Congrats on your purchase!

(it was a purchase.. right?..right...?)

1/10 (randomly chosen) reviews will receive one of my other assets for free, how's that for reciprocation :)

Report issues or request features:
https://bitbucket.org/jjahuijbregts/inventorysystemv2/issues?status=new&status=open

Core concepts

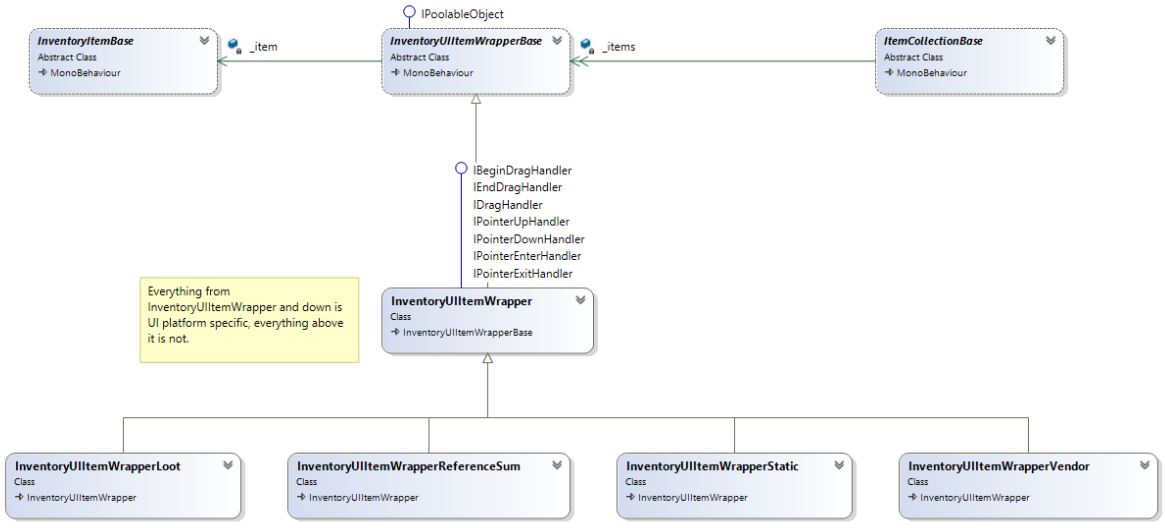
If you don’t care about the internals of the system and don’t wish to extend it with code, you can skip this section.

Wrappers

As the name implies a wrapper wraps around an object. The UIItemWrapper wraps UI code around the InventoryItemBase.

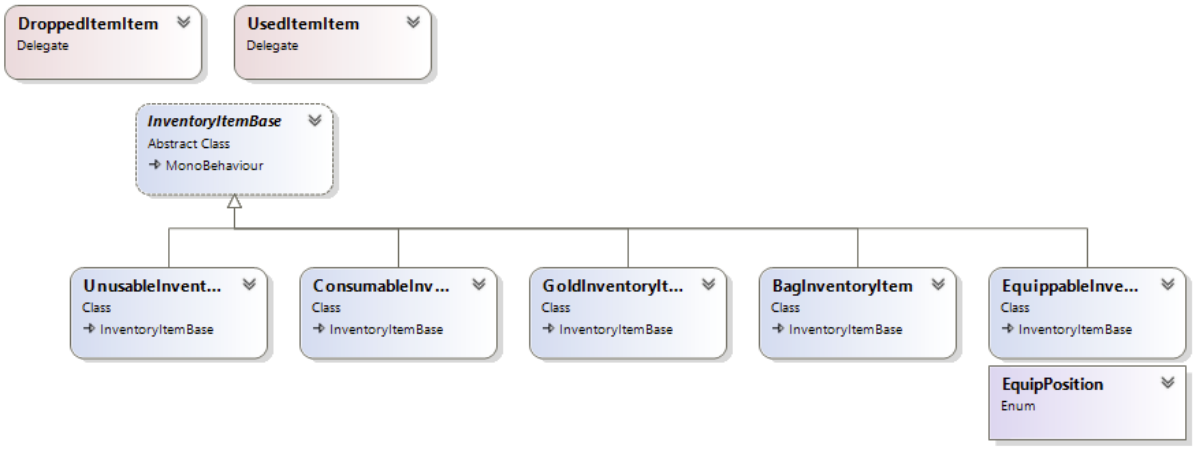
InventoryItemBase: Default abstract inventory item with no connections to UI elements.
UIItemWrapper: An Unity UI specific implementation that triggers the InventoryItemBase when UI events happen, such as a click of a button.

When creating custom item types, make sure to extend of the InventoryItemBase (or any of it’s parents), that way UI code stays separated, and will remain functional in case of a UI change.



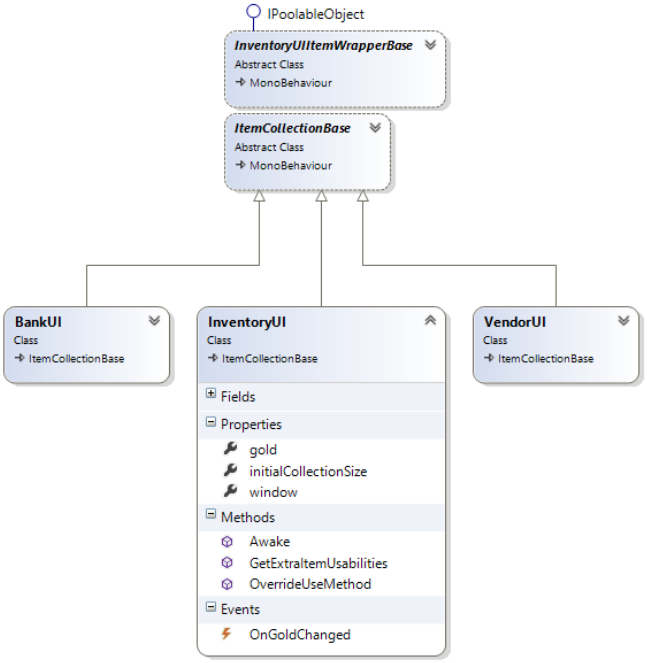
Inventory item types

By default there are 5 item types, item types are very easy to create, [more on this here.]



Collections

The inventory, bank, vendor, treasure chest are all collections. A collection manages the items inside it.



Partial classes

Almost all classes inside the InventorySystem are marked as partial. This means when you create a 2nd script with the exact same name (and namespace) the compiler will merge them together as if it were a single file / class. That way you can add your own code and methods to the system without overriding any internal stuff.
Because, when an update is released, and you've edited files inside /InventorySystem/, it will override the files in the folder, overriding your work, which is quite far from ideal.
Using partial classes outside the folder will prevent this :), so be sure to use them.

For more info on partial classes check
https://msdn.microsoft.com/en-us/library/wa80x488.aspx

The demo's

Inside the Demo/Scenes folder you'll find the scenes marked from 1. … n. These match the "tutorials" shown in this documentation.

Be sure to try out the demo's as they show of all the InventorySystem's features in full glory.

Prefab drop ins

Inside the **InventorySystem/Demos/Assets/UI/RPG_PrefabDropIns** you'll find a bunch of default windows used in the RPG demo. If you want to get to work quickly, grab some prefabs and modify them to your liking.

1. Getting started

Assuming you've imported the asset into your project and no errors occured, let's set up the project so we can get to work.

Demo projects

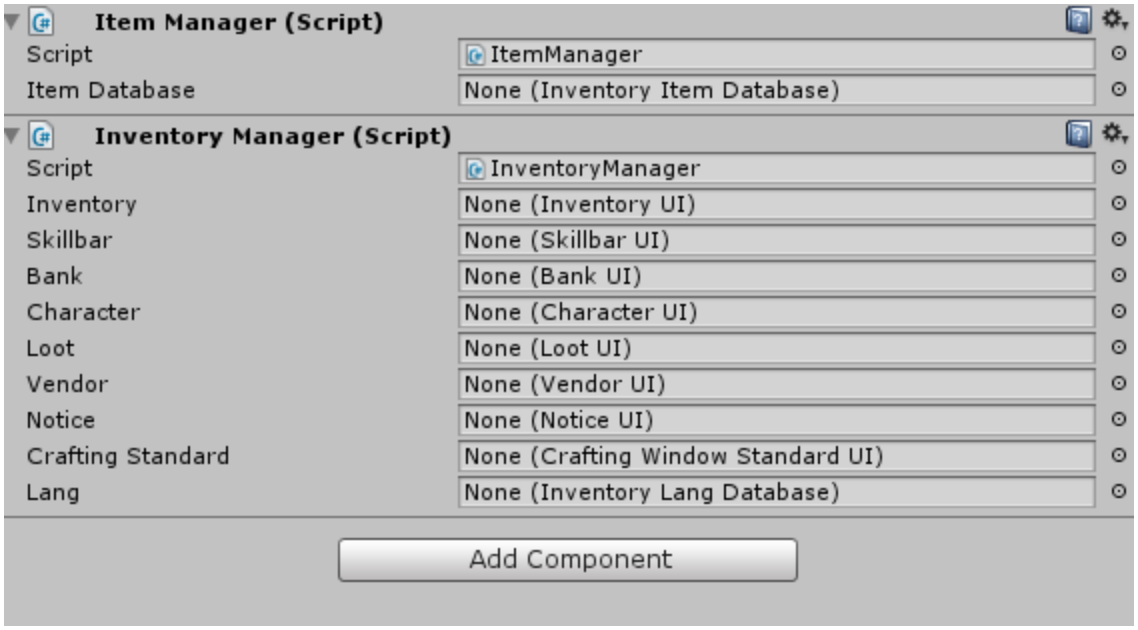
In your assets folder /InventorySystem/Demos/Scenes/ you'll find the demo scenes, these contain all features the InventorySystem has to offer, be sure to take a peek.

A clean project

Let's create a new clean scene and start from there. The first thing we have to do is create the managers. The managers handle the items, and are convenience classes to quickly add items to the inventory and other collections.

1. Create a new empty gameObject
2. Attach the InventoryManager component to it, which you can find under **InventorySystem/Managers/InventoryManager**
3. You'll be given 3 components, for now let's hide the InventroySettingsManager.

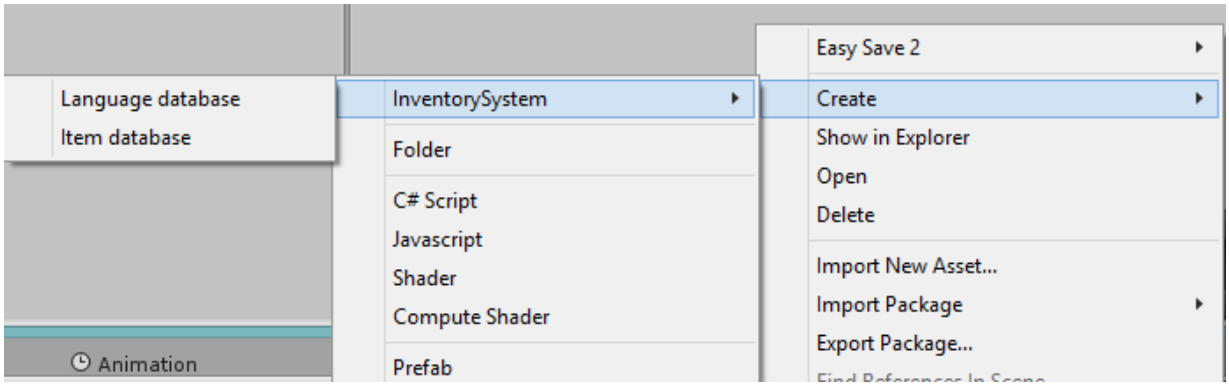
The ItemManager contains the item database which contains all the items, as you might have imagined, you can create managers per scene, and use a different database for each scene. In other words you can create multiple databases and use them in the scenes you like.



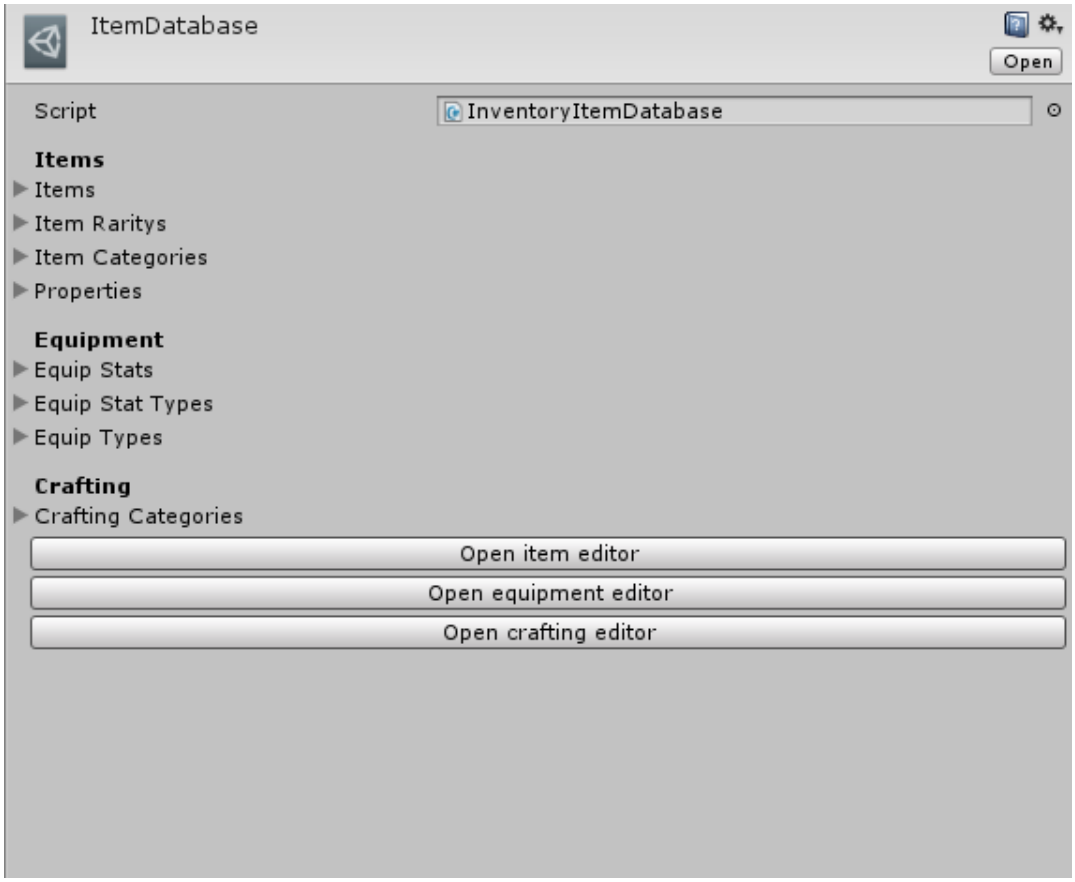
Databases

The next step is to create an ItemDatabase.

1. Go to your project window and find a place where you want to create the Item database.
2. Right click (or hit create, or go to create/InventorySystem/Item database) to create a new Item database.
3. Once created the database will be selected in your project pane.
4. **Drag the item into the ItemManager slot.**

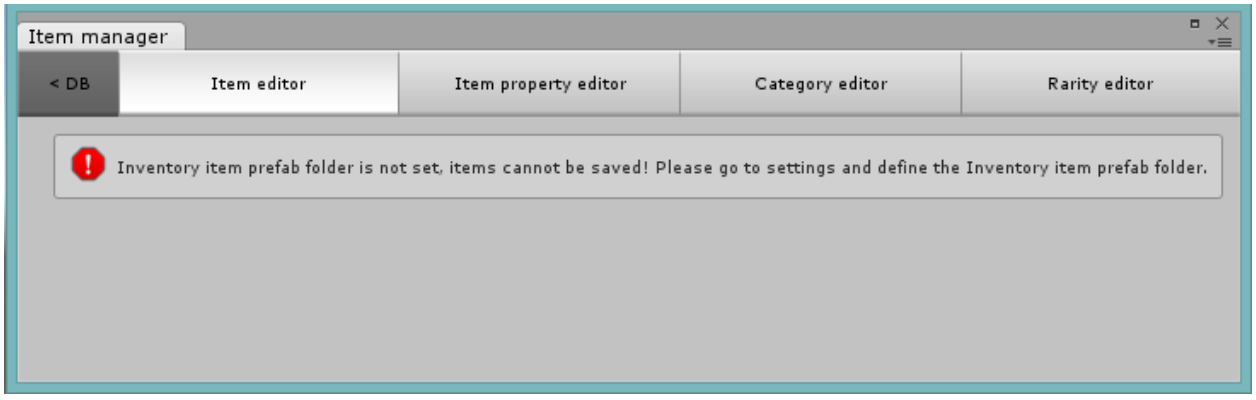


Once created, you'll be presented the following window:



Let's click "Open item editor" for now.
Note that you can also open the editors from the context menu Tools/InventorySystem/<managers>

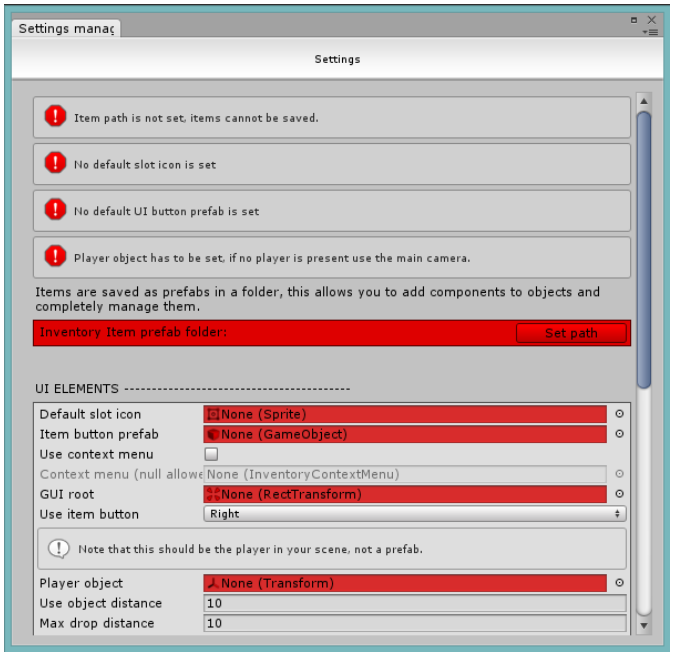
The first time we do this we'll receive an error, this is because all items are serialized inside Unity as prefabs. This approach was chosen to allow the user full control over the created items, instead of an obfuscated database you now have the ability to add custom components, models, whatever you like.



So how to solve this problem?

Well let's go to /Tools/InventorySystem/Tools/Settings in the main menu. Which will show the Inventory Settings manager, and show us even more errors :), gotta love errors.

All fields that are tinted red require immediate attention, and the system won't be able to run, if you don't fix it.



- In a new scene no Unity UI canvas will be present, so let's create one right now.
    - Go to GameObject/UI/Canvas and Unity will create a canvas, as well as an EventSystem.

  And lastly set the canvas in your settings editor.

First things first,

- The Inventory item prefab folder is where all your items will be saved that are created through the editors. Simply click the "Set path" button and **choose a location inside your assets folder** where your items should be saved.

- Default slot icon is the icon used by… default. If you use a custom design pick your own if not, pick one from the demo designs.

- Item button prefab is the default UI element, used to display items in collections. By default this is UI_Item_PFB.

- GUI root is your Unity UI canvas.

- Player object is a reference to our player, note that is not the prefab of your player, but the actual player inside the scene. For the sake of this tutorial let's use the standard asset Ethan, a character created by Unity. **You can find a prefab of Ethan in InventorySystem/Standard Assets/Characters/ThirdPersonCharacter/Prefabs/** There's also a prefab camera controller, that can come in handy for prototyping.

- Confirmation dialog is a reference to the standard confirmation window. This is allowed to be empty as long as you uncheck Show dialog when dropping item.

Other settings

- The context menu is a little menu that pops up when an item is right clicked, of course it can be triggered under any custom condition. Also useful for mobile games.

- Disable windows when dialog is open is used to disable certain windows whenever a dialog is active. This can - for example - be useful to block interaction in the Inventory while trying to unstack an item. When not disabled the user could try to unstack an item, drop the item while the dialog is still open, and continue with the unstack action.

We should now be able to run the project without any problems or errors occurring, however no Inventory is visible, what gives? Well, we obviously have to create one first.

To make your life a little simpler **I've created some prefabs that you can drop in directly** and modify to your liking. **They can be found in InventorySystem/Demos/Assets/UI/RPG_PrefabDropIns**

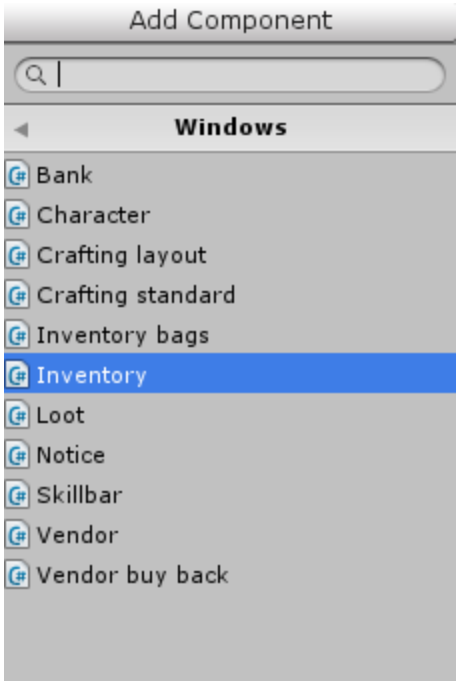
However for the completeness of this tutorial let's start from scratch and create our own custom Inventory window.

2. Creating a custom Inventory window

Let's start by creating a new UI panel inside the previously created UI canvas.

Your structure should look something like this (order doesn't really matter right now).

Alright, now let's add an Inventory component you can find it under **Add component/InventorySystem/Windows/Inventory.**



The Inventory UI

The InventoryUI script inherits from the ItemCollectionBase class, which is the default collection class. **All your collections have a collection name, this name is used to identify the collection, so make sure it's unique, that's up to you to manage.**

The Item Button prefab, can be used to override the default Item button prefab defined in the settings. When left empty the default will be taken from the settings.

The sort button sorts the collection and re-stacks wherever necessary, of course the sorting behaviour can be modified. Leave empty if you have no sorting button.

The Inventory extender collection is a collection used to extend the inventory with bags that can be equipped / unequipped to a special collection. Leave empty if you don't want to use inventory extending.

The Is loot to Inventory checkbox indicates if items should be directly looted to this inventory when picked up. You can create multiple inventories (more on this later) and restrict them to your liking.

Use item move to bank moves an item to the bank if the bank window is open and the user "uses" an item. By default this would be right clicking the icon. If disabled the user will have to manually drag icons to the bank.

Use item sell sells an item whenever the vendor window is open and the item is "used". By default this would be right clicking the icon. If disabled the user will have to manually drag the icon to the vendor window.

Loot priority defines the priority of the collection. For example when there are 2 inventories Inventory a with a priority of 10 and Inventory B with a priority of 80, all items will be stored in B (as long as allowed by restrictions)

Initial collection size are the amount of slots created on the start of the game, this is the amount of slots you'll have in your inventory.

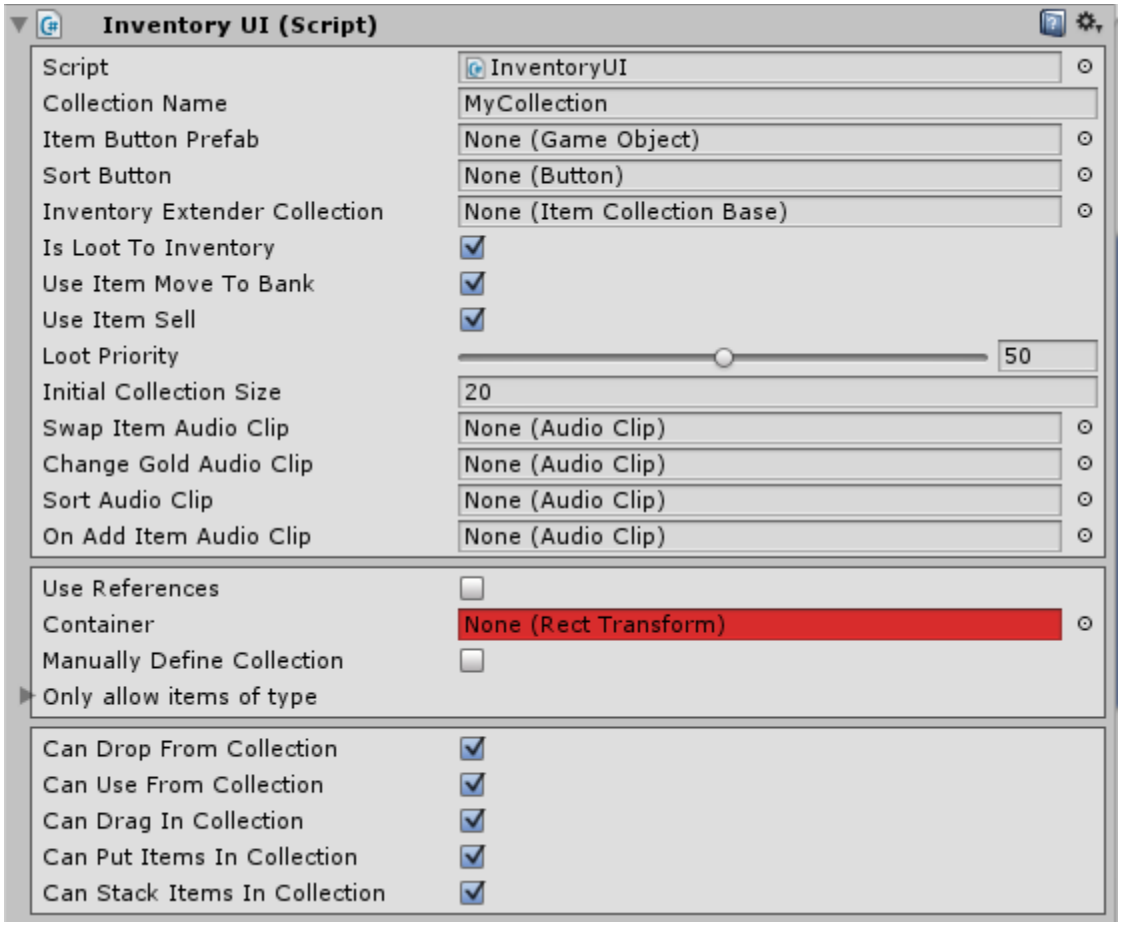
Use references is a default collection variable, when Use references is enabled items will not be placed inside the collection but a reference to that item will be made. For example the skill bar, items are not placed in it, just references to said item).

The Container is where you're UI Item wrappers (visual UI elements) are stored, this field (as you can see) is required.

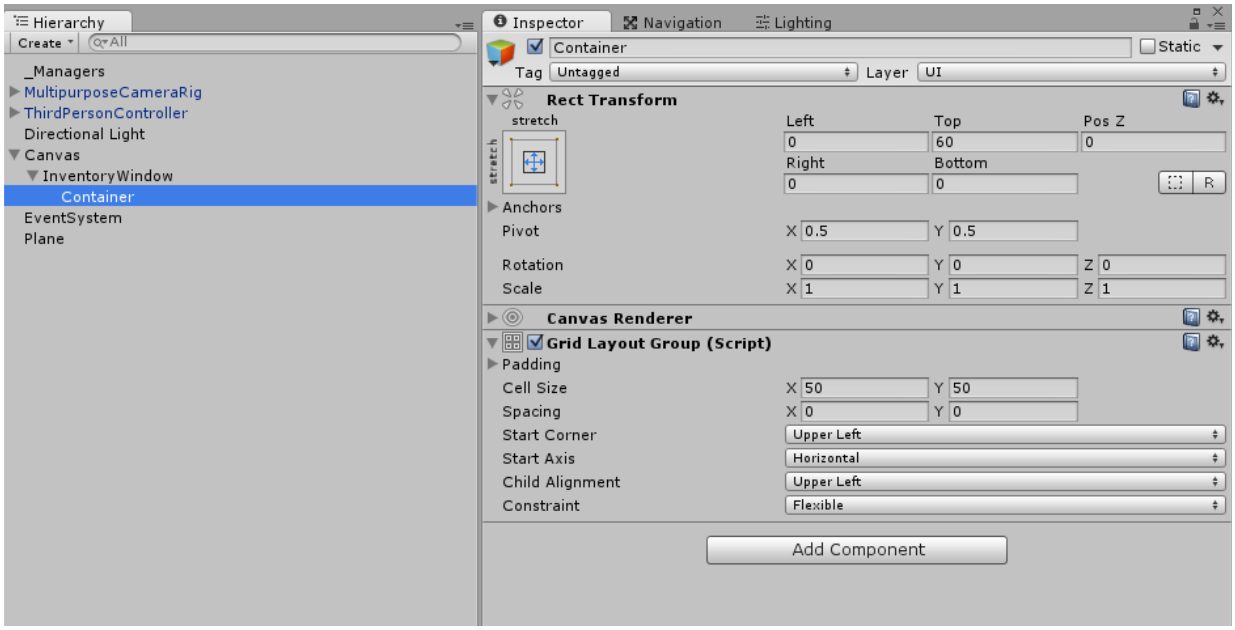
Only allow items of type is a useful feature that allows you to restrict an inventory (or any other collection) to only allow items of a certain type. For example a quest bag, that can only hold items that are quest related, crank up the priority and all your quest items will auto. be stored in a quest bag.

Can drop, Can use are quite self explanatory.

Manually define collection can be used if you don't want to generate items on start. When enabled initial collection size will be ignored and it will be up to you to set the items. This can be useful if you know how many slots your collection will have, and wish to add custom behaviour to each slot (Character equipment for example).



Since we have only 1 required field, let's create a container first.
For this tutorial I'll use a grid layout, but you can use any layout group you like, even write your own if you want to get creative.



Assign the container to the InventoryWindow, and lastly assign the InventoryWindow to the InventoryManager. And we're ready to give it a test drive (assuming you didn't forget to assign the Item button prefab in the settings).

As you may have noticed a 2nd component was added to the InventoryWindow gameObject when we've added the InventoryUI component.

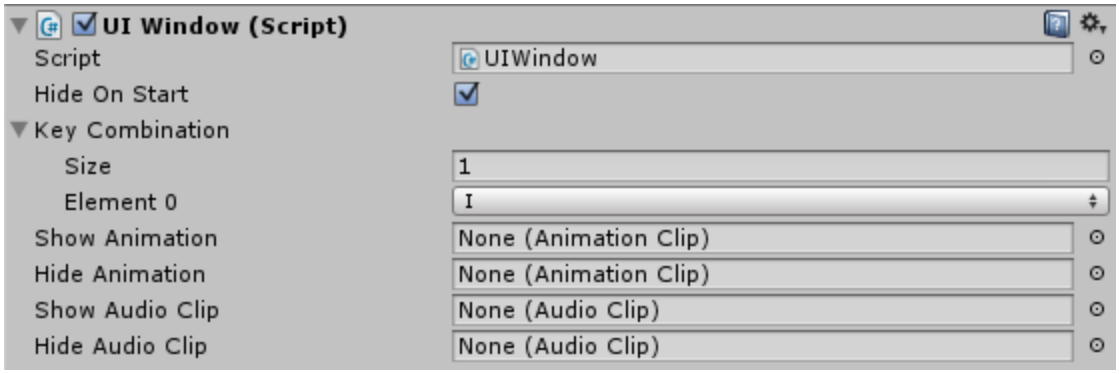
Note:

When testing this setup out, and you aren't able to use an item or drop it, check your inventory window's settings, make sure you enabled item usage and item dropping.

The UI Window

The UI Window is responsible for the management of windows as well as animating them. Optionally you can define a key combination to trigger the window, for example “i” to open the Inventory window.

When assigning animations, don’t forget the controller on the Animator component.



Draggable windows

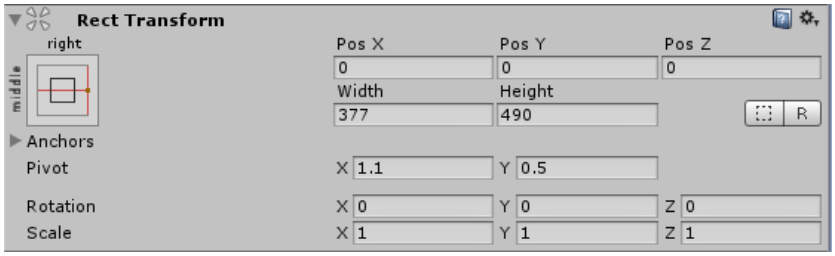
And last but not least, we can also make our windows draggable, simply go to Add component/InventorySystem/UI Helpers/Draggable window.

And that’s all there’s to it, we should now a have working Inventory, but no items yet, so for the next chapter we’ll dive into the item creator for a bit.

Positioning your windows

When the game starts windows are reset to position 0,0. It does this to allow you to design windows outside the viewport, so that you don’t have to throw them on 1 large pile inside the viewport.

You can manipulate the window’s position using the pivot x and y coordinates.A X pivot of 1.1 means 10% offset on the right side of the window. While -0.1 would mean 10% offset on the left.

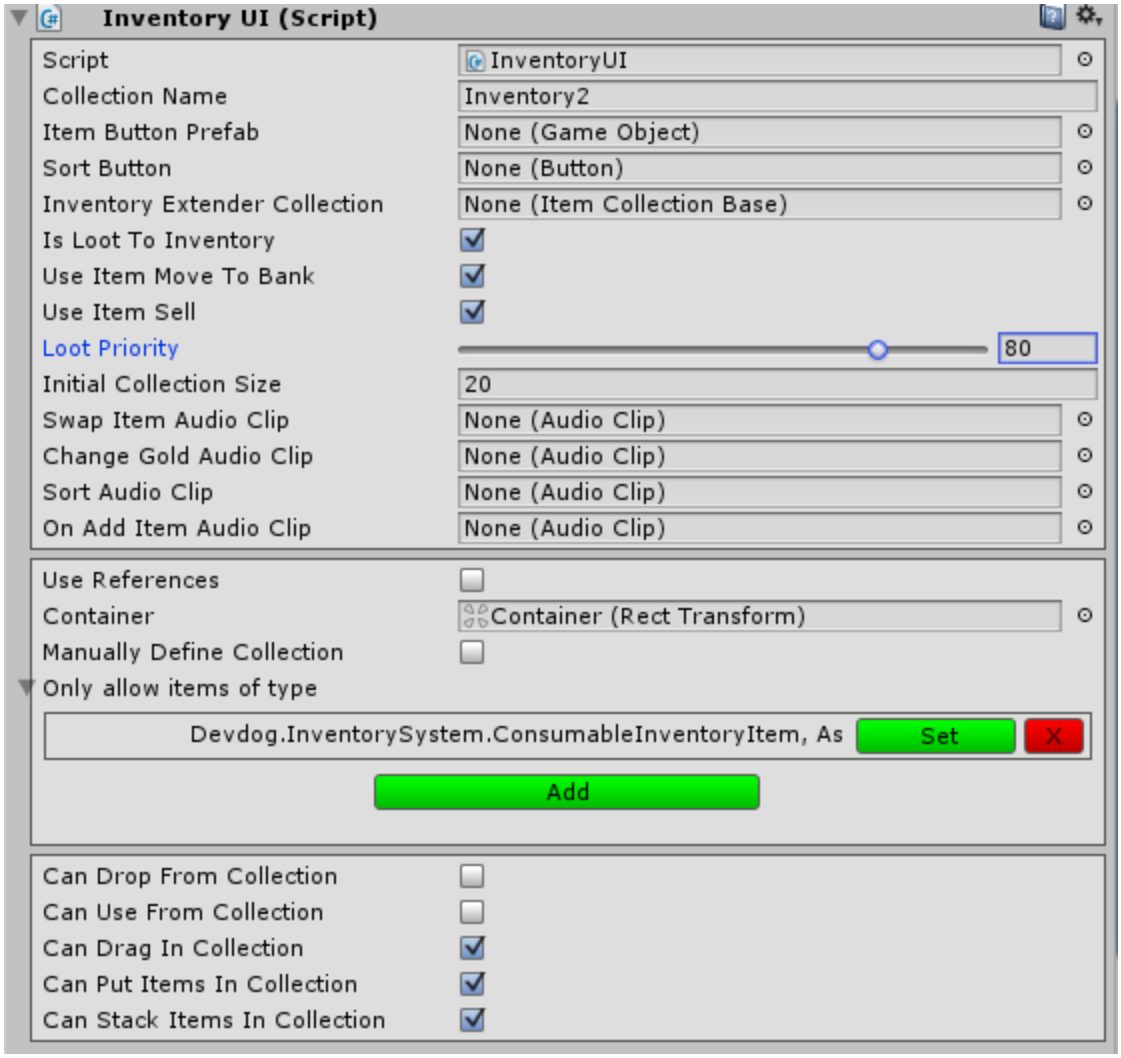


Multiple inventories

Just for fun, and to show off this awesome features, you can create multiple inventories. Simply duplicate your inventory window (and don't forget to assign that unique collection name).

All inventories are by default "Loot to collections" this means that when an item is looted, it can be placed in the collection.

Let's set the Only allow items of type to consumable and the priority to 80. Now, when an item is looted - that is consumable - it will be placed directly into the Inventory2 collection. Pretty cool huh?

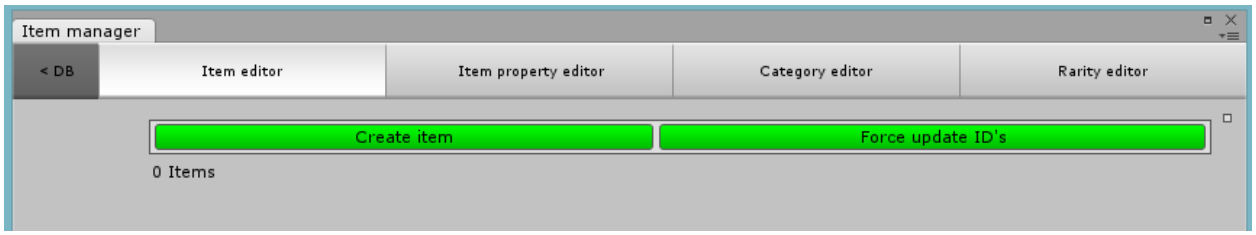

Inventory UI (Script)
Script
InventoryUI
Collection Name
Inventory2
Item Button Prefab
None (Game Object)
Sort Button
None (Button)
Inventory Extender Collection
None (Item Collection Base)
Is Loot To Inventory
Use Item Move To Bank
Use Item Sell
Loot Priority
80
Initial Collection Size
20
Swap Item Audio Clip
None (Audio Clip)
Change Gold Audio Clip
None (Audio Clip)
Sort Audio Clip
None (Audio Clip)
On Add Item Audio Clip
None (Audio Clip)
Use References
Container
Container (Rect Transform)
Manually Define Collection
Only allow items of type
Devdog.InventorySystem.ConsumableInventoryItem, As
Set
X
Add
Can Drop From Collection
Can Use From Collection
Can Drag In Collection
Can Put Items In Collection
Can Stack Items In Collection

3. Item creation (Editor)

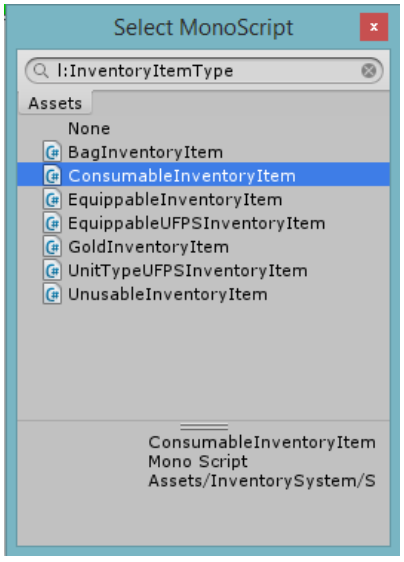
Let's open our Item editor by going to **Tools/InventorySystem/Item manager** if you've configured your settings right, no errors or warning should be shown.

Item editor

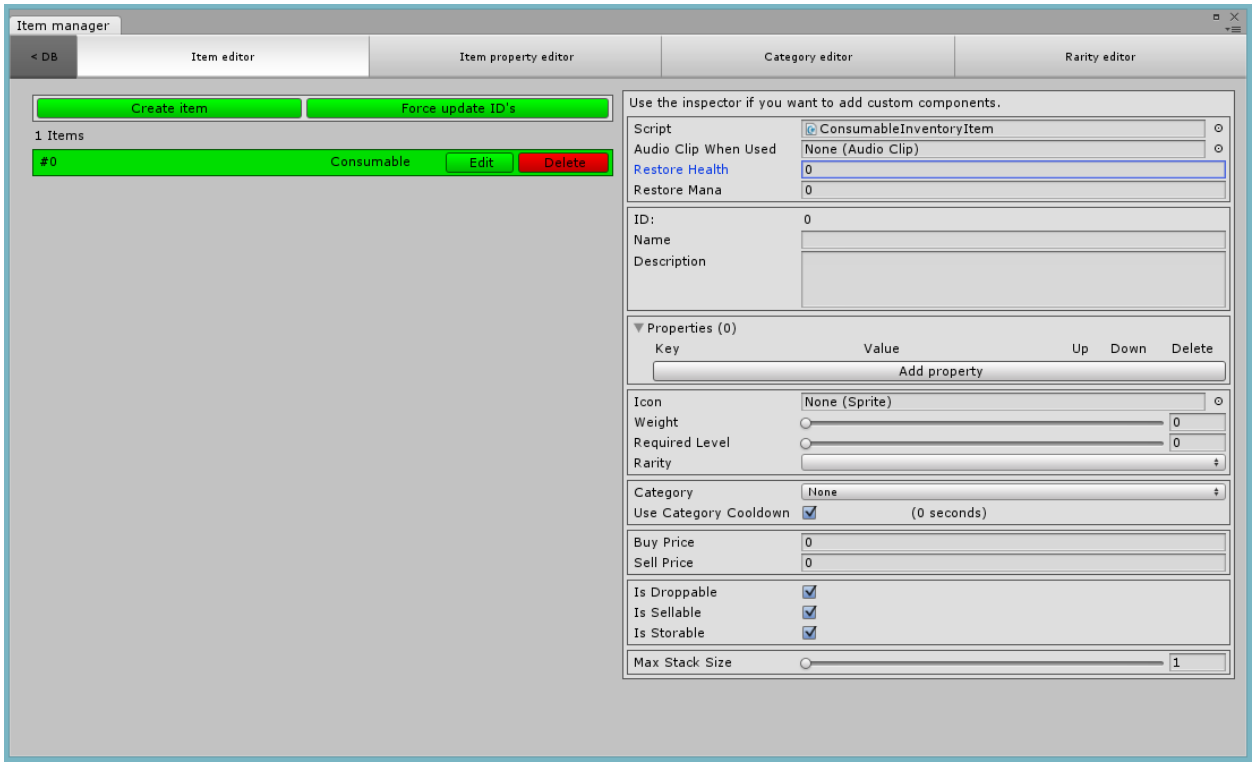
Let's start by creating our first item, by clicking the Create item button, once we do an object picker will be shown. Inside the object picker we can choose our item types. Every item type has a different behavior or purpose, for example consumable items are used to consume and reduce 1 in stack size when used, while weapons will equip to the character screen.



Let's grab a ConsummableInventoryItem for now, and configure it.

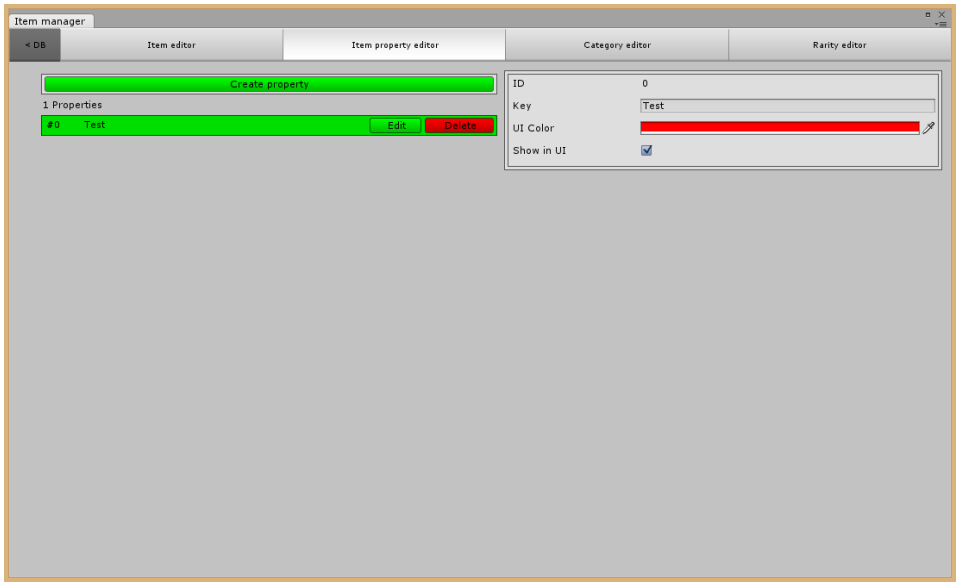


Once created the item will show up at the bottom of the list, click the edit button and a editor will appear on the right. As you might have noticed, the item will also be selected in your project folder, and shown in the Unity inspector, inside the inspector you can add custom components, change the model, etc.



Item property editor

The property editor can be used to add your own custom properties to an item, simply click the Create property button, configure it to your liking. Once done you'll be able to add properties to the Item editor's items.

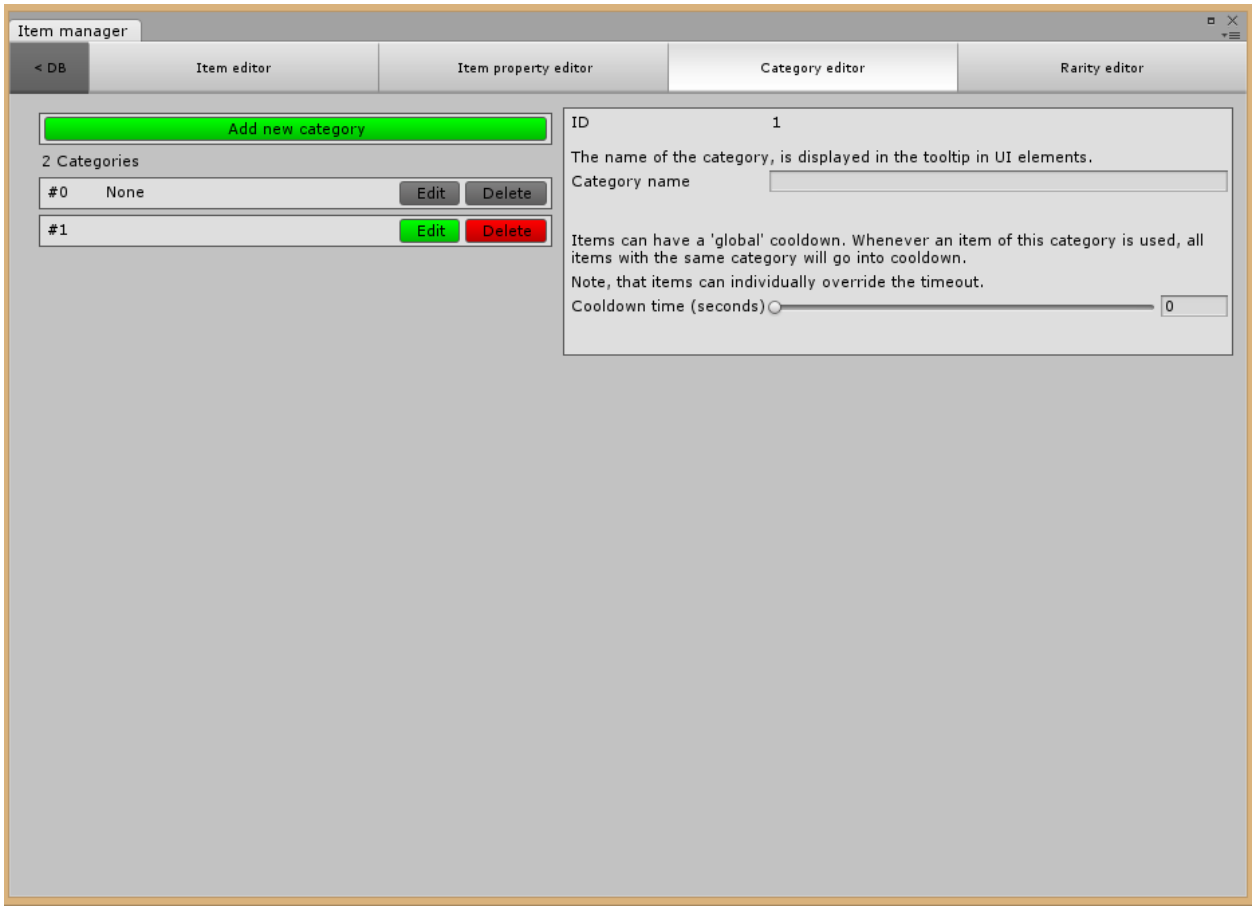


Category editor

Item categories are basically sub-categories, as we already have the item type when creating an item (remember?). For example, an equippable item is the item's type and defines it's custom behavior (equip / unequip), the category defines whether it's cloth or chainmail, this does not directly alter its behavior.

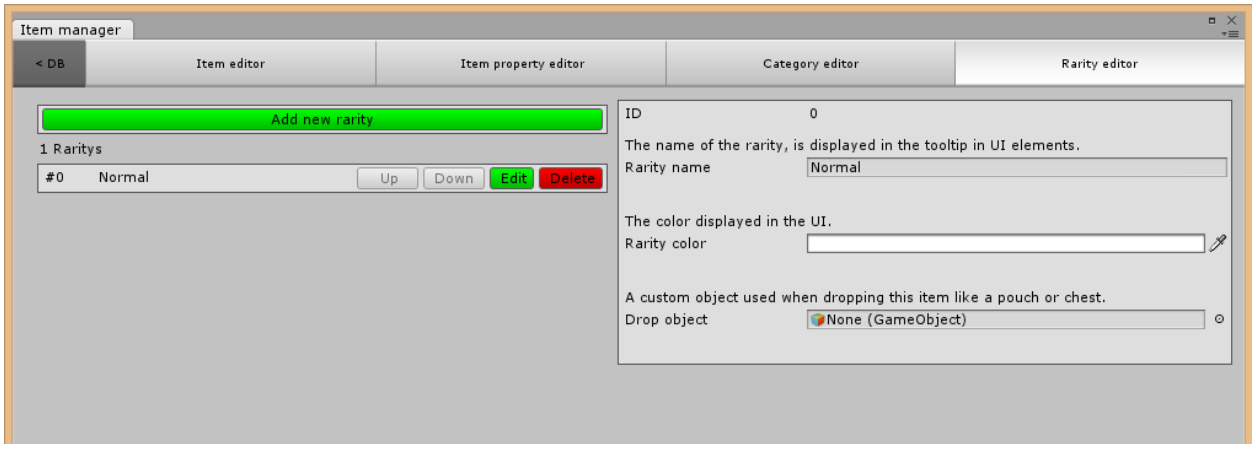
When creating a category you'll notice that there is a cooldown slider. This can be used to create a global cooldown for a given category. All items sharing this category will also go into cooldown.

Note that items can override it's category cooldown behavior in the Item editor tab.



Rarity editor

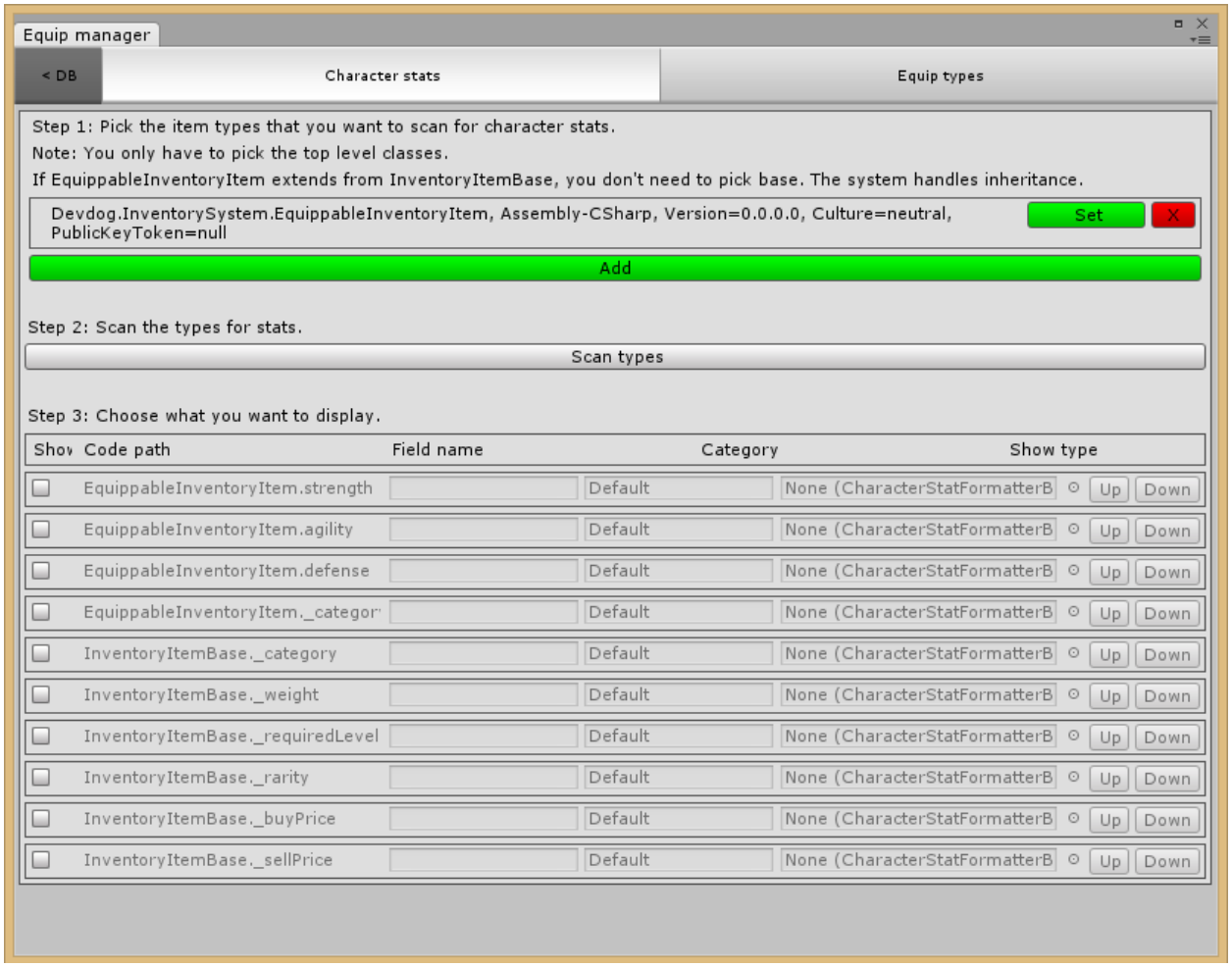
Rarities are commonly used in traditional RPG games like WoW, Guild wars 2, you name it. This does not define any behavior and is currently just used as color inside the UI elements. You could however generate particle effects or glows based on the rarity using a custom component.



4. Equipment

Character stats

The equipment system is very flexible and can be used for almost any equip / attachment system. Character stats are defined by choosing one or multiple item types, using the editor you can pick which stats will be calculated and ultimately showed.

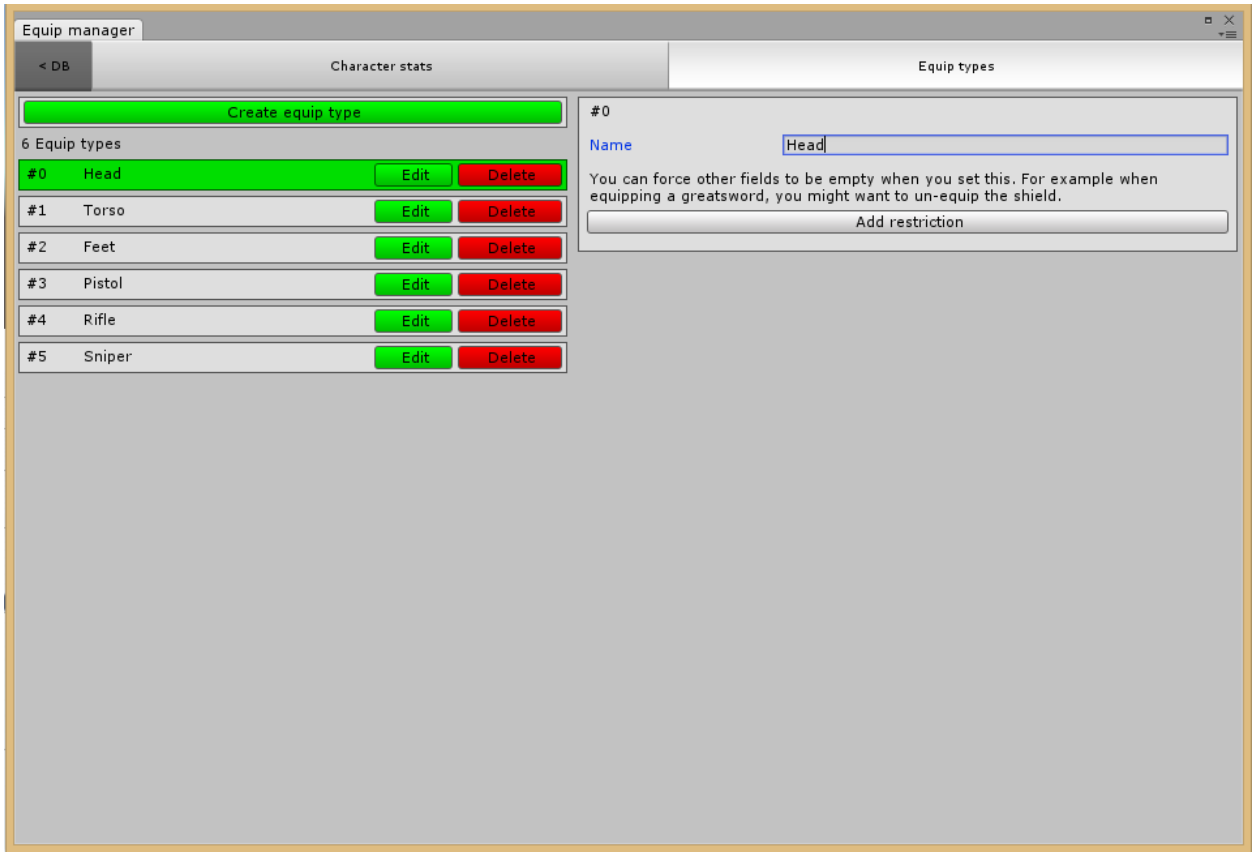


Let's add the type EquippableInventoryItem to the character stats in step 1. Next click the Scan types button, and a list of available stats will show up at the bottom.

Use the toggle box on the left to choose which stats should be calculated, alternatively you can write your own custom formatter to create custom calculations (default is sum).

Equip types

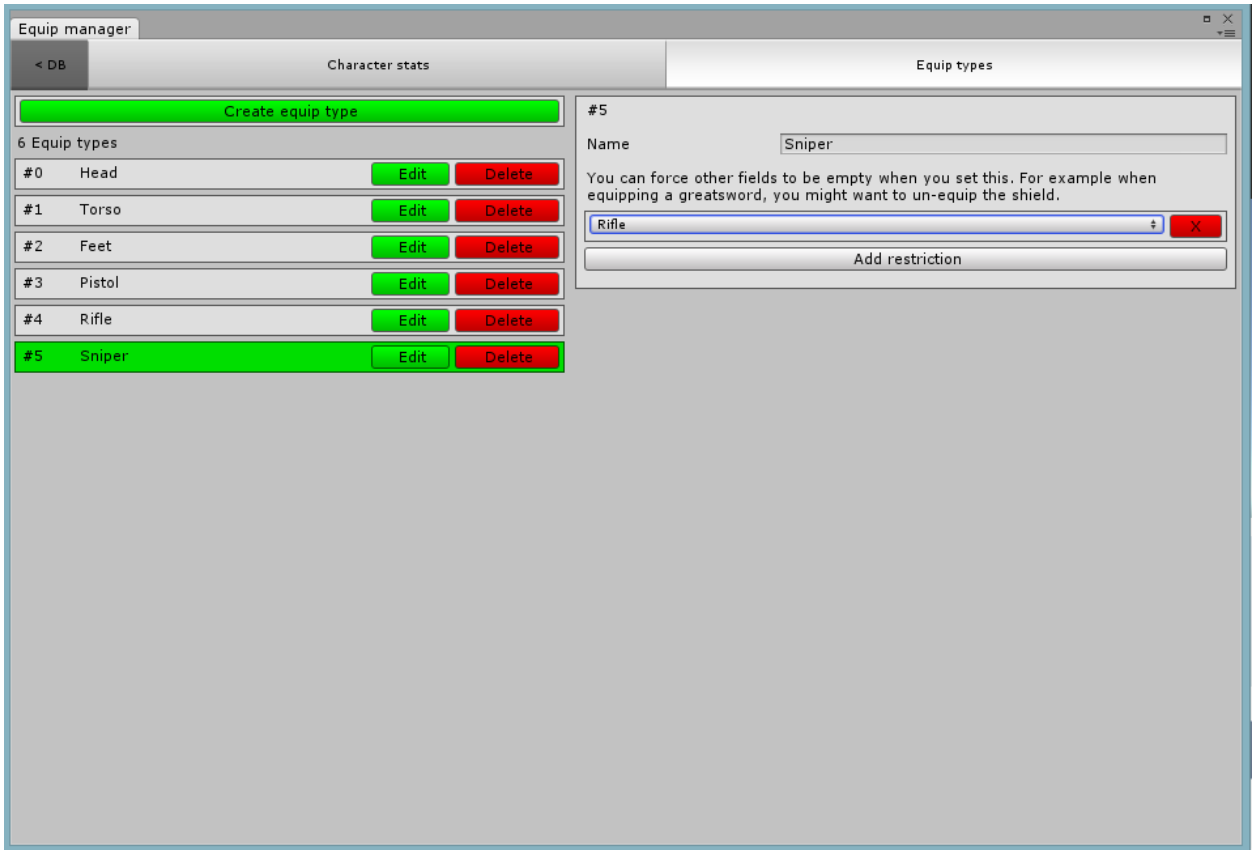
Let’s create some simple equip types for a survival game. Note that these are equip types, not locations (we’ll get to that soon). An equip type is every “type” of potential equipment. For example a bow, handgun, sword and shield might all equip to the same slot, yet are different types.



Restrictions

You can apply restrictions per item type, **a restriction makes the item type “incompatible” with another item type.**

For example, as shown in the screenshot below, we can make the sniper and the rifle incompatible, when the user equips a sniper, and tries to equip a rifle as well, the sniper will auto. be un-equipped, after all they’re not compatible :).



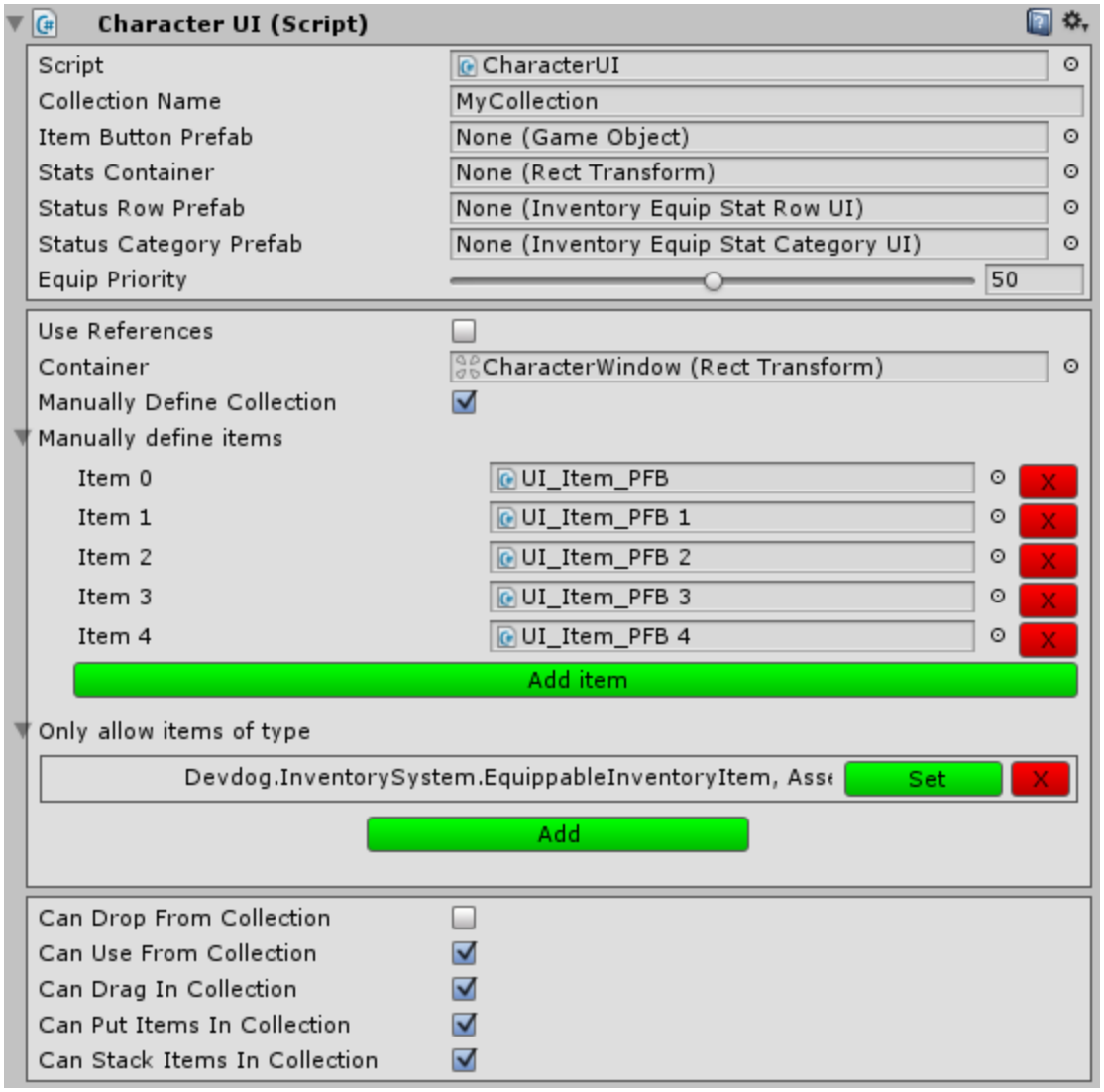
Alright, now that we’ve defined the equip types, let’s create the interface.
For the sake of this tutorial we’ll use the default item wrapper prefab, however you can of course create your own custom one.

First create a new panel, if you don’t know how click here, and assign the CharacterUI component to it, you can find it under **/InventorySystem/Windows/Character**

Manually defining the collection

Because we know how many equip slots there will be, and want to manually define the layout we can define the collection by ourselves.
Only allow items of type allows us to limit this collection to a certain time, equippable items in this case, after all equipping a consumable strawberry would be rather odd… even by my standards.

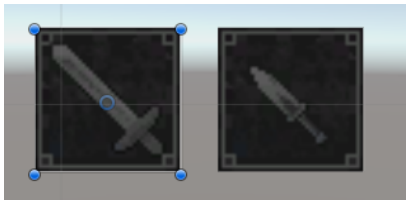
Note that at the bottom, I disabled Can drop from collection, as this likely wouldn't be desired behavior, as well as Can stack items in collection, as we don't want our equipment to stack.

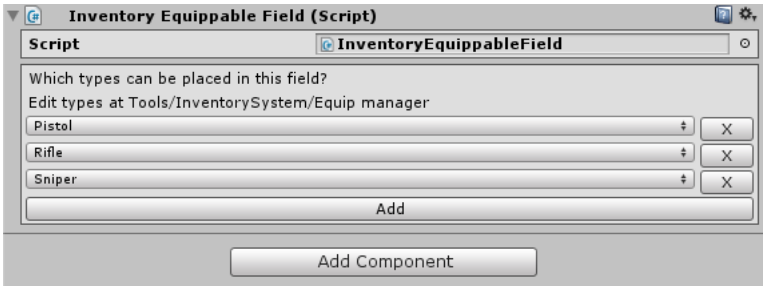


Next, we have to create the items and assign them to the character window. As I stated earlier, we'll be using the default UI item wrapper, which you can find in your project folder at **/InventorySystem/Demos/Assets/UI/UI_Prefabs/UI_Item_PFB.**

Once you've defined your layout using the UI item wrappers, select all of them and add the "InventoryEquippableField" component, which you can find under **/InventorySystem/UI Helpers/Equippable field.**

The InventoryEquippableField defines a location where a given type can be equipped. For example, you might have a pistol, rifle and sniper, that can all be equipped in the same slot, however the player has 2 weapon slots, so can therefore equip 2 weapons. (Check the screenshot below, as I suck at explaining things…)





Define all equip slots, assign them in the character window. Of course we'll need to create some items to test this out, but luckily you know how to do this, right? When creating the item, do make sure it's of type Equippable.

Getting an error when you start?

I for one, forgot to assign the character window to the InventoryManager.

5. Bank & Physical triggers

Now that we know how to set up the Inventory system, we actually also know how to set up (almost) any other collection, as they are almost all the same. However some collections require a physical trigger, such as a bank. A bank is represented by a vendor, or specific location, when we leave the location, the bank will close and won't be re-openable unless we're close enough.

First things first, let's create a bank window, easy as pie, the steps are the same as creating an inventory, except we won't be adding the Inventory UI component, and will instead use the Bank UI component, which you can find under **/InventorySystem/Windows/Bank.**

Now that we have our bank window created (and assigned in the InventoryManager, right?), let's create the trigger to open the window. This can be any model, character, animated blob you like it to be, but I'll use a sphere, feeling creative today :).

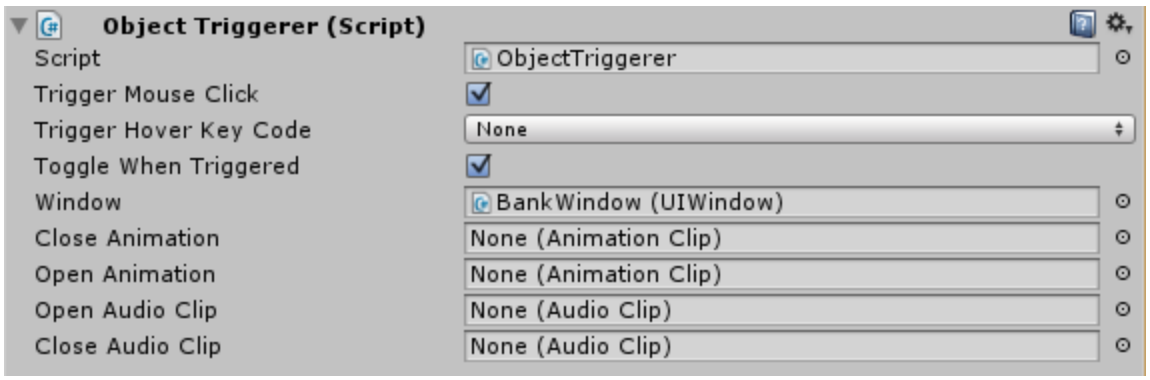
Grab your trigger object and add the **ObjectTriggerer** component, which you can find under **/InventorySystem/Triggers/Object Triggerer**

The Object Triggerer can be used in 2 ways, directly handling an UI window, or to only fire events, so that another component can handle the job for it. Certain triggers, like the vendor for example, require extra data (the items to sell) and are therefore split from the ObjectTriggerer.

Trigger mouse click triggers the window when clicked.

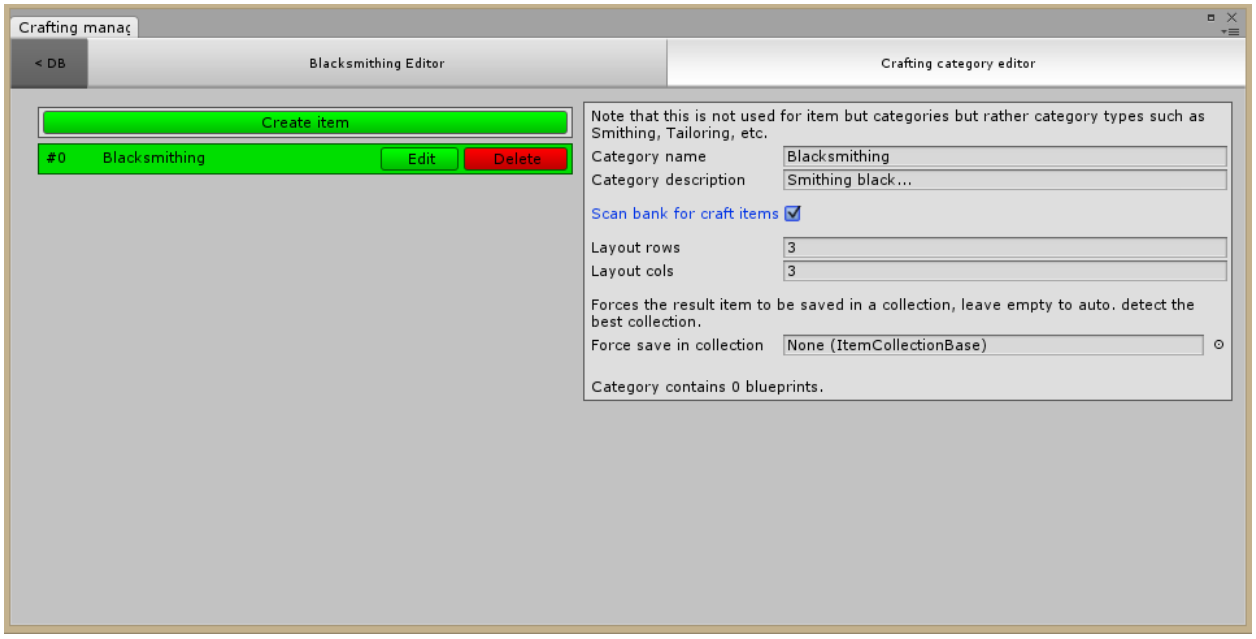
Trigger hover key code triggers the window when the object is in the center of the screen and the keycode is pressed (useful for first person games).

The animations and audio clips are for the object that triggers the window (such as a NPC), not the window itself.



6. Crafting

The crafting manager has categories that allow you to create any time of “crafting”, be it cooking, blacksmithing or leatherworking.
Let’s create a blacksmithing category, and start to create some blueprints.



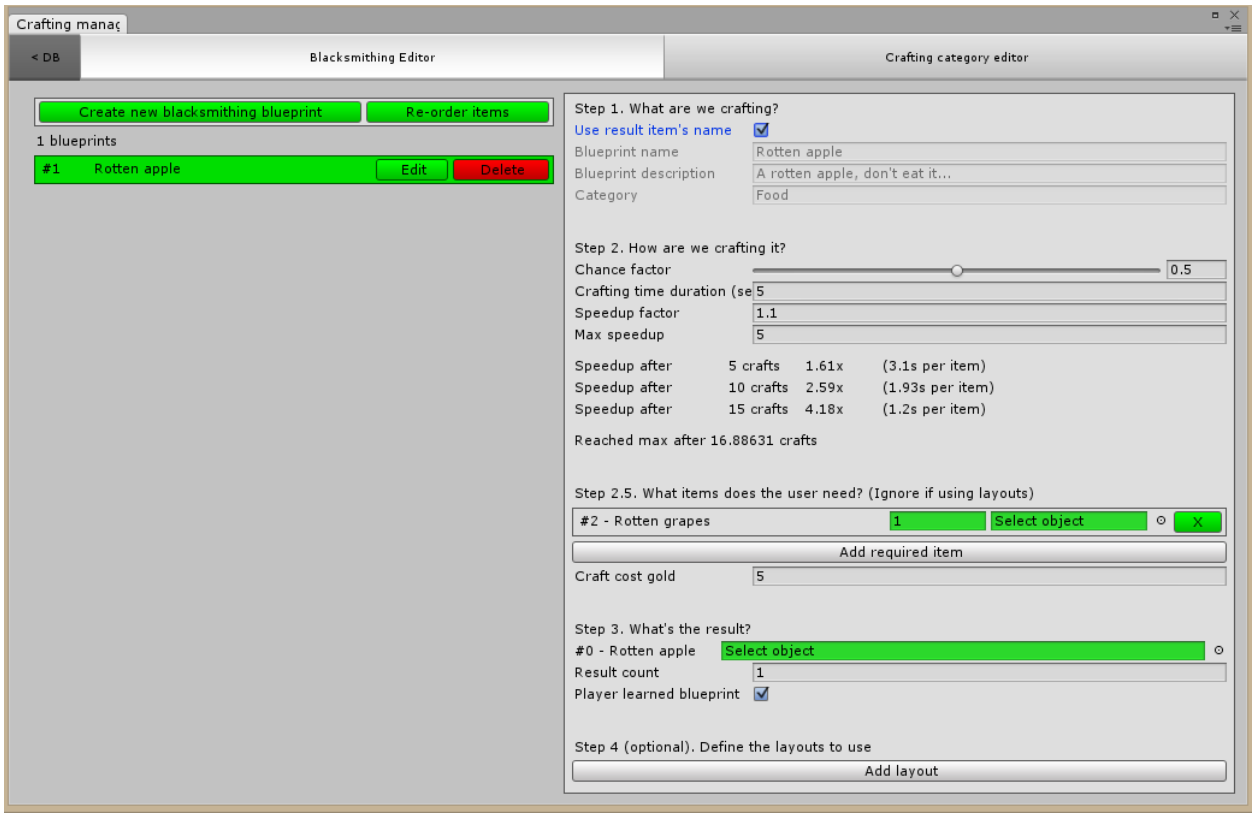
Blueprints

By default the result item's name - the item received after a successful craft - will be used as the blueprint name, but you can of course, like always, configure this.

The chance factor indicates how likely the craft is to succeed, a craft chance of 0.5 has a 50% chance, and a chance factor of 1.0 has 100% chance of succeeding.

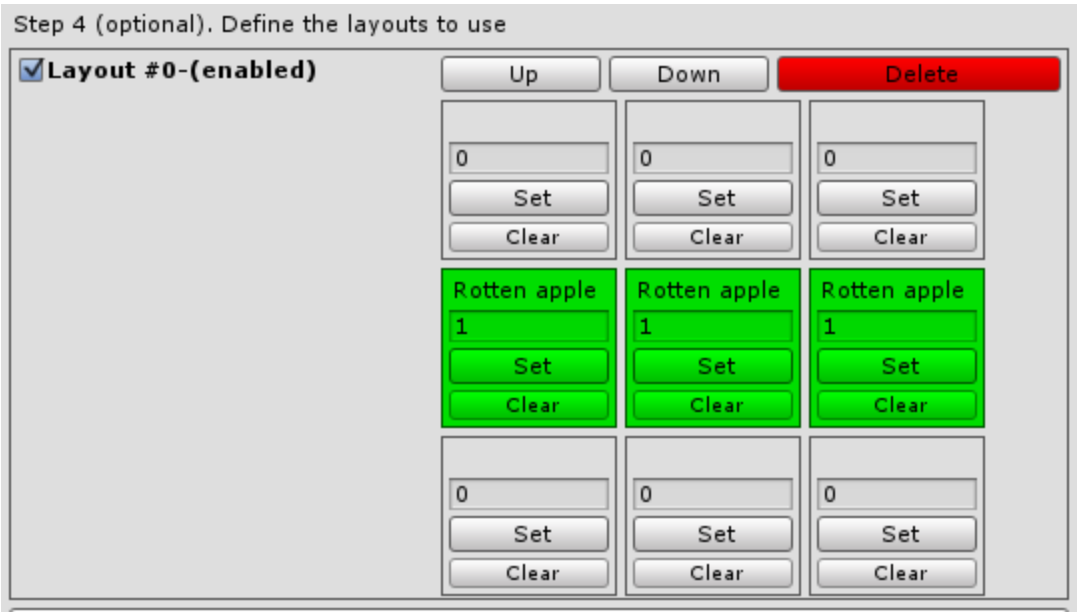
The speedup factor goes paired with the crafting time duration, in many games (like WoW) crafting the same item becomes faster when creating an entire batch at once. For example when crafting 10 Rotten apple's, the first item takes 5 seconds, the 2nd is 1.1 (10%) faster, which comes down to 5/(1.1^n).

The required items indicate how many items are required to craft the given item, this is seperate from the layouts, that can be defined at the bottom.



At step 4 layouts can be defined, a layout is used in games like Minecraft, where the user has to place items in a certain “layout” to craft an item.

As seen below, the user has to place 3 rotten apples in a horizontal row, to trigger crafting the item. **This is not restricted by the “required items” in step 2.5.**



Both systems can be used at the same time, but for now let’s set up the more traditional wow style crafting, using the required items.

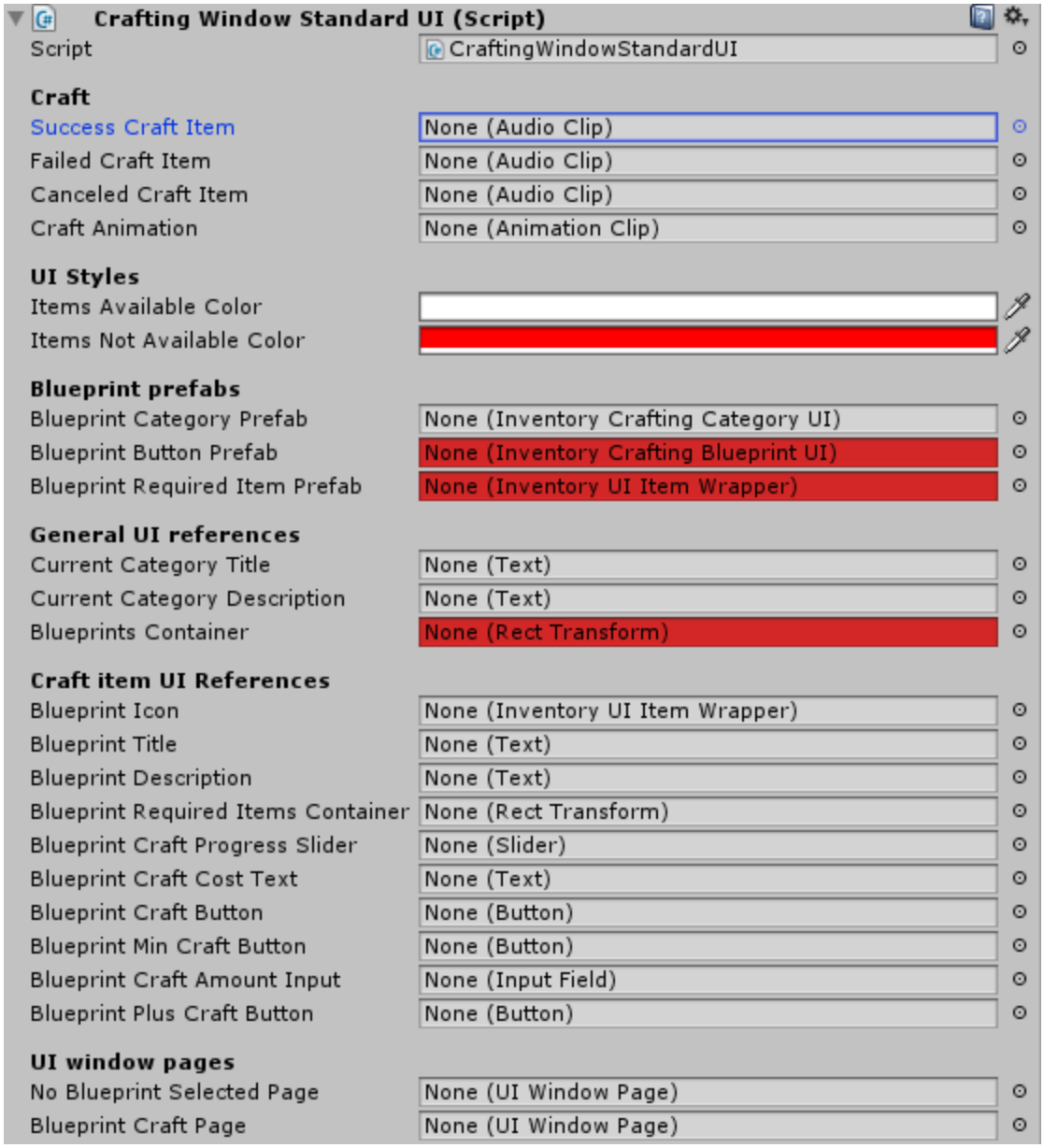
You can find the default crafting component under **/InventorySystem/Windows/Crafting Standard**

And you can find the layout based crafting component under **/InventorySystem/Windows/Crafting layouts** (just in case you want to be all assertive and divert from this finely written tutorial).

Crafting window standard

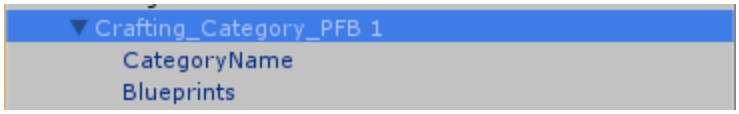
The crafting window requires quite a bit of configuration, as you can see below, to allow for various use cases a lot of options are implemented, we can just use what we need, and leave the rest blank. Fields marked red, are obviously required.

The Blueprint category prefab is the visual prefab that is used to draw categories in the UI. This allows us to design a custom category UI, create a prefab of it, and the system will generate your categories.
The same applies for the Blueprint button prefab & the Blueprint required item prefab.



Let's create the category UI element first.
Create a new UI object and attach the InventoryCraftingCategoryUI element to it.

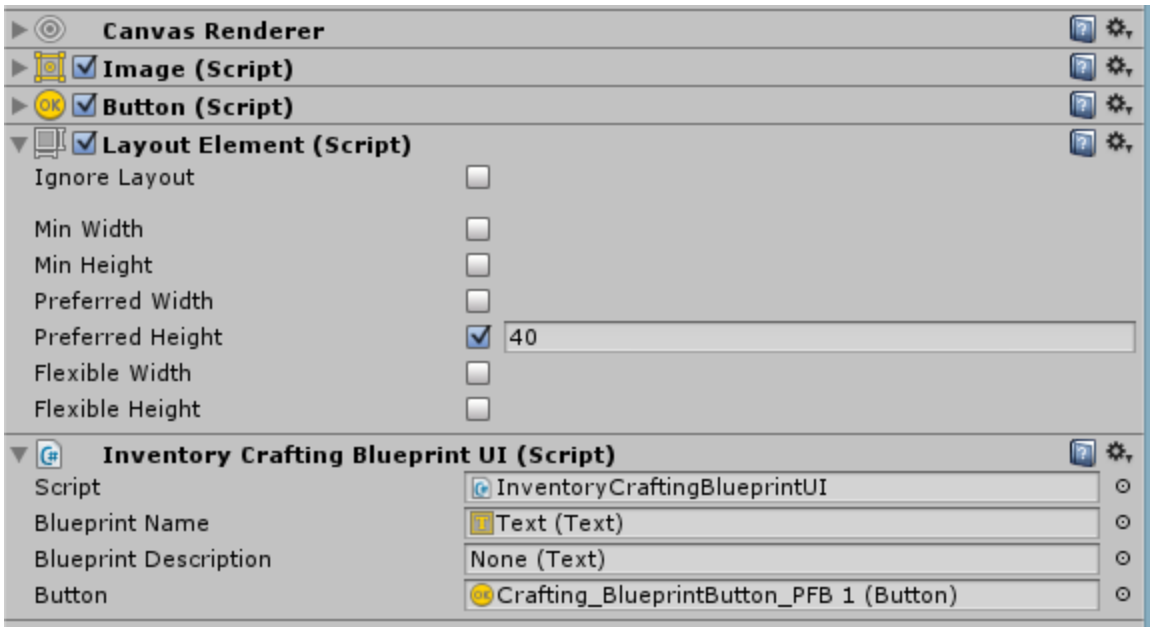
I've used the following structure, of course you are not obligated to use this structure.



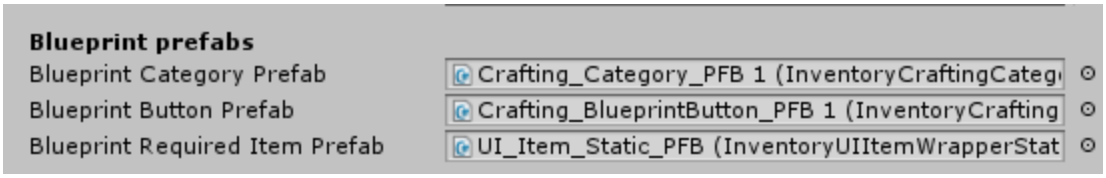
The InventoryCraftingCategoryUI has a title attribute that displays the category's name, as well as a container. The container is used to store the blueprint buttons, that will ultimately display the actual blueprint.

Create a prefab of the just created object, and assign it to the Crafting window standard UI.

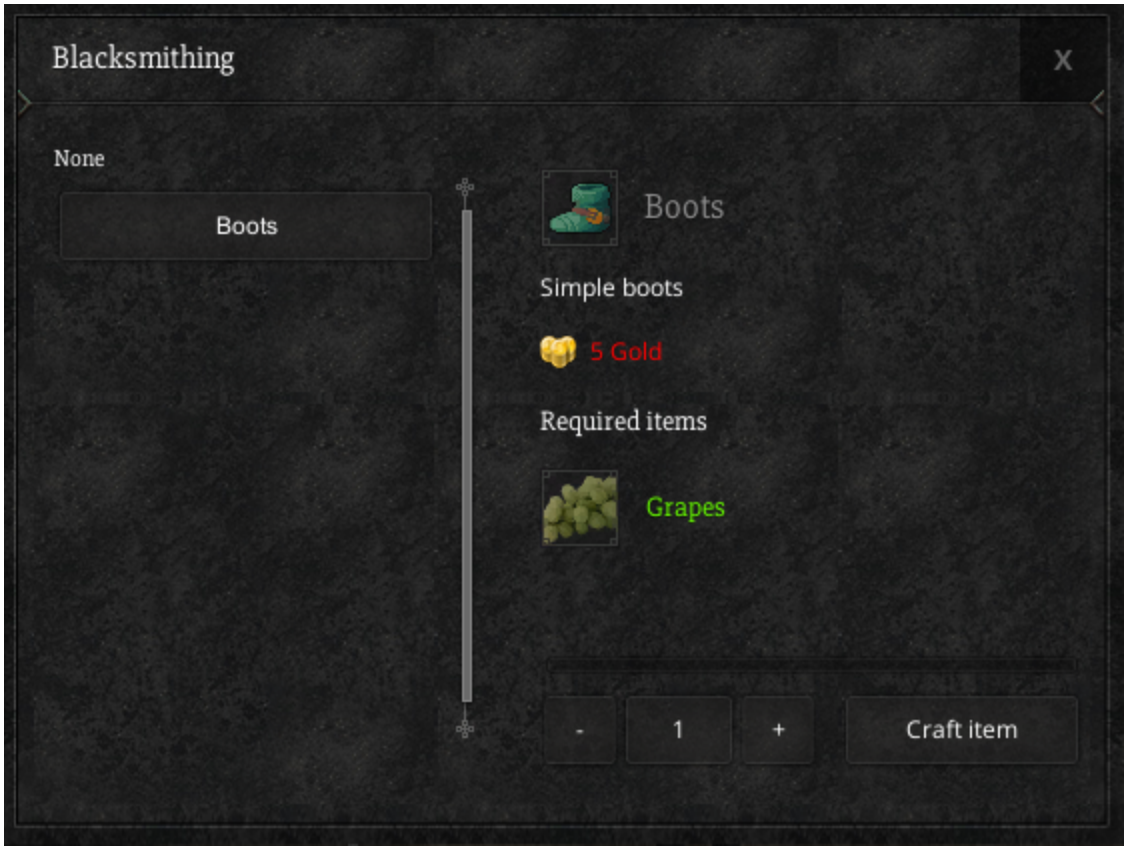
Next up the Inventory crafting blueprint UI, this is used to trigger the blueprint, allowing the user to craft it.
Once you've created the object, create a prefab of it, and assign it in the Crafting window standard UI.



If all went right you should now have the following setup (prefab names may differ).

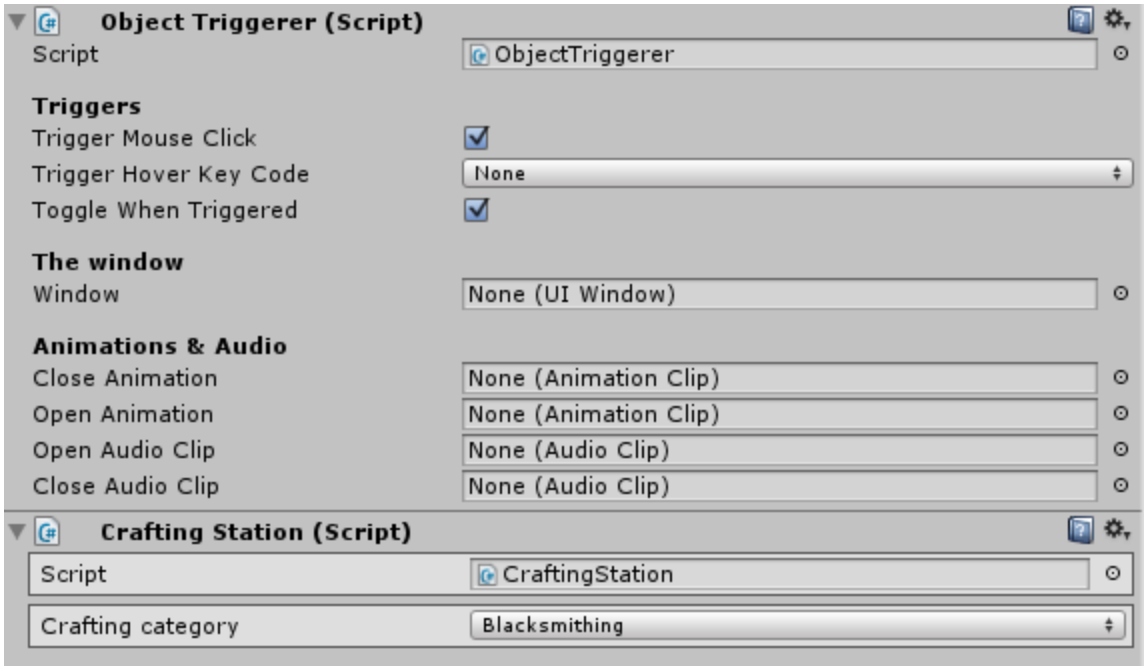


Next we'll have to design the blueprint page, this is the page that the users sees, when attempting to craft an item.
You can view a fully worked out example in scene 9 (no tutorial, just the final product).
For example:



Just one thing remaining, when we created the bank, we also created a physical trigger. The crafting system also needs a physical trigger / location where the window is opened.

The CraftingStation component uses the ObjectTriggers events to set the crafting category. Crafting station auto. detects the ObjectTrigger and knowns where to find the right window, so you're allowed to leave window empty.



Diving into code

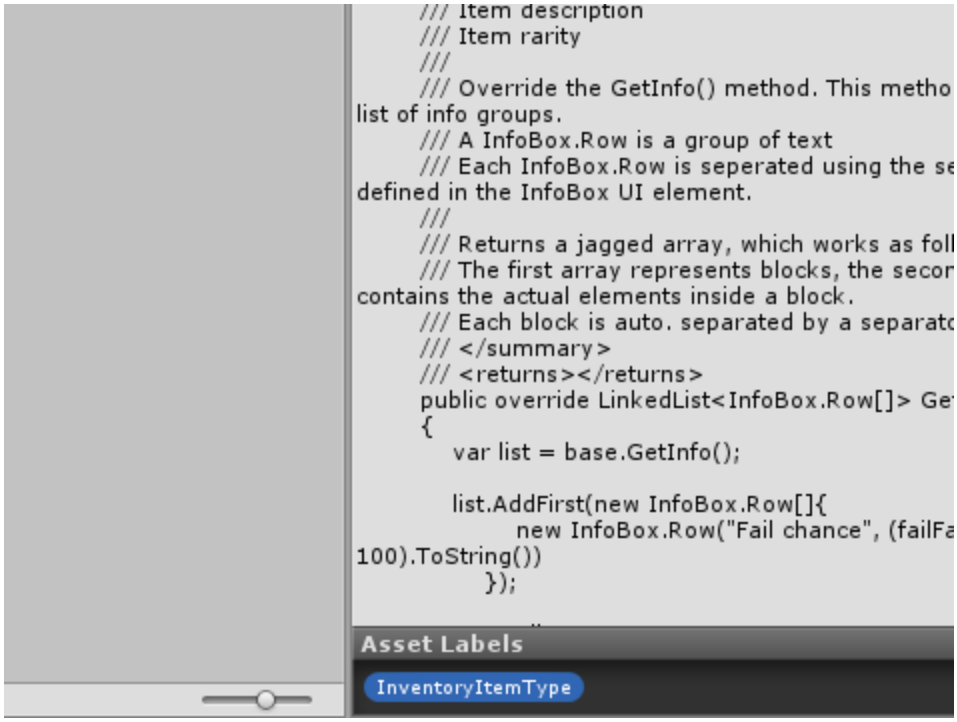
Let's write some code! - wHOOp! (I did 33 pages without code I deserve this moment… >.>)

Creating a new item type.

Let's say, for the sake of this tutorial that we want to create an object that has a 50% chance of breaking when used. And when we use it, it gives the player +50 health.

First of create your script, let's call it MyAwesomeInventoryItemType.

Make sure you add the label InventoryItemType (inside the Unity assets inspector) to the newly created class.



```
public class MyAwesomeInventoryItemType : InventoryItemBase
  {
     public float failFactor = 0.5f; // 0.0f is always success, 1.0f is always failure, 0.5f is 50% chance.

     public override int Use()
     {
        int used = base.Use();
        if (used == -1)
           return -1;

        // Do something specific...

        // 50% chance.
        if (Random.value > failFactor)
        {
           currentStackSize--;
           NotifyItemUsed(1);
           return 1; // No need to worry about anything else, the stack will be removed if no objects are left.
        }

        NotifyItemUsed(0);
        return 0; // 1, the item was used, but no items were used (stack decrease) in the process
     }
}
```

It's that simple! Next up, let's create an item with our newly created class.

1. First, go to the InventorySystem manager in the main menu, and click the Item editor.
2. Click the Create item button in the top left corner.
3. In the object selector pick your class. If you can't find your class in the list, check for compile errors, and make sure you've added the label InventoryItemType to the script asset.
4. Your object will appear in the list of items. Click the edit button, and there you go!
5. (Optional) If you want to place the item in the world click the Edit button, once you do, the item will be highlighted in the Project pane. All that's left to do is to simply drag it into the scene.

Improving our MyAwesomeInventoryItemType class

Alright, we've made our first Inventory item type, now lets extend it a bit further. Perhaps we want to use a variable to define the fail percentage of our item when used.

In the code below you can see I also used override GetInfo(). This method returns the information of the item type. A linked list was used to easily shift items around without messing with the memory.

```
public class MyAwesomeInventoryItemType : InventoryItemBase
{
   public float failFactor = 0.5f; // 0.0f is always success, 1.0f is always failure, 0.5f is 50% chance.

   public override int Use()
   {
      int used = base.Use();
      if (used == -1)
         return -1;

      if (Random.value > failFactor)
      {
         currentStackSize--;
         NotifyItemUsed(1);
         return 1; // No need to worry about anything else, the stack will be removed if no objects are left.
      }

      NotifyItemUsed(0);
      return 0; // 0, the item was used, but no items were used (stack decrease) in the process
   }

   /// Override the GetInfo() method. This method returns a list of info groups.
   /// A InfoBox.Row is a group of text
   /// Each InfoBox.Row is seperated using the seperator defined in the InfoBox UI element.
   public override LinkedList GetInfo()
   {
      var list = base.GetInfo();

      list.AddFirst(new InfoBox.Row[]{
            new InfoBox.Row("Fail chance", (failFactor * 100).ToString())
         });

      return list;
   }
}
```

Extending the InventorySystem basics

To keep the system flexible I made sure (almost) all classes are partial. This means that if you create a class with the same name, and define it partial, Unity will merge it for us.

So how does this actually work?

```
namespace Devdog.InventorySystem
{
   public partial class InventoryItemBase
  {
        // Ads the method Sell to all Inventory items.
        public virtual void Sell()
        {
              // Do some selling stuff.
        }
  }
}
```

An in-depth look about customization

Collections:

All windows that can contain items are called "Collections". A collection is basically a glorified array that holds all the items and allows stacking, merging, swapping, etc. The basic collection (class ItemCollectionBase) has some default settings that allow quick and easy tweaking without having to dive into the code.

- useReferences - This doesn't directly store the items inside the collection. When an item is placed inside the collection a reference is created.
- InitialCollectionSize - This sets the size of the collection when the game starts. You can change this at run-time but be sure to instantiate UI elements accordingly (check documentation for more info).
- container - The container is the parent of all inventory slots. You can easily create a grid / horizontal or vertical layout by using uGUI's buildin automatic layouts.

- onlyAllowItemsOfType - This allows you to block items to a certain type. For example when extending the inventory there are 4 slots to place bags, by using the onlyAllowItemsOfType the collection can easily be limited to items of type bag. Example: This can also be useful if you want to limit a bag to only allow potions or consumable goods.
- canDropFromCollection - Can you drop directly from the collection? Ignored if useReferences is enabled.
- canUseFromCollection - Can an item be used directly from the given collection? Example: Can you use a potion directly from the bank?
- canDragInCollection - Can items be re-arranged inside the collection? Disable if you want to have a static collection that the user cannot modify. For example a loot window.
- canPutItemsInCollection - Can the user put items inside the collection or is it read only?
- manuallyDefineCollection - If you don't want your collection to be auto-generated you can manually define it inside the Unity inspector.

Tooltips (InfoBox)

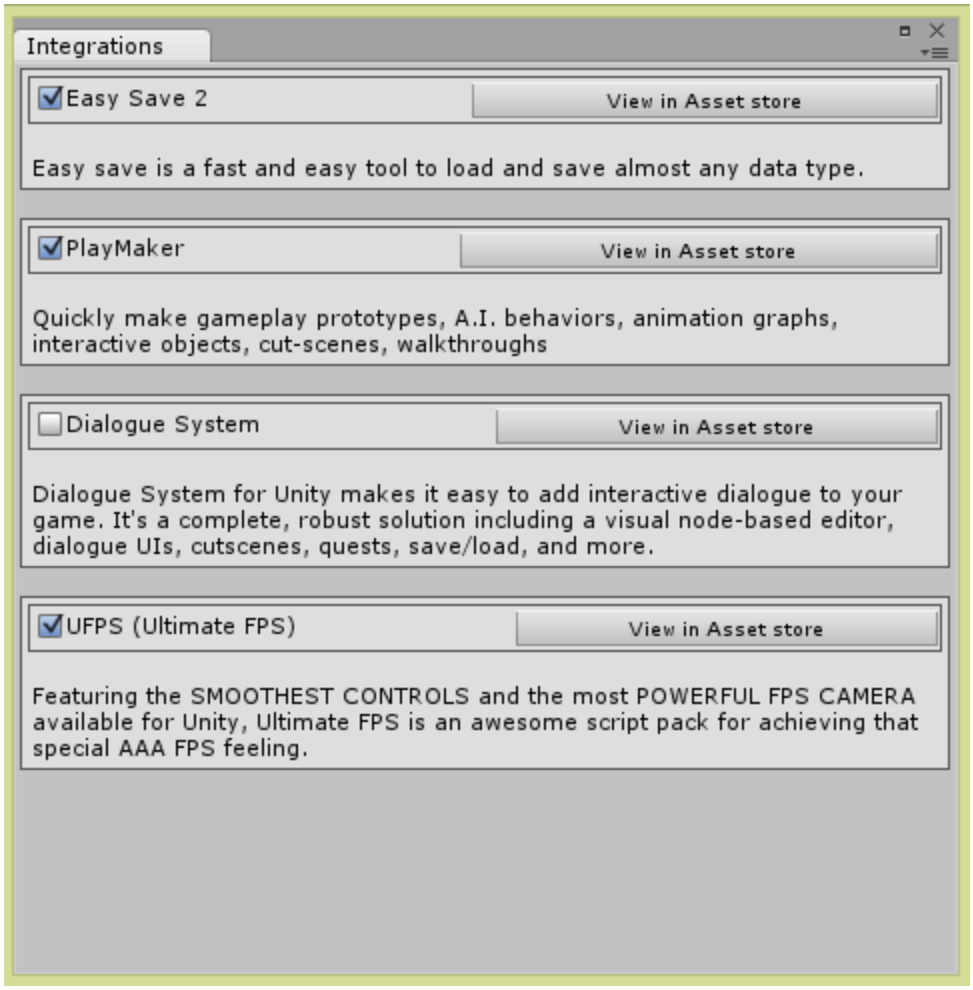
Tooltips are shown when the user hovers over an item icon. These tooltips can be modified per item type. For example a consumable item might want to show how much health it regenerates while a weapon would likely want to show how much damage it deals.

By default the tooltip shows some basic information, but you are in no way forced to use these.

Integrations

All integrations can be enabled through the integration window. Go to /Tools/InventorySystem/Tools/Integrations to open the integration window.

Simply tap the systems you want to use (and have imported in your project). When you do this Unity will re-compile code, so it might take a second before changes show up.



PlayMaker

Integrating playmaker is easy as pie, enable it in the Integrations window and that's it. Once Unity recompiled the code you'll see the items show up in the PlayMaker actions browser.

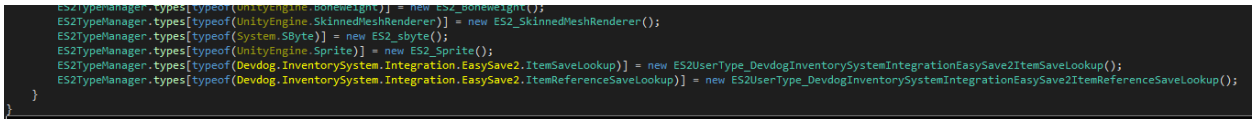
Easy save 2

Assuming you enabled Easy save 2 in the integrations window, a few small changes have to be applied to make it work.

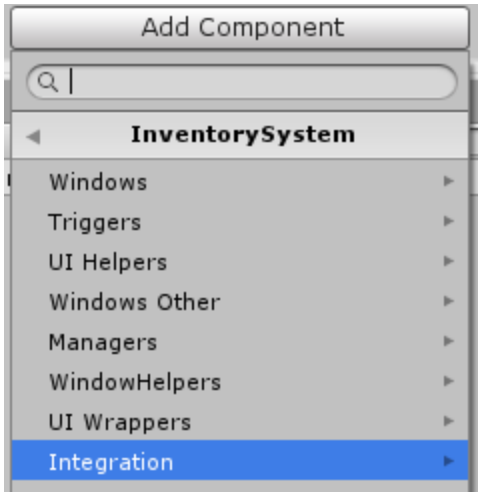
Navigate to your ES2Init.cs file, and paste the following 2 lines of code at the bottom of your file:

```
ES2TypeManager.types[typeof(Devdog.InventorySystem.Integration.EasySave2.ItemSaveLookup)] = new ES2UserType_DevdogInventorySystemIntegrationEasySave2ItemSaveLookup();
ES2TypeManager.types[typeof(Devdog.InventorySystem.Integration.EasySave2.ItemReferenceSaveLookup)] = new ES2UserType_DevdogInventorySystemIntegrationEasySave2ItemReferenceSaveLookup();
```

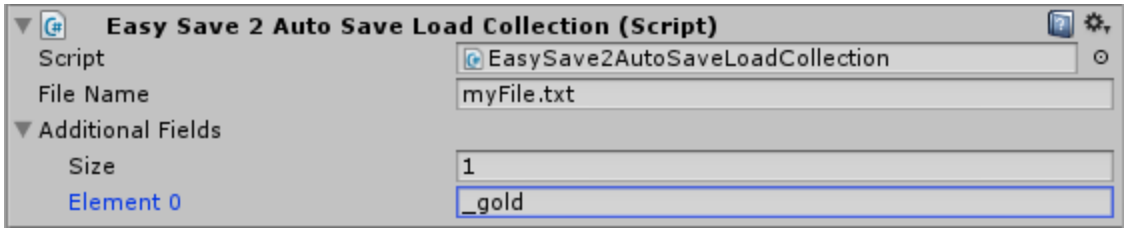
Like this:



Add a the Auto save load collection to any collection you like saved and loaded.



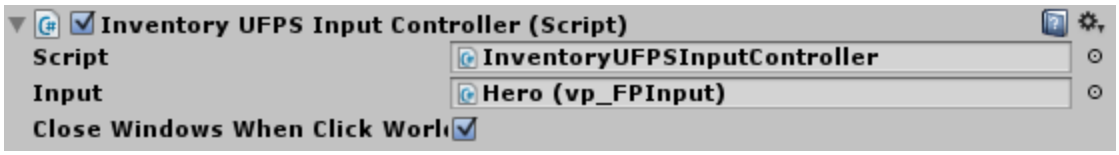
Optionally you can add “additional fields” such as _gold, which will save and load the gold variable that is specific to inventory collection.



UFPS

Assuming you enabled UFPS in the integrations window.

All that we have to do is add the Inventory UFPS Input controller component to the UFPS character object. This will handle UI elements and lock the cursor whenever a window is shown.



You'll notice that all UIWindow components now have an extra field named "Block UFPS Input", this will disable the UFPS input controller (shooting, etc) while this window is open.