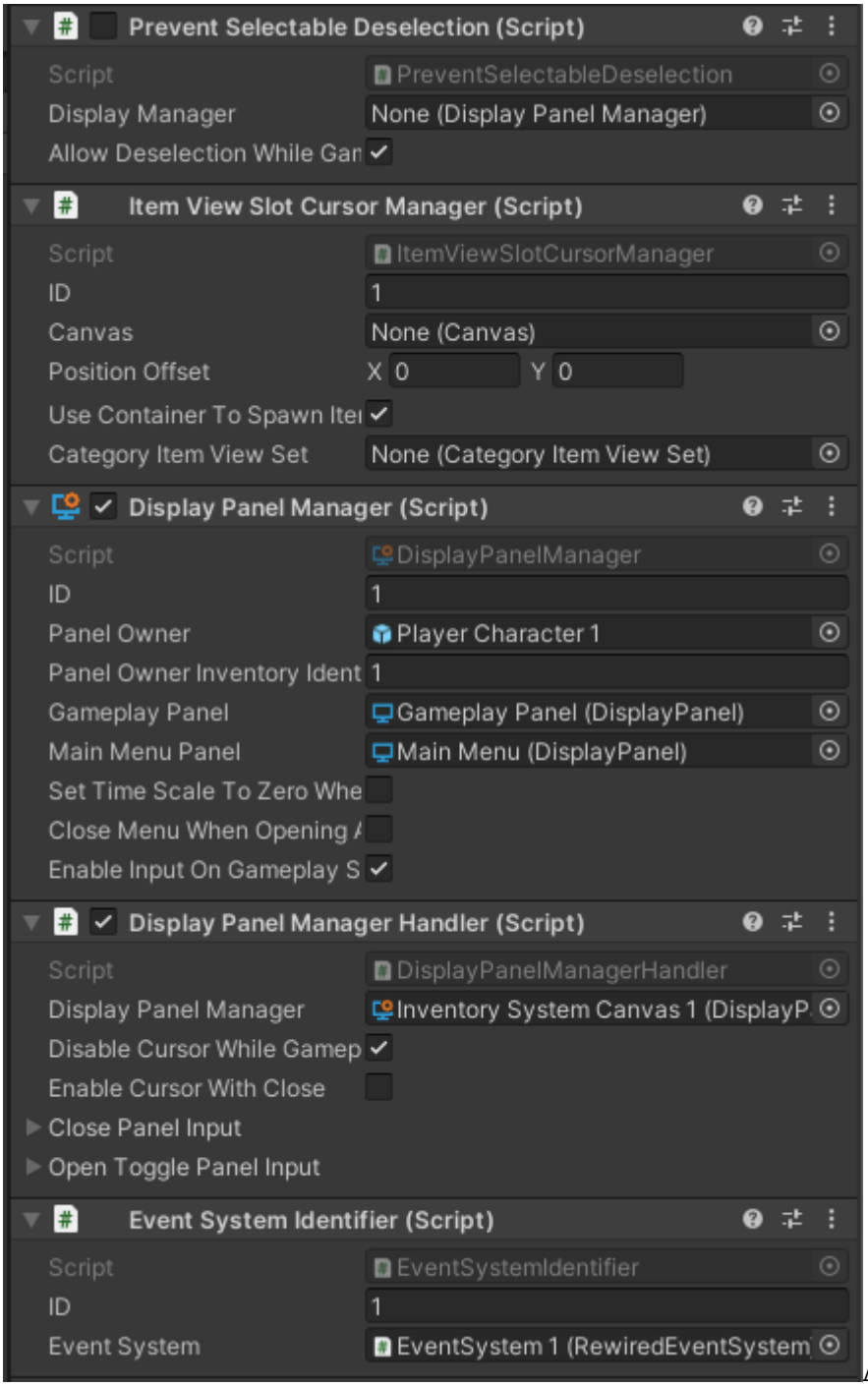
Rewired

1. On these Event System Game Objects add two components: Rewired Event System and Rewired Standalone Input Module. Both are part of Rewired.
2. Make sure to set the appropriate ID on each Rewired Standalone Input Module.
3. The “Always Update” field must be set to true on both Rewired Event



System

5. Setup the Canvas:
   Assign different IDs for each Inventory Canvas. The ID should be set to the same ID as the Inventory Identifier ID you plan to use. The Panel Owner should then be assigned by reference or automatically using the ID.



Add a Event System Identifier to your Canvas. It should reference the correct Event System. The Event System Identifier must be set on the Canvas, as the Selectable components within our system are selected using a Event System Manager. The Event System Manager is used to find the relevant Event System for the Selectable. As a result an Event System Manager should be added on the Game Game Object, with the other managers.

6. Additional steps for the Demo scene:
    1. If you are using the Demo scene make sure to manually fix the Inventory Monitor and HP Monitor references.
    2. If you are using a 3D model of Player Character in the UI make sure to create a new Texture and assign it in the UI character camera and the Raw Image Texture

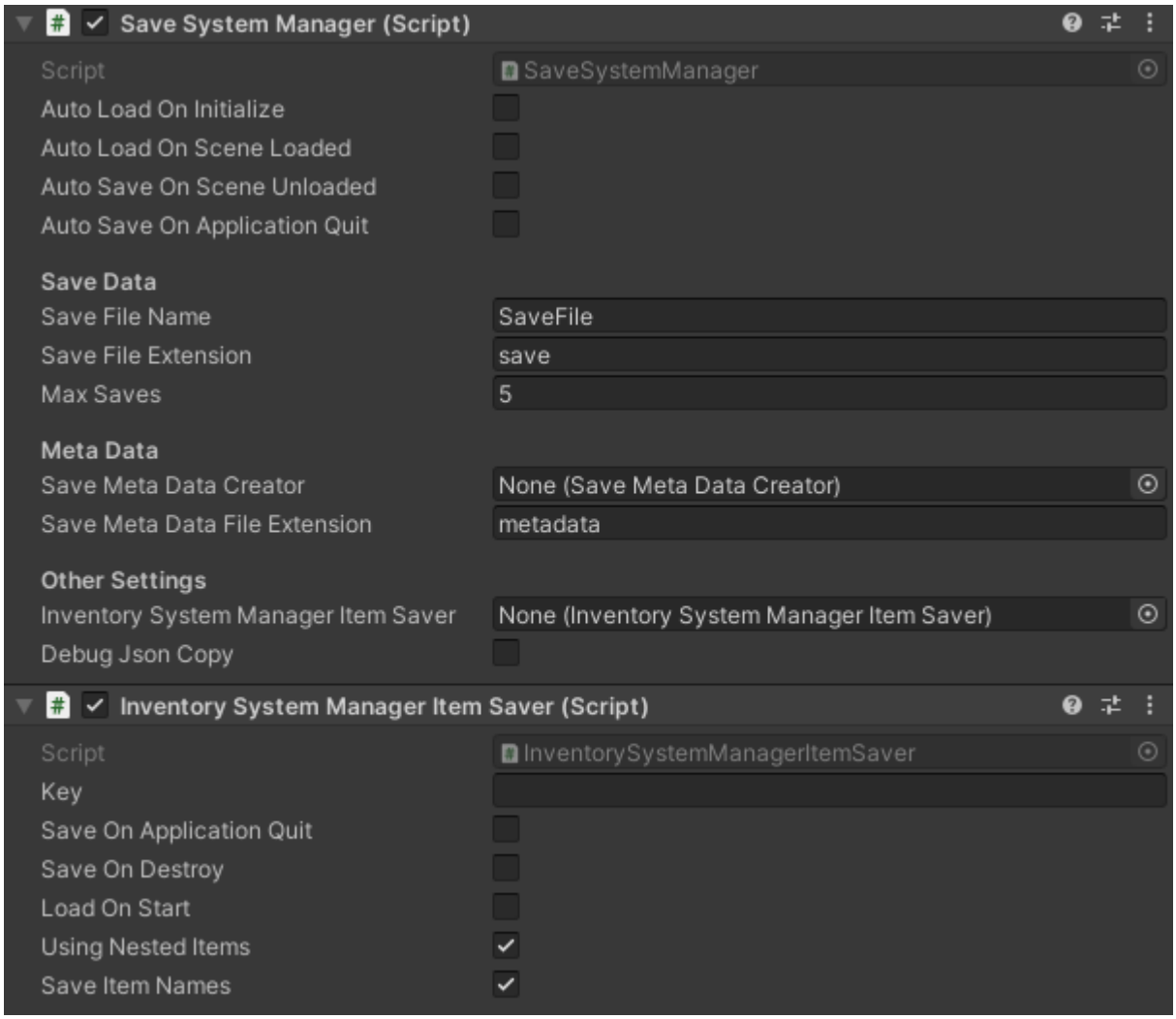
Save System

The save system can save the current state of the inventory, and other component, to disk. This state can then be restored by loading the file that it was saved to.

The Ultimate Inventory System Save System is meant to be either extended or used within an existing save system.

Save System Manager

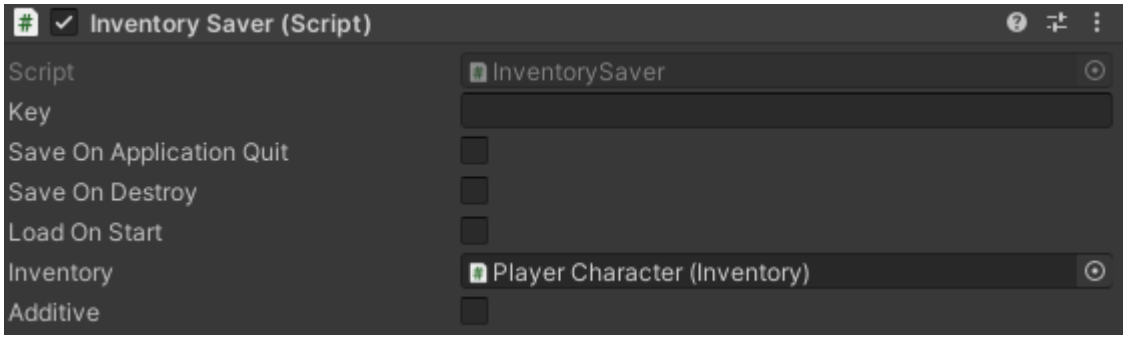
The Save System Manager is responsible for saving the data to the disk. Serialized data can be saved by registering it with the SaverBase component. When the manager is saving it will loop through the SaverBase components and retrieve the data that needs to be saved. It will then serialize that data to disk.



At the time of loading the Save System Manager will go through all registered Savers and load the data. In most cases if Saver component will register itself to the Save System Manager on it's Start function. Therefore the Savers must "Load On Start" if the Save System Manager Loads the Save Data before the Saver is registered.

Saver Components

Any components that contain data that should be saved will inherit the SaverBase class. The Inventory Saver component is an example which contains data that should be saved. The Inventory Saver will serialize and deserialize the items that are within of the Inventory. Your Inventory content will be saved/loaded by the Save System Manager if the Inventory Saver component is added to the same GameObject as the Inventory.



The Savers available out of the box are:

- **InventorySystemManagerItemSaver** : The main saver component. It is used to save the Item states (attribute values, names, etc…). All savers which save Items only save the item IDs. They notify this component to save the actual Item by serializing it once even if it is used in multiple places. Without this saver the "InventorySaver" and other savers which save Item IDs won't work (except for Immutable & Common Items)
- **InventorySaver** : Saves the content of an Inventory component by saving the Item IDs and amounts in each Item Collection.
- **CurrencyOwnerSaver** : Saves the content of a Currency Owner.
- **InventoryGridSaver**: Saves the positions of your Items within the Inventory grid.
- **GameObjectSaver**: A generic save component that can be used to save position, rotation, scale and active/inactive states of the components.
- **ItemShapeGridDataSaver**: Saves the Item Shape Grid Data to keep track the positions of the items within the shape grid.

Some Savers are specific to integrations, for example Ultimate Character Controller (UCC) is incompatible with the standard Inventory Saver:

- **InventoryBridgeSaver** : Saves the content of an Inventory which is bound to a UCC character.

Save Data

Save Data is the object that is saved to disk. This contains the serialized data of the inventory state as well as metadata such as the time that it was created. Since the save system is very extensible it may save anything which can be serialized by our serializer.

Save Meta Data & Save Meta Data Creator

To prevent all the saves files to be loaded in memory at all times to preview the information about the save within the UI (for example the time the file was saved), we split the save files in two. The Save Meta Data just includes some context about the save file you wish to actually load.

The Save Meta Data and Save Meta Data Creator can be extended to add custom information such as the last level loaded, time played, player progress, etc… The save meta data is accessible in the Save Menu UI to customize the Save Views.

Here is an example of a basic Save Meta Data and Save Meta Data Creator

```
/// <summary>
```

```
/// A basic save meta data creator.
/// </summary>
[CreateAssetMenu(fileName = "BasicSaveMetaDataCreator", menuName =
"Opsive/Save System/Save Meta Data Creator.")]
public class BasicSaveMetaDataCreator : SaveMetaDataCreator
{
    /// <summary>
    /// Create the save meta data.
    /// </summary>
    /// <param name="saveSystemManager">The save system
manager.</param>
    /// <param name="saveDataInfo">The save data info linked to that
meta data.</param>
    /// <returns>The save meta data.</returns>
    public override SaveMetaData CreateMetaData(SaveSystemManager
saveSystemManager, SaveDataInfo saveDataInfo)
    {
        return new BasicSaveMetaData(saveSystemManager, saveDataInfo);
    }
    /// <summary>
    /// Create an empty save meta data.
    /// </summary>
    /// <returns></returns>
    public override SaveMetaData CreateEmpty()
    {
        return new BasicSaveMetaData();
    }
}
/// <summary>
/// The save meta data which can be serialized.
/// </summary>
[Serializable]
public class BasicSaveMetaData : SaveMetaData
{
    [Tooltip("The date and time in ticks.")]
    [SerializeField] protected long m_DateTimeTicks;
    public long DateTimeTicks => m_DateTimeTicks;
    /// <summary>
    /// Default constructor.
    /// </summary>
    public BasicSaveMetaData() : base()
    {
        m_DateTimeTicks = new DateTime().Ticks;
    }
    /// <summary>
    /// Overloaded constructor.
    /// </summary>
    /// <param name="saveSystemManager">The save system
manager.</param>
```

```
    /// <param name="saveDataInfo">The save data info linked to that
meta data.</param>
    public BasicSaveMetaData(SaveSystemManager saveSystemManager,
SaveDataInfo saveDataInfo) : base(saveSystemManager, saveDataInfo)
    {
        m_DateTimeTicks = DateTime.Now.Ticks;
    }
    /// <summary>
    /// Set the date of the save data.
    /// </summary>
    /// <param name="newDateTime">The new date.</param>
    public void SetDateTime(DateTime newDateTime)
    {
        m_DateTimeTicks = newDateTime.Ticks;
    }
}
```

Save Data Info

The save data info is a struct with:

- The save slot index
- The save Meta Data
- The Save Data

Save Menu

The save menu uses the Save System Manager to display the saves available to load/save. Check the UI documentation to learn more about the Save Menu.

Save System Manager API

Save Load and delete save files

```
// Save in file 0
SaveSystemManager.Save(0);

// Load in file 0
SaveSystemManager.Load(0);

// Delete save at file 0
SaveSystemManager.DeleteSave(0);
```

Get the save data

```
// Get the current state of the cached save data
var saveData = SaveSystemManager.GetCurrentSaveDataInfo();

// Get the cached serialized data of a specific Saver component
var saveData = SaveSystemManager.TryGetSaveData("The Saver Key", out
```

```
var serializedData);
```

Nest our Save System in a Third Party Save System

In a lot of cases you may use a custom or third party save system for your game as the main save system. But you may still wish to use our save system to save and load items without needing to create additional save files.

In the example below we assume you have an interface that has a function for Saving called “RecordData” and a function for loading call “ApplyData”. The actual functions, parameters and return types will depend on your save system implementation.

Here is an example on how this can be achived:

```
public class NestedInventorySaver : ISave
{
    public int saveSlot = 0;
    public object RecordData()
    {
        // Save the game without saving to disk by specifying the
"false" parameter
        SaveSystemManager.Save(saveSlot, false);
        var saveDataInfo = SaveSystemManager.GetCurrentSaveDataInfo();
        var saveData = saveDataInfo.Data;
        // Serialize the save Data with a custom serializer perhaps?
        return saveData;
    }
    public void ApplyData(object data)
    {
        if (data == null) return;
        // Deserialize the save data with a custom serializer perhaps?
        var saveData = data as SaveData;
        // Load the save data in a specific slot, by specifying the
save data we don't need to read from disk.
        SaveSystemManager.Load(saveSlot, saveData);
    }
}
```

Custom Saver Component

To create a custom Saver component inherit the SaverBase class and override the Serialize and Deserialize methods.

```
/// <summary>
/// An example of a Saver component.
/// </summary>
public class ExampleSaver : SaverBase
{
    /// <summary>
    /// The examples save data.
```

```
    /// </summary>
    [System.Serializable]
    public struct ExampleSaveData
    {
        // Add you serializable data here.
        // To save items or currency use IDAmountSaveData.
        // public IDAmountSaveData[] ExampleAmounts;
    }
    /// <summary>
    /// Serialize the save data.
    /// </summary>
    /// <returns>The serialized data.</returns>
    public override Serialization SerializeSaveData()
    {
        // Get the item you want to save
        // var saveAmounts = GetComponent<Example>().Amounts;
        var saveData = new ExampleSaveData() {
            //Set the save Data Value
            //ExampleAmounts = saveAmounts
        };
        // Serialize the save data using our custom Serialization.
        return Serialization.Serialize(saveData);
    }
    /// <summary>
    /// Deserialize and load the save data.
    /// </summary>
    /// <param name="serializedSaveData">The serialized save
data.</param>
    public override void DeserializeAndLoadSaveData(Serialization
serializedSaveData)
    {
        // Get the save data, Make sure the type matches the type
saved
        var savedData =
serializedSaveData.DeserializeFields(MemberVisibility.All) as
ExampleSaveData?;
        if (savedData.HasValue == false) {
            return;
        }
        var exampleSaveData = savedData.Value;
        //Apply the loaded save data on the object
        // GetComponent<Example>().Amounts =
exampleSaveData.ExampleAmounts;
    }
}
```

Interaction System

The inventory system requires a lot of interactions to pickup items, open chests, open a menu, etc. An interaction system has been created in order to handle these type of interactions.

The interaction system uses the interfaces, IInteractor and IInteractable. An interactable can be selected, deselected, and interacted with. The interactor can add and remove interactables.

For convenience the Inventory Interactor component can be used as the IInteractor and the Interactable component can be used as the IInteractable.

Inventory Interactor

The Inventory Interactor should be a neighbor of the Inventory.
If “Auto Interact” is false the Inventory Interactor will expect an input to trigger the interaction. For convinience use the (Standard) Inventory Input component to set the input.

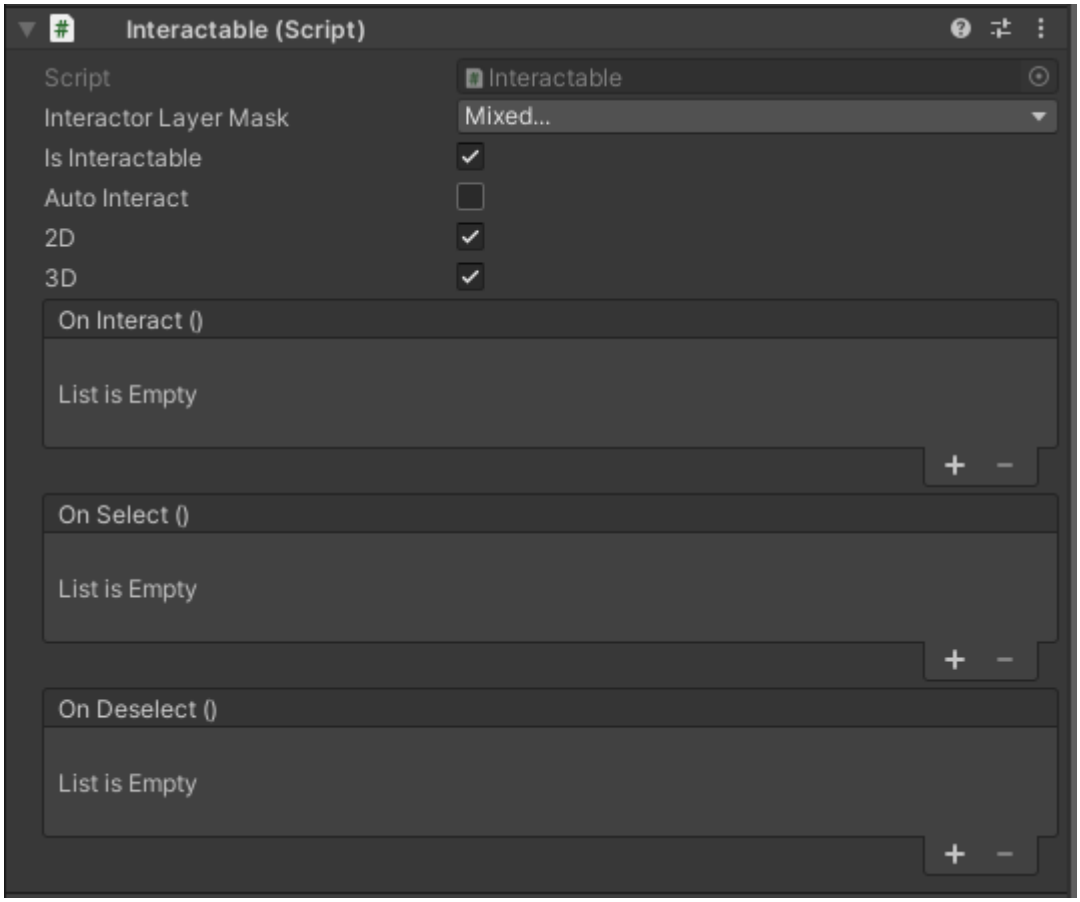
Important: The Interactor requires a Collider to detect the Interactable.

Interactable

Add the Interactable component on any component the Interactor should interact with.
Make sure the Interactor Layer Mask includes the Interactor game object Layer.
Examples of use cases for interactables are : Item Pickup (or any Pickup component), Menu Openers Interactable Behviour (Used to open menus with the Inventory Interactor)

Set “Auto Interact” to true to interact on trigger instead of waiting for the Interactor to interact with the Interactable.

Important: The Interactable component requires a Trigger Collider to detect the Interactor.



Interactable Behavior

Interactable Behaviors are components which sit next to the Interactable component and listen to the events on Select, Deselect and interact.
Examples of Interactable Behaviors are: PickupBase, MenuInteractableBehavior, etc…

Create your own Interactable Behavior

```
/// <summary>
/// Example Interactable Behavior.
/// </summary>
public class ExampleInteractableBehavior : InteractableBehavior
{
    /// <summary>
    /// can the interactor interact with this component.
    /// </summary>
    /// <param name="interactor">The interactor.</param>
    /// <returns>Returns true if it can interact.</returns>
    public override bool CanInteract(IInteractor interactor)
    {
        return base.CanInteract(interactor);
    }
    /// <summary>
    /// The event called when the interactable is selected by an
interactor.
```

```
    /// </summary>
    /// <param name="interactor">The interactor.</param>
    public override void OnSelect(IInteractor interactor)
    {
        base.OnSelect(interactor);
        // Do something.
    }
    /// <summary>
    /// The event when the interactable is no longer selected.
    /// </summary>
    /// <param name="interactor">The interactor.</param>
    public override void OnDeselect(IInteractor interactor)
    {
        base.OnDeselect(interactor);
        // Do something.
    }
    /// <summary>
    /// On Interaction.
    /// </summary>
    /// <param name="interactor">The interactor.</param>
    protected override void OnInteractInternal(IInteractor interactor)
    {
        // If you require the interactor to be an inventory
interactor.
        if (!(interactor is IInteractorWithInventory
interactorWithInventory)) { return; }
        var inventory = interactorWithInventory.Inventory;
        // Do something.
    }
}
```

User Interface (UI)

Games can differentiate one another by having a unique user interface (UI). Therefore when building the inventory UI system we took the approach to make it easy to customize the UI. The result is a modular system that relies heavily on components, prefabs and Scriptable Objects.

But with so many modular components is often easy to lose track of the all the possible features and how to hook them up. That's why we created the UI Designer. An editor window specifically made to create, find and edit UI components for your Inventory related UI. We recommend using the the UI Designer to get started quickly to set up a UI that is close to what you envision.

The Inventory System is completely independent from the UI therefore it is not required to use it, and thanks to its modular approach you may pick and choose the components you are interested about.

The components which are used extensively in the UI system are

- Item View
- Item View Slot
- Item View Slots Container
- Display Panel

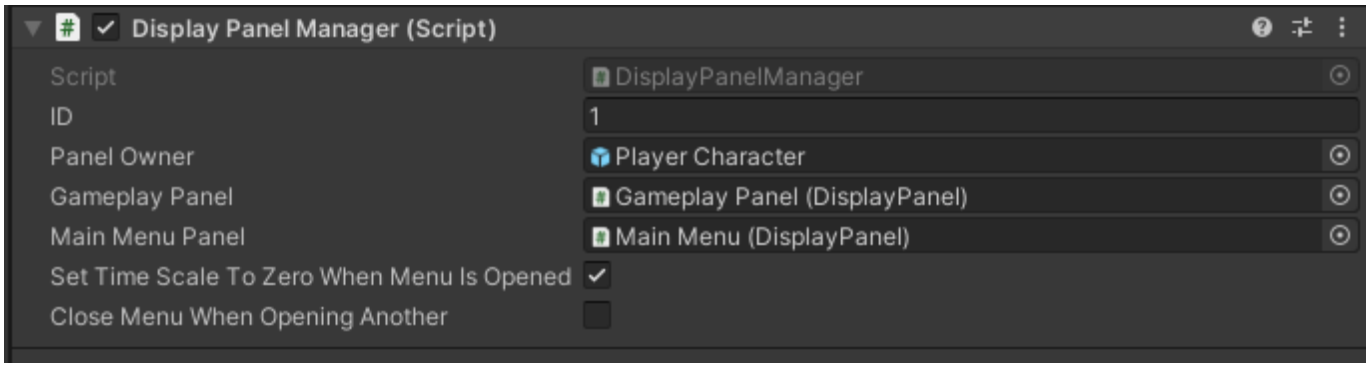
Check out the Getting Started page to familiarize yourself with the UI and how to quickly customize it to make it your own.

Display Panel & Manager

Panels are used to be opened and closed. Each time a panel is open it can be given a reference to the panel that was previously opened such the previous panel may be selected once the current panel is closed. This allows total control on the order panels are opened and how it rolls back when closing panels. The display manager keeps track of all the panels and more.

Important: If your game already has a UI system, it is not required to use the Display Panel Manager and Display Panel. It may require some coding but the Inventory Grid, Item Hotbar, Item View, etc… can be used without Display Panels. If assistance is required checkout the forum.

Display Panel Manager



The purpose of the Display Panel Manager is to keep track of all the panels in it's children. In the case of menus we restrict to being able to only open one menu at a time. You may chose to prevent opening a menu if one is already opened, or close the current menu when opening the new one. You may also chose to set the time scale to 0, otherwise it is recommend to listen to the "GameObject_OnPanelOpenClose_OpenClosePanelInfo" event to disable input for example.

The Display Panel Manager can be selected by ID from the Inventory System Manager.

The Panel Owner is very important. It is a GameObject that will receive events about panels closing, opening and more.
It is highly recommend (in some cases even necessary) to use the character game object (The game object with the Inventory, Item User, Inventory Input, etc… components) as Panel Owner.

Tip: Multiple display panel managers can be created for a split screen setup.

```
// Get the Display Panel Manager by ID from the Inventory System
Manager singleton.
var displayPanelManager =
InventorySystemManager.GetDisplayPanelManager(ID);

// Get the Panel Owner. It is a GameObject, usually the character
GameObject with the Inventory component.
// The panel owner also receives events about panels opening/closing
and more.
var panelOwner = displayPanelManager.PanelOwner;

// Get a panel by the panel name.
var panel = displayPanelManager.GetPanel(panelName);

// Open a panel from the panel managers, this uses the current
selected panel as the previous panel.
displayPanelManager.OpenPanel(panel);

// Get the selected panel.
var selectedPanel = displayPanelManager.SelectedDisplayPanel;

// Get the selected menu.
var selectedMenu = displayPanelManager.SelectedDisplayMenu;

// Close the selected panel.
displayPanelManager.CloseSelectedPanel();

// Register to the event for opening/closing panels, (All events can
be found in EventNames.cs)
EventHandler.RegisterEvent<OpenClosePanelInfo>(m_PanelOwner,
EventNames.c_GameObject_OnPanelOpenClose_OpenClosePanelInfo,
HandlePanelOpenedOrClosed);
```

A Selectable object must always be in focus in order to have proper keyboard or controller support. To assist with this the Prevent Selectable Deselection component will always select the previous selected Selectable if there is no selection.

Display Panel

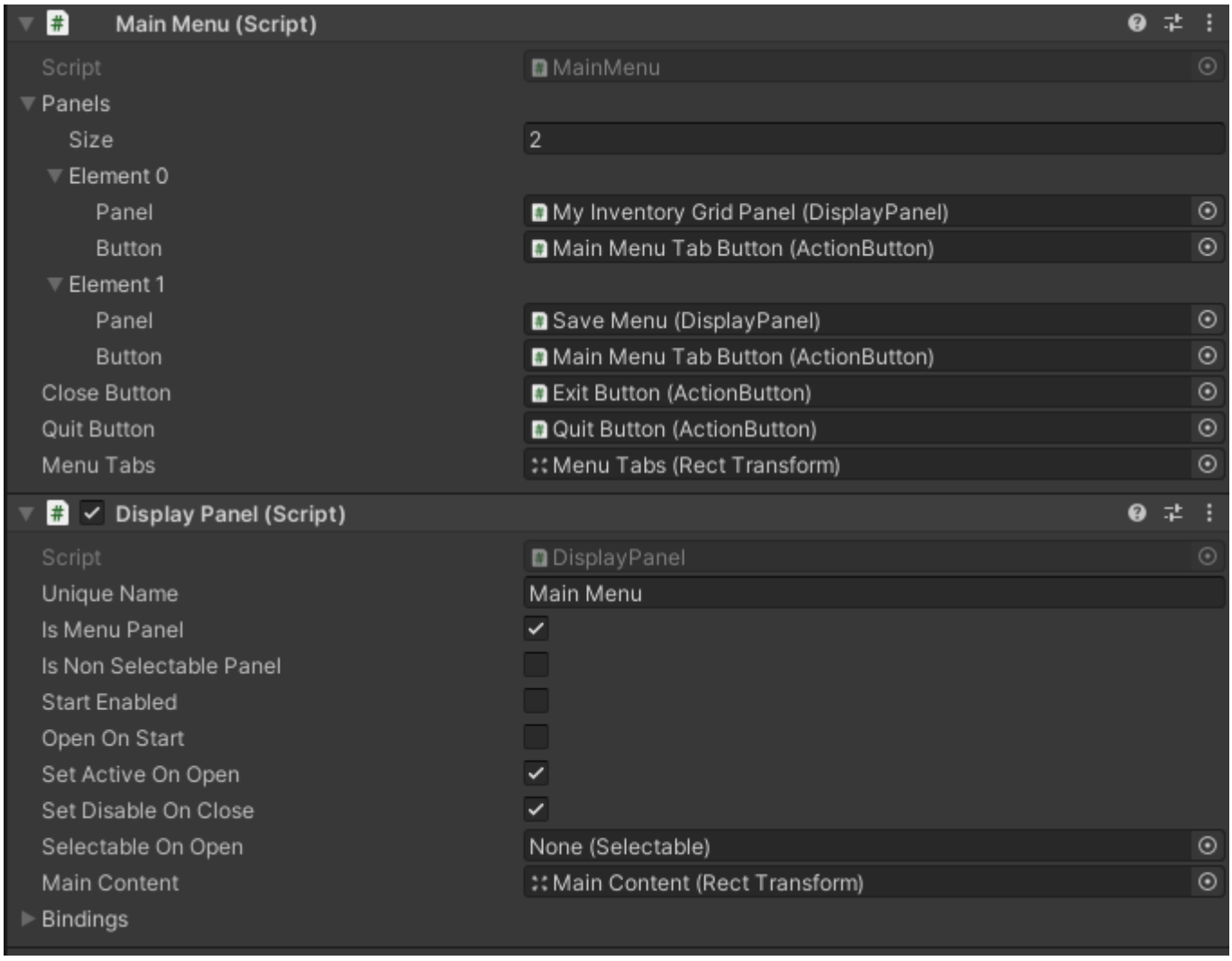
Display Panels are used to open and close UI. There are a few options which allow to customize the functionality of the panel:

- *Unique Name*: This allows the Display Panel Manager to create a dictionary matching names to panels, allowing for easy access from anywhere in the code.
- *Is Menu Panel*: If true the panel will act as a menu, which means the Display Panel Manager will only allow one to be open at any one time.
- *Is Non Selectable*: This prevents the Display Panel Manger to register the panel as selected. It is particularly useful for tooltips as you may wish to open & close it without changing the chain of panels.

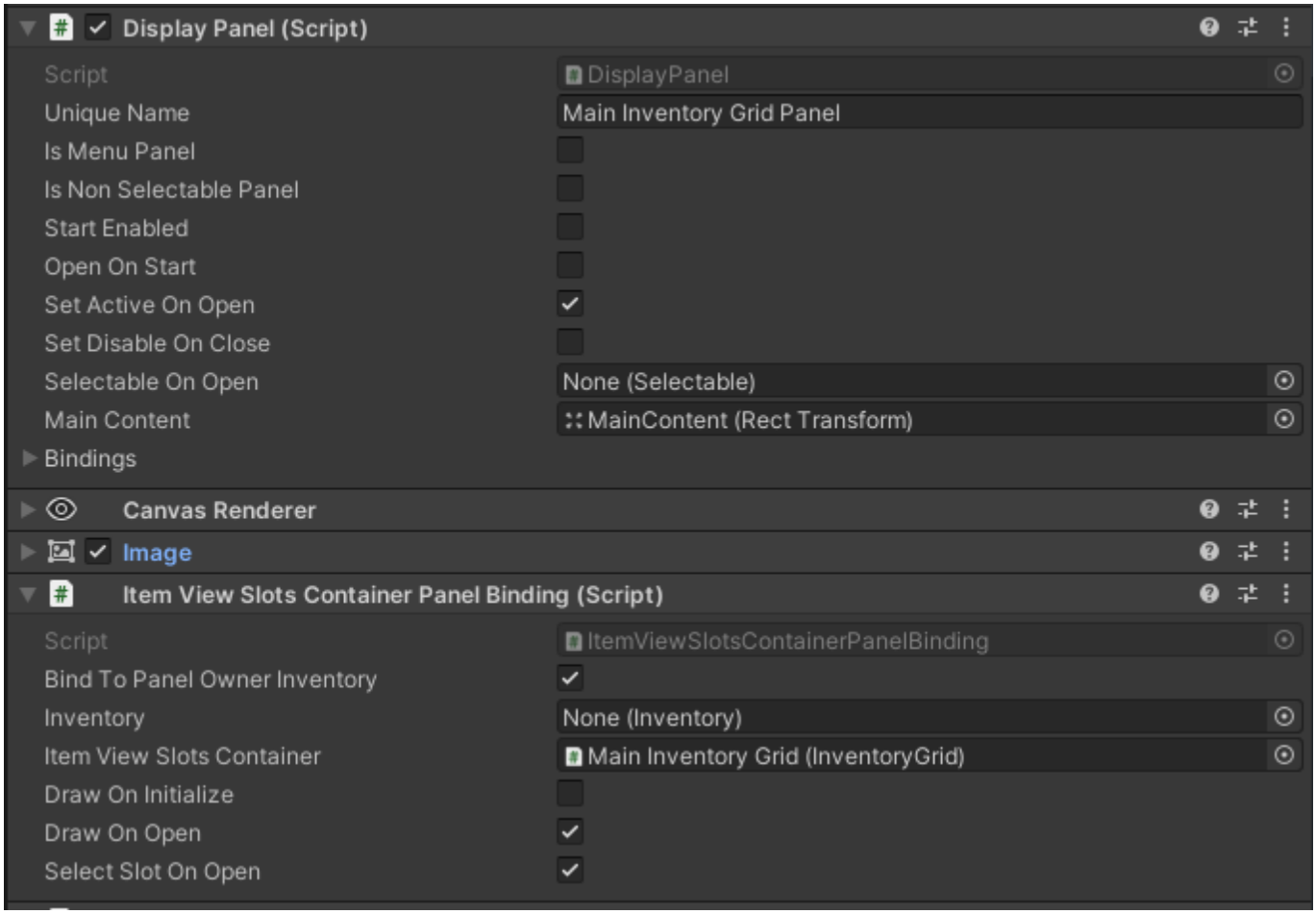
- *Start Enabled*: By default all panels start disabled, you may choose to start enabled.
- *Open on Start*: Similar to start enabled, except it will send a OnOpen event right at the start.
- *Set Active On Open*: Set the game object active on open.
- *Set Disable On Close*: Sets the game object inactive on close.
- *Selectable On Open*: Choose a selectable which will be selected once the panel is opened.
- *Main Content*: This is used extensively by UI Designer to know where prefabs should be spawned.
- *Bindings*: Bindings do not have to be set manually they are found on awake if they are set on the same game object. Panel Bindings get events for Open, Close, etc.

The interesting part with panels are the panel bindings, for example all Menus (Main Menu, Shop Menu, etc…) are all Panel Bindings. All they need to know is when to initialize, Open and Close. The most common examples of Display Panels & Binding set ups are:

Display Panel Menus, Example Main Menu



Item View Slots Container Panel Binding (For an Inventory Grid, Item Hotbar, Equipment Panel, etc.)



Tip: For Item View Slots Container Panel Binding you may choose to Bind To Panel Owner Inventory, this is very useful as you may set up the inventory to use in a single place.

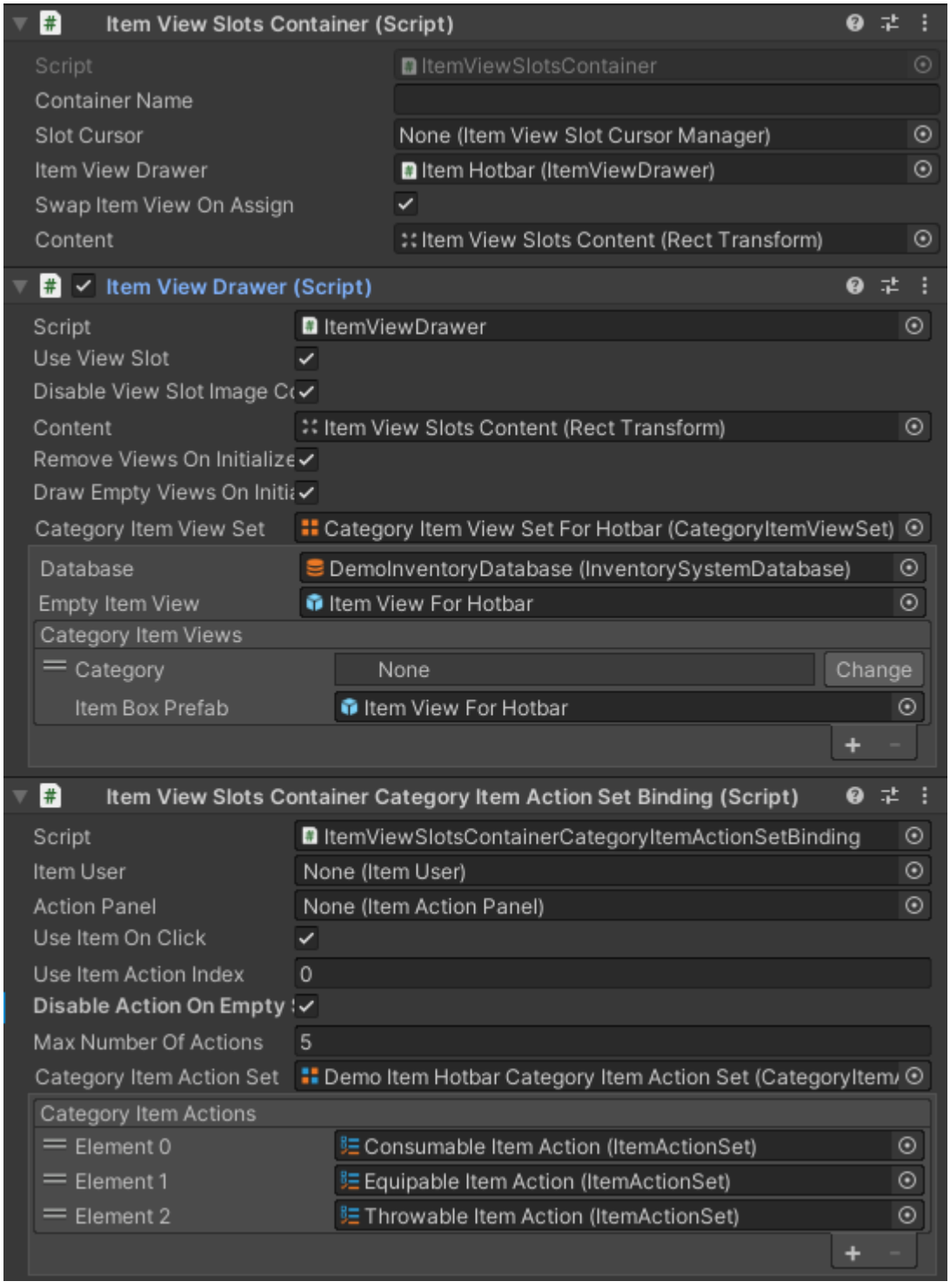
```
// Open the panel using the Panel Manager which uses the currently
selected panel.
panel.SmartOpen();
panel.SmartClose();

// Get references to the objects and the state of the panel.
DisplayPanel previousPanel = panel.PreviousePanel;
Selectable previousSelectable = panel.PreviousSelectable;
bool isOpen = panel.IsOpen;
var panelManager = panel.Manager;
string panelName = panel.UniqueName;
List<DisplayPanelBinding> panelBindings = panel.Bindings;

// Open the panel specifying the previous panel and selectable.
panel.Open(previousPanel, previousSelectable)

// Close the panel where the parameter selectPrevious is a bool which
true will open the previous panel once this one is closed.
panel.Close( true )
```

Item View Slots Container

The base class “Item View Slots Container Base” is used by the Inventory Grid, Item Hotbar, Item Slot Collection View and more.

Some options in the UI Designer are common between Item View Slots Containers and can be found on this page.

The grid allows for a common way to interact with Item View Slots: select, click, move, add, remove, and exchange. It can be bound to an Inventory (inventory Grid, Item Hotbar) or an Item Collection (Item Slot Collection View) but that is not necessarily the case.



The Item View Slots Contain contains the following properties:

- *Container Name*: Is used to differentiate the Item View Slots Container. it's especially useful for Item View slot Drop Action Conditions.
- *Slot Cursor*: A reference to the Item View Slot Cursor Manager (optional).
- *Item View Drawer*: Used to swap out the Item Views depending on the Item set in an Item View Slot. Note that if specified the Item View Slots Container "Content" and the Item View Drawer "Content" must specify the same transform.
- *Swap Item View On Assign*: If true the Item View will change when a new item is set in an Item View Slot
- *Content*: Points to the transform which has all the Item View Slots as children.

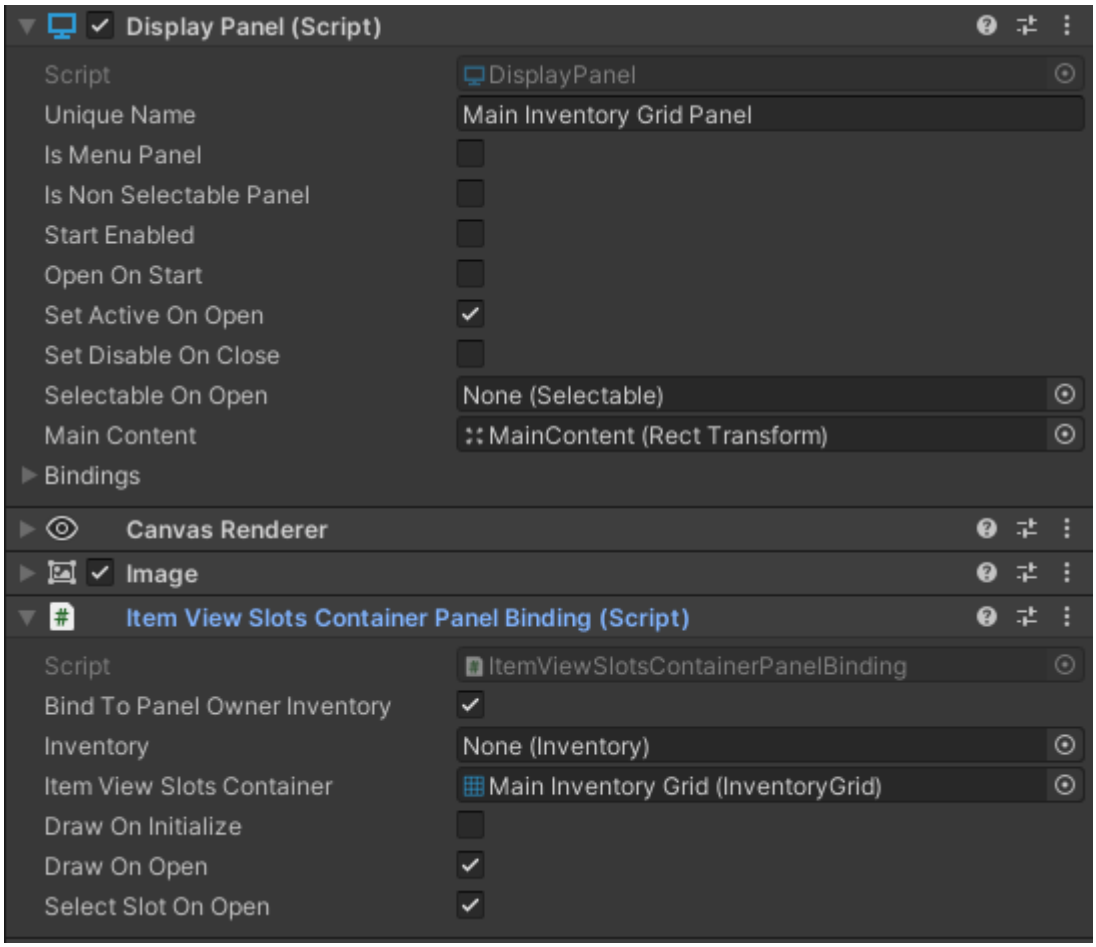
Useful API methods include:

```
// Listen to events on the Item View Slot Container (there are more
than the ones below).
m_ItemViewSlotContainer.OnItemViewSlotSelected +=
HandleItemViewSlotSelected;
private void HandleItemViewSlotSelected(ItemViewSlotEventData
slotEventData){
        /*An Item Was Selected*/
        // You may get the item view slot, which contains the item
info.
        var itemInfo = eventdata.ItemViewSlot.ItemInfo;
}

m_ItemViewSlotContainer.OnItemViewSlotClicked +=
HandleItemViewSlotClicked;
private void HandleItemViewSlotClicked(ItemViewSlotEventData
slotEventData){ /*An Item was clicked*/ }

m_ItemViewSlotContainer.OnItemViewSlotEndDragE +=
HandleItemViewSlotEndDrag;
private void HandleItemViewSlotEndDrag(ItemViewSlotPointerEventData
slotPointerEventData){ /*An Item Stopped Being Dragged*/ }

// Get the list of Item View Slots.
var itemViewSlots = m_ItemViewSlotContainer.ItemViewSlots;

// Add, Remove or Move items to specific slots.
m_ItemViewSlotContainer.AddItem(itemInfo, slotIndex);
m_ItemViewSlotContainer.RemoveItem(itemInfo, slotIndex);
m_ItemViewSlotContainer.MoveItem(sourceIndex, destinationIndex);

// Get the item in a slot.
var itemViewSlot = m_ItemViewSlotContainer.GetItemViewSlot(slotIndex);
var itemView = m_ItemViewSlotContainer.GetItemView(slotIndex);
var itemInfo = m_ItemViewSlotContainer.GetItemAt(slotIndex);

// Select and Get the selected slot.
m_ItemViewSlotContainer.SelectSlot(slotIndex);
```

```
var itemViewSlot = m_ItemViewSlotContainer.GetSelectedSlot();

// Refresh the container by drawing.
m_ItemViewSlotContainer.Draw();

// Set an Item View Slot Action Event that will happen once only.
Useful when moving an item to another slot without the mouse without
triggering Item Actions for example.
m_ItemViewSlotContainer.SetOneTimeClickAction(itemViewSlotActionEvent)
;
```

Item View Slots Container Panel Binding

Most of the times the Item View Slots Container will be bound to a panel such that it can be initialized on setup and refreshed whenever the panel opens. It is also a good place to set the Inventory that should be bound to the Item View Slots Container (if it needs one).



Item View Slots Container Bindings

The Item View Slots Containers are often accompanied by other useful Components called Item View Slots Container Bindings. These Include:

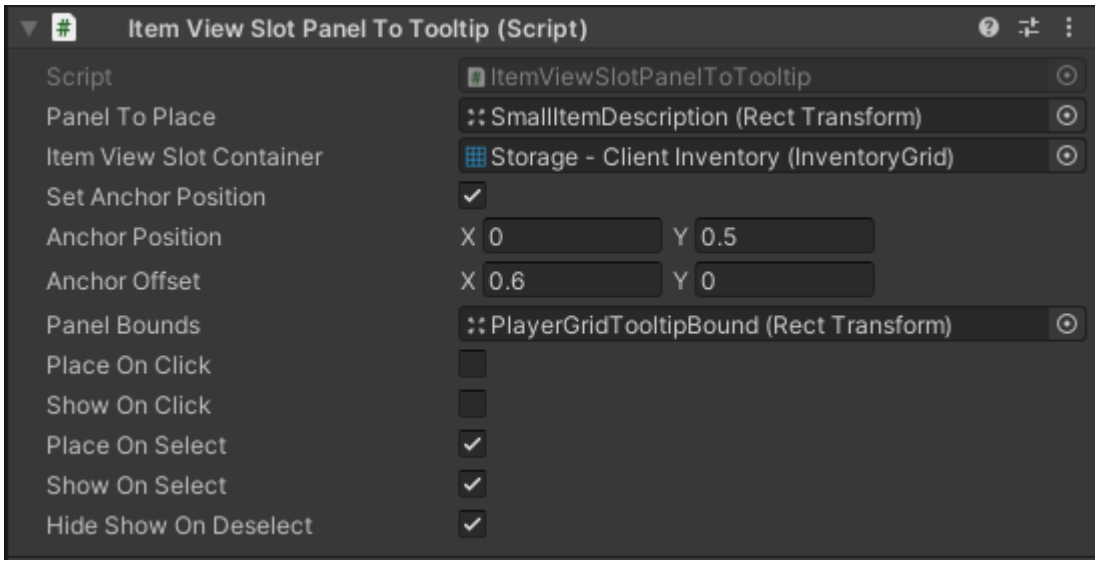
- *Item View Slot Move Cursor*: a component used for moving items from a slot to another, using the Unity UI Event System instead of drag & drop.
- *Item View Slots Container Item Action Binding*: A component used to bind Item Actions and use the selected item on click or through script.
- *Item View Slots Container Category Item Action Set Binding*: A component used to

bind Item Actions and use the selected item on click or through script.
- *Item View Slots Container Description Binding*: Show the Description of an Item on select or on click.

All the Item View Slots Container Bindings can be added with ease using the UI Designer

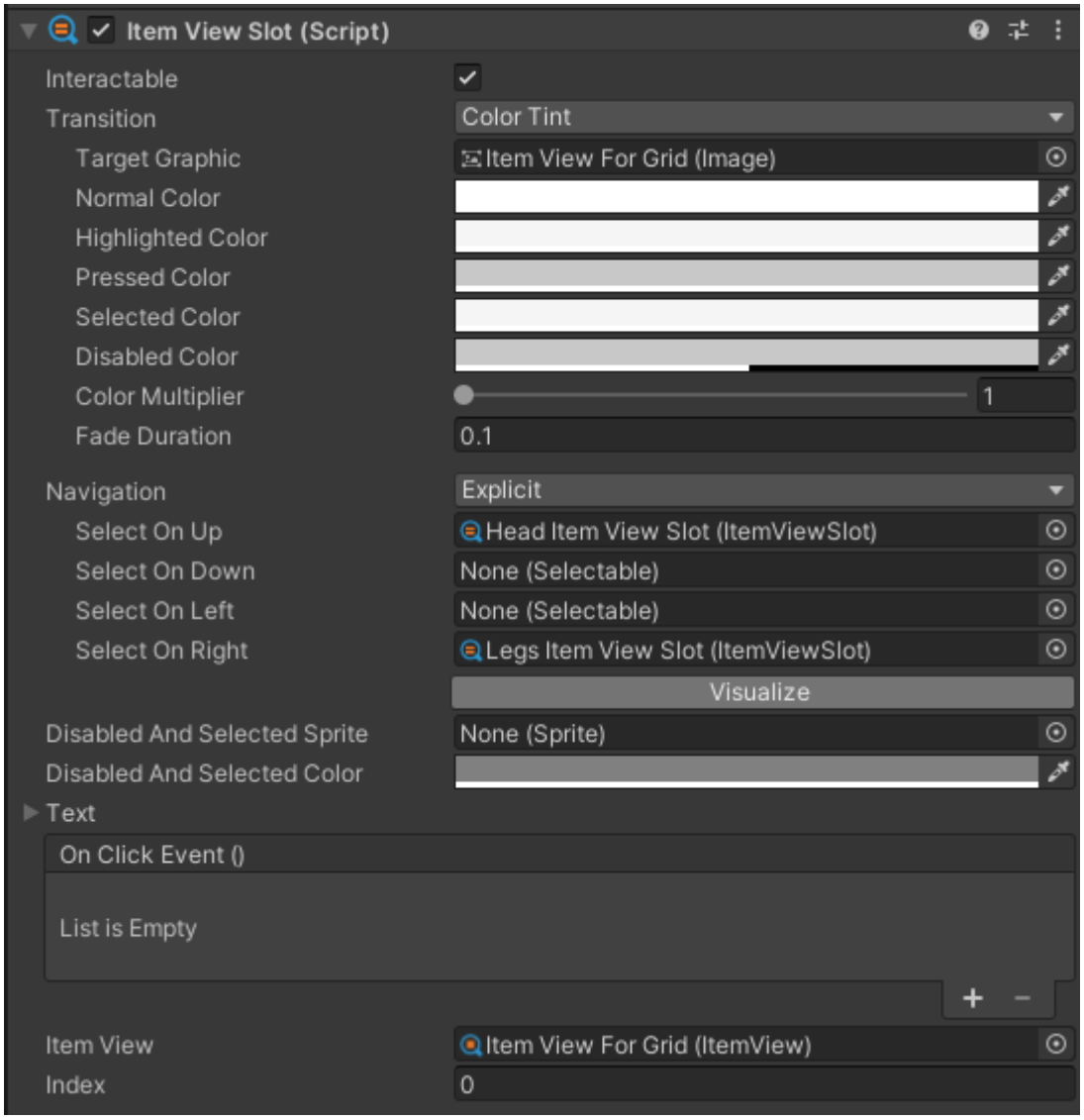
Item View Slot Panel To Tooltip

Converts any Rect Transform into a tooltip when selecting or clicking an Item View Slot within an Item View Slot Container.



Item View Slot

The Item View Slot is an Action Button which is used to detect clicks, selection, or drag & drop for an Item View. It is usually used as the parent of an Item View.



The Item View Slot is used extensively in the Item View Slot Containers (Inventory Grid, Item Slot Collection View, Item Hotbar). Useful API methods include:

```
// Set the Item Info in the slot.
m_ItemViewSlot.SetItemInfo(itemInfo);

// Set the Item View.
m_ItemViewSlot.SetItemView(itemView);

// Check if the Item Slot can contain an Item Slot (this uses Item
View Slot Restrictions).
bool canContain = m_ItemViewSlot.CanContain(itemInfo);
```

Item View Slot Restrictions

Item View Slot Restriction components, such as “Item View Slot Category Restriction” can be used to limit the type of item info the Item View Slot can contain. Restrictions can be added with the Item View Slot Restriction component next to the Item View Slot. To create a custom Item View Slot Restriction simply inherit the base class and override the “Can Contain” function.

Inventory Grid

Grid UI and Grid Event System

When the Unity Scroll View loads content it loads an element for every list item without pooling or reusing elements. This is inefficient, so instead the inventory UI uses a grid where the number of buttons is fixed. When the items are displayed they are mapped to an index which is then compared to the index of a View. By moving the Index of the objects to draw up and down we can achieve an efficient scrolling.

The Grid Event System is used to detect any event from the buttons. The grid uses an Action Button class which allows for more event notifications than the standard Unity Button provides. The Grid Event System will invoke events when a button is selected, deselected, clicked, etc. It also invokes an event when the Unity Event System tries to move outside the grid. The Grid UI uses those events to scroll or even change tab.

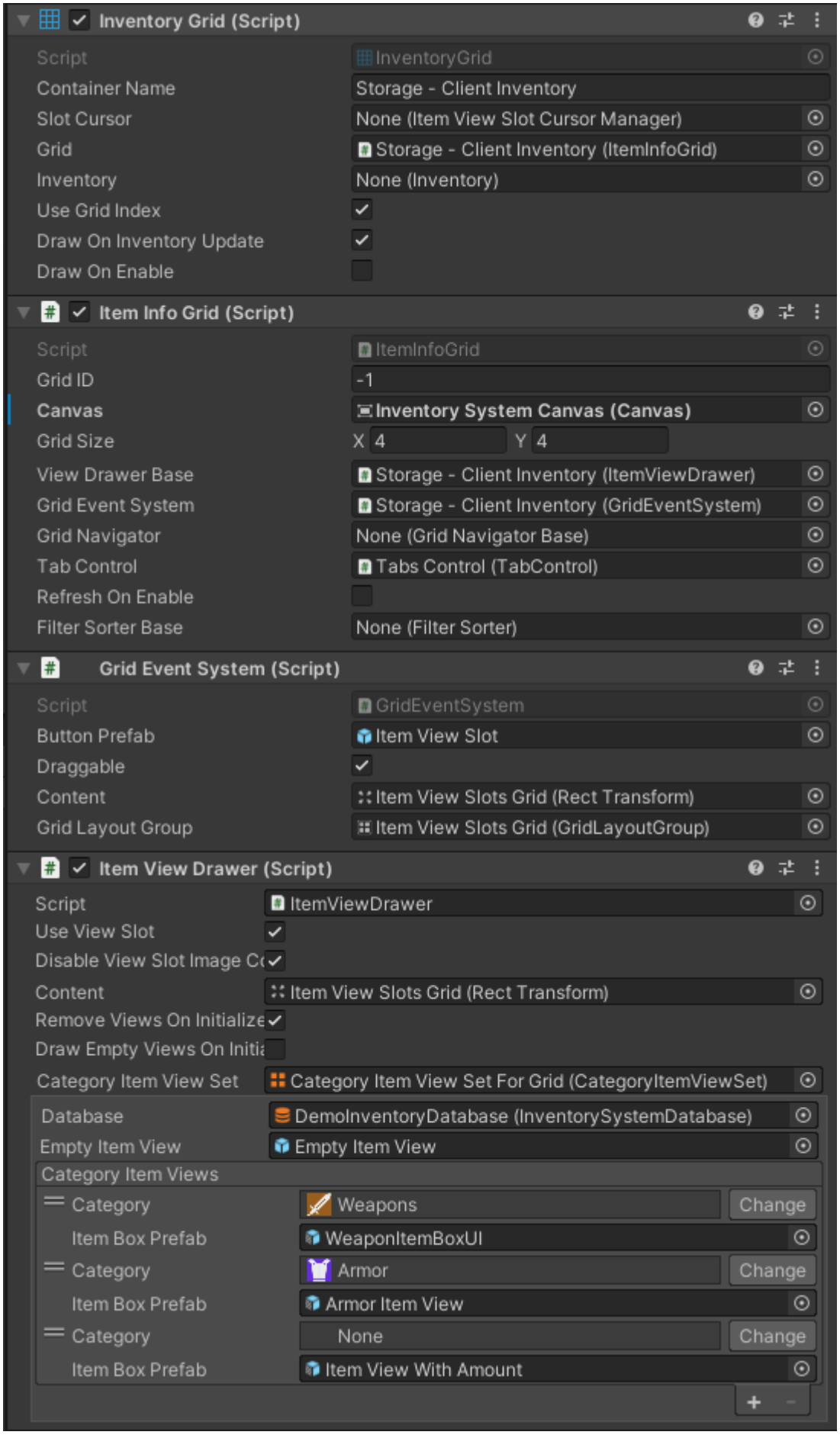
Inventory Grid

The Inventory Grid is an Item View Slots Container, which uses the power of the Grid system for tabs, navigation and more. This is a perfect fit for displaying the content of an Inventory whether it is the Inventory of the player , shop, storage, etc…

An Inventory Grid can be setup and edited using the UI Designer Inventory Grid tab.

As an Item View Slots Container the functionalities such as Item Description, Item Action and more are easily accessible for the Inventory Grid.
To learn more about Item View Slots Containers see this page.

The Inventory Grid requires a few components to work: Item Info Grid, Grid Event System and Item View Drawer. These are the essential components but more are available to customize functionality of the Grid.



- *Container Name*: Used to differentiate Item View Slots Containers from one another.
- *Inventory*: The Inventory can be set directly in the inspector, or it can be set by code or using an Item View Slots Container Panel Binding
- *Use Grid Index*: Use the Grid Index such that the items remember where they were

placed when moved, allowing empty spaces in the grid.

Useful API methods include:

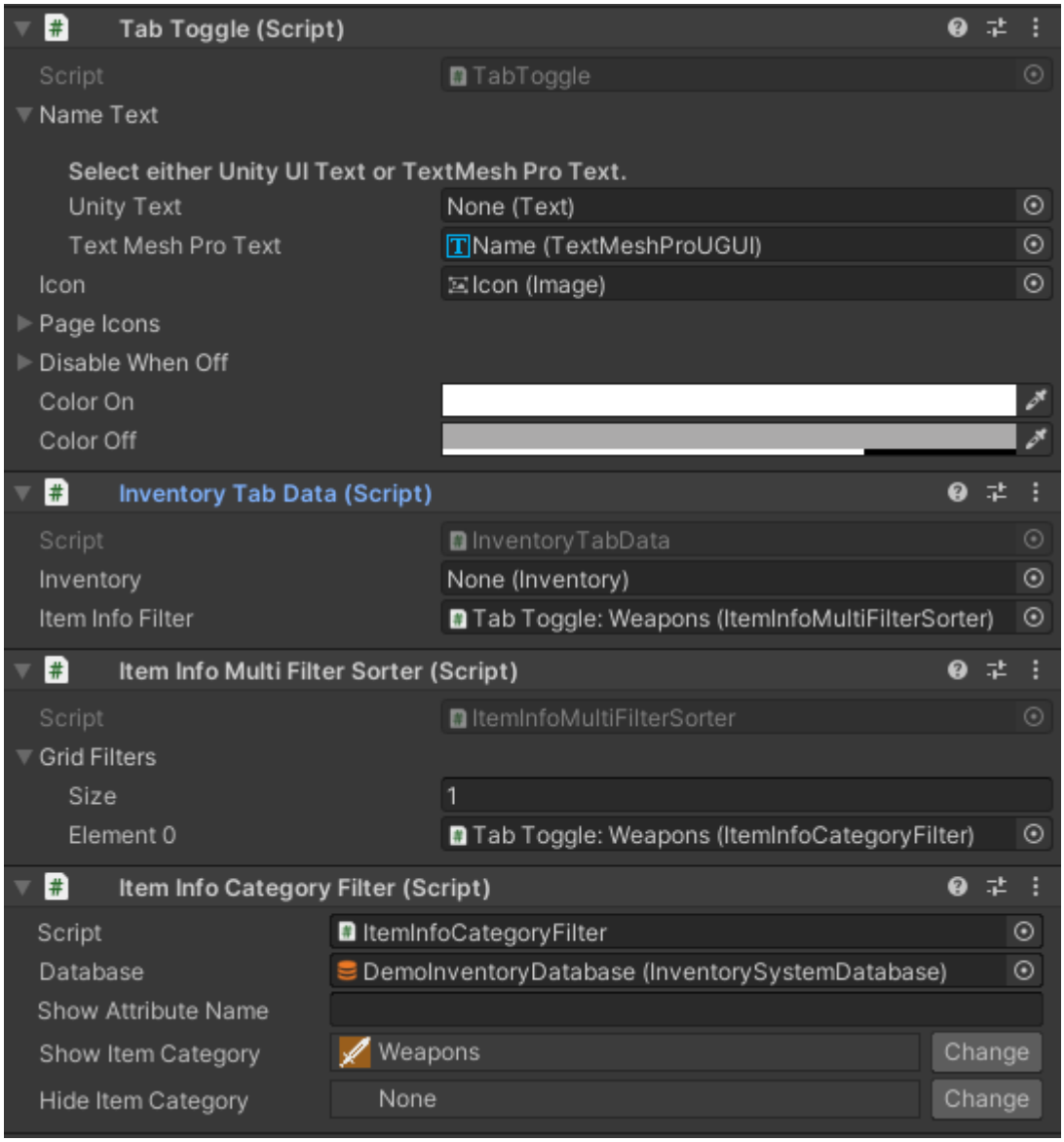
```
// Get the Grid ID used to differentiate grids.
m_InventoryGrid.GridID

// Set the Inventory manually.
m_InventoryGrid.SetInventory(inventory);

// Sort the items in the inventory grid.
m_InventoryGrid.SortItemIndexes(itemInfoComparer);

// Bind a filter/sorter to the inventory grid (this can be done
directly on the Item Info Grid).
m_InventoryGrid.BindGridFilterSorter(itemInfoSorterFilter);
```

Grid Navigator

Grid Navigators are used to navigate a Grid. For example, scrolling, paging, or scrolling with a scroll view. Use the UI Designer to set this up in no time.

Inventory Grid Tab Control Binding

The Inventory Grid Tab Control Binding, binds a Tab Control to the Inventory Grid. It watches the Tab Toggles being turned on for an Inventory Tab Data component. The Inventory Tab Data component can have filters/sorters and more which will change what items the Inventory Grid shows



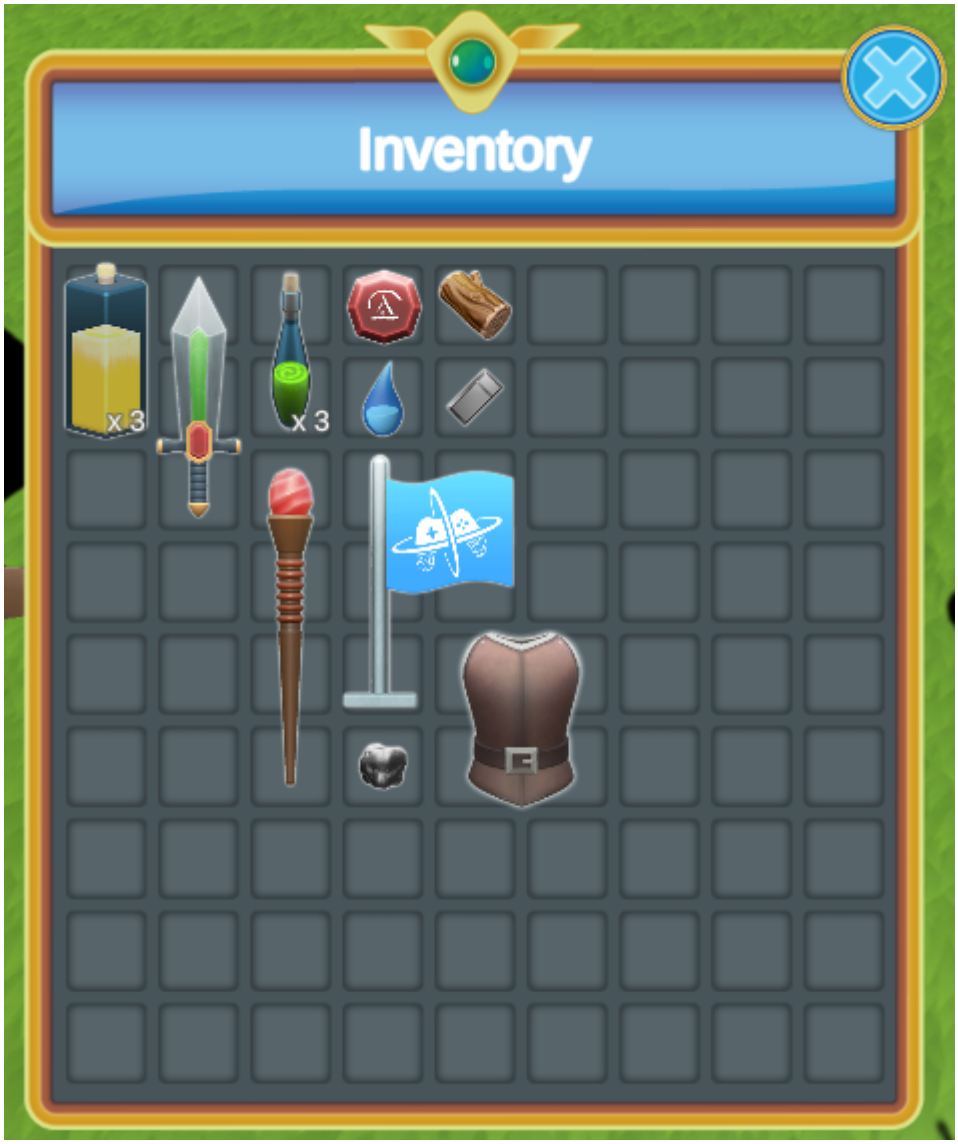
Item Info Filters/Sorters

Adding and removing Filters for the Inventory Grid is easy with the UI Designer. The list of available filter, even custom ones, will appear in the drop down when creating a new filter/sorter. Learn more about filter & sorters on this page.

Item Shape Inventory Grid

IMPORTANT NOTICE: The Item Shape Grid may receive refactor changes in the near future due to design decisions which may have limited it too much, we are sorry for the inconvenience, we are working on improving this feature and others.

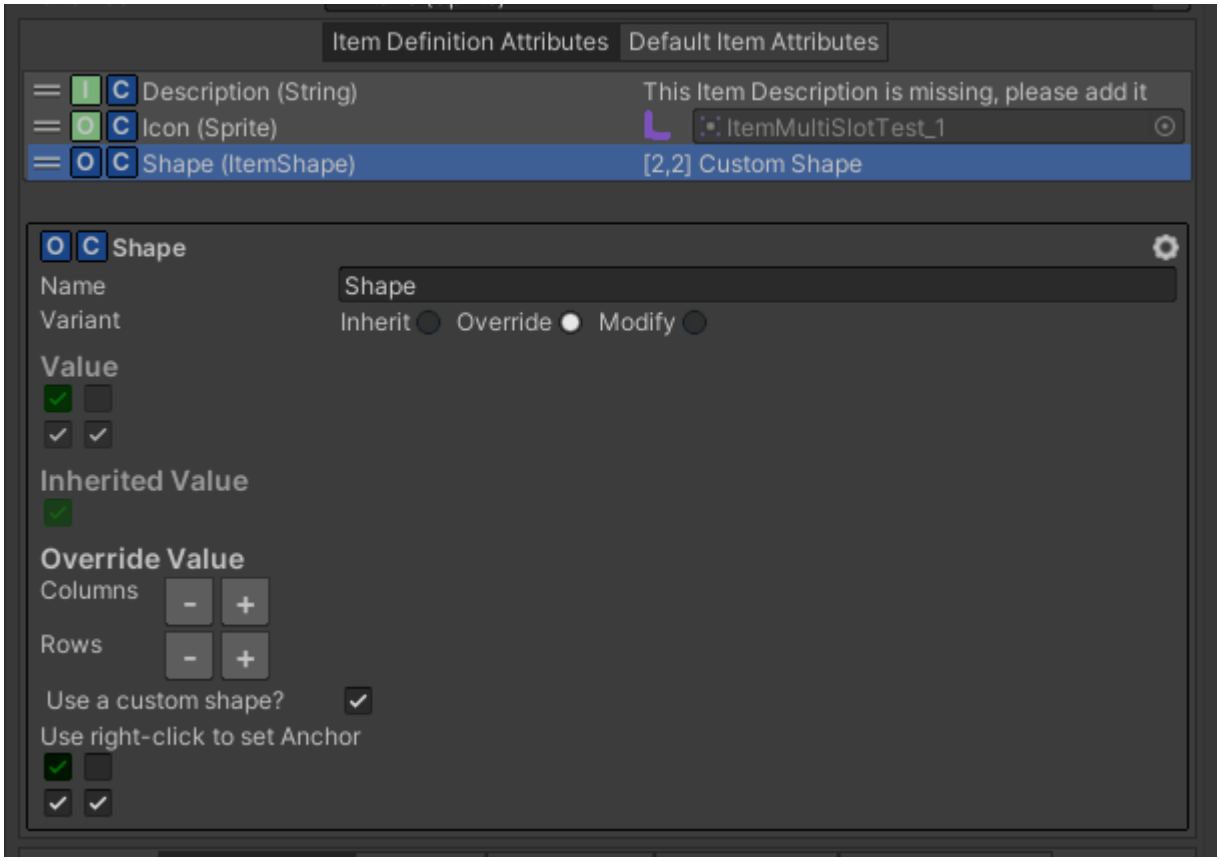
Item Shape Inventory Grid is a special type of Inventory Grid which allows for items which take more than one slot in the grid.
Since it is a bit special compared to how the i Inventory Grid is usually used there are restrictions and requirements to make it work.



Note: In the example above the background grid is a simple tillable image background. It is aligned perfectly (by hand) to the Item View Slots. We are looking into improving this in the future.

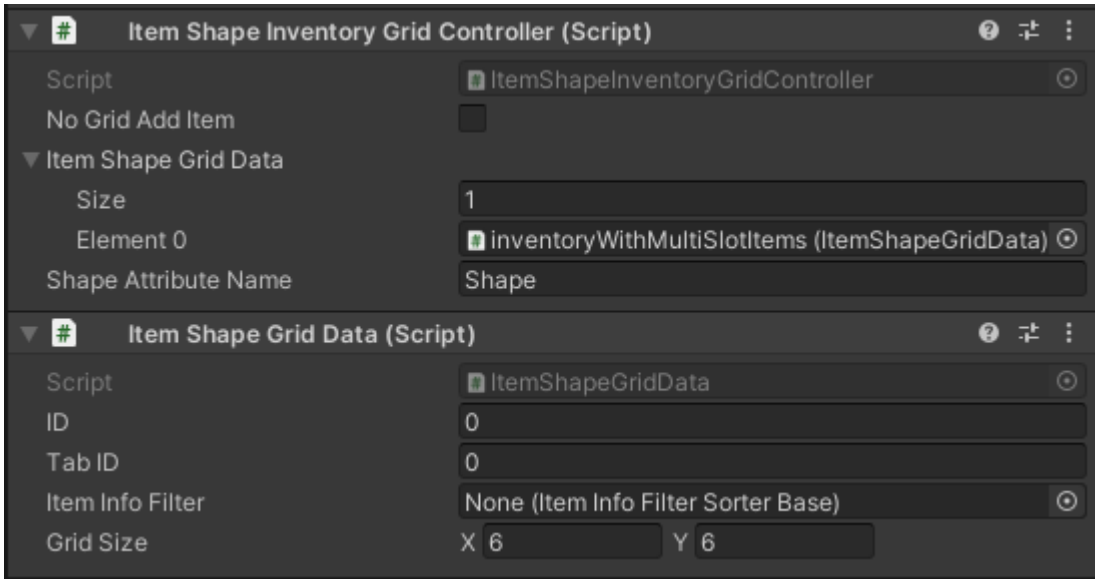
Attributes Define the Item Shape

The Item Shape allows you to define the size and shape of the Item within the grid. It uses a grid of boolean with an anchor. The anchor must be on an index which is “True”. The Item Shape can be set on items are Item Category/Definition Attribute. The rest of the Item Shape system will take care of using that information to place the items correctly:



Item Shape Inventory Grid Controller

The Item Shape Inventory Grid Controller will use the Item Shape Attributes on the Items to know whether or not they fit in different grids. The Item Shape Grid Data is the component which keeps track of where the item is in a grid. This components must be set next the the Inventory they affect.

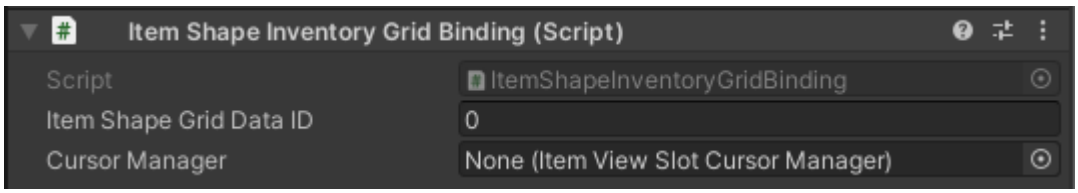


The ID and Tab ID can be used to differentiate the different grids. The Item Info Filter can be used to restrict the type of items which gets added in the Item Shape Grid. Of course the Grid Size must match the Inventory Grid Size in the UI.

Item Shape Inventory Grid Binding

The Item Shape Inventory Grid Binding sits next the the Inventory Grid component and it is

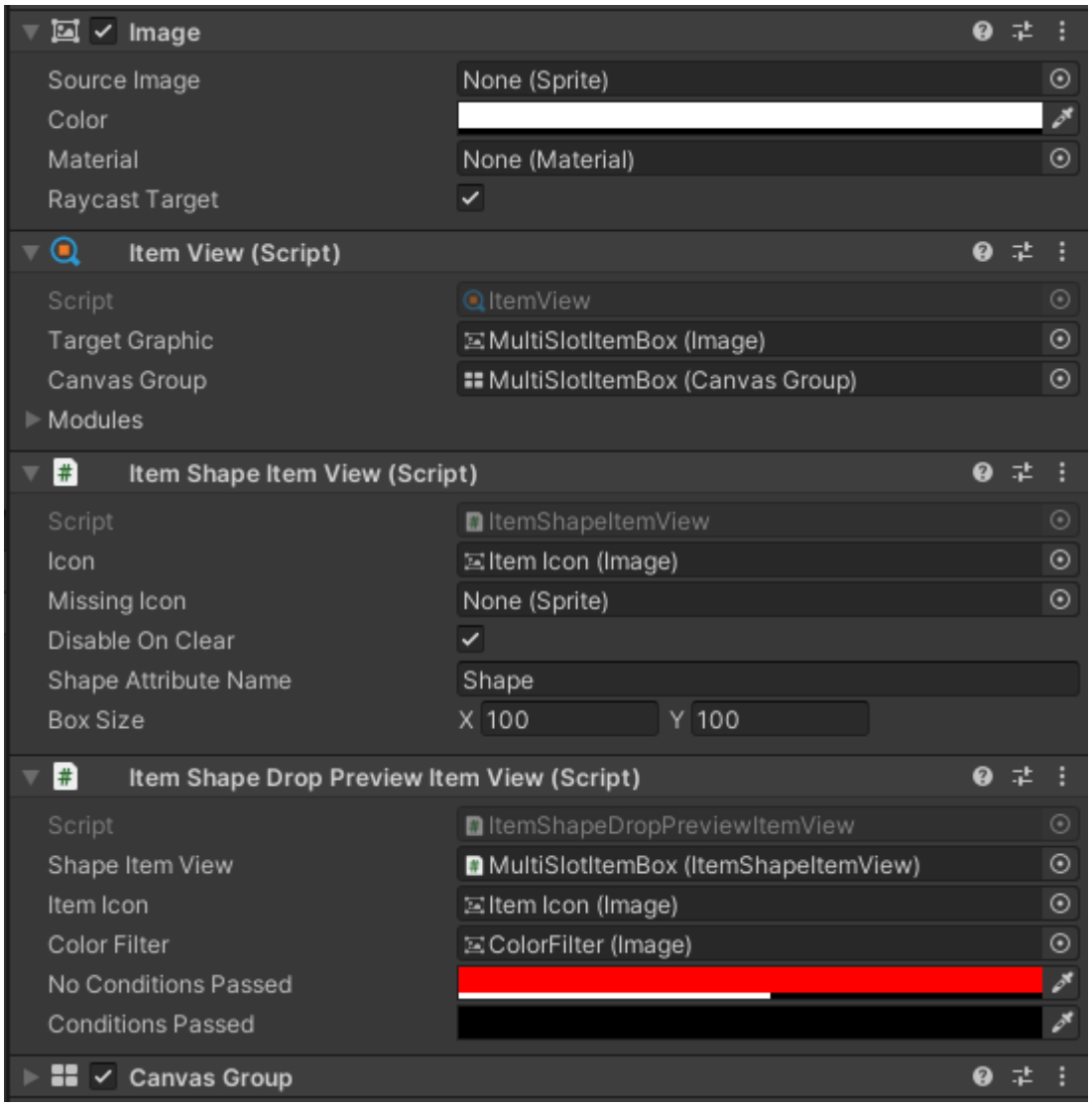
used to bind the Item Shape Grid Data to the Inventory Grid to display the items within the grid.



The Grid Item Shape Data Id can be used to specify which Inventory Item Shapes Grid Data should be used from the Inventory Item Shapes Grid Controller.

Item Shape Item View

For the Item View to display correctly in the Inventory Grid it must have some special Item View Modules.



The following components are used:

- *Item Shape Item View*: A component used to show the image of the item at the correct size.
- *Item Shape Drop Preview Item View*: A component used to preview whether the item can be moved by swapping icons of the Item Views and changing the color whether the item can be moved or not.

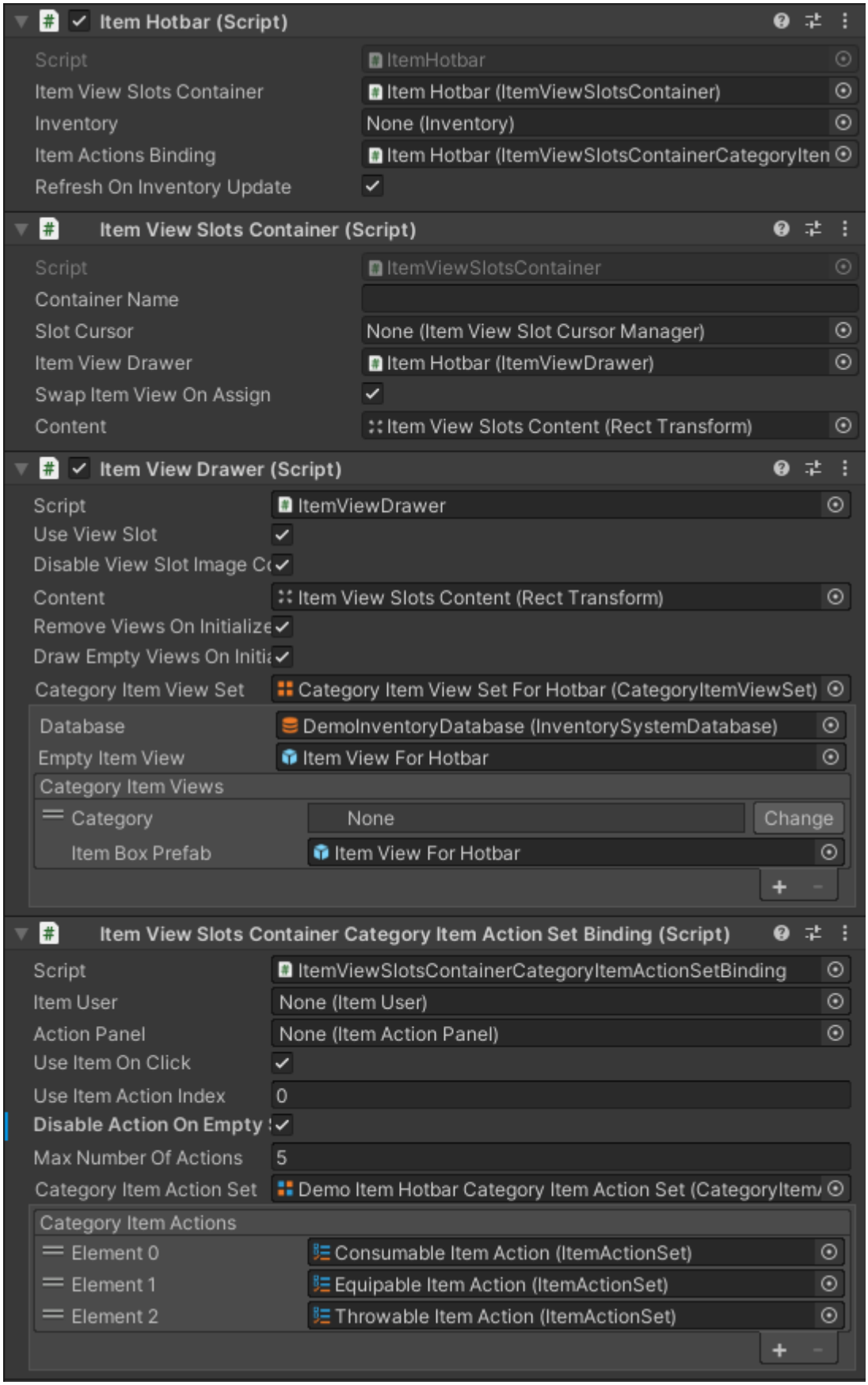
- *Canvas Group*: The canvas group is important as it will be used to hide the image and disable interaction when necessary.

Item Hotbar

The Item Hotbar uses an Item View Slots Container to display and use Items. To create an Item Hotbar with use the UI Designer Hotbar tab. To learn more about Item View Slots Containers see this page.

> Note: Hotbars are very game specific. Our solution allows you to assingn the items to monitor in each slot. The items do not move within the Inventory.
>
> Another option is to use the Item Slot Collection View (aka Equipment UI) as a hotbar. A Feature Demo scene called “Hotbar” showcases this use case.
>
> If neither of these options suits your needs it is recommended to create your own custom hotbar by inheriting the Item View Slot Container Base class which will automatically work with item view drawer, drag/drop and item actions.

An Item Hotbar requires an Item View Slots Container and also benefits from an Item View Drawer and an Item View Slots Container Category Item Action Set Binding.



The Item View Slots Container orders items by index which allows the hotbar to easily get the Item View Slot for any index. The Item Hotbar uses events to capture input from the Inventory Input it is bound to. The hotbar can be refreshed each time the bound inventory is updated to keep the item amounts/states always up to date.

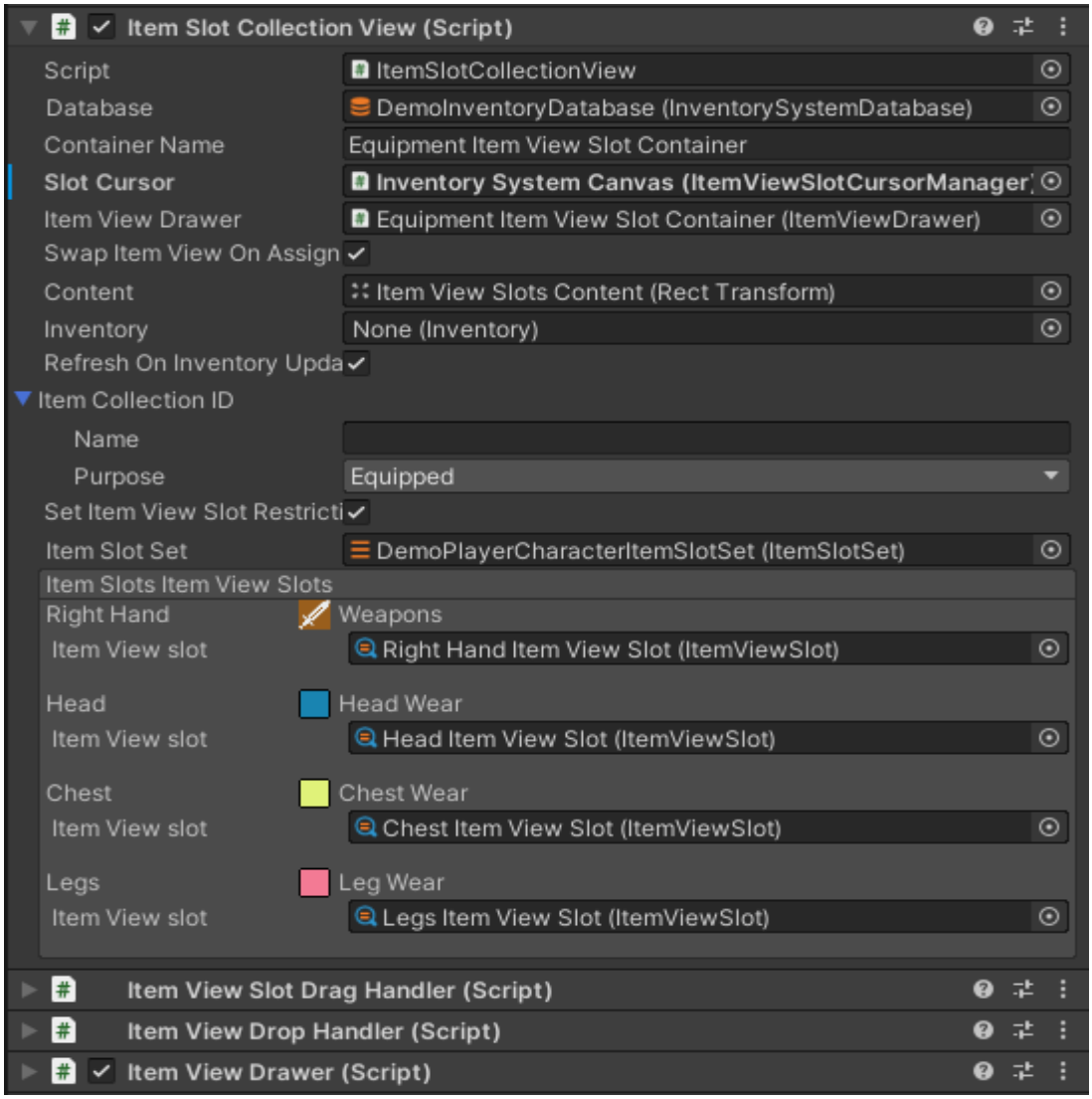
```
hotbar.AssignItemToSlot(itemInfo, itemSlotIndex);
```

```
hotbar.UseItem(itemSlotIndex);
```

Similar to other Item View Slots Containers, the component is initialized by a panel binding which also sets the inventory to monitor.

Item Slot Collection View (Equipment)

The Item Slot Collection View is an Item View Slots Container that binds to an Item Collection within an Inventory which is perfect for making equipment panels. Create and setup an Equipment panel using the UI Designer Equipment tab. To learn more about Item View Slots Containers see this page.



The Item Collection ID field must be pointing to an Item Slot Collection which has the same Item Slot Set as specified in the Item Slot Set field. Map each slot to an Item View Slot by setting them in the Item Slots Item View Slots field. The Set Item View Slot Restriction field will make sure that the Item View Slots have a restriction to prevent adding items with the wrong category in a slot.

Useful methods include:

```
// Get the Item View Slot using an Item Slot name.
var headItemViewSlot =
```

```
m_ItemSlotCollectionView.GetItemViewSlot("Head");

// Get the Item Slot from an Item View Slot.
var headItemSlot =
m_ItemSlotCollectionView.GetItemSlot(headItemViewSlot );

// Bind an inventory to the Item Slot Collection View.
m_ItemSlotCollectionView.SetInventory(m_Inventory );
```

Move Items (Drag & Drop)

Items can be moved within and between Item View Slots Containers using drag & drop (mouse) or with special Item View Slots Container Item Actions (keyboard/gamepad). The Movement components can easily be added and setup on any Item View Slots Container using UI Designer.

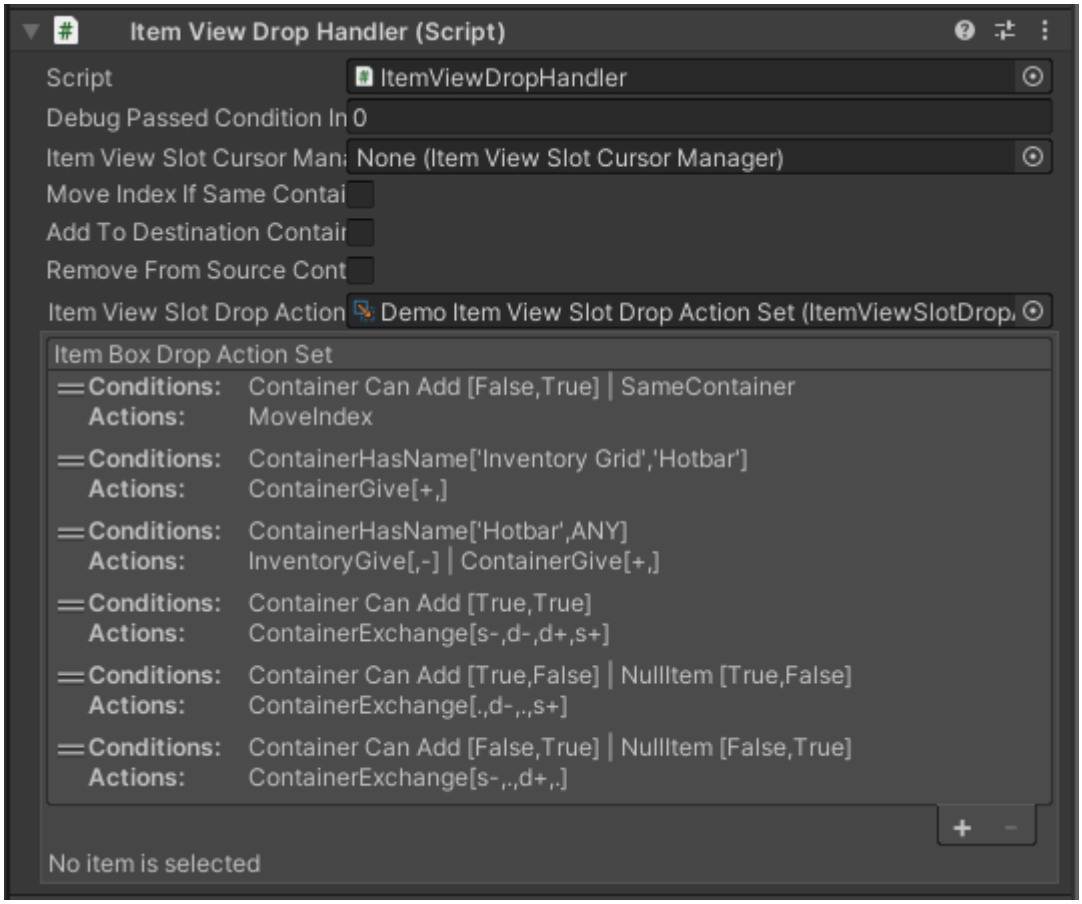
The Item movement in the UI is separated in two main parts Item View Slots Cursor Manager and Item View Drop Handler. In addition depending whether to move with drag & drop or by the Move Item Action another component is required: Item View Slots Drag Handler or Item View Slot Move Cursor.

Item View Slots Cursor Manager

The Item View Slots Cursor Manager usually sits next to the Display Panel Manager on the Canvas. This component’s job is to keep track of the source of the Item being moved and spawn the Item View that will move across the screen while the player is dragging or moving the item. The source Item View and the moving Item View will have their “IViewModuleMovable” Modules notified such that they may change their visual.

Item View Drop Handler

The Item View Drop Handler is the brains behind the drag and drop movement operations. It takes the information about the source Item View Slot and the destination Item View Slot and checks the Item View Slot Drop Action Set to know what what actions to perform when the drop happens: exchange, give, add, or remove. You may create your own custom drop condition and actions.



The Item View Drop handler will listen to these events on the Item View Slots Containers:

- *OnItemViewSlotDropE*: Triggered when something is dropped on an Item View Slot
- *OnItemViewSlotSelected*: Used to know the Items being hovered on while an Item View Slot is moving
- *OnItemViewSlotDeselected*: Used to know when the moving Item View stops hovering an Item View slot

These events all pass in an “Item View Slot Event Data” object which contain information about the Item View, Slot, Container and index. Using that information it sets up an “Item View Slot Drop Handler Stream Data” with the information of the source and destination Item View Slots. This stream data can then go through the Item View Drop Action Set to know which action (if any) should potentially be invoked when the Item is dropped.

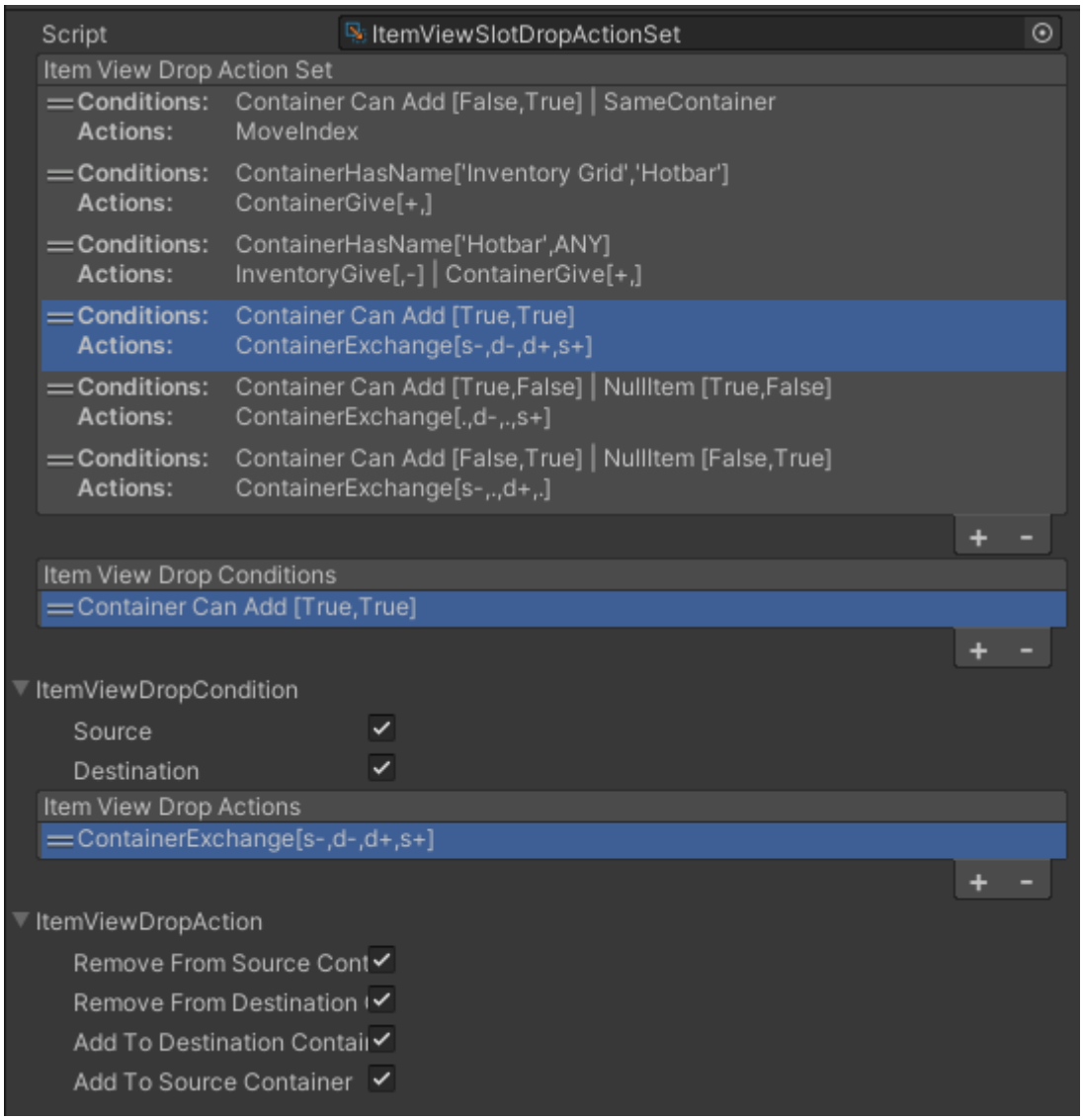
Since the stream data is an object with public properties it may be modified while going through the Item View Slot Drop Action Set. It is highly recommended not the change it while checking conditions.

The Item View Modules with the “IItemViewSlotDropHoverSelectable” interface will be notified when the Item View is being hovered. The module will get a reference to Item View Drop Handler, which it can use to display a preview of the drop, a green/red background whether any or none of the conditions were met.

Item View Slot Drop Action Set

The Item View Slot Drop Action Set is a list of conditions and actions defines what should happen depending on the source and destination of the Item View Slots. Create a new Item View Slot Drop Action Set in the project view using right-click Create -> Ultimate Inventory

System -> UI -> Item View Slot Drop Action Set.



An example of condition/action could be a “Container Can Add” condition where the destination can add the source item and vice versa, with a “Container Exchange” action where the items are removed from the source and destination, to be added to the destination and source.

The options of conditions and actions will appear automatically when pressing the “+” icon to add elements to the lists.

Here are a list of available conditions and actions:

Drop Conditions

- *Item View Drop Container Can Add Condition*: Check if the source and/or the destination can accept an exchange.
- *Item View Drop Container Has Name Condition*: Check the names of the source and/or destination container to see if they match with the names specified.
- *Item View Drop From Item Collection Condition*: Check if the source item collection matches the item collection specified.
- *Item View Drop Null Item Condition*: Check if the source and/or the destination item is Null.

- *Item View Drop Same Container Condition*: Check if the Item View Slots come from the same Item View Slots Container.
- *Item View Drop Container Can Move*: Check if item can be moved from an index to another within the same Container.

Drop Actions

- *Item View Drop Container Exchange Action*: Exchange Items between containers by removing from one and adding to the other.
- *Item View Drop Inventory Exchange Action*: Similar to the above but instead of adding/removing items through the Item View Slots Container, do so directly on the Inventory.
- *Item View Drop Container Give Action*: Give the item from the source to the destination. Choose whether the item is removed and/or added.
- *Item View Drop Inventory Give Action*: Similar to the above but instead of adding/removing items through the Item View Slots Container, do so directly on the Inventory.
- *Item View Drop Move Index Action*: Moves the index of the Item Stack within the same Item View Container.

You may create your own conditions and actions simply override the “Item View Drop Condition” and “Item View Drop Action” abstract class:

```
// Condition.
[Serializable]
public class ItemViewDropSameContainerCondition :
ItemViewDropCondition{
    public override bool CanDrop(ItemViewDropHandler
itemViewDropHandler)
    {
        return itemViewDropHandler.SourceContainer ==
itemViewDropHandler.DestinationContainer;
    }
}

// Action.
[Serializable]
public class ItemViewDropMoveIndexAction : ItemViewDropAction
{
    public override void Drop(ItemViewDropHandler itemViewDropHandler)
    {
itemViewDropHandler.SourceContainer.MoveItem(itemViewDropHandler.Strea
mData.SourceIndex, itemViewDropHandler.StreamData.DestinationIndex);
    }
}
```

Item View Slot Drag Handler

The Item View Slot Drag handler component can be set next the the Item View Drop Handler if you wish to use drag & drop.

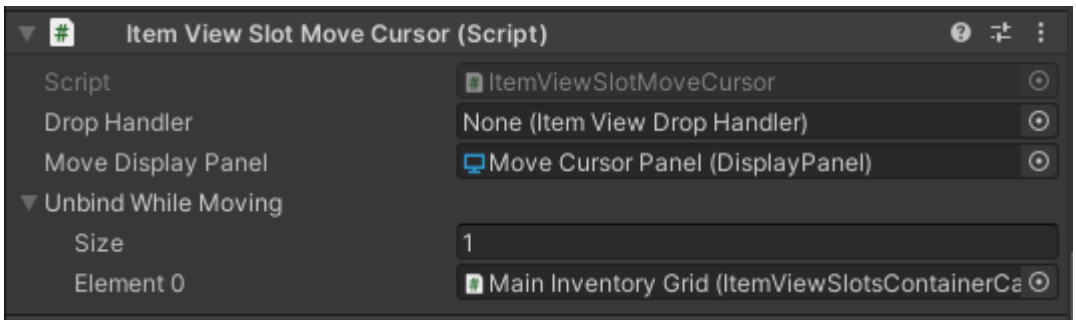
The Item View Slot Drag Handler will listen to these events on the Item View Slots Containers:

- *OnItemViewSlotBeginDragE* : Triggered when the Item View Slot is starting the drag.
- *OnItemViewSlotEndDragE*: Triggered when the Item View Slot finishes the drag.
- *OnItemViewSlotDragE*: Updates every frame the Item View Slot is being dragged.

By listening to these events the drag handler can set up the Item View Slot Cursor Manager with the source Item View Slot information and tell the manager the position of the mouse while the dragging is taking place.

Item View Slot Move Cursor

The Item View Slot Move Cursor component allows you to move items from one slot to another without the need for drag & drop.



The Move Display Panel is optional. It allows you to cancel the movement by pressing the back button (escape by default). This input closes the currently opened panel which will by the Move Display Panel while the movement is taking place. It is important to note the *Unbind while Moving* field which should be used to unbind the Item Actions while the move is taking place.

The Item View Slot Move Cursor will listen to the following events on the container:

- *OnItemViewSlotSelected*: An Item View Slot was selected.
- *OnItemViewSlotClicked*: An Item View Slot was clicked.

The component can use those events to setup the Item View Slot Cursor Manager and the Item View Drop Handler with the source and destination Item View Slot. All that is needed is to call the “StartMove” function. This is usually done by the Move Item Action, but can be done directly with custom code.

Item Info Filter & Sorters

Item Info filters and sorters are used by Item Info Grid, Inventory Grid and other components to filter and sort Item Infos from a list.

The available Item Info Filters and Sorters are:

- *Inventory Search Filter*: Use an input field to search for an item in an Inventory Grid. It replaces the Bound Filter/Sorter by a sorter of choice while the search is in use.

- *Item Info Category Filter*: Filter the item by its category.
- *Item Info Item Collection Filter*: Filter the item by the item collection it is stored in.
- *Item Info Amount Sorter*: Sorts the Item by amount.
- *Item Info Attribute Value Sorter*: Sorts the Item by the value of an attribute. The attribute value must be “IComparable”.
- *Item Info Category Name Sorter*: Sorts the Item by the Item Category name.
- *Item Info Name Sorter*: Sorts the Item by the Item name.
- *Item Info Multi Filter Sorter*: Combine multiple sorter and filters together. The order matters.

Custom Item Info Filters and Sorters.

Creating your custom filter and sorters for Item Info is possible with a little bit of code. Inherit the IFilterSorter<ItemInfo> interface or if you which to create a component inherit from one of those three MonoBehaviours: “Item Info Filter Sorter Base”, “Item Info Filter Base”, “Item Info Sorter Base”.

Filter Sorter Example

An example of multi filter sorter

```
public class ItemInfoMultiFilterSorter : ItemInfoFilterSorterBase
{
    [SerializeField] internal List<ItemInfoFilterSorterBase>
m_GridFilters;
    public List<ItemInfoFilterSorterBase> GridFilters =>
m_GridFilters;
    public override ListSlice<ItemInfo> Filter(ListSlice<ItemInfo>
input, ref ItemInfo[] outputPooledArray)
    {
        var list = input;
        for (int i = 0; i < m_GridFilters.Count; i++) {
            list = m_GridFilters[i].Filter(list, ref
outputPooledArray);
        }
        return list;
    }
    public override bool CanContain(ItemInfo input)
    {
        for (int i = 0; i < m_GridFilters.Count; i++) {
            if (m_GridFilters[i].CanContain(input)) { continue; }
            return false;
        }
        return true;
    }
}
```

Sorter Example

An example of sorting by name:

```
public class ItemInfoNameSorter : ItemInfoSorterBase
{
    [SerializeField] protected bool m_Ascending = false;
    protected Comparer<ItemInfo> m_ItemNameComparer;
    public override Comparer<ItemInfo> Comparer => m_ItemNameComparer;
    protected override void Awake()
    {
        base.Awake();
        m_ItemNameComparer = Comparer<ItemInfo>.Create((i1, i2) =>
        {
            if (i1.Item == null && i2.Item == null) { return 0; }
            if (i1.Item == null) { return 1; }
            if (i2.Item == null) { return -1; }
            if (m_Ascending) {
                return i2.Item.name.CompareTo(i1.Item.name);
            } else {
                return i1.Item.name.CompareTo(i2.Item.name);
            }
        });
    }
}
```

Filter Example

An example of a simple Filter by Item Collection:

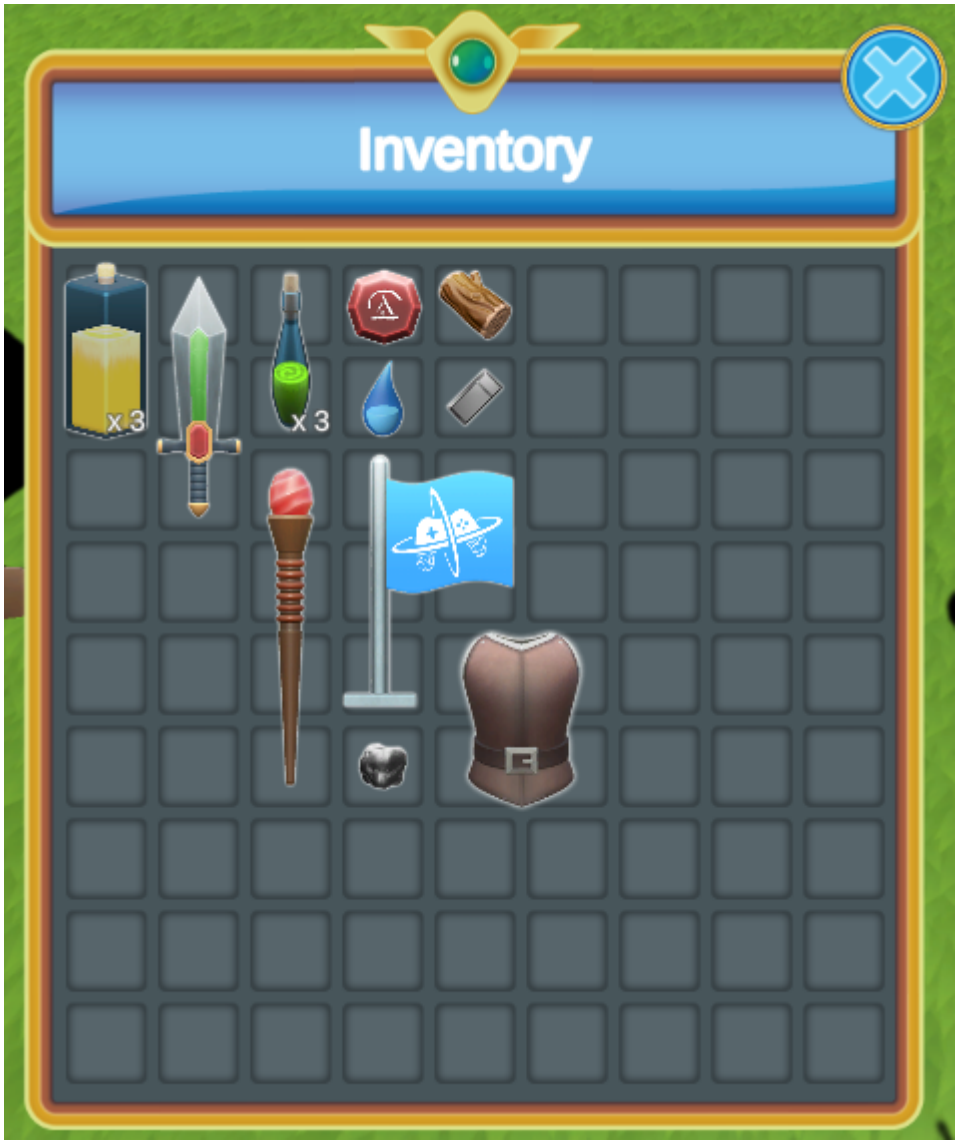
```
public class ItemInfoItemCollectionFilter : ItemInfoFilterBase
{
    [Tooltip("The item collections to show, Show all if null.")]
    [SerializeField]    protected ItemCollectionID[]
m_ShowItemCollections;
    [Tooltip("The item collections that should not be drawn.")]
    [SerializeField]    protected ItemCollectionID[]
m_HideItemCollections =
        { ItemCollectionPurpose.Loadout, ItemCollectionPurpose.Hide };

    // Return true to show the item, false to hide it.
    public override bool Filter(ItemInfo itemInfo)
    {
        var show = m_ShowItemCollections.Length == 0;
                for (int i = 0; i < m_ShowItemCollections.Length; i++)
{
            if
(!m_ShowItemCollections[i].Compare(itemInfo.ItemCollection)) {
continue; }
            show = true;
```

```
            break;
        }
        if (show == false) { return false;}
                for (int i = 0; i < m_HideItemCollections.Length; i++)
{
            if
(m_HideItemCollections[i].Compare(itemInfo.ItemCollection)) { return
false; }
        }
        return true;
    }
}
```

Item Shape Grid

Item Shape Grid may seem quite similar to an Inventory Grid at first but it is quite different in a few ways. The Item Shape Grid allows for items which take multiple slots in a finite grid. It allows any shape, it is not limited to rectangles.
To allow this functionality the Inventory itself must contain the grid data. As a result the Item Shape Grid requires a bit more attention than the standard Inventory Grid.

You may create an Item Shape Grid very easily using the UI Designer Item Shape Grid tab.



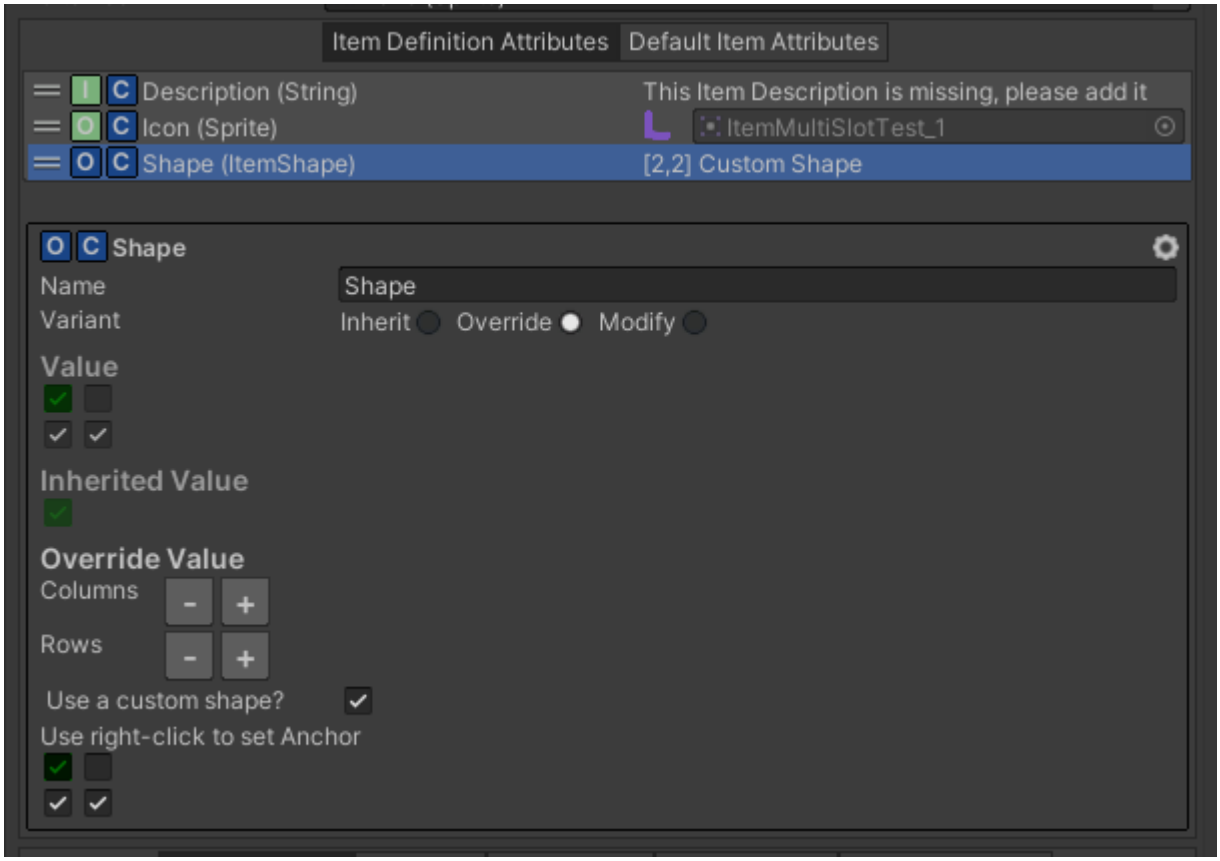
The Item Shape Grid is split in two layers.

1. *Background Layer:* This shows the squares the size of one cell in the grid. It can be used to show the item being selected or as a preview when dragging items. The preview and selection finds all the slots that the item occupies and enables/disables color filters.
2. *Foreground Layer:* This layer show the item icon scaled to the appropriate size. It is not intractable (the clicks pass-through to the background layer).

Both layers use the same Item View which enables/disables the different components depending on its layer state.

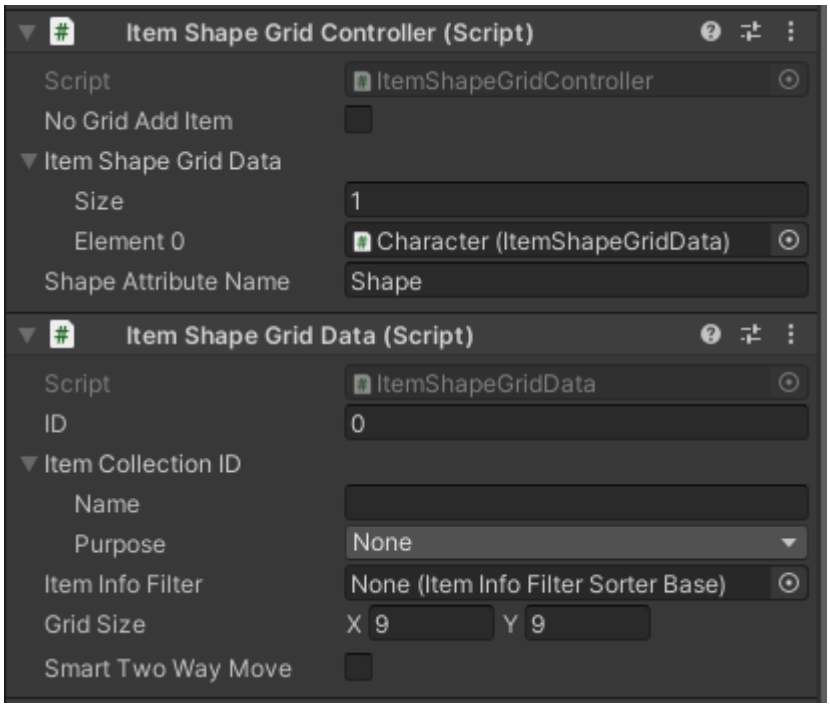
Attributes Define the Item Shape

The Item Shape allows you to define the size and shape of the Item within the grid. It uses a grid of boolean with an anchor. The anchor must be on an index which is “True”. The Item Shape can be set on items are Item Category/Definition Attribute. The rest of the Item Shape system will take care of using that information to place the items correctly:



Item Shape Grid Controller & Item Shape Grid Data

The Item Shape Inventory Grid Controller will determine whether an item can be added in the inventory depending on the state of all the Grid Data. To determine if an item fits or not it checks each Grid Data Item Collection ID and Filter as well as checking the Item Shape Attribute on the Items. The controller also chooses in which grid data the item will be added if there are more than one. The Item Shape Grid Data is the component which keeps track of where the item is in a grid. These components must be set next the the Inventory they affect.

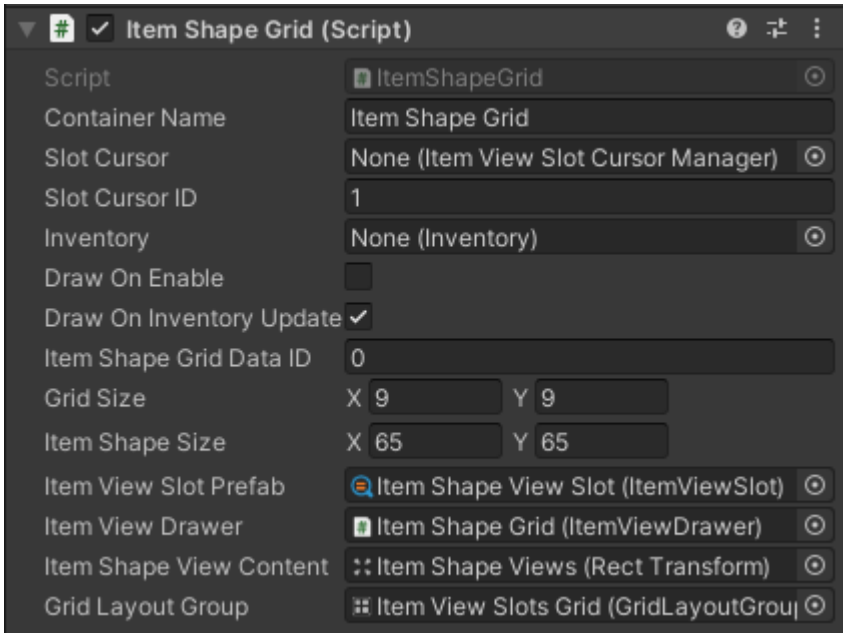


The Item Collection ID must be set to None if you wish to take into account all Item

Collections. Otherwise the Grid Data may only take into account a single Item Collection. For example you may have two grid data: one for the main item collection and the other for your equipped items. The Item Info Filter can be used to restrict the type of items which gets added in the Item Shape Grid. Of course the Grid Size must match the Item Shape Grid Size in the UI.

Item Shape Grid

The Item Shape Grid is an Item View Slot Collection. It can easily be setup through the UI Designer (which also checks that your inventory grid data matches the UI size). The Item Shape Grid will use the information in the Grid Data (found by its ID on the bound Inventory) to draw the items in the grid.

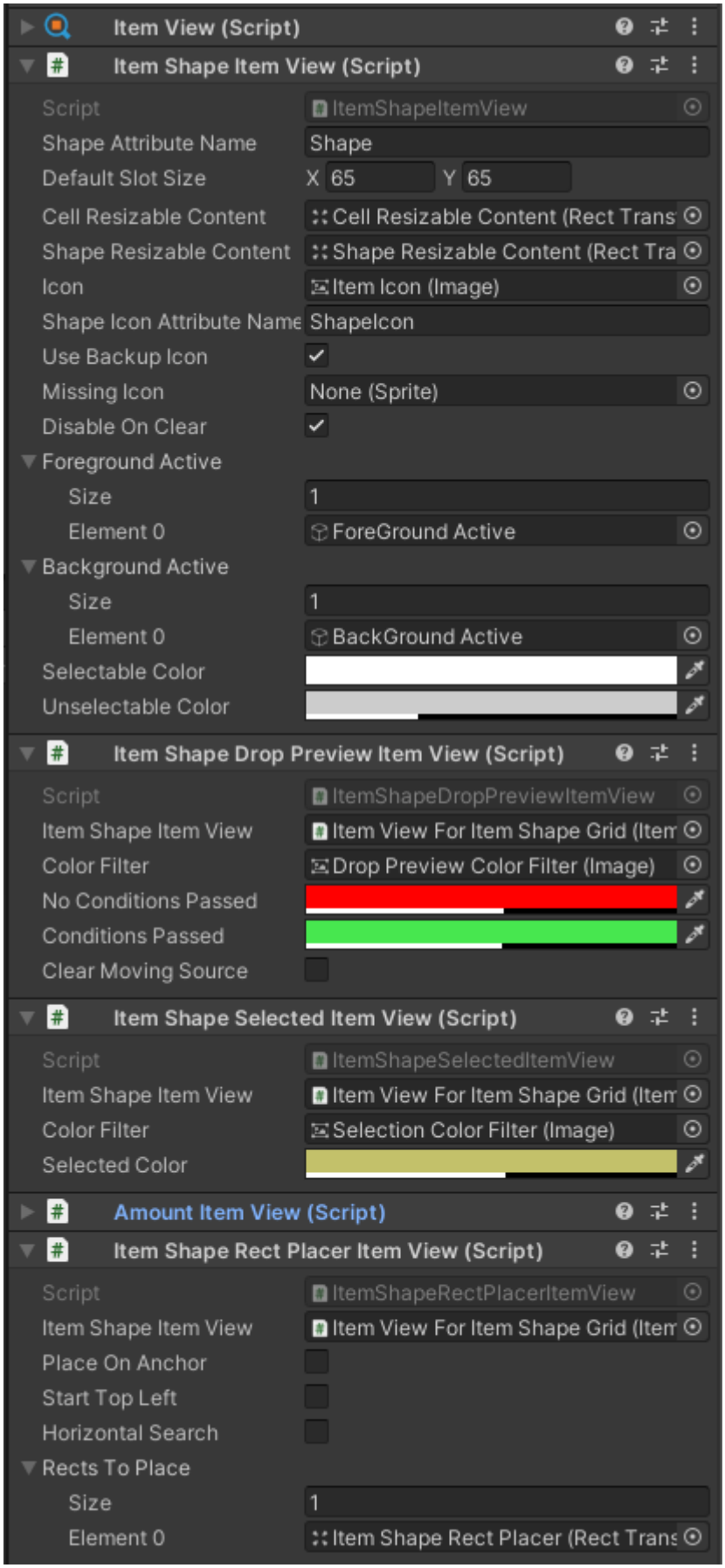


It’s important that the Grid Size matches between the Item Shape Grid and the Item Shape Grid Data.
The *Item Shape View Content* is the transform where the foreground Item Views will be spawned. The background Item Views will be spawned in the Item View Drawers Content like any other Item View Slot Container.

Whenever possible use the UI Designer to change the Grid Size or the Item Shape Size (Cell Size). This will make sure that all the components will be setup correctly (This might not work well with prefabs).

Item Shape Item View

For the Item View to display correctly in the Item Shape Grid some special Item View Modules are required. There are also some optional but extremely useful Item View Modules specific to Item Shape Grids. It is highly recommended to check the “Item View For Item Shape Grid” prefab in the Schema for an in depth look.



The following components are used:

- *Item Shape Item View*: The View Module is the main component for the Item Shape Item View. It is used to show the image of the item at the correct size. It also allows to scale transforms to the cell size or item shape size. There is also the option to enable/disable GameObjects depending on whether the view is set on the background

or foreground layer.
- *Item Shape Drop Preview Item View*: A component used to preview whether the item can be moved by changing the color of some filter images of Item Views in the background layer. This allows the player to preview if an item can be dropped or not.
- *Item Shape Selected Item View*: This uses a similar technique as the drop preview color filter to show the selected Item by adding a color to the appropriate slots in the background layer.
- *Item Shape Rect Place Item View*: If you wish the add an amount in the bottom right of the item, it may be complicated if the item has a weird shape. This is where this component is very useful. It allows you to place a Rect Transform within the Item Shape by offsetting it in different axes depending on the shape and cell size.
- *Canvas Group*: The canvas group is important as it will be used to hide the image and disable interaction when necessary. For example the interaction is disabled when part of the foreground layer.

Views

The View and View Module classes allow for extensive customization when displaying the items, recipes, attributes, etc. These classes allow the View to display things differently depending on the value selected.

View & View Modules

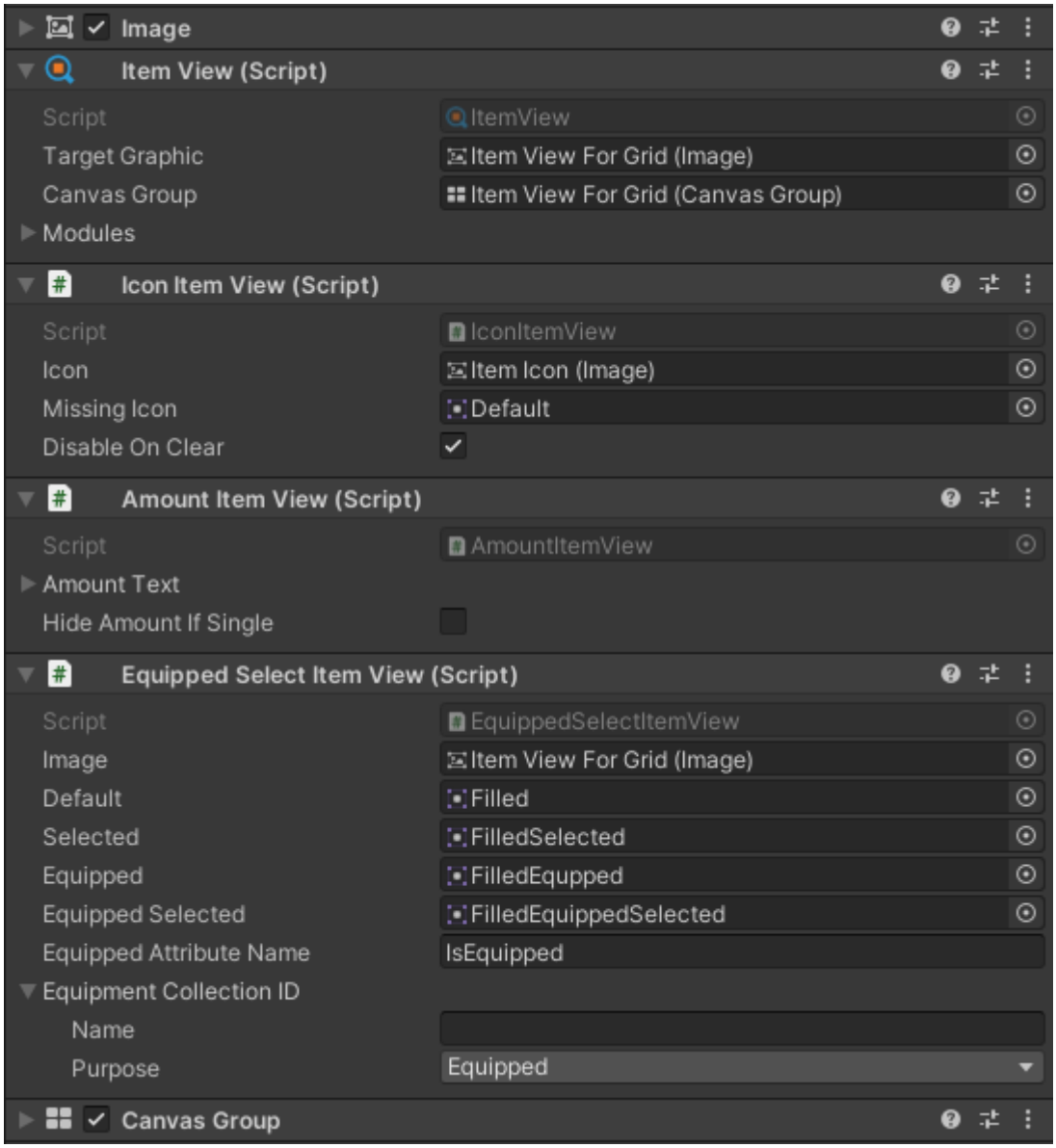
A View is a generic class that can be extended. The Item View extends View class, which is used by items to perform the following functions (and more):

- Clear()
- Select(bool select)
- Click()
- Hide(bool hide)
- SetValue(T value)
- Refresh()

The View will pass on these actions to its View Modules. There can be any number of View Module components. This allows different components to be mix and matched on a prefab to dynamically change the visuals when the state changes.

As an example, the Item View can be used with the components Icon Item View and Name Item View. This allows the item icon and name to be dynamically displayed on the Item View depending on the Item set.

Since these components are small they are extremely easy to write and highly reusable. In the demo scene you'll find a few View prefabs in action, which are used in grid and list layouts. The screenshot below is an example of one.

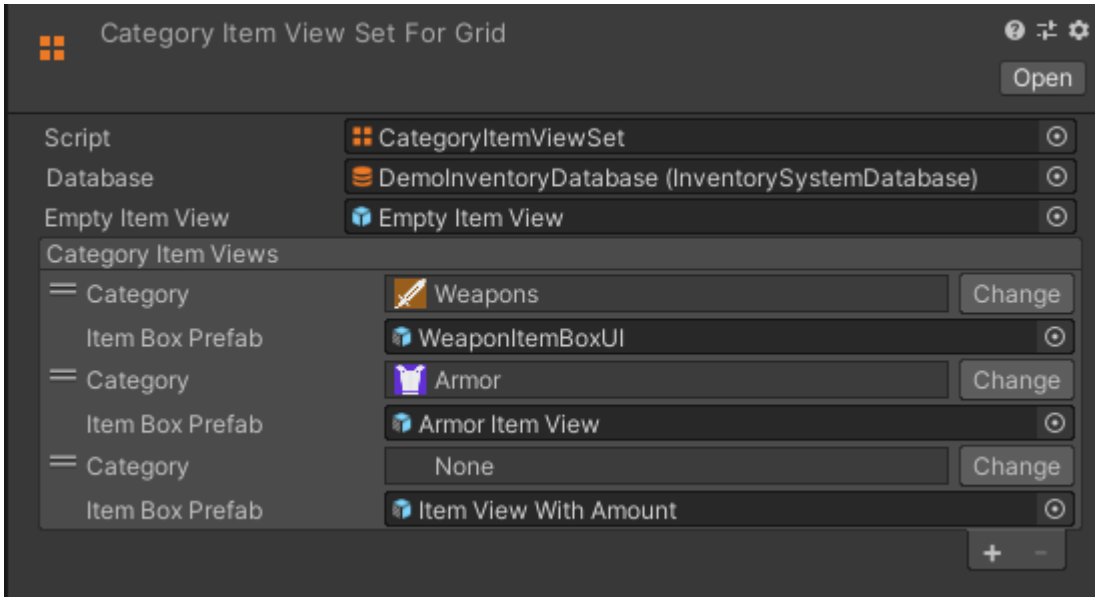


Each View Module component is used for one purpose only and they are grouped by the view.

Item View Drawer & Category Item View Set

Views are normally used in groups. The View Drawer works with the View Module components to spawn new views and send them events such as selected or clicked.

The Item View Drawer uses a Category Item View Set object. The purpose of this object is to map an Item Category to an Item View prefab. With this setup when the Item View Drawer draws an item it knows which Item View prefab it should use. An equivalent object exists for Recipes.

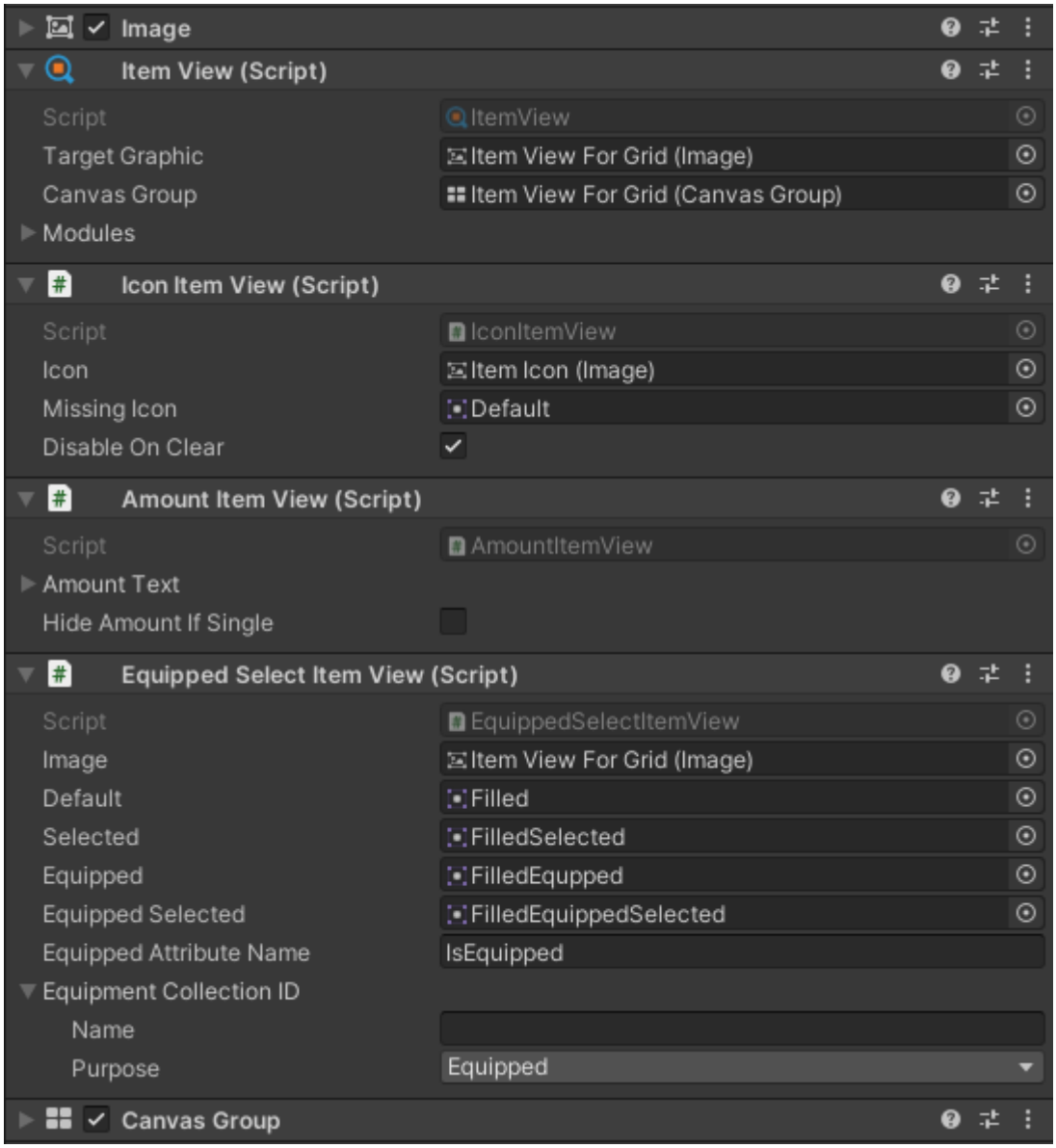


The Item View Drawer invokes events which lets you listen before and after an Item View is drawn. An example use case is the Shop Menu. An item price might depend on the shop or the character, and the item doesn't contain this information. The Shop Menu can listen to the AfterDraw event to set and display the item post-processed price.

Item View

Item Views are used to display an Item in the UI.

Item Views can easily be setup using UI Designer Item View tab



Item Views are often paired with an Item View Slot to detect selection, clicks and more. Find more about Item View Slots here

Item View Modules

Creating a custom Item View Module is possible, simply Inherit the Item View Module base class

Example:

```
/// <summary>
/// An item box component to display an amount.
/// </summary>
public class AmountItemView : ItemViewModule
{
    [Tooltip("The amount text.")]
    [SerializeField] protected Text m_AmountText;
```

```
    [Tooltip("Do not show amount if it is 1 or lower.")]
    [SerializeField] protected bool m_HideAmountIfSingle;

    /// <summary>
    /// Set the value.
    /// </summary>
    /// <param name="info">The item info.</param>
    public override void SetValue(ItemInfo info)
    {
        if (m_HideAmountIfSingle && info.Amount <= 1) {
            m_AmountText.text = "";
        } else {
            m_AmountText.text = $"x {info.Amount}";
        }
    }

    /// <summary>
    /// Clear the value.
    /// </summary>
    public override void Clear()
    {
        m_AmountText.text = "";
    }
}
```

There are some special Item View Modules abstract classes and Interfaces which are very useful when creating custom Item View modules

- IInventoryDependent: This ensures that the Item View Module can have an external Inventory referenced. Example: “Amount Comparison Item View”
- IViewModuleSelectable: receive Select events from the Item View. Example: “Equipped Select Item View”
- IItemViewSlotDropHoverSelectable: receive Hover events from the Item View Drophandler. Example: “Drop Hover Icon Preview Item View”

Some of the most popular Item View Modules include (Use UI Designer to see all of the options):

Select Image View

Generic box component which changes the sprite of an image depending on the selection state of the box.

Name Item View

Displays the name of the item.

Icon Item View

Displays the item's "Icon" attribute value.

Amount Item View

Displays the number of items that the inventory contains.

Equipped Select Item View

Changes the image depending on the Item Collection that the item belongs to. This can be used to show if an item is equipped.

Int Attribute Item View

Displays the value of the integer attribute.

Item Slots Item View

Enables or disables GameObjects depending on the Item Collection that the item belongs to. This can be used to show if an item is equipped.

Item Shape Item View

A special Item View Modules required when using an Item Shape Grid.

Cooldown Item View

Show the cooldown of an Item used by the Use Item Action Set Attribute item action.

Attribute View

Attribute View is similar to Item Views except it is used to display Attribute values. It is often used within Item Descriptions.

Attribute Views can be created using the UI Designer Attribute View tab.

The following components are included as Attribute View Modules (see all available options in the UI Designer):

Float Value Attribute Box

Displays a float attribute. The number of decimals can be changed with format string.

Int Value Attribute Box

Displays an integer attribute.

String Attribute Box

Displays a string attribute.

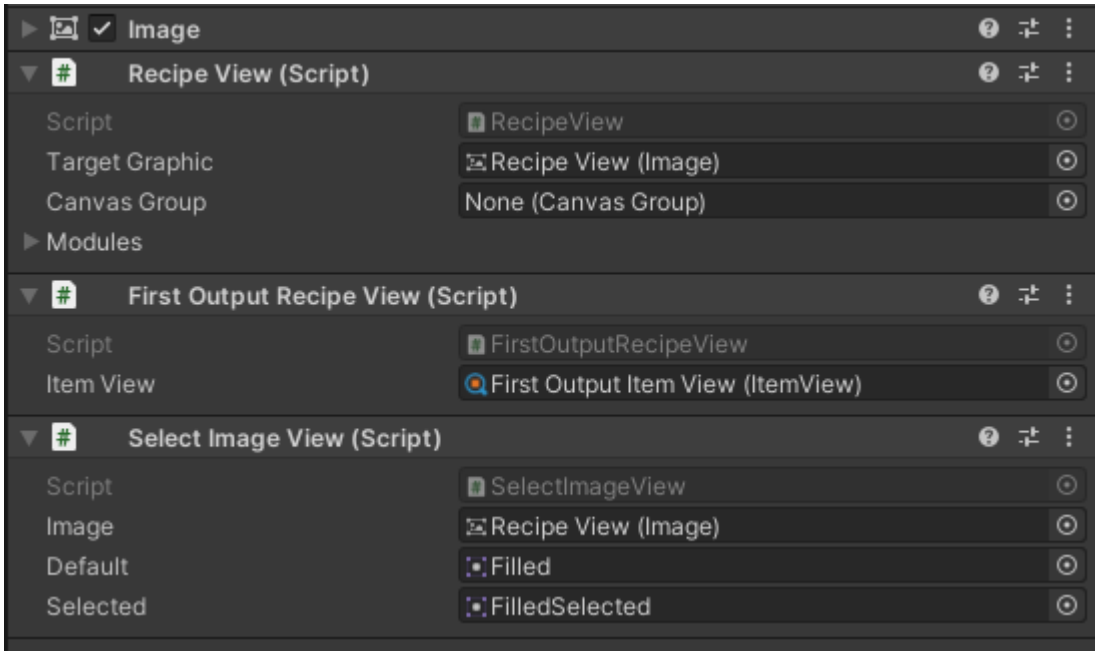
Name Value Attribute Box

Displays the name and the value of an attribute. The value can be any object type.

Recipe View

The Recipe View are just like Item Views except for recipes instead of Item Infos.

Often the Item Recipe View is used to show the output of the recipe



The included Recipe View Modules are:

First Output Recipe View

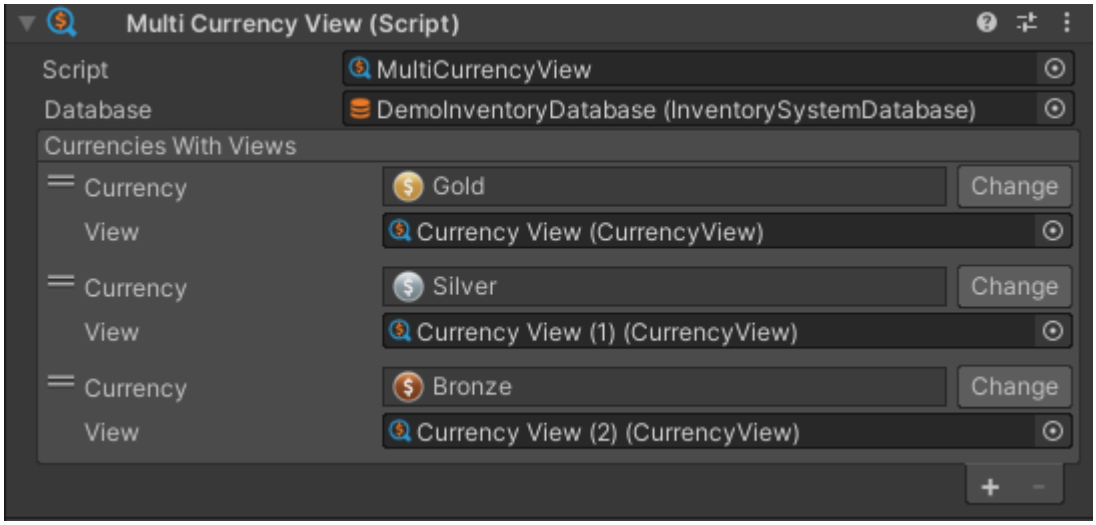
References the Item Box Object to display the first output result of the recipe.

Multi Currency View

Create Currency Views and Currency Owner Monitors with using the UI Designer Currency tab.

Multi Currency View

Currencies are displayed using a Multi Currency View. It maps a Currency to a Currency View. Once a Currency Amount or a Currency Collection is set the Multi Currency View will display the amount of each currency specified.

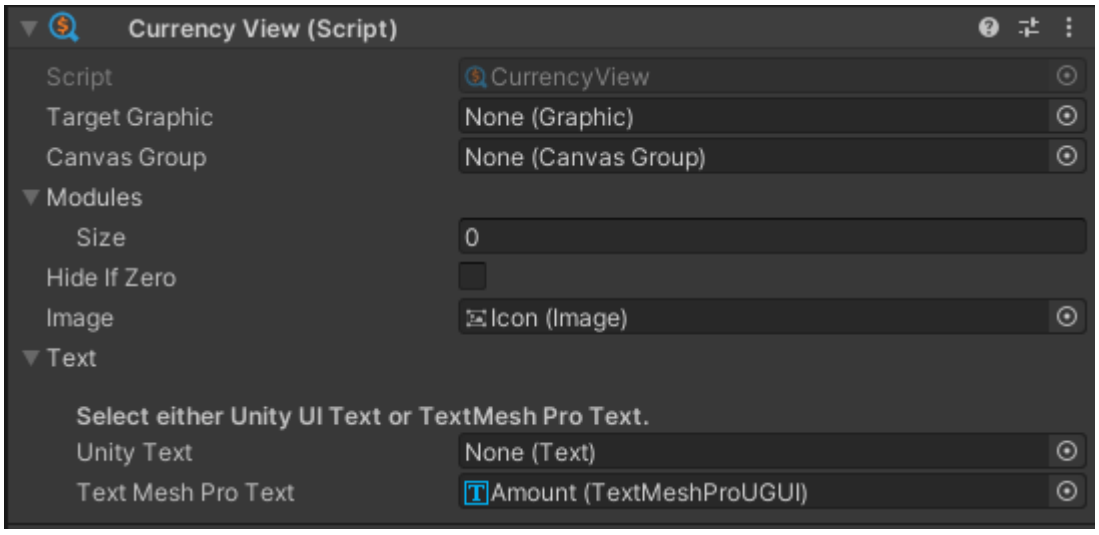


Useful API methods include:

```
// Draw the currency on the Multi Currency View by using a Currency
Collection or a Currency Amount array.
m_MultiCurrencyView.DrawCurrency(currencyCollection);
m_MultiCurrencyView.DrawCurrency(currencyAmounts);

// Set the text color, useful to shop or crafting menus to display if
there is enough currency for example.
m_MultiCurrencyView.SetTextColor(color);
```

Currency View

Similarly to Item Views, Currency Views can be customized. Although usually all that is suggested is to specify an Image component to display the Currency Icon set in the editor.



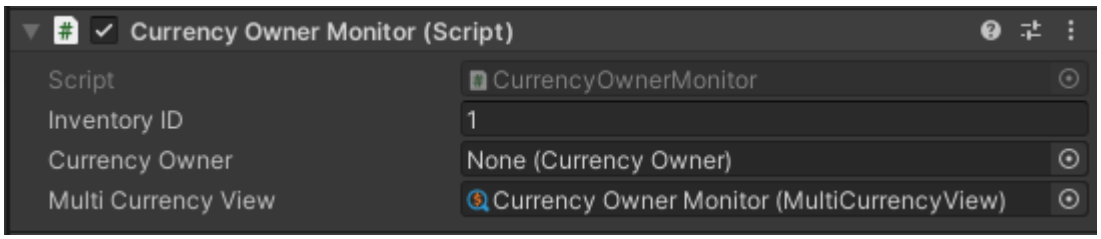
Useful API methods include:

```
// Set the amount of currency to display.
m_CurrencyView.SetValue(currencyAmount);
```

Currency Owner Monitor

The Currency Owner Monitor uses a Multi Currency View to display a Currency Owner

amount of currency. The Currency Owner can be easily found using an Inventory Identifier ID.



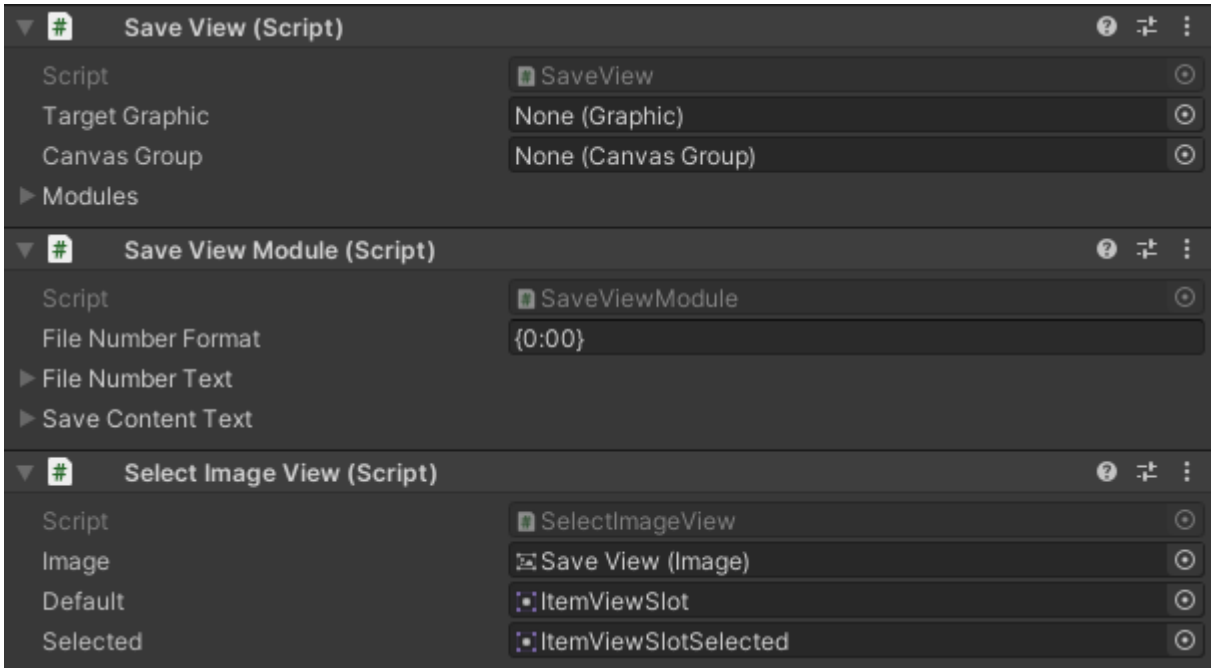
Useful API methods include:

```
// Set the Currency Owner to Monitor.
m_CurrencyOwnerMonitor.SetCurrencyOwner(currencyOwner);
```

Save View

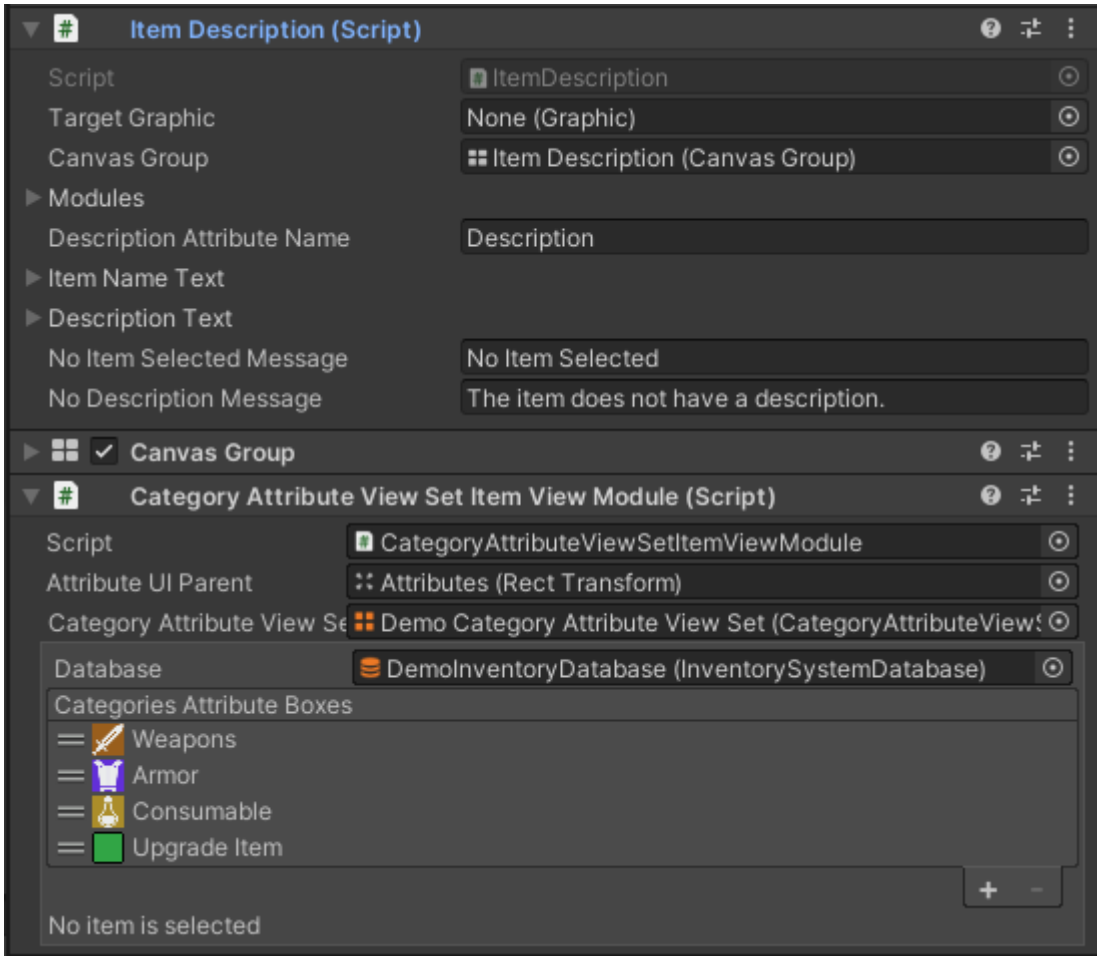
The Save View are similar to Item Views except they are used by the Save Menu to show the Save Data Info.

If you use a custom Save Meta Data you will be able to use custom Save View Modules to customize the content shown in the Save Menu UI.



Item Description

The Item Description is a special Item View which has some predefined options for writing the Item Description. It can be created with using the UI Designer Item Description tab. This page explains more about Item Views.



As an Item View, the Item Description can be coupled with any Item View Module. Item Descriptions are often coupled with the “Category Attribute View Set Item View module” which allows you to dynamically spawn the Attribute Views depending on the Item being described.

Item View Slots Container Description Binding

The Item Description can be bound to any Item View Slots Container (Inventory Grid, Item Hotbar, etc.) such that the item description displays the selected item view slot.

Item Description Panel Binding

An Item Description can be bound to a Display Panel via a Item Description Panel Binding component, which makes sure the Item Description is initialized before use.

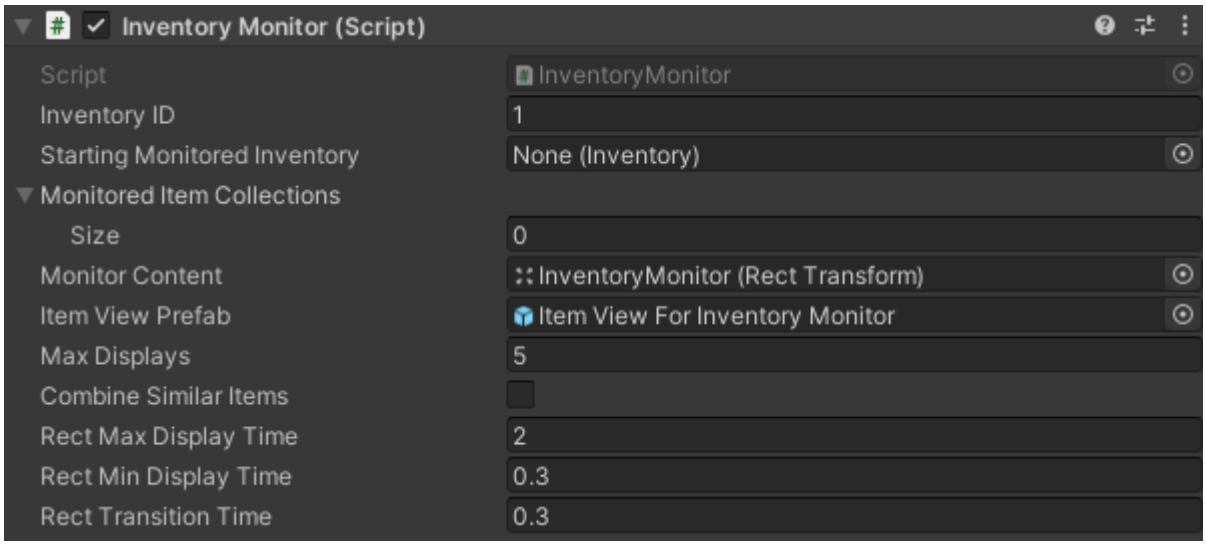
Item Description As Tooltip

Item Description are also sometimes used as tooltips, to do so use the “Item View Slot Panel To Tooltip” component, which can be used to position the item description panel (or any panel) to somewhere near the select Item View Slot of an Item View Slot Container. See the Item View Slots Container page to learn more.

Inventory Monitor

The Inventory Monitor listens to an Item Collection on an Inventory component and pops up Item Views to display the items which were just added to the Inventory.

Create and set up an Inventory Monitor with ease using UI Designer Inventory Monitor tab

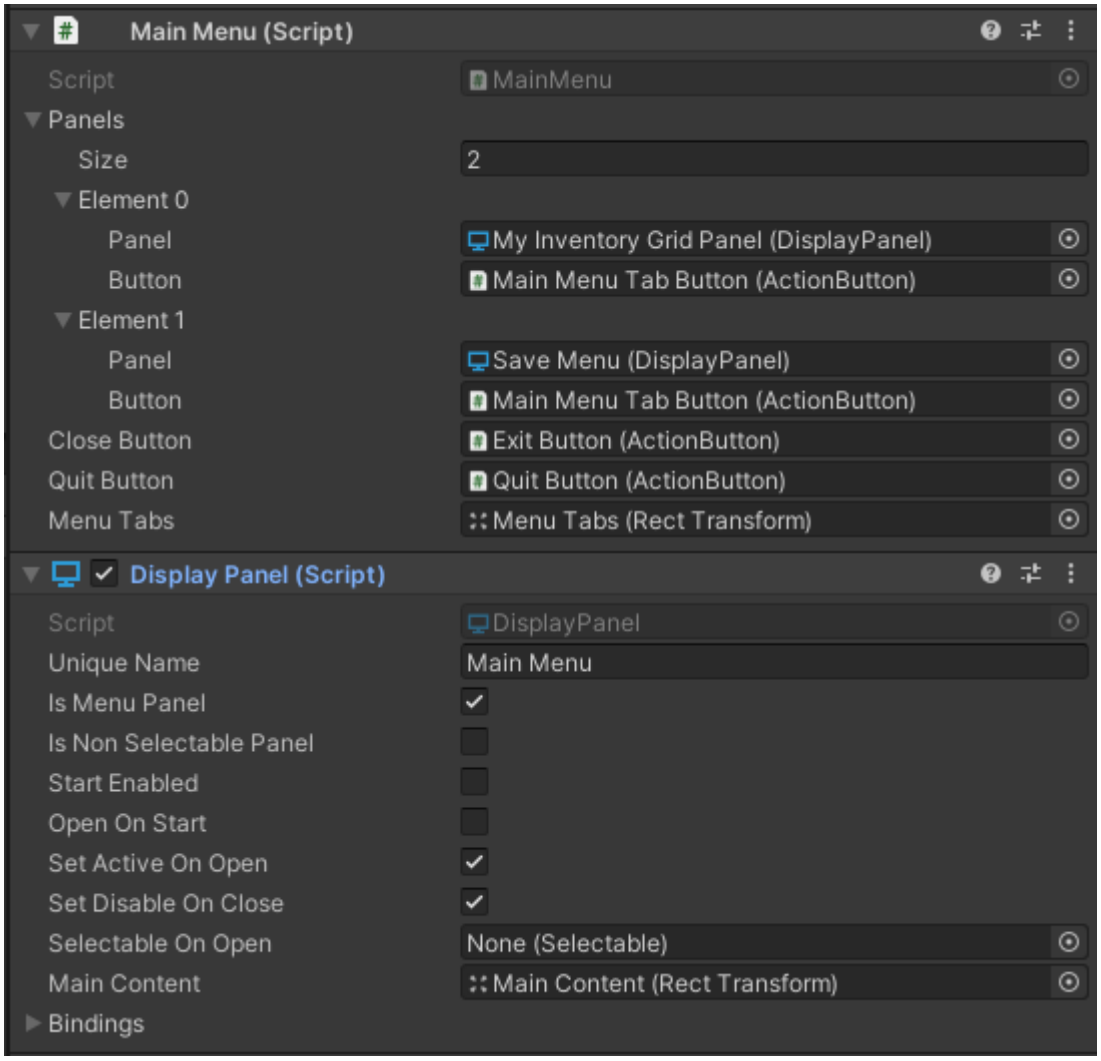


Use an Inventory Identifier ID to monitor the Inventory without needing to reference it directly in the inspector.

You may choose the Item Collections that are monitored. As well as combine items that are added within the same view if they are similar.

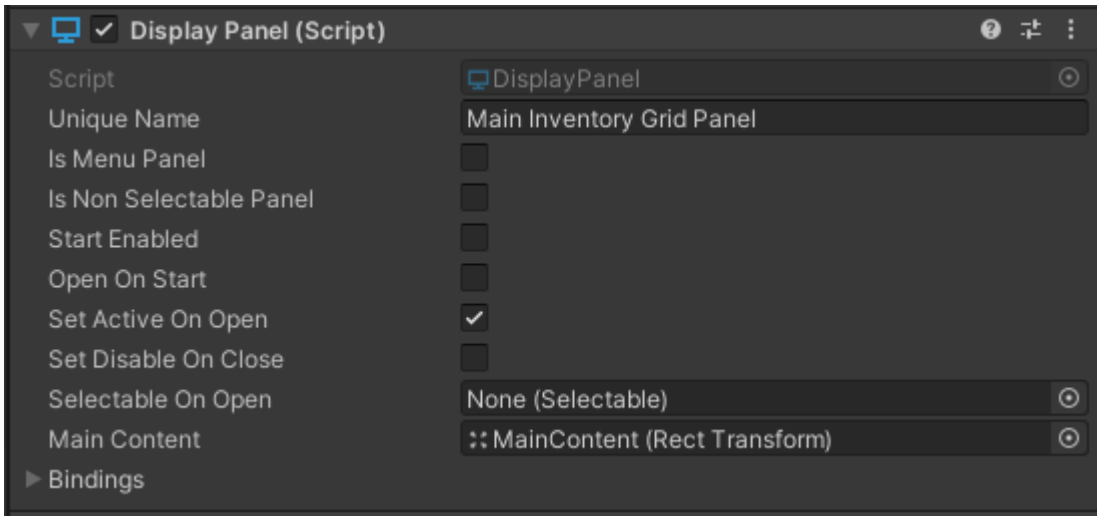
Main Menu

The Main Menu is a Panel Binding which has sub panels. The sub panels can be opened using Action Buttons. To create and setup a Main Menu with use the UI Designer Main Menu tab.



Pay close attention at the Display Panel for the Main Menu and if the *Is Menu Panel* option is selected. In this case the Main Menu has two sub panels: Inventory Grid and Save Menu. Use UI Designer to add more easily.

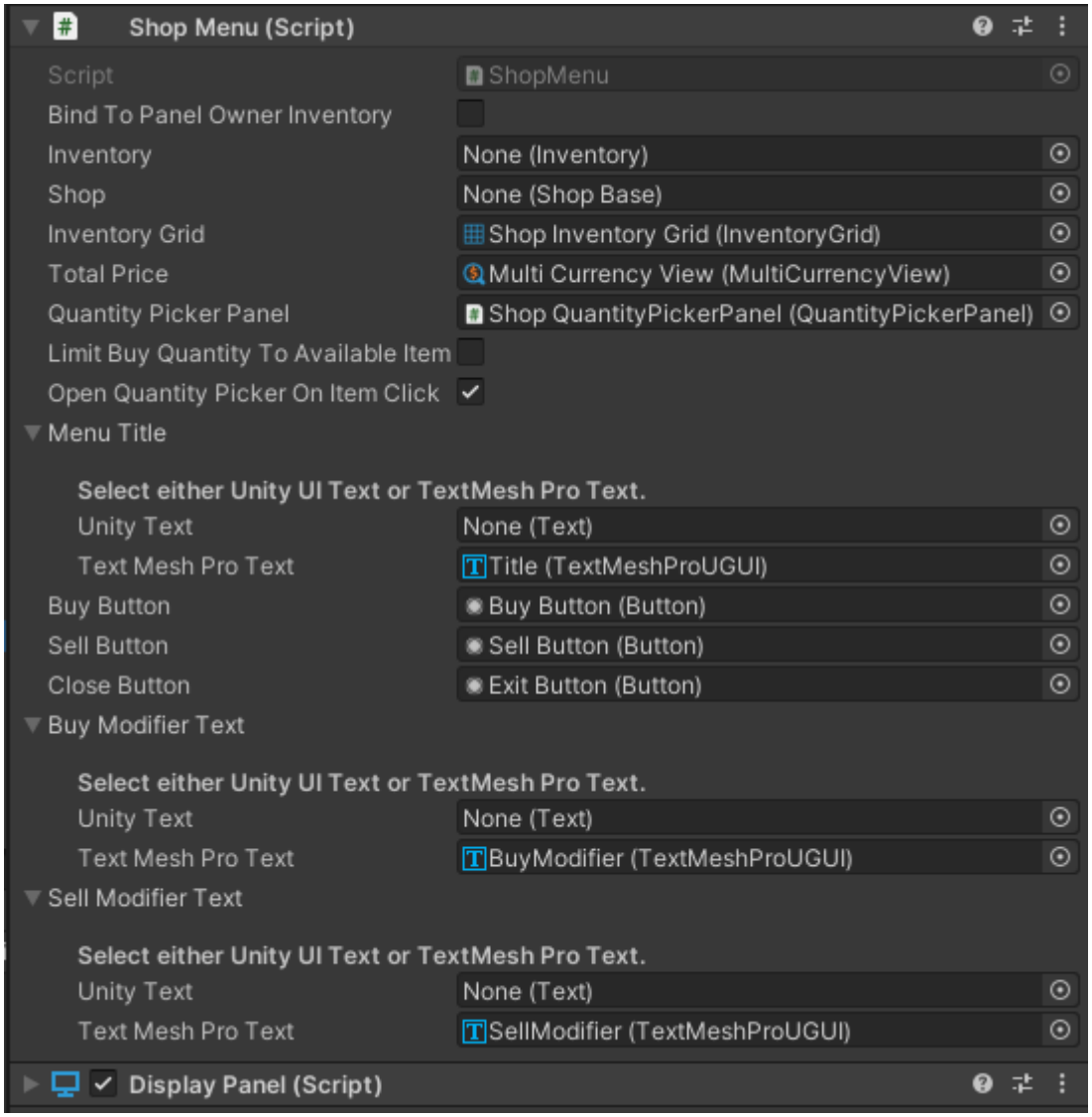
If the *Set Disable On Close* is deselected the Main Menu will display the sub panel while selecting another one.



The Main Menu can be referenced by the Display Panel Manager such that it may be opened/closed.

Shop Menu

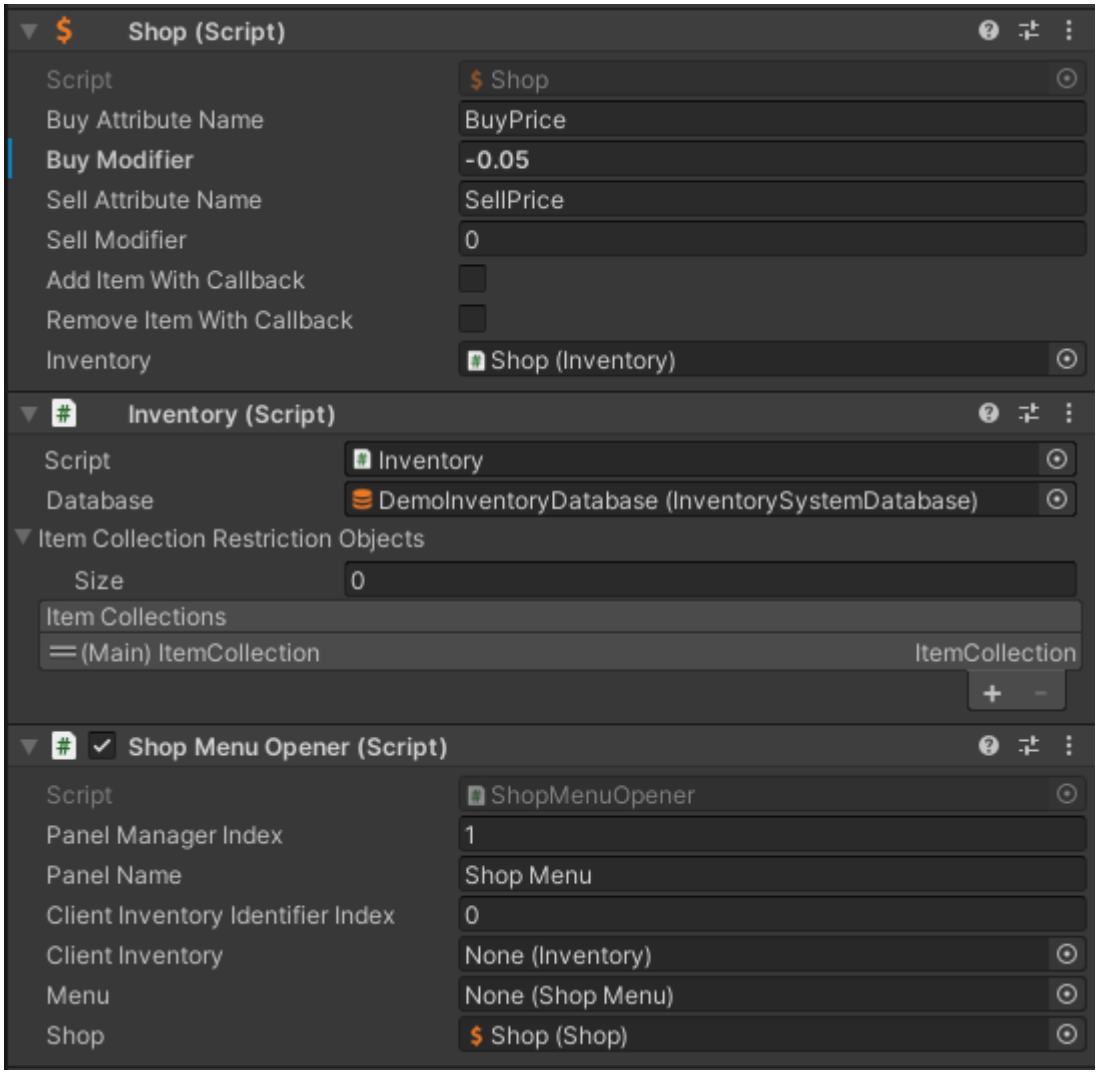
The Shop Menu is used to allow player to buy and sell items from a Shop. Create a Shop Menu with the UI Designer Shop tab. See this page for more information about the Shop component.



It uses the following components:

- *Inventory*: The Inventory is the client inventory (the player).
- *Shop*: The Shop component takes care of all the logic for Buying and Selling. As well as what Item can be bought and sold.
- *Inventory Grid*: The Inventory Grid displays the Items To Buy and Sell.
- *Multi Currency View*: The Multi Currency View shows the total price of the item defined by the Shop.
- *Quantity Picker Panel*: The Quantity Picker Panel lets the user choose the amount to buy/sell.

The Shop component can be set directly next to the Shop Menu, or it can be set through code, allowing multiple shops in the game to use the same UI. The easiest way to do so is by using a “Shop Menu Opener” component.



Save Menu

The Save Menu is used by the player to select a save file to load, save or delete. It uses the Save System Manager to do so.Use UI Designer to set up a Save Menu with ease in the Save tab. To learn more about the Inventory System Saving System go to this page.

Storage Menu

You may use UIDesigner to create and edit a Storage Menu in a few clicks.

The Storage Menu is usually used to exchange items between the player inventory and a storage inventory. It references a pair of Inventory and Inventory Grid for both the player and the storage.

The Inventory Grid is assigned a custom click event that opens a quantity picker panel to choose to amount of item to store and/or retrieve. This means that the Storage Menu is not compatible with Item Actions.

More details can be found on this page.

Important: The Storage Menu is one solution to store items in another Inventory. Feel free to create your custom Storage Menu from scratch or even create storage with just Item Actions and/or drag and drop using Inventory Grids in floating panels.

Chest Menu

The UI Designer can be used to create a new Chest Menu in a few clicks. More details can be found on this page.

The Chest Menu is usually used to retrieve items from a Chest component. The Chest components in the scene will automatically find the Chest Menu and assign themselves to it when opened. The easiest way to get started with the Chest Menu is to simply spawn a "Chest" prefab from the Demo folder in your scene. When the player interacts with the chest through the Inventory Interactor and Interactable components, the Chest Menu will be opened and bound the the chest the player interacted with.

The Inventory Grid is assigned a custom click event that opens a quantity picker panel to choose to amount of item to retrieve. This means that the Chest Menu is not compatible with Item Actions.

Important: The Chest Menu is one solution to store items in another Inventory. Feel free to create your custom Chest Menu from scratch or even create with just Item Actions and/or drag and drop using Inventory Grids in floating panels.

Any component inheriting the IChest interface may be assigned to the Chest Menu using the function:

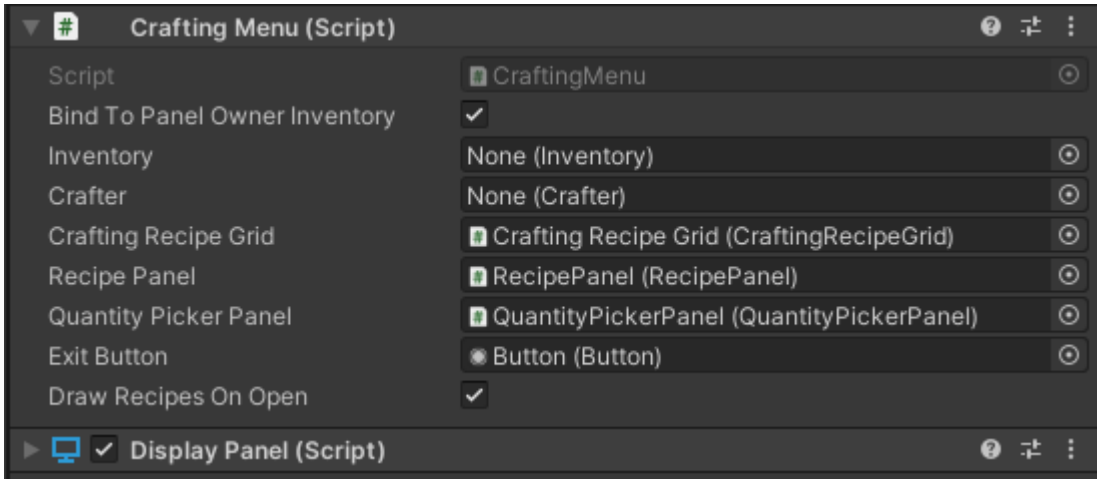
```
chestMenu.SetChest(myChest);
```

Crafting Menu

The Crafting Menu is used by the player to select the recipe to craft and display the ingredients required.

Important: This Crafting Menu is one solution for crafting, but it is highly recommended to extended the crafter or the crafting processing to create custom crafting solutions. Of course this would mean creating a custom Crafting Menu. Head to the forum to discuss possible solutions for your use case and share your experience with other users.

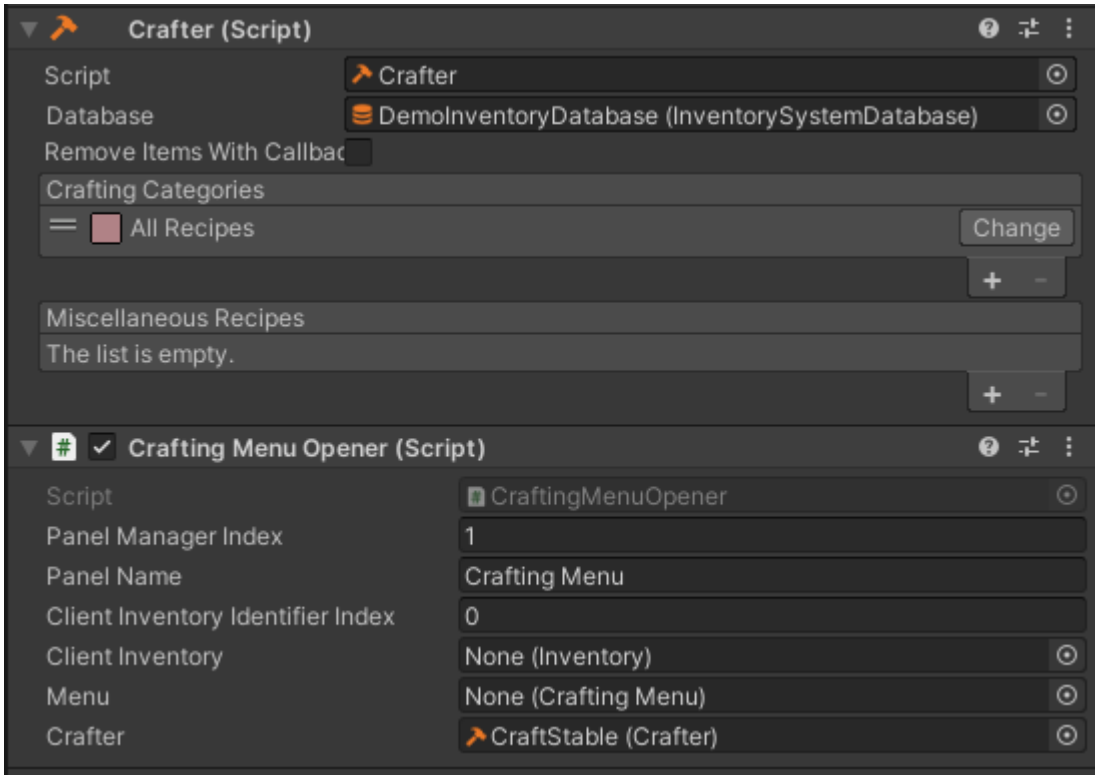
Create a Crafting Menu using the UI Designer Crafting tab. To learn more about the Crafter component go to this page.



The crafting menu uses the following components:

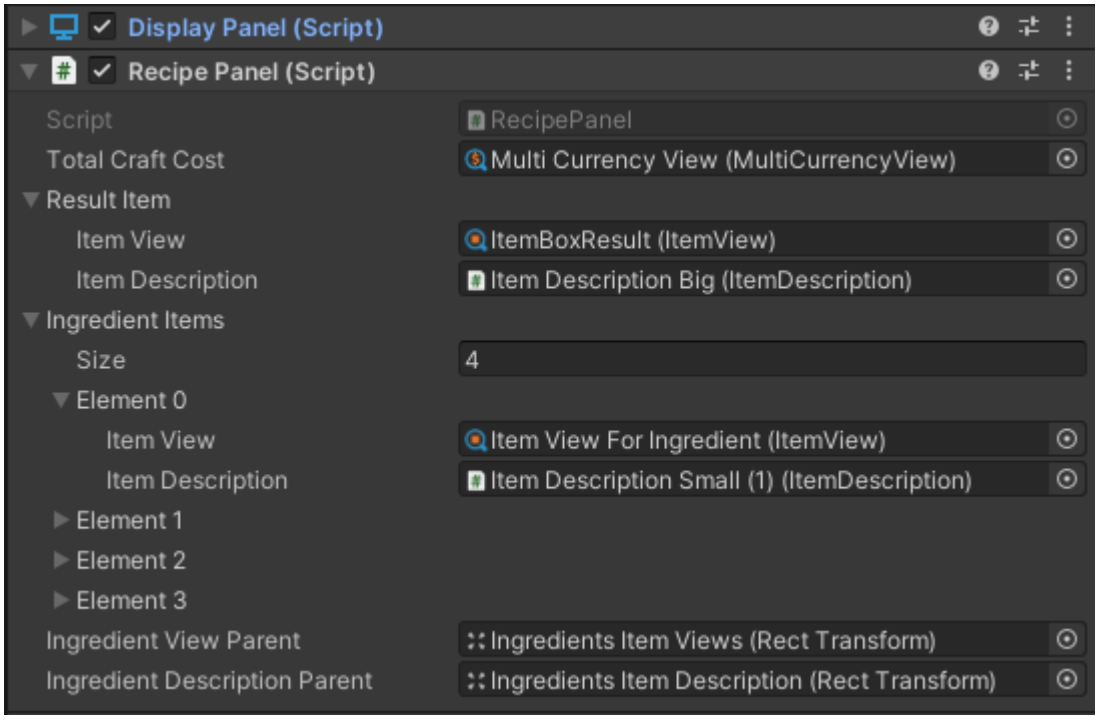
- *Inventory*: The client Inventory (the player).
- *Crafter*: The Crafter component which has the list of recipes, and the crafting processor logic.
- *Crafting Recipe Grid*: A grid of Recipe Views used to select the recipe to craft.
- *Recipe Panel*: This panel shows the crafting output and the ingredients required for crafting.
- *Quantity Picker Panel*: Lets the user choose the amount of item to craft.

Similar to the Shop Menu, the Crafting Menu can have the Crafter within the UI or it can be bound to the Crafting Menu through code. The easiest way to do so is to use the “Crafting Menu Opener” component.



Recipe Panel

The Recipe Panel can be edited with ease in the UI Designer Crafting tab. The Recipe Panel is used to display the item to craft and it’s ingredients/cost.



Menu Character

The demo shows a character next to the inventory in the main menu. There are a few ways this could be set up from scratch. Here are two options:

Option 1: Make a UI Character by Duplicating your character

This character visually shows the items equipped, without having any game play logic. This can easily be set up using an Equipper script.

The steps are as follows:

1. Duplicate your character and remove all the scripts from it leaving only the animator. Make sure the Animator is not set to Root Motion.
2. Add an Equipper script (Avoid using the VisualCharacterEquipper or the CharacterEquipper script as those are demo only scripts)
3. Create an Item Slot Set. Right-click in the project view -> Create -> Ultimate Inventory System -> Inventory -> Item Slot Set
4. Set the Item Slot Set in the in the Equipper
5. As the UI character does not usuallly use item you may ignore the Usable Item Prefab Attribute Name. Simply set the Prefab Attribute Name, which should point to the prefab that you will wish to see when the character equips a weapon.
6. Make sure to set The Inventory to monitor and set the Item Collection to point to the equipped items.
7. Finally set the Item Object Slots by selecting each slot in the list. you may choose whether the object is skinned or not. If not you may choose the parent transform for the item.

At that point the UI character should be set, The next steps are to display it in the UI.

1. Create a Render Texture. Right-click the project view -> Create -> Render Texture.

Assign it a reasonable size (The size you plan to use in the UI).
2. In the UI add a Raw Image game object and assign it your Render Texture. This is the image that will show your character.
3. Add a new Camera inside your UI and make it look at your UI Character you created in the first steps. Make sure the Camera and the character is very far from the game world (underneath it if possible) so that it cannot be seen. Another option is to simply put it on another Layer and Cull it from the main camera.
4. Set the camera background to solid Color, Clear Flags to Solid Color and the background alpha to 0. This will render the character with a transparent background.
5. The camera should have its Target Texture field set with the Render Texture.

With these steps you should now have a UI character displaying the equipped items.

Most of the times you character will need to animate holding the weapons, There is unfortunately no way to do this easily. You must create a custom Equipper to do this and smartly play the equip animations. A more versatile solution is planned for the future.

Option 2: Place a camera in front of your player character

1. Create a Render Texture. Right-click the project view -> Create -> Render Texture. Assign it a reasonable size (The size you plan to use in the UI).
2. In the UI add a Raw Image game object and assign it your Render Texture. This is the image that will show your character.
3. Create a Camera inside the player Character game object in your scene and rotate it such that it looks at the character
4. Assign the Render Texture to the camera Target Texture Field
5. Set the camera background to solid Color, Clear Flags to Solid Color and the background alpha to 0. This will render the character with a transparent background.
6. Tweak the Culling Mask, and Clipping planes to show the character and its weapons

This solution is quite a lot simpler than option 1 but it also comes with a few quirks. For example this won’t work for first person view characters. Also since the character being rendered is in the scene it will react to anything in the world, you won’t be able to pause the game while the menu is up or play custom animations for the UI character

ResizeableArrays and ListSlices

Resizable Arrays and List Slice are custom types for holdings set of data. If used carefully they can improve performance and productivity. There are advantages and disadvantages using these types compared to regular Lists or Arrays.

Resizable Array

A Resizable Array is a class which contains a single array. As it name suggests it allows you to resize that array. Internally the array length is only increased when necessary. If the size is reduced then the internal element count value is reduced. The Resizable Array should only be used when the size does not frequently. A List should be used if you’re unsure which object type should be used.

List Slice

A List Slice is a slice of a list. The List Slice denotes a start and end index and allows you to iterate through the elements within that start and end index. A List Slice is a struct which makes it light weight. Any object that implements the IReadOnlyList<T> interface can be converted to a List Slice. This includes Arrays, Lists and Resiazable Arrays. It is important to remember that a List Slice is not a copy of the original IReadOnlyList. If the internal list changes then the List Slice will change as well.

Events

The inventory system uses two types of events. It uses the C# event system as well as the Opsive Event Handler which is used throughout the set of Opsive assets.

Important: All of the Event Handler events can be found within the *EventNames.cs* file. Event Handler event names are static constants which are used throughout the system. The names have a particular syntax such that can be easier to undertsand what the target and parameters are. If unsure how the event should be handled check the comments in the *EventNames.cs* file or by finding where the event is being executed:

```
public const string
c_TargetObjectType_EventDescription_Parameter1Type_Parameter2Type_...

public const string c_ItemCollection_OnAdd_ItemAmount

// Global events do not have a target and have the syntax below.
public const string
c_Global_EventDescription_Parameter1Type_Parameter2Type_...
```

The code below will register, execute, and unregister from a particular event. The parameter is a simple “int”. Both the event name and the parameter type must match for the event to be listened to properly.

```
//Generic Examples

// Includes the correct namespace.
using Opsive.Shared.Events;

// Registers for myEventName on myObject with a single int parameter.
When the event is sent the MyFunction will be executed.
EventHandler.RegisterEvent<int>(myObject, myEventName, MyFunction);

// Executes myEventName with an int parameter on myObject.
EventHandler.Execute<int>(myObject, myEventName, 7);
```

```
// Unregisters from the event.
EventHandler.UnregisterEvent<int>(myObject, myEventName, MyFunction);
```

It is important to understand that the event is executed with a list of parameters (can be none), the listeners (when using RegisterEvent) method handler needs to match the parameters type exactly. If not an error will occur.
Therefore the Event Handler Execute function defines the pattern that should be used for that event name. You should never execute the same event name with different parameter types.
You may create your own events by following the examples above, or you may simply listen to the events we execute throughout the system.

Examples Specific to the Inventory System

```
//Examples Specific to the Inventory System
The code below will register, execute, and unregister from a
particular event.

/// <summary>
/// Register to events on awake or on start.
/// </summary>
private void Start(){

    // Listen to the inventory add item event.
    // The first parameter is the target, in this case the inventory.
    // The second parameter is the event name which can be found in
the EventNames static class.
    // The third parameter is the function that will be called when
the event is executed.
    // Note the function must have the parameters ItemInfo and
ItemStack as specified in the '<>' of the event execution. In this
case it is executed in the ItemCollection class
    EventHandler.RegisterEvent<ItemInfo, ItemStack>(m_Inventory,
EventNames.c_Inventory_OnAdd_ItemInfo_ItemStack, HandleOnAddItem);

    // Listen to the inventory remove item event.
    EventHandler.RegisterEvent<ItemInfo>(m_Inventory,
EventNames.c_Inventory_OnRemove_ItemInfo, HandleOnRemoveItem);

}

/// <summary>
/// Handle the item being added to the inventory.
/// </summary>
/// <param name="orignItemInfo">The origin of the item that was
added.</param>
```

```
/// <param name="addedItemStack">The item stack where the item was
added.</param>
private void HandleOnAddItem(ItemInfo orignItemInfo, ItemStack
addedItemStack){
        if (addedItemStack== null) { return; }
        //use the orignItemInfo and addedItemStack to know where the
item came from, where it was added, how much was added, etc...
}

/// <summary>
/// Handle the item being removed.
/// </summary>
/// <param name="removedItemInfo">The removed Item info.</param>
private void HandleOnRemoveItem(ItemInfo removedItemInfo){
        //The removedItemInfo contains the amount that was removed in
Amount and the amount that is still left in ItemStack.Amount
}

/// <summary>
/// Make sure to unregister the listener on Destroy.
/// </summary>
private void OnDestroy()
{
    EventHandler.UnregisterEvent<ItemInfo, ItemStack>(m_Inventory,
EventNames.c_Inventory_OnAdd_ItemInfo_ItemStack,
OnAddedItemToInventory);
    EventHandler.UnregisterEvent<ItemInfo>(m_Inventory,
EventNames.c_Inventory_OnRemove_ItemInfo, OnRemovedItemFromInventory);
}
```

Integrations

Bolt

There is no additional downloads required to make Bolt and the Ultimate Inventory System work together.

Important: Bolt and UIS both use the Antlr3.Runtime.dll. If both dlls are in the project this will cause problems, therefore one of them must be removed.
In the project view go to Opsive/Ultimate Inventory System/Plugins and remove or rename the Antlr3.Runtime.dll file.

Bolt works using reflection and therefore all available functions and types in the Ultimate Inventory System will be available to you. To make sure the graph shows the Ultimate Inventory System functions you must specify the Assemblies in the Bolt Setup Wizard.

In the Bolt Setup wizard under the Assemblies tab make sure to specify the "Opsive.UltimateInventorySystem" assembly.

In the Types tab it is recommended to add the following types:

- ItemInfo
- ItemAmount

Dialogue System

The Ultimate Inventory System is integrated with the Dialogue System allowing your inventory to be managed by the Dialogue System. The Dialogue System integration is maintained by Pixel Crushers. Please find the link to the documentation here.

You can download the integration on the Downloads page.

Opsive Behavior Designer

The Ultimate Inventory System is integrated with Behavior Designer allowing you to use Behavior Designer to manage your inventory. The tasks can be used using the standard Behavior Designer workflow and no extra steps are necessary. If you'd like to see any new tasks created please request those actions on the forum.

The integration can be downloaded on the Downloads page.

Opsive Character Controllers

The Ultimate Inventory System integration documentation with all the Opsive Character Controllers can be found on this page.

Playmaker

The Ultimate Inventory System is integrated with Playmaker allowing you to use Playmaker to manage your inventory. The Playmaker actions can be used using the standard Playmaker workflow and no extra steps are necessary. If you'd like to see any new actions created please request those actions on the forum.

The integration can be downloaded on the Downloads page.

Quest Machine

The Quest Machine integration allows your inventory to be managed by Quest Machine. The integration is maintained by Pixel Crushers. Please find the link to the documentation within the integration package.

You can download the integration on the Downloads page.

Input System

If your project makes use of controllers it is highly recommend that you use an asset dedicated to controller support such as Rewired, InControl or the new Unity Input System. The Ultimate Inventory System is integrated with each of these assets and the integration can be downloaded here.

The Player Input component is base component that the Unity Input and integration components derive from. By using the Player Input component any new input can be added by just swapping out the component and the rest of the code does not need to change.

The Player Input component is shared between the Ultimate Inventory System and the Character Controllers. To learn more about the Input System integration and how to set it up, go to the Character Controller integration documentation

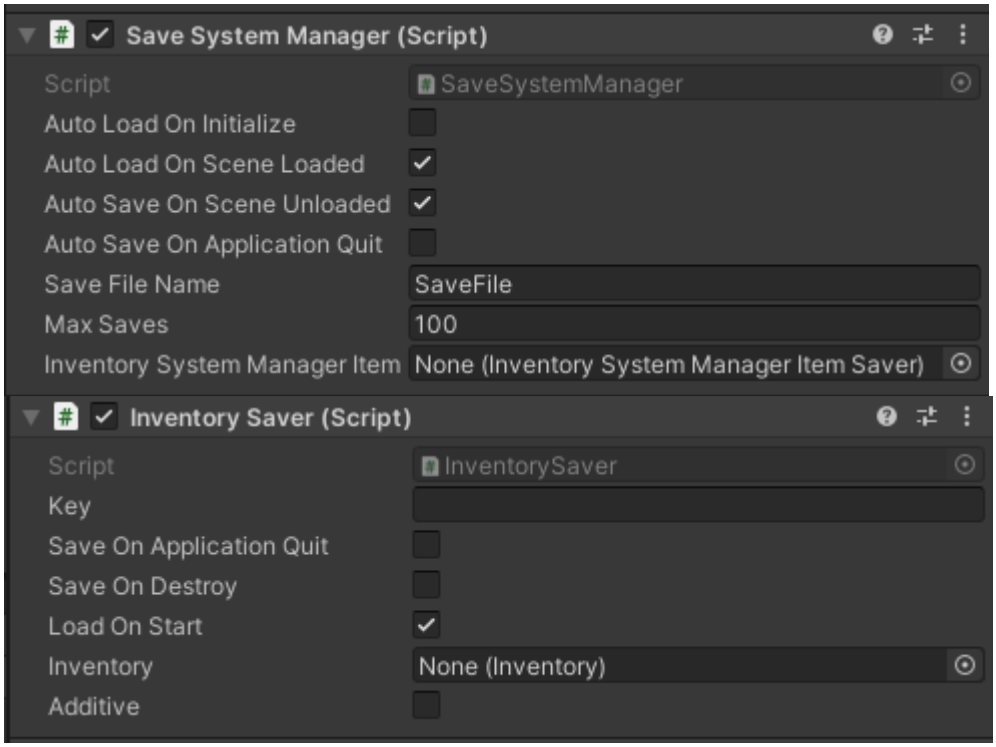
Scene Transitions

When transitioning from one scene to another you usually want your character items to carry on from the previous scene. There are many ways to achieve this. Each solution below has pros and cons, so make sure to read thoroughly to know which ones suit you best.

When transitioning scene you must choose whether or not you which to carry over the managers, player, and/or UI. If the character is not carried over between scenes you may choose to save the game before unloading the current scene and load the game when the new scene is loaded.

It is important to understand how the Save System Manager works. The Save System Manager saves and loads all Saver data. When changing the scene the Save System Manager will be initialized first. The Savers will each be initialized as the execution order defines. This means that the Save System Manager may be told to load data while the savers are not yet registered to the Save Manager. Therefore the Saver component must load the save data in its start function. This is particularly important for the Inventory Saver.

If you wish to use the *Auto Save On Scene Unload* and *Auto Load On Scene Load* you must set each saver to *Load On Start* and *Save On Destroy*. Since the Savers can be initialized after the save is loaded, they might not have loaded the save data as the Save System Manager did not know of its existence in the scene. The Scene Unloaded event is sometimes called after the GameObjects have already been destroyed, which requires the Savers to be saved before they are destroyed.



You can manually choose when to save and load the data when unloading/loading scenes. This will give you the most control over how the items saved and loaded. A coroutine can be used to wait a frame between the scene loading and the Load function call to make sure the Saver components are registered to the manager. Learn more about the save feature here.

Each Scene has Everything

This is a naive approach where each scene contains all the components you need in your game: Character, UI, Game Managers, etc. Each time you unload the previous scene completely and load the new one from scratch. Before switching to the next scene save the game, and once the next scene is loaded wait a frame to allow all objects to initialize and then load the save data.

Pros:

- Easy to set up

Cons:

- Does not scale well, make if one of the values is changed, all the scenes need to be updated.

Have a Starter Scene

With this approach none of the scenes have the managers except for the starter scene. Make sure the managers GameObject is set as DontDestroyOnLoad such that they stay when transitioning scenes. The character and UI can also be part of GameObjects to load at start and never destroy.

Pros:

- More efficient load and initialize the managers (and the inventory database) only once

at the very beginning instead of each scene.

Cons:

- The only way to test your scenes will be to load the “Start” scene before switching to the scene you wish to test.
- Coming back to “Start” scene from any level will cause the managers to be duplicated causing errors.

Smart Managers Load Component

Using a similar approach to option 2. Except each scene has a custom component that checks if the managers are loaded if not you may load a prefab or a scene with all your managers.

Pros:

- More efficient load and initialize the managers (and the inventory database) only once at the very beginning instead of each scene.
- Each scene can now be tested without having to manually load a “Start” scene.

Cons:

- Slightly less intuitive.
- Requires custom code.

You’ll need to bump the Execution Order of that component to -500 in the project settings to make sure it is called before anything else.

Here is an example of a component which will load the managers:

```
public class SceneLoaderManagerExample : MonoBehaviour
{
   [SerializeField] protected GameObject m_ManagerPrefab;
   [SerializeField] protected int m_ManagerSceneBuildIndex= - 1;

   private void Awake()
   {
       if (!InventorySystemManager.IsNull) {
           //This gameobject is only needed to load the managers
remove it once it is done.
           Destroy(gameObject);
           return;
       }

       // Instantiate a prefab of managers (Make sure to
DontDestroyOnLoad).
       if (m_ManagerPrefab != null) {
           var managers = Instantiate(m_ManagerPrefab);
           //Optional don't destroy on load here instead of within the
```

```
manager components
            //DontDestroyOnLoad(managers);
        }

        //Instead of a prefab load an additive scene with the manager
(Make sure to DontDestroyOnLoad).
        if (m_ManagerSceneBuildIndex < 0) {
            SceneManager.LoadScene(m_ManagerSceneBuildIndex,
LoadSceneMode.Additive);
        }

        //This gameobject is only needed to load the managers remove it
once it is done.
        Destroy(gameObject);
    }
}
```