

Version 3.0



SQUAREBITE INVENTORY SYSTEM



Radoslav Bunic - Bjakuja

Squarebite inventory system created with Unity3D and Playmaker. Using this system you will be able to create powerful outstanding inventory. It will also help you to learn, understand and love Playmaker even more.

C# Programmer

gamemakingalliance@gmail.com

<http://squarebite.proboards.com>

Table of Contents

FEW WORDS BEFORE WE START	3
LEGEND	5
FSM Legend.....	5
HOW THIS SYSTEM WORK	6
What this package contains:.....	6
What this system supports:	6
SETTING UP SCENE AND OBJECTS.....	8
Layers, tags and initial projects setup.....	8
Editor Menu	11
Create your item	12
Define your item in container.....	16
UI ROOT.....	20
INVENTORY SYSTEM.....	22
Hash Tables and Arrays.....	22
Interact FSM.....	23
Arrange FSM.....	24
ItemDrop FSM.....	26
FSM FSM	27
Transfer FSM	28
UseItem FSM.....	29
Sounds FSM.....	30
IMPORTANSE OF PLAYER STATS AND EQUIP	31
CONTAINER SYSTEM AND ITEM EXPLANATION.....	32
Container FSM components	36
SHOP SYSTEM.....	37
CRAFTING SYSTEM	39
SERIALIZER AND MANU MENU	41
PLAYER STATS AND GUI	44
FEW WORDS FOR THE END.....	47

FEW WORDS BEFORE WE START

Edit – this is new GUIDE for version 3 and up, read it from the start to the end because there are a lot of changes and almost everything is changed since system not using global variables anymore and other things are more simplified and easier to use.

We all know that without good inventory system you can't make good RPG game. My inspiration for this Inventory was Gothic game serial where you have your Square inventory system that nicely sort items, scroll through and reorganize when items are added / removed. This is not dragable system so you can't drag and drop items, this is Square inventory where you have columns and rows so when items reach horizontal edge of inventory they will spawn in next row etc. This will be possible in future updates for sure.

My aim was to create almost perfect inventory for my game that I develop and since I don't work anymore for Cracked Piston I reserved all rights to publish it on Asset Store as mine, because it is completely mine 😊. I searched through asset store for inventory of this type where I can easily display my items display certain item types such as weapons, apparel, consumable with a single click and inventory that looks awesome and guess what none of those inventory exist on Asset Store so I decided to create my own.

On this pretty long road to have better and better inventory I created this Squarebite inventory system and instead of using it for my own game I want to share it with people around the globe and I have strong reasons for that: first because I think it will help a lot to all people who love to program with Playmaker, second because I want to help people to understand Playmaker even better and to create great games or any other successful project with this Asset, my help and Playmaker. Third reason is because I think that my Squarebite system can be easily integrated in any game that requires inventory, if you read this guide and watch some upcoming tutorials you will do just great. Final reason is because this Asset is made entirely with Playmaker and I just love this system so I wanted to share it and to see it integrated into many future projects.

This is not just inventory, this is more than just ordinary inventory system, you will get many other things that will be included in this package such as Great character Model that was meant for our game A.F.T.E.R, shop, loot and chest systems, lockpicking, character stats with skills, simple models of some potions, apparel, weapons and bunch of sounds that fits any inventory system.

So we can finally start with this guide and I hope I get all aspects that are required to be explained in order for you to use Squarebite without problems. S

Watch on my Youtube channel video tutorials for Squarebite in order to help you even more, and here we go. ;) Just type squarebite tutorials and you are ready to go ;)

Important changes!

Read this GUIDE because there are a lot of changes in version 3.0, almost everything changed

No more global variables, no more multiple game objects for each Hash Table, now you have everything inventory and everything works with local variables which means no more bugs.

Important for save feature!

Now for the start you will need Playmaker license in order for this package to work and also if you want to save your game and use save / load features you will need to import Unity Serializer (Unity Save) into your project. There are a few things you should do after you import Serializer, first is to import Playmaker package from serializer Addon folder, you can find under [Plugins/whydoidoit.com/AddOns/PlayMakerAddOn_v0.5](https://plugins.whydoidoit.com/AddOns/PlayMakerAddOn_v0.5)) and that package contains Playmaker actions that are necessary for saving / loading games using Serializer and Playmaker.

In Squarebite folder under SystemResource you will find a small package called SaveGame and it contains a SaveGameManager prefab with all necessary actions and save / load states for Playmaker and Serializer + SaveExample scene. So after you import Serializer and that Playmaker Addon you will be able to completely save your game using SaveGameManager that must be on your scene.

Related to Serializer next thing to do is to create SaveManager on scene using Serializer Window and to store information for every HashTable in your project. Just click on game object with HashTableProxy component and click Store information using Serializer window. When you have to decide what to save using store information you just have to decide what you want to store, for example in inventory you want to store Hash Tables because they contains values about your inventory items and since you have two hash tables (one for count and one for items) you have to save them both, but don't worry about that too much because when you click to store hash table it will automatically select all hash tables on current game object.

You can find more about in chapter Serializer and Main Menu on this guide.

LEGEND

This is meant for easier reading and understanding variables and other things. Read what each color represent in this GUIDE:

Variable – when you see marked as purple that is variable

Edit – this is note for something being edited compared to previous GUIDE

Notes – this is mark for notes and something very important

FSM – blue is mark color for fsm components on game objects

Event – this is event that is called in game obejcts

Game Object – I use this this color to mark game objects that are children of UI Root

Actions – This color is used for Playmaker actions

FSM Legend

In playmaker you have the option to color states so here is some legend about how most of the FSMs are colored inside of this package:

RED – States that will probably finish some set of actions (player dead, loop complete...)

GREEN – Most of the time will represent when you begin something or continue some action

YELLOW – Yellow will be for states that starting whole bunch of action (generate list, load...)

BLUE – Blue state color is meant for checking conditions (exist in inventory? YES or NO events for example), when you have multiple choices most of the time.

PURPLE – States that are marked as purple will be some small changed or changing index for certain variables

CYAN – This color will represent some initial values or resetting game objects in most cases to prepare for some big events

ORANGE – Similar to green, use this color in same FSMs just to make difference and to get better look of my states 😊

HOW THIS SYSTEM WORK

This is inventory system created with Playmaker and without Playmaker license it can't work so first thing you need to do is do download Playmaker if you don't have one already and then import this package into your project. If you don't have any previous Playmaker knowledge you can still use this system by following this guide, most of the things you need to know are explained here and by using this system you will understand some advanced things and actions in Playmaker.

What this package contains:

- ❖ Complete INVENTORY and ITEM system.
- ❖ Complete SHOP interaction system
- ❖ Complete CONTAINER system
- ❖ Complete crafting system
- ❖ Player Stats, level system and skills
- ❖ Health and energy bars / easily setup
- ❖ Basic Character
- ❖ Basic low poly models such as potions, weapons, shop, chest, armors.
- ❖ Basic particle effects, heal, fire
- ❖ A lot of inventory sounds
- ❖ Complete SaveGameManager when you import Unity Serializer.

What this system supports:

- ❖ Item sorting
- ❖ Inventory list scrolling
- ❖ Pick items
- ❖ Preview items
- ❖ Highlight items
- ❖ Item stack, any item can be stackable
- ❖ Item drop
- ❖ Item hit
- ❖ Item use
- ❖ Item equip / unequip
- ❖ Buy / sell items
- ❖ Preview items

- ❖ Confirmation / discard after shopping
- ❖ Scroll through items
- ❖ Numerical purchase / sell if item number is above limit
- ❖ Transfer item to inventory
- ❖ Transfer items back to container
- ❖ Take all items at once
- ❖ Lockpick system (based on random number of attempts)
- ❖ Unlock with keys stored in inventory
- ❖ Level and experience system
- ❖ Health / energy restoring
- ❖ Attribute increasing (Strength, Dexterity, Wisdom, Health, Energy)
- ❖ Skill points to use
- ❖ Stat points to use
- ❖ Basic Character controller supports:
- ❖ It's Mecanim related so it uses basic Mecanim animations, supports moving forward, backward, crouching (basic crouch for picking items and chest opening, it's not accurate animation, for example when you pick item character will just crouch, when dropping item character will just put hand straight forward etc...), item drop/use and idle.
- ❖ Camera movement that follows Player
- ❖ Player is rotating with mouse look action using mouse X value. If you want to have Mecanim rotation using animation to rotate you have to create your own, because this is inventory example not character controller system.
- ❖ Saving inventory/shop/container items into hash table and restore back on load
- ❖ Saving stats into hash table and restore it back
- ❖ Saving scene items, equipped items into hash tables and restore on load.

SETTING UP SCENE AND OBJECTS

Layers, tags and initial projects setup

Before importing this project you have to make sure that you have proper layers and camera culling masks in your project, otherwise you will be unable to see some items or some random collision problems can occur.

Edit – Layers are changed, now you have one important layer to set in your project and that's GUI layer

Make sure you have following layers in your project

❖ **Layer: GUI.**

If you don't have GUI layer, you have to define them and to set MainCamera to not cull these layer because you don't want to see inventory or stats until you open them. Next you have to CameraGUI to cull only GUI layer. Last thing to do about layering is to set items and backgrounds in proper layers, take all of your GUI items, all backgrounds / labels that will appear on inventory (regular and equipped items) in project and set them in layer GUI (include children), same goes for Inventory, just add whole inventory to the GUI Item layer (include children) and it will be culled properly. It's basically like this, each item and every GUI object has to be in GUI layer and CameraGUI set to cull only GUI layer

As you can notice all items that are equipped, used or dropped, they are set to be kinematic, this is important because of collision problems, If you don't set this up correctly then you can have situation where some random collision can happen especially with player he can jump fly or any random bug can happen. So make sure that before importing project you have proper layers and uncheck collisions in physics matrix if you want something specific.

After this step, make sure you have items in proper layers. All GUI items have to be in GUI Item layer, just mark them all and change layer to all items including children and when you create new item and if for some reason layers messed up just change them to GUI Item.

Next are Tags, there aren't too many tags, only important tags here to mention are **Player** (already included in settings default), **Container**, **Item**, **Workbench**, **NPC** and **Shop**.

Tags will probably appear when you import this package, in case they don't then you have to define those tags.

First tag you Player to be Player, this way it will interact with shops and containers properly.

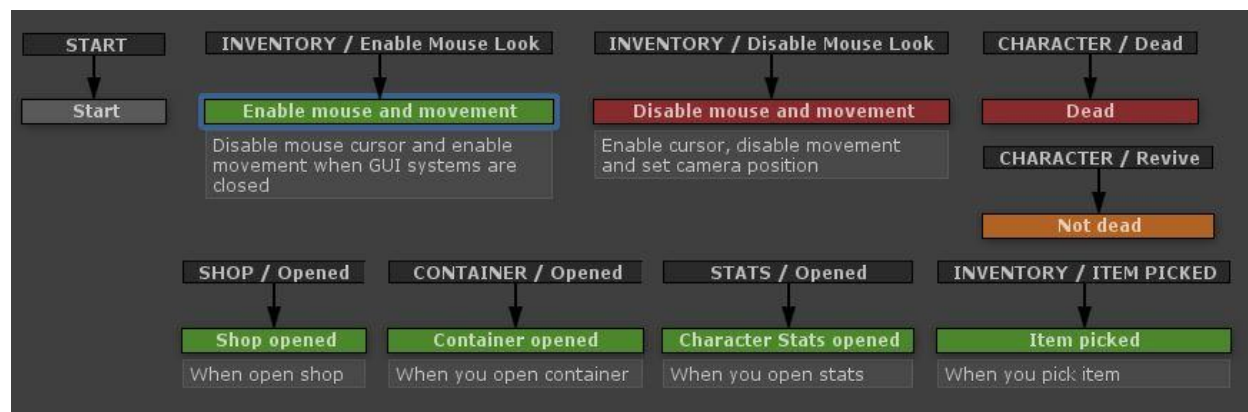
After this, you have to tag every chest, dead body container or any other container that can contain items you have to tag as **Container**, because of ObjectInteractSystem that should be parented to your Player in order to detect items, containers and shops.

Edit: Object interact system now has one FSM and with that fsm it will detect everything on scene so no bugs can occur with different objects

When you are close to some objects, depends on tag of that object, Object Interact System will send proper events, if that is container it will send event to that container to open if you press E key. If we have all of this setup we can detect those containers, shop, and items, but why do we have to set Player to be tagged as Player? Because of circumstances that can happen in game so when you open chest for example.

InteractObjectSystem has child also, that is some empty game object that we use as drop item location, so when items are dropped from your backpack, they will appear on that ObjectDrop location.

Edit / Player is changed completely, now has only one FSM that will define every game object you need.



Picture 1

As you can see, this image is FSM of Player and this is everything you need of events, but this FSM also has some variables that needs to be defined as well.

amuletHolder	amuletHolder	⊙
armorHolder	armorHolder	⊙
beltHolder	beltHolder	⊙
helmHolder	helmHolder	⊙
itemUseHolder	useHolder	⊙
leftArmbandHolder	leftArmbandHolder	⊙
objectInteractSystem	ObjectInteractSvs	⊙
oneHandedWeaponHolder	oneHandedWeapon	⊙
playerStats	PlayerStats	⊙
rightArmbandHolder	rightArmbandHolder	⊙
ringHolder	ringHolder	⊙
shieldHolder	mixamoria:LeftFoot	⊙
twoHandedWeaponHolder	twoHandedWeapon	⊙
useHolder	useHolder	⊙

Picture 2

On this image you can see many game object variables that has to be defined. Each variable is written with small capital first letter and big capital for first letter of each next word.

All holders are game objects that you have to fill by dragging your desired parents of equipped objects. For example amuletHolder is game object that is parented to bone on your neck, and after amulet is equipped it will be parented to that amuletHolder. Same goes for all other Holders.

Except holders, here we have some other values to specify, first objectInteractSystem, just drag from project objectInteractSystem and parent it under

player, after that drag that objectInteractSystem to this objectInteractSystem variable.

Other important game object is playerStats, when you add it to scene and parent it under player, just drag it to playerStats variable and you are ready to go. This playerStats system will hold all player values in one place.

There are few FSM components of playerStats, first one is **Attributes** which will hold all player stats values, stats (strength, dexterity, wisdom), experience, skill points, health and all other. Big change here is that you have baseAttribute and equippedAttribute which means that when you equip something it will increase only equippedAttribute and if you drink elixir or level up you will increase baseAttribute. Than for GUI stats to show those values are combined and displayed as one using int operator to add one value to another. Other FSM components are **LevelSystem, Health and Energy**. Health and Energy FSM components serve as calculator for health and energy but values are still taken from Attributes FSM.

Editor Menu

Under Tools/Squarebite menu you have several options that you can use for creating certain systems and items or to access GUIDE, forum, online resources etc.

When you want to create system on your scene you can use this menu to do so and system will be created on 0, 0, 0, position and rotation, or you can just drag system into your scene and for inventory and character stats systems it's important to have 0, 0, 0 location and rotation. Available system to create with editor menu are UI Root, Inventory, PlayerStats, PlayerStatsGUI, ObejctInteractSystem, GameManager,ShopSystem, ContainerSystem and LootSystem. Shop system place where you want your shop to be and scale collider to fit your needs and for chest system it should be put under your chest and you have to specify your cover for chest so it can rotate it and open.

When you want to create items with EditorMenu you have several options here so you can choose what item type you want. Available types are Apparel, Consumable, Weapon and Other. When item is created it will automatically add two items to your project under folder NewCreatedItems and in both 3D and GUI item will be created. This is important because every item needs **TWO** components, 3D item that will appear on scene and GUI Item which will be visible when you open your inventory, shop or container. After items are created you have to specify item values, names, sounds, information, UID and other fields that you will use but most important is to connect those two items, you will read more about item component and item connection in next section *"Create your item"*

Those items and systems that Editor will create in your project are stored in folder called SystemResource and Editor will just create copies of those prefabs (for systems they will appear on scene and for items they will be duplicated in project folders). So don't change names of those prefabs and don't move them into other folders, otherwise Editor will not work because it has defined paths to know where to search and system/item names to know what to search.

You can create items with one simpler way by pressing control D on desired item and it will be duplicated, you have to do this for both 3D item and GUI item and to connect them by dragging each other into each other field for this purpose (more in next section).

Other Editor options that can be helpful to you are online resources which will lead you to youtube channel so you can watch some tutorials and important videos, or you can go to Squarebite forum, post there if you have some problems, requests, questions or search for your answers.

Under editor you will find this GUIDE and about option so you can track latest updates and change log for Squarebite Inventory.

Create your item

Edit – Items are changed completely, now you have only 1 FSM on each item, items still has two components for referencing database but no more FSMs and complicated detection. Now system will check everything for each item and also all GUI that is displayed, it is set on Inventory system, not on items anymore. All parameters you define GUI items and after you click on some item system will get all values and based on results it will send proper events. Same goes for item usage, drop, equip and everything else (much better approach).

After you finish with setting layers and tags you can start to add your systems and items to the project and scene.

First you will have to add your character and main camera to the scene (you can find them both under Squarebite / Character folder (**you have make sure that main camera is set to render everything except GUI layer, that layer is rendered by Inventory camera**)).

Next you have to add ObjectInteractSystem to your main character, this object will be used to detect and interact with every item and container from this system. It's used for picking items, detect item name, shop, workbench and chest containers.

Next you have to add UI Root game object to your scene and for stats to work you have to create PlayerStats too and set it under player. After this you have to drag PlayerStats and ObjectInteractSystem to player game objects variables so system can know where they are and we don't have to use find game object.

Next you will need some chests so drag ContainerSystem to your chest object and define items inside of that chest.

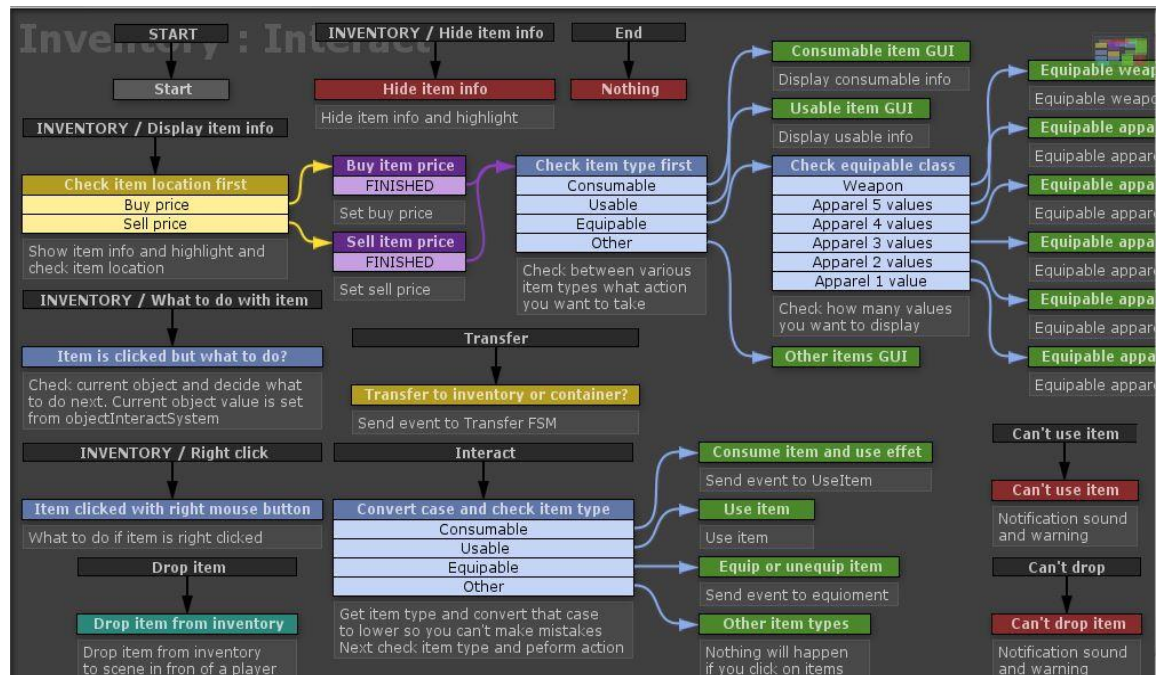
If you need shop on your scene, you just drag ShopSystem and place it where you need shop.

To add items inside of a chest or shop you will have to create items first.

To create item you will need two things for every item, here is the explanation:

1. **GUI component** – This is your item GUI that will be displayed in inventory or container. When it's clicked with left or right mouse button it will send event called "What to do with item" to inventory "Interact" FSM, and from there system will decide what to do with clicked item. If left clicked than first we check string value called "currentObject" this value is set on objectInteractSystem when you approach some something so if you approach container, currentObject value will be set as container, if it's shop, value will be shop. This is very important because after we check that value we know where are we and what we can do, if for example shop is opened you can't drop items, equip them

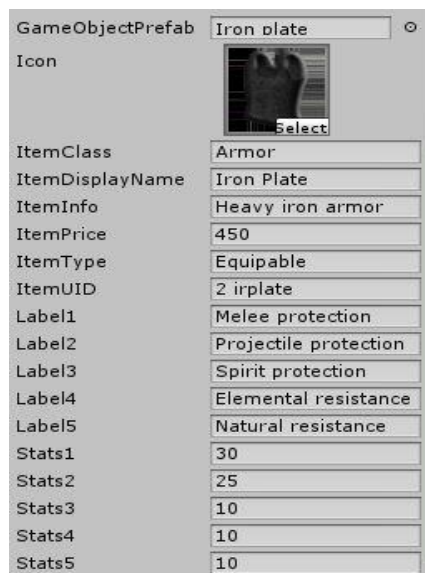
etc, it will automatically send event to **Transfer** FSM where we check **ItemLocation** and based on that we transfer item to inventory or from inventory. Have a look at this example image of that interact FSM:



Picture 3

So when we want to display item info we have to check item class, current item classes are: Amulet, Armor, Armbands, Belt, Helm, One Handed, Ring, Shield and Two Handed.

On pictures 4 and 5 you can see GUI item FSM components for armor and one handed:



Picture 4



Picture 5

Here you define everything related to item. Let's explain the armor GUI: First thing to define is **GameObjectPrefab**, it represents physical component of an item so drag your object with collider and rigid body tagger as Item In Layer item on this field. Next is item **Icon** which will be ItemGUI texture for displaying on screen. Next parameter to define is **ItemClass** which is important to define in order for system to know what kind of item is that. This is armor class so if you equip it, system will create system on armorHolder and equip to armor slot. Next is **ItemDisplayName** which is exactly that, your item name. **ItemInfo** is your item information, **ItemPrice** is item value (if you want to change shop values or item values you can define desired values in Interact FSM on inventory and in **Transfer** FSM of inventory). **ItemType** is value that defines what to do with item if you click on it, if it's equipable it will equip / unequip certain item, if that value is for example consumable then event will be sent to **UseItem** FSM in inventory. **ItemUID** is your item unique code for storing it in database (Hash Tables) and each item needs unique code. Rest of the values are labels for item specification and their values, **Label1** is Melee Protection and **Stats 1** is value for that Melee Protection which will be increased if you equip that item. This GUI item has 5 components to display, some of them have 2 or 3 values, you can define from 1 to 5.

2. **GameObjectPrefab** – This will be item physics component. What is important here is to put your mesh under this object and set collider and rigid body on this object, not on the mesh. This is important because this way you will avoid pivot point problems of models.

So when you have your ItemPrefab ready for game you just have to connect it with desired GUI item, in this case we want that Iron plateGUI (armor) to connect with Iron plate game object prefab (physical component of our item) so just drag that prefab in field called "GameObjectPrefab" (picture 6). After that you have to drag GUI prefab to the field GUIItemPrefab of your 3d game object you just dragged as a prefab:



Picture 7



Picture 6

This way we connected our physical object to the desired GUI object and now if drop or equip our game object than this defined object will appear on the scene. So what exactly this will do? This will make connection between those two items so if you find this physical object on the scene and pick it up then this GUI defined object will appear in inventory. You can define item pick and drop sounds or you can call them what you want, depends on your needs, or it may be equip sound or any other. System is set to play generic sound defined in inventory for every weapon or apparel that is equipped. If you define your own sounds in those fields system will play those sounds if you leave fields `soundEquip` and `soundUnequip` empty then system will play generic sounds defined in inventory `Sounds` FSM. How's this done? Well if item want to play sound when equipped it will compare sound objects.

Now you have your item ready for game and for storing / extracting from inventory or any other container. Same goes for other item classes, for weapons you only have to specify some additional values:

Attribute	Dexterity
AttributeRequired	15
Damage	20
DamageType	oneHandedDamage
GameObjectPrefab	Scimitar
Icon	 Select
ItemClass	One handed
ItemDisplayName	Scimitar
ItemInfo	Great one handed we
ItemPrice	140
ItemType	Equipable
ItemUID	1 scimitar
Label1	One Handed Damage
Label2	Dexterity required to

Picture 8

As you can see first value here is `Attribute` and this is very important for your weapons since they will use attribute requirements to equip, in this case it's Dexterity (note that this must be defined as Upper case because this word will be used in build string to get / set values in PlayerStats) and `AttributeRequired` is 15 which means you need 15 Dexterity in order to equip this item. `Damage` is how damage your weapon will provide and `DamageType` is name of variable to increase in your PlayerStats, in this case that is oneHandedDamage. `GameObjectPrefab` is your 3d game object component (your item mesh) same as we said in armor description and the rest of the variables are same except for `Label1` and `Label2` those are item labels for your info, `Label1` say in this case One Handed Damage on left side and on the right will be your

`Damage` integer value converted to string and displayed on screen, same goes for `Label2` except that value displays your requirements.

Define your item in container

Because this inventory system has power to sort items and place them in nice order you have to define item UID's that way. Inventory will sort items alphabetically and because we want to have everything in order we have to define our items in groups, so for example weapons will be shown first, one handed melee weapons will show on top, after them one handed ranged weapons, then two handed melee, then two handed ranged such as bows or whatever you want. Next will be shown apparel items, armor first, then boots, then greaves the helmets, necklaces, rings, shoulders, shields.... Next elixirs, potions.....food or any other consumable item you define, next miscellaneous items such as documents, maps, miscellaneous for last you can display keys, quest items....

So how we do this, it looks so complicated but I will show you how can you make those groups with just two or three single letters so take a look and system can recognize items and place them where you want. It is all related to item UID, which you define for each item and using that UID inventory system knows in which order items have to be.

Let me explain first how Hash Tables works with Playmaker, pay attention cause this can help you a lot to understand them, many people using Playmaker faces problems when comes to HashTables, Array Maker and Array Lists.

Basically is like this: Hash Tables are used for storing information in database and then we use ArrayList to get and display that information. When you add HashTableProxyComponent to game object you have to define Prefill type of that Table. It can be defined as many types such as string, float, game object, integer, boolean, vector 3..... We need only two for this inventory and those are Game Object and Int. Game object to store items inside and Integer to count how many items are in table. Hash Table works using key pairs, key is item UID (unique identification) and second component of pair is your defined variable, in our case it will be game object and Int value. Have a look at these images picture 9 and 10:



Picture 9



Picture 10

Chest and other container has 2 Hash tables, Hash table item count which references as “count” and there will be stored item numbers as integer values and other Hash Table is called “items” which will store game objects. We store items Hash table data base by dragging items inside and for each item we define UID key. Have a look at the image number 9, you will see on the left it says 1 scimitar, that is our game object UID and on the right we just drag our scimitar GUI game object. We do same for Hash table count except that we don’t drag values we simply define how many items we want in container. Have a look at the image number 6 and you will notice different stuff then in image number 10. It is integer hash table so we just define item UID which is in our case 1 scimitar and on second field we define how many Scimitars we want in our container, in this case we will say that we want 3 Scimitars.

So how we actually define our item UIDs, why I used 1 scimitar as my item key, why not just Scimitar? Here is the answer, if I used just Scimitar then if I used just potion or just amulet or just regular name and when list is sorted you will get messed up inventory where items are sorted alphabetically but it will be chaos for players cause they will have a real hell until they find desired item even if list is sorted. This is not a problem when you have few items, but if you have 200 or 300 items in your inventory then it’s a problem. So that’s why I used 1 scimitar which is in this case: 1 weapon and scimitar is item name.

Here is the list of items I used here in inventory and how I group them to be organized well:

1 – weapon
2 – apparel
3 – consumable
4 – miscellaneous
5 – key

This table is just example of item organization, you can use numbers if you prefer or more letters to be more specific about items. Our scimitar is one handed weapon so if we want to have more organized weapons we can use 1 Oscimitar and for two handed we can use 1 Tsword , this way one handed weapons will be displayed before two handed weapons. If you want more control over sorting you can use for example 1 OMscimitar which means 1 for weapon, O means One handed, M means Melee weapon and scimitar is name of our weapon so if you have a Bow called Long bow you can set Its UID to be 1 TRlongbow, so 1 is weapon again, T is two handed, R is ranged and long bow is item name. There are lot of variations so you can use anything you like, it depends on your desire of how to sort items and what to be displayed before and what after.

Main thing of this inventory is ability to display certain types of items. Depends on how you create your item UIDs system will display what you want to be displayed. Here is example image of this item type display:



Picture 11

As you can see on image 11 there are 6 icons of what you can display on your inventory. In this case on image I clicked on fourth icon (potions) and I got displayed only my consumable items. First icon will display all sort of items, second will display only weapons, third only apparel, fourth only consumable, fifth will display miscellaneous items (you can put in this group items such as maps, some jewelry or what you want) and last icon is set to display only keys,

lockpicks or you can use it for maybe to display only quest items etc.

This can be also used for backpacks of inventory, for example you want to display only weapons in your inventory you just have to define item group you want to show and system will hide the rest of the items. Here is next image example in inventory [FSM](#) fsm (it's component of inventory system that is responsible for inventory display, not for item icons etc cause that is done in Inventory Array list component, but for showing / hiding inventory, how to open inventory when shop or container is opened...)



Picture 12

As you can see on image number 8 there are multiple state and events that are called when you press any icon mentioned above. It will set item type in Inventory Array list (referenced as "items") so items that contain string for example 1 will be displayed and items that doesn't contain will be skipped. This way we display what we want on the screen by pressing certain icon button.

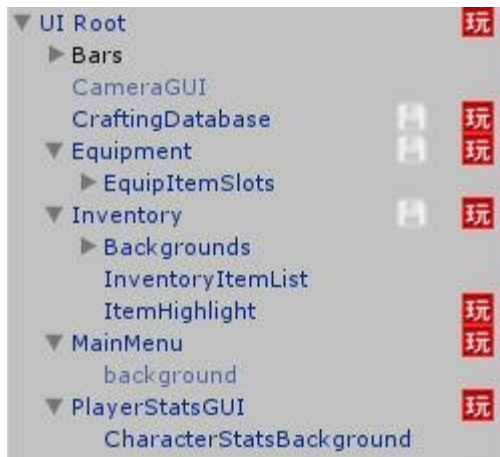
For creating new items in your project you have two options, one is to just duplicate items in your project, for example you need another elixir, you will take any elixir GUI + elixir 3D Prefab, then duplicate them, rename and connect them by dragging inside described fields above. After you have your items you will just have to define your item specification, add effect, info, icon and for 3D item to define item mesh, change collider if necessary and you have item ready for game. Other option is to use Squarebite Inventory menu in editor where you have some options to create items and systems. When you choose to create item it will be duplicated in folder called new items, both GUI and 3D component will be created but totally empty so you will have to fill them up with values and you are ready to go. Choose options that will suits you best, I prefer to duplicate items since they already have everything inside, just redefine values and that's it because if you need potion you will duplicate potion if you need one handed sword you will duplicate that scimitar that's already there etc.

You can also use editor menu to add systems to your scene, systems included: Inventory system, character stats system, chest and shop systems object interact system and game manager system.

Note about editor menu – When you create some system on scene using Editor Menu that item will be created as clone and renamed automatically so it will not be prefab instance, it is instantiated to scene and not a prefab. To use prefabs you have to drag items to your scene.

UI ROOT

Edit – This chapter is earlier called Inventory System but now it's changed to UI Root because Inventory is reworked completely and this chapter about inventory system is totally different then before so now every GUI is parented to UI Root and every game object you need as GUI it will be under this game object.



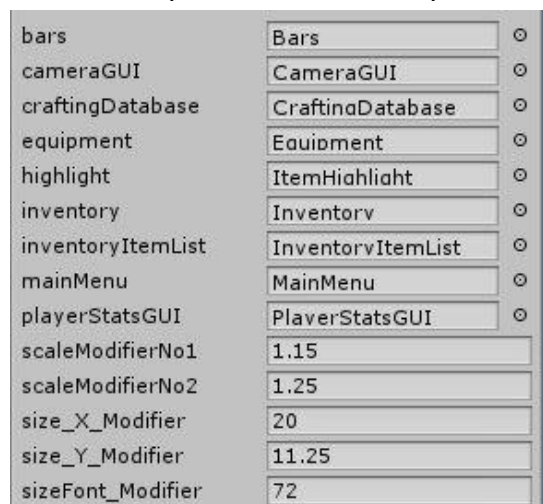
Picture 14

As you can see on picture 13 UI Root is our main game object and motor of all things. That game object only has one FSM component called **FSM** by default. Here we drag and defined every game object we need and on start we calculate all values. After that we send event **Reset** to every children and each game object will get values from **UI Root**. Here we have **Bars** which is empty game object but childrens are **EnergyBar** and **HealthBar**. Next we have **CameraGUI** that is required to render our GUI layer only. **PlayerStatsGUI** is game

object required to display our stats (note that actual stats are defined in PlayerStats under player, this

PlayerStatsGUI is for GUI only). **CraftingDatabase** is required for crafting (totally reworked crafting system, now you just have to define here your result item and that's it), **Equipment** is game object required for equipping items on your Player. This is completely reworked and now you have slots where you can equip and it is much more efficient than before. And finally we have **Inventory** game object, which has children **Backgrounds** and **ItemHighlight**. Backgrounds will display your GUI textures and item highlight is active when you mouse over any item in inventory or container. Last game object is **MainMenu** which is responsible for displaying options, menu buttons etc.

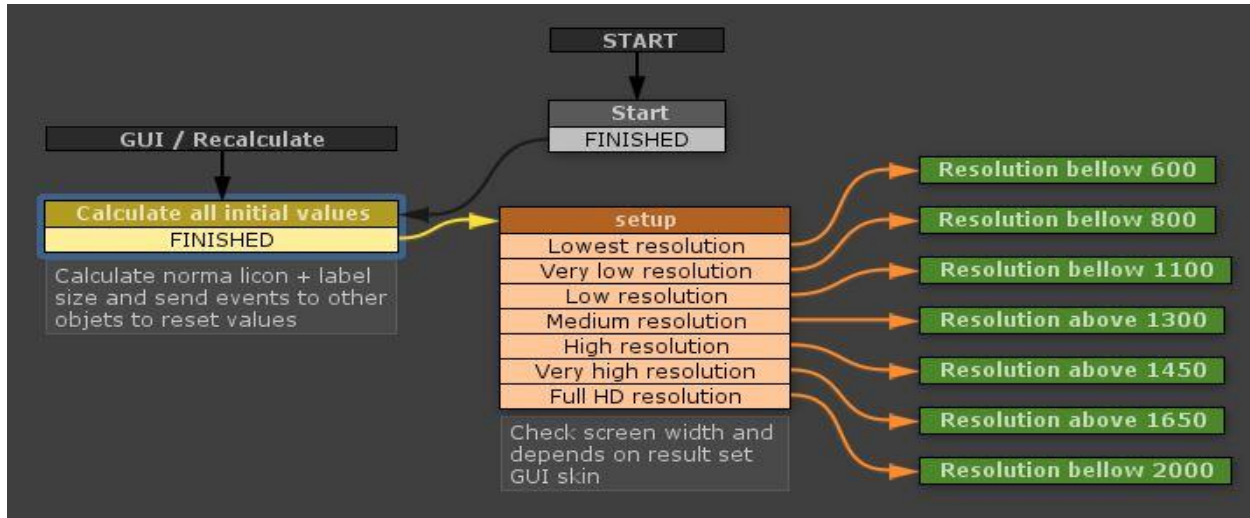
Before everything you have to specify some values and drag some game objects to your variables in UI Root. Drag your game objects to those variables in UI Root and next you have to set your GUI values, **scaleModifierNo1** is value of scaled icon which is 1.15 times bigger than normal icon and **scaleModifierNo2** is 1.25 times bigger than **scaleModifierNo1**. This is used for icons, slots,



Picture 13

scaled icons etc.

When scene starts first thing to do in UI Root is to find player and store it as player, next we get playerStats from player since we defined that game object variable in player we can easily use action **GetFsmGameObject** to get any values we want (no more global storing). Everything in this system is based on this, get value using get FSM value actions, change value and store value back using set value actions. So after we get values from player next we have to calculate all values we will need for our GUI. Have a look at Picture 15 that is UI Root FSM:



Picture 15



Picture 16

As you can see on Picture 16 all we need to do is to calculate every value and that begins with **Get Screen Width** and **Get Screen Height** and based on those values we calculate everything we need. This will calculate your icon sizes, scale sizes, equipment slots, font size and then it will choose proper GUI skin for current resolution. After everything is finished we send events to children of UI Root and to player, event that is sent is called "**Reset**". This event will serve as initial values on all those game objects. For example we can't use initial values on start state in inventory because we don't have calculated values yet.

In the next few pages you can read about **Inventory** game object since it is most important game object for your items and there are a lot to explain components.

INVENTORY SYSTEM

Inventory game object has 2 Hash Table proxy components, first one is called “items” and it stores game objects, second is called “count” and it stores integer values for each item. There is also 1 Array List Proxy component on this game object and will get values from hash tables, used for sorting and organizing things mostly. This Array list will create items and using HashTableKeys action will get keys from your items HashTable and it will sort, display your items on screen as GUI items. Read next about all FSM components in inventory

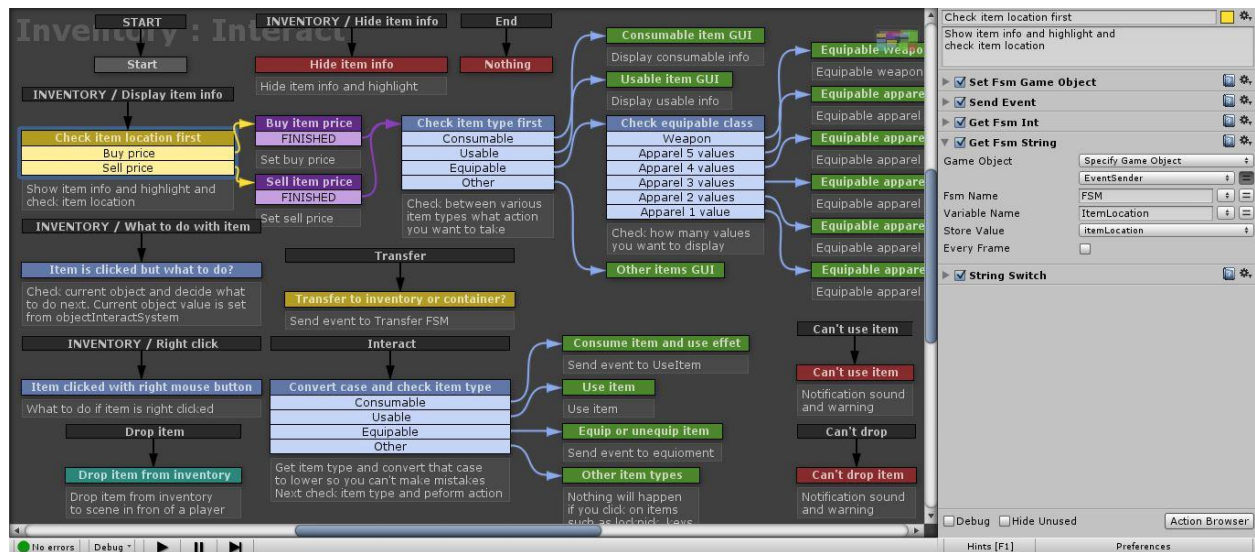
Hash Tables and Arrays

You can put anything in array list and also a lot of actions that support ArrayMaker. If you didn't have a chance to use ArrayMaker until now trust me it's very good addon for Playmaker and you will love it when you discover what you can do with it. You can set values by yourself or you can get them from Hash Tables so basically it's like this: Hash Tables are used to store values with KeyPairs and ArrayLists are used to display, organize those values. In this Inventory case we store all items in Hash Tables and we use Array List to get those items, sort then and display on screen. So how we do that? Depends on your choice you can store in Hash Table any value including game objects so for every item we store inside of Hash Table we have to define Key which is UID of item (Unique Identification) so my items have their codes as I explained already. Each code has to be unique and logical, use your best methods for defining keys for your items.

Inventory System uses two array lists and two hash tables. First array list is keys and second is items, this is only system that has two array lists, and this is done because we have to check what items to display in inventory, you can filter you can filter your items by clicking on inventory buttons on top near label. Why we have Two Hash Tables, one is game object type called “items” and there we store all items as game objects, and the other one is integer type called “count” and there we store count number for each item in our “items” hash table. Other containers use same method for storing items. When inventory is opened, GUI list of items will be generated, first we hide complete list and based on filter and index we display certain items.

Array lists are usually cleared before populating with items, then we use action Hash Table keys and get all items from “items” hash table and store those values as string, then we sort list so we can sort items as well, then we get UIDs one by one and based on them we get item as game object from our “items” hash table and item is created under your item list. When you transfer items from container to inventory or vice versa, items will be removed from hash table of no other items of that kind left in container.

Interact FSM



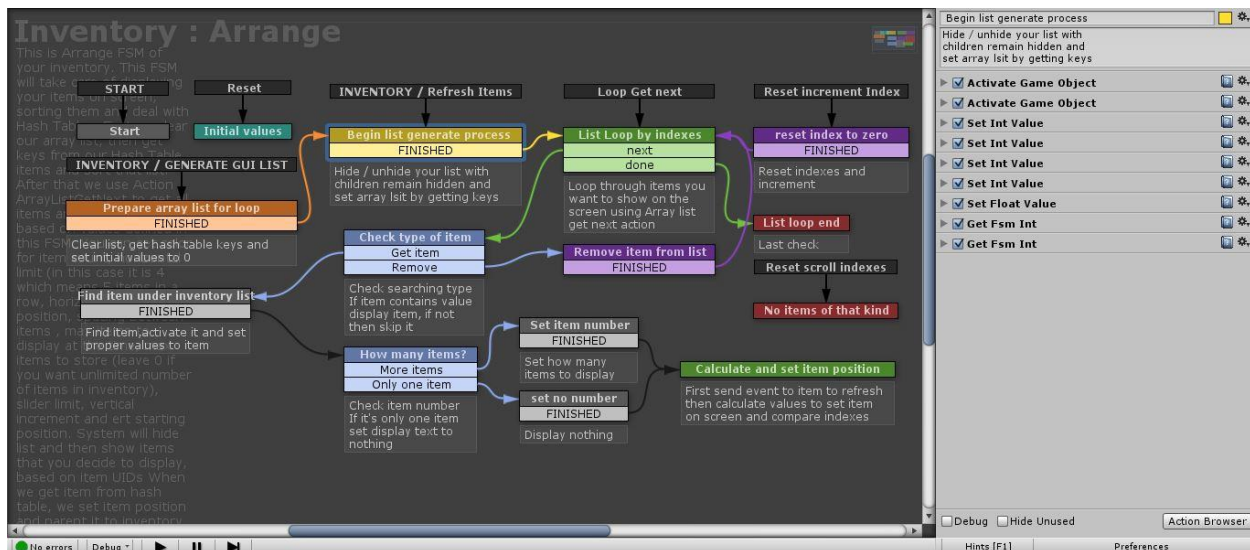
Picture 17

Not first in alphabetical order but have to explain this FSM first. Everything starts here related to your items or any interaction with them. First this **Interact** FSM have game object value called “currentObject” and it is set by your objectInteractSystem that detects objects you interact with. When you mouse over GUI item, that item will set itself as **EventSender** game object to this FSM and that way we can use get fsm actions to get any value we want from that item. Event **INVENTORY / Display item info** will be sent when you mouse over and system will check item location, item class, and send proper event to display certain values on screen. Item types can be consumable, usable, equipable or other (you can change all of that if you more types or you don’t like this setup). So if you mouse over some equipable item then we have to check item class, if that is a weapon we will display only item damage and requirement for equip, if it is armor then we need 5 values displayed, if amulet we need 3 etc.

When you click on item with left mouse button that means that you want to interact with item, we have to check item type and if you clicked on consumable, system will set that EventSender to UseItem FSM (on inventory game object) and send event **INVENTORY / Item use**. If you click on equipable it will set EventSender and send event to Equipment game object under UI Root to check there item class and to equip / unequip (if equipped) item from certain slot.

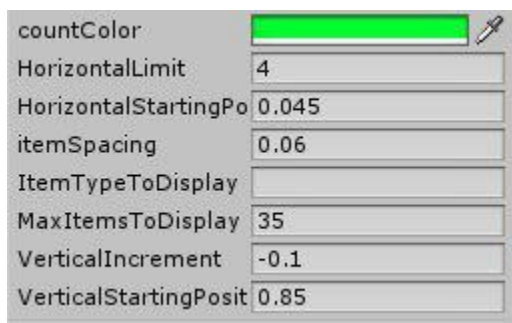
If you click with right mouse button system will set EventSender to DropItem FSM in inventory and item will be dropped to the scene.

Arrange FSM



Picture 18

In this **Arrange** FSM we create item list, sort items, use specific algorithm to place items on screen and rearrange items if you scroll through the list. Everything begins with list creation, then getting hash values from Hash Table items to our array lists and get one by one item and display it on screen. First we use Array List clear action to clear our list, then to get values from Hash Table we use action Hash Table keys, this will copy all values from Hash Table to Array List and then we sort the list. When list is prepared for we get values one by one using action ArrayListGetNext, we store that value as string and use that value to get item from our items Hash Table and store it as game object. Then we create that game object, parent it to our ItemList and set parameters for each item and their position.



Picture 19

There are some values in the inspector that you can set you can and change change depends on how you want your items to be displayed.

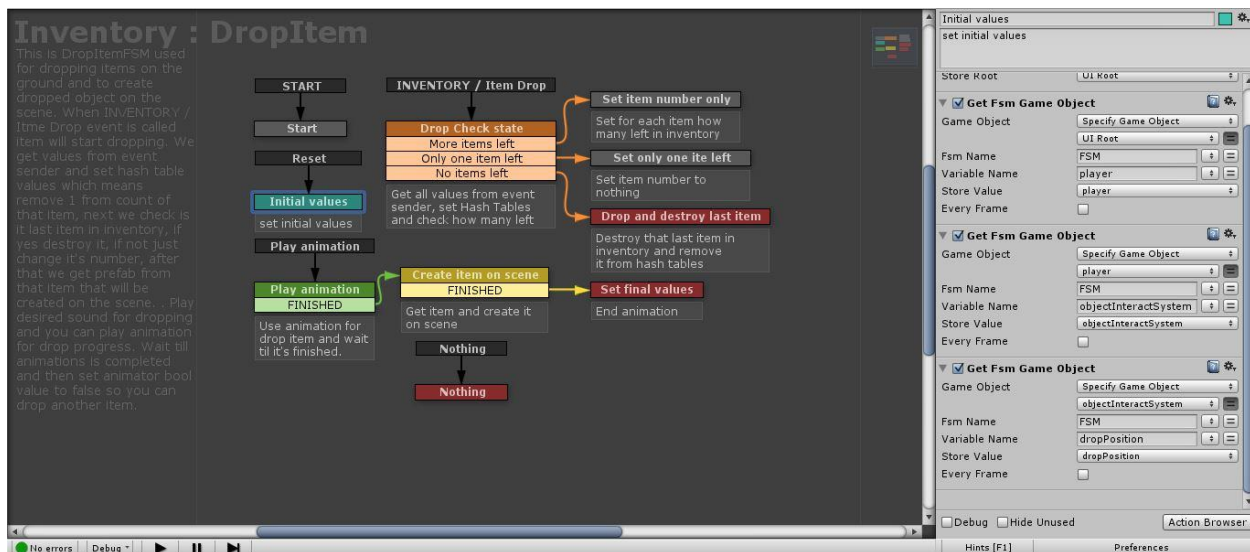
First you can set color of your item count numbers, this default is green so for each item count that color will be used in you inventory. Horizontal limit is set by default to 4 which means 5 columns will be displayed because zero is also counted.

HorizontalLimit is set to 4 which means 5 items in a row since zero is counted as well, **HorizontalStartingPosition** which is set to 0.045 in my case that is that position from left to right

where you items will begin to display on X axis so I set my items to start displaying almost at the beginning of the screen. Vertical Starting position is position where item should appear on Y axis. Item Dimension is increment from left to right, so this means that first item will appear on 0.045, second will be 0.105, next will be 0.165 etc. and we get this by adding increment to every next item. For every item that appear on screen we add 1 to item index so when that index go over 4 or horizontal limit then we will have to set it back to 0 because we want next item to appear from 0.045 but we add one increment to vertical increment and next item will appear below our first item, in this case Vertical Increment is -0.1 which means that first item will appear at this position $x = 0.045$, $y = 0.85$ and sixth item $x = 0.045$, $y = 0.75$ and so on.

This same method is used for container (chest and loot) and shop systems so you can set those values there if you don't like those default values.

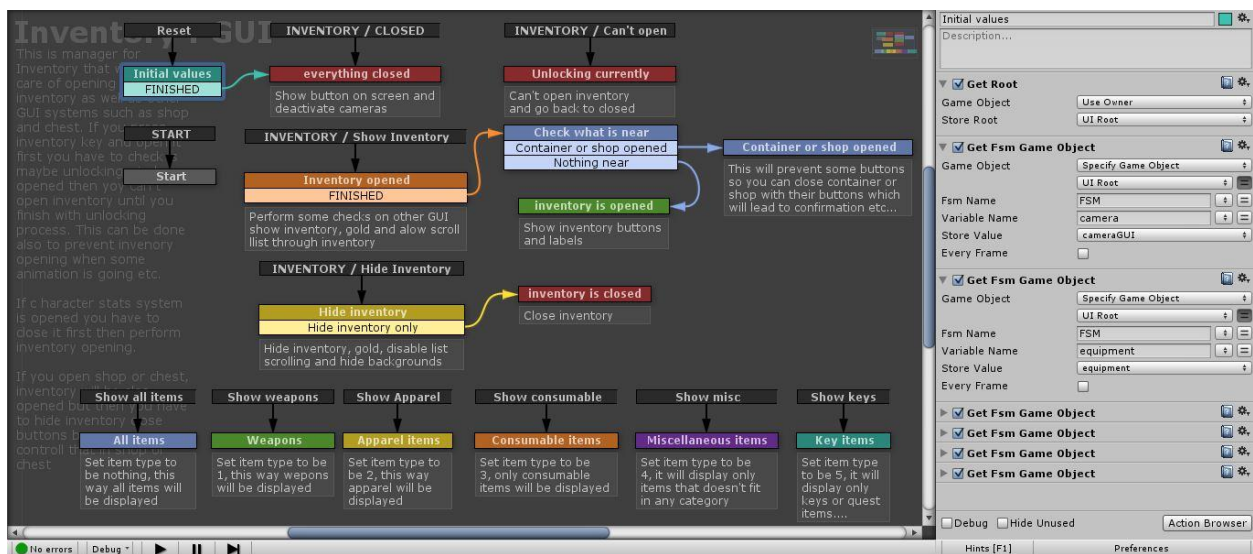
ItemDrop FSM



Picture 18

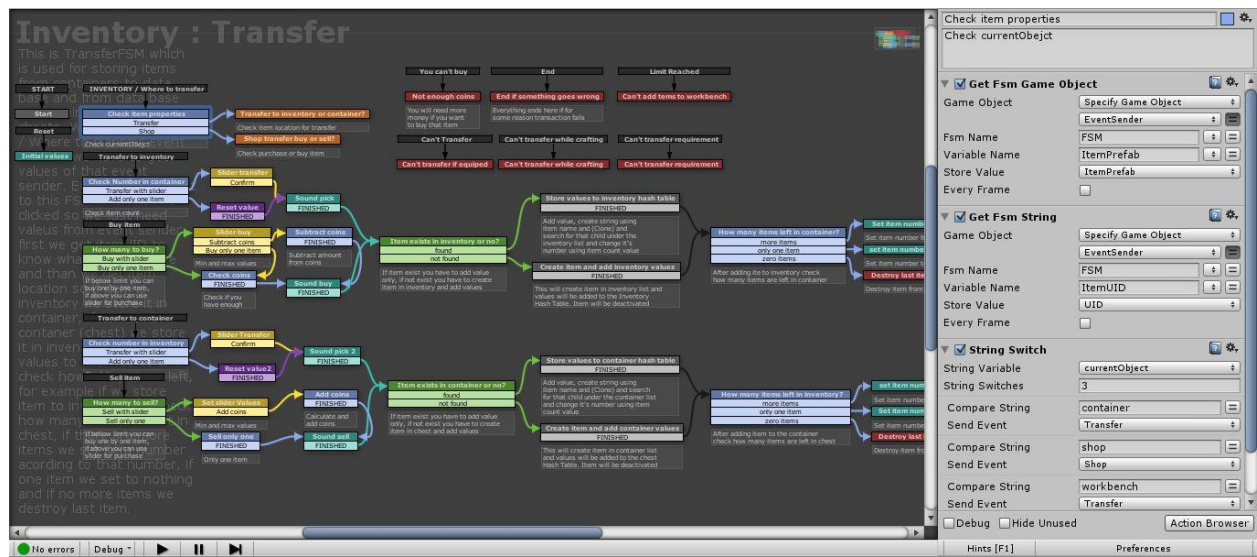
Next on image No. 18 you can see ItemDrop FSM component which will be activated when item is dropped from your inventory. When any item is clicked with right mouse it will set values in this FSM such as event sender and object to drop so the system can know what item to subtract from hash tabel and what item to drop on the scene and also item will send event to start drop process and that event is “INVENTORY / Item Drop”. When item is subtracted from inventory count hash table and created on scene it will check then how many items of that kind are left in inventory, if it's last item it will be destroyed from inventory hash table database. If there are more then one item in inventory left it will get value from hash table count and set that value to item so it can display item number normaly usin GUI text which is component parented to every item and it serves to display item count for each item. If only one item left it will then set item to be without item number which basically means when you have one item you don't need to display label 1 for that item it's obvious that is one item, text labels for item count are necesarry when you have more items, not just one. There are conditions that will prevent you from dropping item from inventory, for example you can't drop equiped item or you can't drop items when shop or chest are opened etc. After everything is set you can procced to Play animation state which can be set play Mecanim Animator action by setting values in animator or you can use anything you like for this purposse, I preffer Mecanim! This state will also play sounds for drop item and you can define those sounds in inspector. You can also define DropTime which means how long will drop progress last. After that time you can drop your next item and if you try to drop item while drop is in progress you will not be able to do that. This time should match your animation time.

FSM FSM



Picture 20

FSM fsm is responsible for displaying and opening your inventory. After **Reset** events is sent from UI Root we will go to everything closed state and there you can set keys or buttons to open your inventory. When you open your inventory we have to get fsm string from **Interact** FSM (will explain later on Interact FSM) on this game object and check currentObject value. That value will be set depends on what objects are colliding with you objectInteractSystem, it can be shop, workbench, container or nothing which will be set if you approach item for example. This is important because based on currentObject we can set what GUI buttons to display on screen, if you open your shop for example you don't want to have your equipment on screen, same goes for container, but if you open inventory with nothing near you will get your equipment slots displayed. Other states will be active if you press certain buttons on screen, those tiny button will switch your display type.



Picture 21

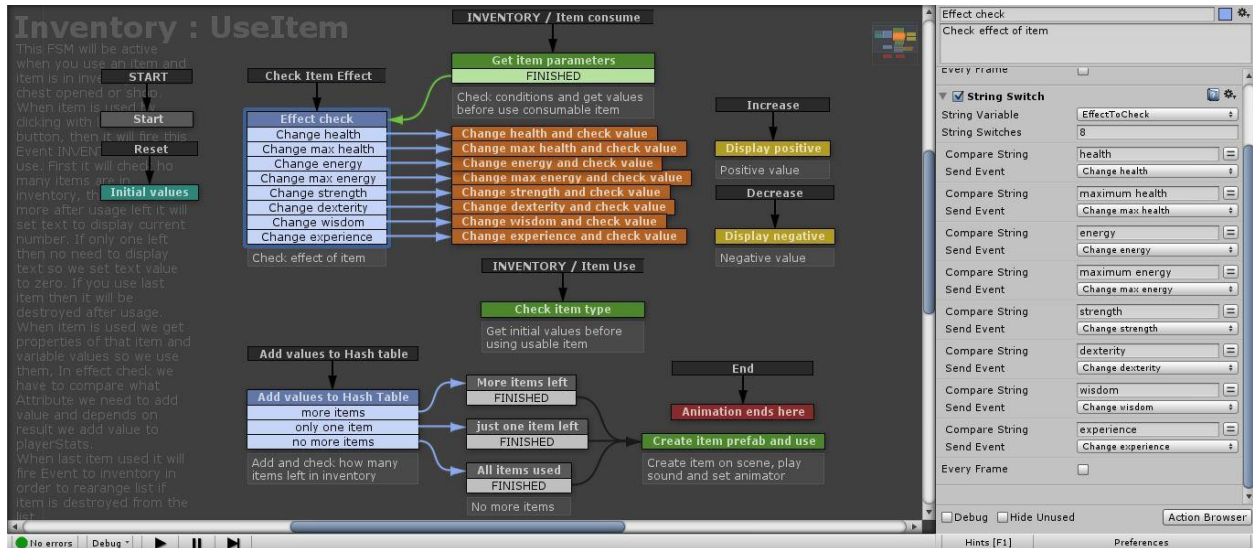
This FSM becomes active when you have something opened other than inventory and you click on GUI item. Since you can't drop, use or equip items while shop, container or workbench are opened you will be able to transfer items to inventory or from inventory to other containers. When item list arrange items, we use set fsm string to set item location, for example you transfer item from your inventory to container, that fsm string will store value "container" this way we know that item is in container so when we click with left mouse it will be transferred to you inventory one by one, or by slider if item count is more than 10.

Based on item location item will be transferred, if that value is “inventory” means that item will be transferred from inventory and based on currentObejct value if that is container system will transfer item to container, if it is shop then system will sell item to shop and vice versa.

Rest of the transfer procedure is same as in shop which includes Hash Table get / set storing values check how many items, hash table remove, slider transfer when items are above define limit.

UseItem FSM

UseItem FSM will be active when you click on any consumable item in your inventory while no shop or container opened. Basically means that when you open your inventory and click on consumable item it will be used and event will be sent to UseItem FSM in inventory.



Picture 22

As you can see on picture above method for using items are similar to item drop fsm but here we have to get item "Attribute" and "AttributeValue" so we can know what variables will change after item usage for what amount in `playerStats`. If you have Health defined as `Attribute` and 50 for `AttributeValue` you will increase your current health by 50 points when item is used. After attribute is changed then we have to check that value, is it positive or negative depends on results display string variable will be formatted and event sent to `CharacterStatsGUI` to `Display` FSM in order to display that change. More about item usage you can find under *Container and item explanation* chapter.

Creating item on scene, usage time subtracting / removing item from inventory HashTables, those operation are same used in `DroptItem` FSM.

Sounds FSM

This FSM is responsible for playing sounds in your inventory. You can notice that inventory game object has Audio Source component and we use that to set sounds we want to play. There are predefined sounds already for you to use but if you want to use your own sounds for each action, you just have to specify those object variable in Sounds FSM and there you go.



Picture 23

When you need custom sounds you just have to specify and set variable **customSound** and send event **SOUND / Custom** and sound will be played.

This sound FSM exist also on PlayerStatsGUI, it works basically the same but with less number of sounds to play, can still work with custom sound as well.

IMPORTANCE OF PLAYER STATS AND EQUIP

Since it's problematic for Serializer to save FSMs properly and global variables we have to use some different system for saving equipped items, this is the reason why we have to put all items that are equipped into HashTable and on load that hash table is checked and using that list will equip all items that were equipped when saved (items are removed from hash table when unequipped). Same method is used for items that are on scene, they are stored and on load recreated on their positions.

When you equip weapon damage will be calculated by using Player attributes and weapon damage. It's simple with weapons because they will only increase damage which is only one value but for apparel it is different story because there are more values that can be added to your stats or protection or even damage.

Currently all apparel items can have up to 5 stats that can be used. For example when you equip body armor it is set to change all your protections which is MeleeProtection, RangedProtection, SpiritualProtection, ElementalResistance and NaturalResistance. So those values will be changed when item is created and system send event to equipped item to get all values from its GUI item and add those values to desired variables. When item is unequipped it's set to subtract those values from variables that are changed when item was equipped. If you don't have all 5 values you can just disable those actions that you see as unnecessary. If you want to change it to another variable you can change it without problems, just don't forget to change Unequip values as well because you will get into problems if values aren't removed.

In PlayerStats we have two types of Attributes, first we have baseValue and equippedValue, those combined are your Attributes. When item is equipped we will add values to equippedValue, for example you equip ring that give you 10 maximum health, we will get fsm int from playerStats called equippedStrength, increase that value by 10 and set fsm string action to set that equippedStrength value in playerStats.

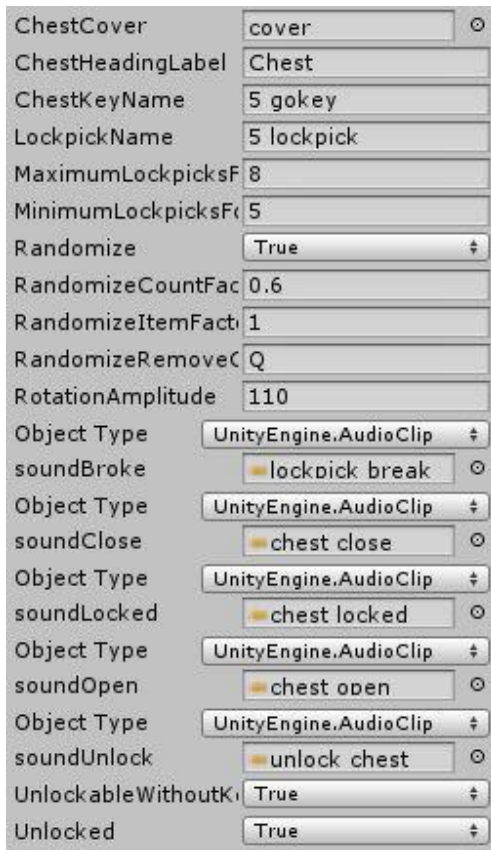
When you drink elixir of strength it will increase base_strength because effect is permanent but in equip case effect is temporary so this approach is necessary. Main reason for this is Save because this way we just store base_strength, not both values and on load we just reequip every item and we get proper values. This is done for every player attribute so every equip will change temporary value and every permanent effect will change attribute directly.

CONTAINER SYSTEM AND ITEM

EXPLANATION

When you create a new chest on scene you have to specify few things first (Picture No. 23):

- ❖ Create chest System to the scene and it must be tagged as chest in order for player to be able to find it when near.
- ❖ This is your chest inspector, have a look at this image, those are things that need to be specified for your chest to work nicely, some of them are required and some are optional



Picture 24

specified for your chest to work nicely, some of them are required and some are optional

- ❖ You have to specify Chest Cover game object in inspector of that chest so you can use rotation to open it or you can just use animation if chest has any. Specify cover by dragging game object that you want to use rotation. This time I used simple rotation, it's just for show, if you want you can download Itween animations, they have many more options for nicer look.

- ❖ You can specify chest open, chest close, locked, unlock and lockpick broke sounds. They are already there, if you want you can change them but those sounds work pretty well.

- ❖ If you want your chest to be opened and no need to unlock it by key or lockpick you just can set bool value "Unlocked" in editor at the bottom of the screen to be true and chest will open without lock check.

- ❖ If you want your chest to be unlocked set bool value "Unlocked" to true in the inspector, otherwise if you set it to false then chest will be locked.
- ❖ If you want for chest to be unlocked only with keys just set bool variable "UnlockableWithoutKey" to be false and it can't be opened without proper keys in inventory. If you set it to true then you will have option to unlock chest with lockpicks. How lockpick system works, when chest is added to scene and game starts it will get random value between minimum and maximum integer values defined in inspector which is called "MaximumLockpicksForUnlock", in this case it is 8 but default will be 5

“MinimumLockpicksForUnlock”, here is 1 but default will be set by default to 5, you change it to any number if you want. This value we get by random will be our chest lockpick attempts necessary for chest to be unlocked so if you have 5 lockpicks in inventory and you get random number 2 you will have to try 3 times in order to unlock it because you will break 2 lockpicks when trying to unlock the chest

- ❖ You have option to set Chest label in inspector which will be name displayed on top of the screen above chest items when you open chest, it's set to “Chest” by default you can change it depends on your container name, for example you want to exchange items with some NPC character or want to pick from crate or barrel you will have to specify what name you want to see when you open container. And also if you change container to be barrel or crate or any other don't forget to disable or change animation for opening the chest.
- ❖ You also have the option to randomize container with setting the value true or false and setting randomize factors which will decide how you randomization works.

When you have your container in place and everything set up for beginning next thing to do is to specify items you want to appear in your container. If chest is empty when you are near it with character you will get label “Open empty chest” because that chest doesn't contain any items inside. To specify items in chest you have to add values to chest Hash tables by adding key pairs.

Every container including shop has minimum two hash tables and one array list. In this case your chest has “items” which will be used for storing game objects inside data base and “count” Hash Table Proxy components on the same game object. You need to set “items” hash Table to have PrefilType set to GameObject and “count” must be set to Integer because we want whole numbers, not floats.



Picture 25



Picture 26

This is how they look without any items (pictures 24 and 25). Reference field is defined in both of them because we have multiple hash tables on same game object (when you have more than one hash table on same object you have to specify table reference Reference to make sure you are working with right one). This is how you fill HashTableProxy component with items:



Picture 28




Picture 27

So how we do this? First you define your items (hash on the left side picture 27) and then you define how many of each items you want in your container. For potion of experience we set it to two because we want only those two kind of potions to appear inside in this example. Every item has two values, one is item key value which will represent item UID, it is unique and other value is game object itself so you just drag game object that is defined with this key to that field. Using item UID system can find items, store them extract from databases and all other operations are going over this UID. Value on the left is item UID (3 poexp means 3 for consumable, po means potion and exp means experience, for next item 3 means consumable, el means elixir and dex is short of dexterity and for the ring 2 means apparel, go means golden or made of gold and ring is a ring). After you define items like this your system is able to recognize items and store them as unique in Hash Tables.

To understand this better, have a look at the next example, potion of health GUI will be explained and how to define your parameters in the inspector properly:

So let's start with first field, **Attribute**, in this case is defined as Health so it means that

Attribute	Health
AttributeValue	50
GameObjectPrefab	Potion of health
Icon	
ItemDisplayName	Potion Of Health
ItemEffect	Restoration effect
ItemInfo	This potion will restore
ItemPrice	100
ItemType	Consumable
ItemUID	3 pohp
UsageTime	0.7

Picture 29

variable **currentHealth** will be changed and increased by value (second field) **AttributeValue** which is 50 in this case. You can define your attribute with lower or upper case, it doesn't matter, system will convert case to lower anyway. **GameObjectPrefab**, this is required field you need to define for your physical item to appear on the scene that has collider, rigid body and other definitions so in that field you specify your physical item by dragging to that field.

Next on the list is item **Icon**, here you define your item GUI texture you want to see on screen, just your icon here.

ItemDisplayName which will represent your item name that displays on the screen.

ItemEffect, just for displaying info when you mouse over, this is heal potion so it has restoration effect, for elixirs such as strength elixir you will set it's permanent effect etc...

ItemInfo, here you define your item information that will be displayed when you mouse over this item, in this case I set info to be: "This potion will restore 50 points of your health when used"

ItemPrice. This is item base price which means item value, you can modify this in your Interact FSM of inventory where you can set what price to display, divide this value or multiply by factor that you can get from current shop or any other value you want.

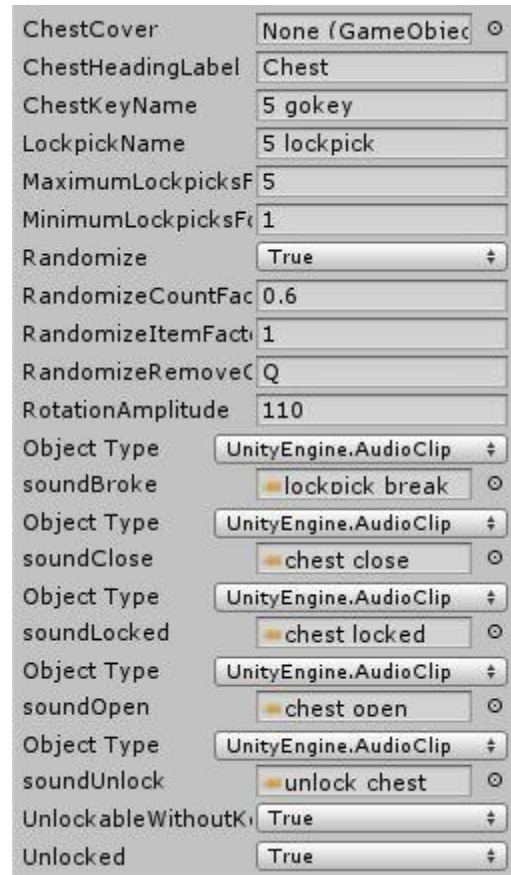
ItemType, this is important for your items because system will recognize few types that can exist (you can change this as well anyway you want), so item can be equipable, consumable, usable or other. If item type is other that means that you can't use, consume or equip that kind of item, those items are lockpicks, keys etc... they will activate upon approaching container

ItemUID, this is very important variable to define correctly. By this value item can be recognized and set to data base because system uses this value to operate with item and this value will be needed when defining items in HashTables. Each item has unique ID and by this value system will sort, organize and display your items properly.

Container FSM components

Container system has 5 FSM components:

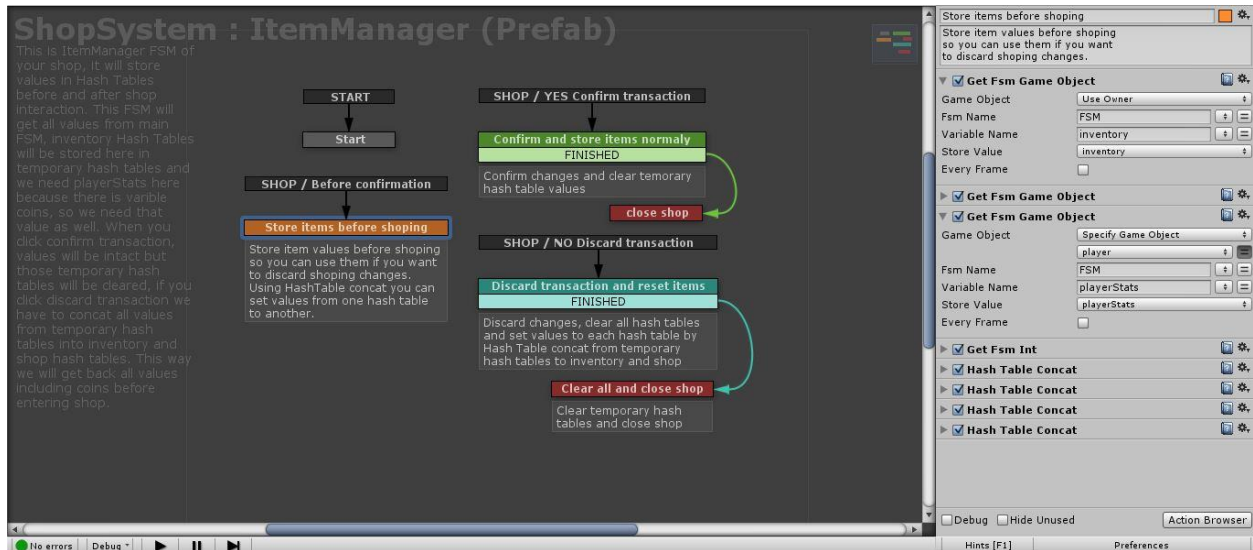
- **Arrange** – this FSM is responsible for sorting and displaying items exactly like in inventory Arrange FSM. It will create list, get all items from “items” hash table and create one by one and parent them to the item list with calculating their position on screen based on values you set for this FSM.
- **FSM** – This is main FSM of container and it will be active upon opening container, event called **CONTAINER / Open** will be sent by objectInteractSystem when you approach container and press key to open. Actions in this state will play animation for opening container and will get all necessary values and game object variables in order for container to work properly. We need to find UI Root and all game objects from that game object including inventory and others. When you close container event **CONTAINER / Close** will be called and container will close with setting necessary values and playing animation. In this fsm as you can see on picture 29 you can set a lot of values, you can set Labels, key name, lockpick name, max and min for unlock, to randomize or not, randomize values and sounds for each action. There is option to set is container unlocked and unlockable without key. If that is true than you can use lockpicks.
- **KeyUnlocking** – This is FSM that will check for certain keys in your inventory, if you have them container will be unlocked, if you don't it will display warning about that key required to open.
- **Lockpicking** – If you defined in your container that you can open container with lockpicks then this FSM will become active and you can begin lockpick process by randoming necessary lockpicks before container is opened.
- **ScrollList** – this FSM is same as scroll in inventory, it will display when you scroll with mouse wheel up and down.



Picture 30

SHOP SYSTEM

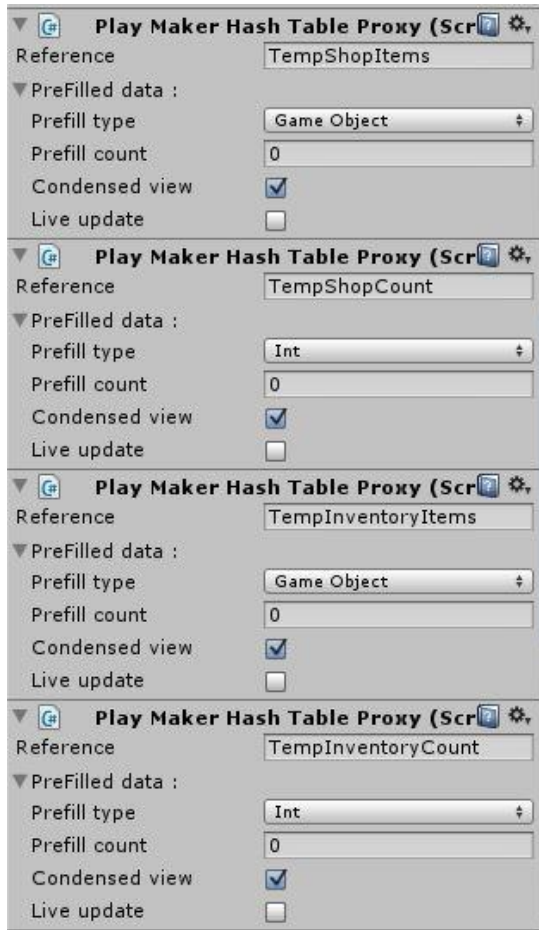
Similar to container system shop indeed has some differences while generating items and hash table item add / remove is almost the same as we saw in chest and inventory, only difference in shop is that we have options to discard and confirm transaction after shopping which requires additional HashTables.



Picture 31

On the picture No. 30 you can see ItemManager of shop system and we use this system to store all items before and after shop. To store values we use Hash Table Concat action and we practically copy value from one hash table to another. Because in inventory we have two hash tables (one for database and one for item count) and also in shop we have those two, to be able to store all items somewhere we need 4 additional hash tables for each hash table that we have in inventory and shop. After confirmation we don't have to restore values because values are already set in the way they need to be, only thing left to do is to clear those additional hash tables for next usage just to keep everything clean and to prevent bugs. But if we choose to discard transaction and want to restore items and gold to be same as before shop we have to concat values from those temporary hash tables into inventory and shop hash tables and to send event INVENTORY / Create items to Inventory that we need to recreate our inventory list using same method I mentioned earlier.

You can see those additional hash tables on picture 31 as you can see I had to specify reference



Picture 32

for each hash table because you have more than one hash table on same item. When you have only one hash table proxy component on item you don't need reference because system know that one table is there however in this situation system would be confused so we have to specify those reference values. TempShopItems is used for game objects and TempShopCount is used to count items in shop. (same is with inventory hash tables).

In shop like in chest and inventory you have option in inspector to define your maximum number of items (default is 35).

Shop like chest don't have get owner action because there are many chests and shops around so when you are close to the chest or shop and press E to open then it becomes current shop or chest.

Shop can be closed by clicking on X button or you can just press Escape and confirmation window will appear.

Rest of the stuff is the same as in other containers, FSM component is main and there will be sent event **SHOP / Open** by objectInteractSystem when is approached. Only difference is that you have **PriceFactor** that will define your item price when you buy them from shop, you can change this value based on the shop or player level or whatever you want, when shop is about to open you can create additional state before opening where you want to calculate your **PriceFactor**.

CRAFTING SYSTEM

Edit – crafting system reworked totally

New crafting system from version 3.0 will allow you to create new items using item components in crafting process. For this system you have to create workbench using



Picture 33

Tools/Squarebite/Create system/Workbench or you can drag it to the scene. Under your UI root there is CraftingDatabase and there you have to define your items you want to be craftable. Here is the example of that (picture 33), as you can see on the left side of the image we have UID codes for each item in our database, based on that system can

know what are components required for crafting. Here we have multiple UIDs for each

item which means that for Elixir of health (on the bottom of the list) we need potion of health (3 pohp is UID for that item), potion of experience (3 poexp) and poison (3 popsn). Those are requirements for that item and if you put those items in your workbench and click craft you will get elixir of health in your inventory, simple as that 😊.

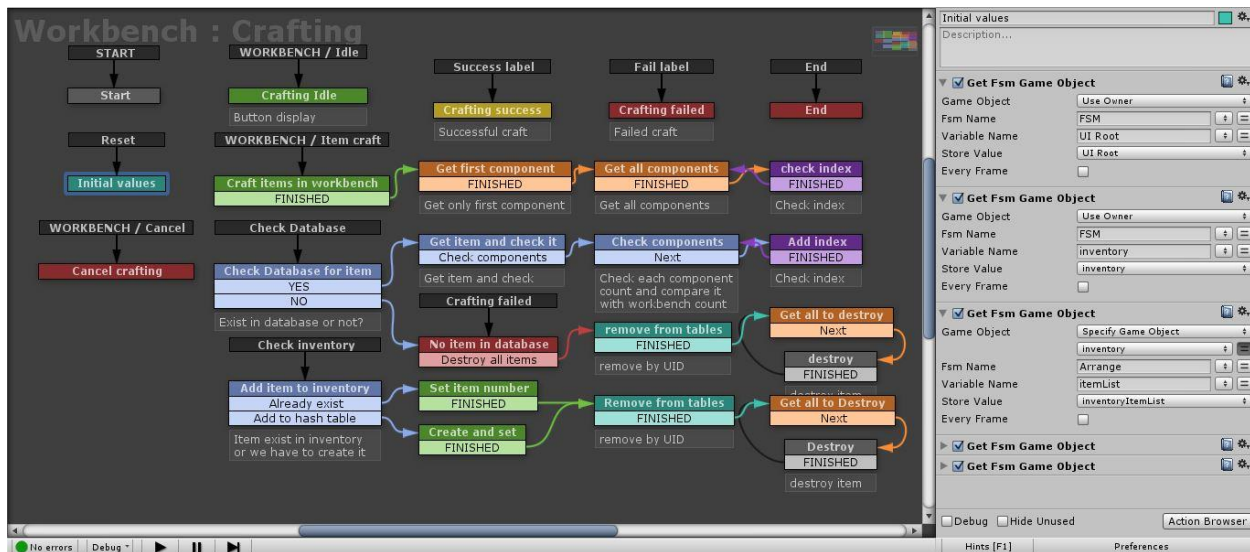
Note that components must be defined in alphabetical order, for that elixir you will notice that UID “2 irplate 4 iringot” is composed of 2 UIDs in alphabetical order, 2 before 4. Same for 3 poexp is set before 3 pohp because that UID is in alphabetical order “e” before “h”

That is crafting with single items, however if you want to craft multiple items for example you have strategy game where you need 50 pieces of wood and 20 pieces of stone to build a barrack you can do that as well with this approach, here is how:



Picture 34

On the picture 34 you can see that we have integer type of Hash Table Proxy component, that is the component of our item we want to craft. Simple add Hash Table Proxy component to item and define by UIDs how many each components you want in order to craft something. In this case this is Crafting test item and for it we need 1 iron plate and 3 potions of health and after we press craft, new item will appear in our inventory.



Picture 36

On the picture above you can see how crafting works and how components are checked. If you click craft system will combine UIDs of all components you put in workbench using format string action, that value and only important thing here is to define your UIDs for crafting in alphabetical order in order to work (I tried to use combination algorithm but that is way too much complicated, this is far easier)

Note about Crafting UID for items – To define UID for your item result you have to define in alphabetical order because system will get all components UID one by one, sort them out format those string values into new value and check is that item UID defined or not in our CraftingDatabase, if yes item will be created, if not you will get warning crafting failed. It doesn't matter in what order you put them in workbench they will be sorted anyway, it is important in what order you define your item UIDs, so always use alphabetical order when defining UID for your crafting item.

There is one option in crafting, you can set to destroy or not destroy components in your workbench if crafting fails. If you choose to destroy than items will be removed from both hash tables and item list of your workbench, however if you choose not to destroy than items will remain in your workbench and you can get them back anytime like from any other container.

SERIALIZER AND MANU MENU

If you want to save your scene and all of your items from inventory or from scene, character stats, equipped items, player position you will have to import Unity Serializer. It's free package for Unity with a lot of possibilities and saving options, we should skip all that and explain how to use it to save scene using Playmaker items, objects etc...

First when you determining what you want to save, you have to make sure that all of your Hash tables have script store information. All Hash tables in this project will be saved and information stored and none of them have FSM component (except for StorageManager) since Serializer not saving FSM components properly. Because of this when saving something you have to expand store information script by unchecking Store All Components checkbox (picture No. 36). When Hash Table information is stored we just have to get that info on load and redefine scene. Here are some crucial things you have to store information in order to save game properly:



Picture 37

- First you will need SaveManager on scene that is provided by Serializer (script is called SavaGameManage script, you can attach that script to empty game object or you can just click in Serializer window to add save manager to scene).
- In Inventory prefab you have to store information for both ItemDatabaseHashTable and ItemCountHashTable. Also for equipped items you have to store one more hash table from inventory, that is the child of EquippedItemsManager and it's called EquippedItems. This HashTable will store only what items are equipped so on load we just send event to those items to equip and they will be on equipped on Character.
- In containers save same hash tables as in inventory which means Database and ItemCount, we don't need equipped Items hash table since container doesn't have it.
- In Shop use same store information as you did for container.
- Use store information for MainCharacter but from character we only need his position so when you expand what to save choose Transform only.
- One more HashTable left to be saved and that is StorageManager. This manager has two HashTables, one is for item prefabs that are on the scene and the other one is for position of those items. So just expand store information and choose to store only HashTables, not FSM component, not Array list only Hash Tables.

Main menu is FSM component of GameManager game object and main purpose of this is ability to save game progress as well as load / delete game.

When you press escape you will enter main menu with some basic commands such as Resume game, Options, Save Game Load Game and Quit Game.

Options are only basic, some for game volume, light and resolution change, nothing much to explain about those options, so focus will be on save / load progress.

Save and load game will use Unity Serializer for that purpose and it's free Asset, it's already included in this package without standard assets and examples, so if you want them you can reimport Serializer from Asset Store.

How save game works with this package and how items are stored? Well as you already know in order to save with Serializer you have to specify what you want to save so in this case I want to save my hash tables for inventory, shops and chests + I want to save my character stats int and string values for stats, want to save all items that are on the scene + all items that are equipped on my character, so how can I do that?

Serializer doing here just great job since it has the options to store game objects which is enough for saving inventory and entire scene with only few hash tables. Using hash table game object values and hash table count values you can just tell your inventory that you want those items to appear in your inventory list in numbers defined in hash table count and that's it. We store all items into hash table, and all items are in table as reference of prefab (this is why all items have two components because we can always get Prefabs from items), so we just store item list and on load we tell our inventory to destroy current item list and create new with items that have to be loaded. Similar concept is used for scene and character items. It would be trouble for system to store entire scene and all items and character with all joints and all fsm's and materials, textures.....which is hell for system. I created some simpler way where we store only list for all items on scene and we get their positions. When you pick item it's automatically removed from this hash table, and when you drop item it's added to hash table and item position is saved. When you load your game, system will check what items to create and where.

For equipped items we use exact same method and after scene is loaded and we get what we stored, only thing we have to do is to find those items under our items list and send events to those items to equip themselves. It's not smart to store everything, especially because every equipped item has a background, label, some of them have two backgrounds, so best way is to find what should be equipped find those items and send events to them to equip.

This exact method can be used for storing enemies and their positions on level, get all enemies, get their positions when you saving game and on load restore enemy list from scene, create all enemies and set their positions from Hash Table.

You can probably do this on your own way, but I wanted to make things simpler because I created this for my own game if you remember ;)

For character stats there are bunch of int values so we serialize them into one hash table that will contain int values and on load we just restore those values from stored hash table.

When you save game it will take save information such as date and time, you have 10 slots for saving games and also options to delete saves. When you press save game button it will serve same as quick save because you don't get option to save to certain slot, instead of that system will save to next slot so you don't have to worry about that, slots will be filled with game info and when you want to load you will have slider to choose which game you want to load.

Look at the image No. 38 to see load menu and example of saved games. Also there is delete button and if pressed you will get same menu and when you click on a game it will be deleted instead of loaded.



Picture 38

You can define your save name, I just set it to be Save, you have save information and system time is added so you can know when you saved and what is latest save you created.

When game is save it will take first free slot to save a game and earlier saves will be replaces with newer saves when no slots are available.

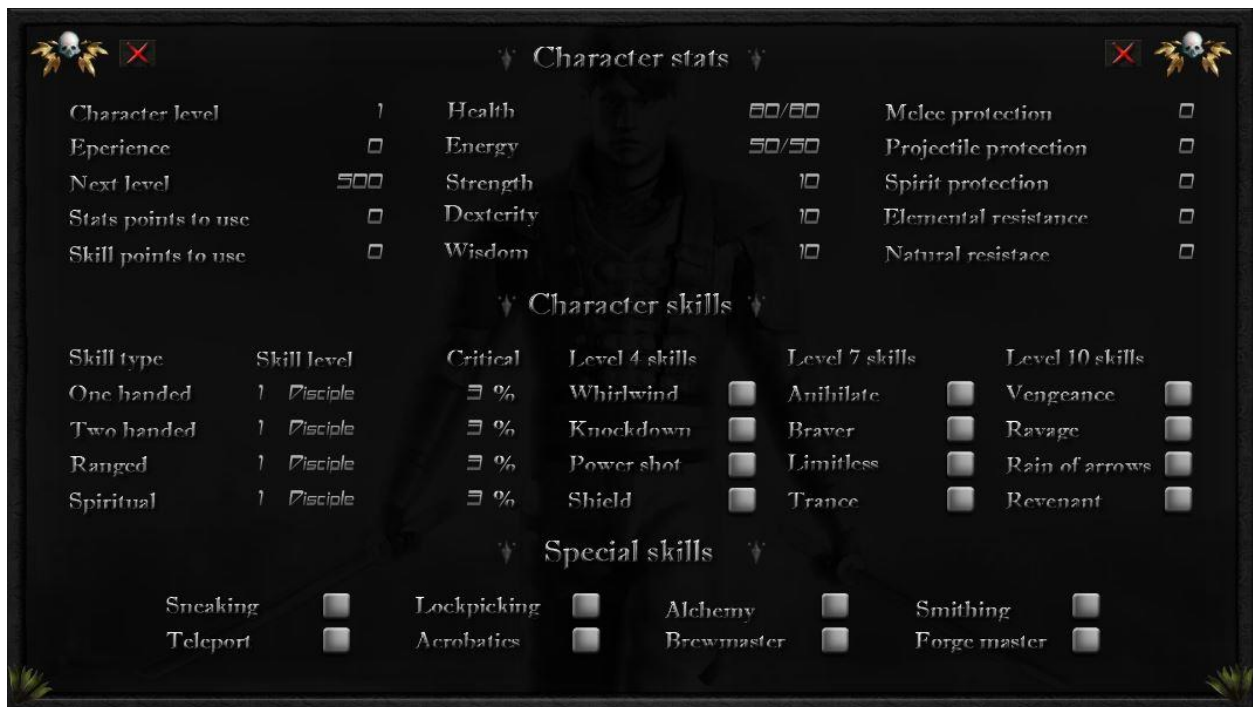
If you don't like this type of saving feel free to post a comment on my Squarebite forum so we can discuss about suggestions and ways to improve this package.

PLAYER STATS AND GUI

Edit – now you have playerStats instead of CharacterStats and also you have PlayerStatsGUI under your UI Root game object. To define player attributes, skills and everything else you have to define that in playerStats game object that is child of your player. PlayerStatsGUI will open / close or show / hide your stats on screen, it will get all values, format them to strings and display on screen using GUI.

As a bonus to this package I give you character stats with some basic and advanced features that you can use if you like this concept. This screen and entire PlayerStats system is created for game A.F.T.E.R. as you can see that background of stats window.

Here you can display your stats, level, experience, health, energy and also you can get skills / stat point to use as you like when enough experience is earned.



Picture 39

As you can see on picture No. 39 there are a lot of some different skills that I wanted to use and in time I will add those skills in my game, you can make your own background image with different type of skills that will fit your game. So what's basically happening here when it comes to display stats on screen and how they are monitored?

There are two different game objects that you need in order to display your playerStats, first you need to add **playerStats** game object under your player and to drag that object into **playerStats** variable of your player, and second is **PlayerStatsGUI** that is already under your UI Root game object. In playerStats we define every value we need for our game and we change those values based on actions. For each attribute in playerStats we have base value and equipped value. For example base_strength is your player strength that you increased with levels or potions, so that value is permanent and will be stored if you save a game, on the other hand we have equipped_strength which is value you get from equipping items. Player actual strength is calculated by adding those values together using IntOperator action.

Health FSM will monitor your health and it will set values to your Health bar so you can see how that value transition from last value to current value. In initial values state this FSM will get bars from UI Root and using GetChild action we can get Health bar game object from Bars.



Picture 40

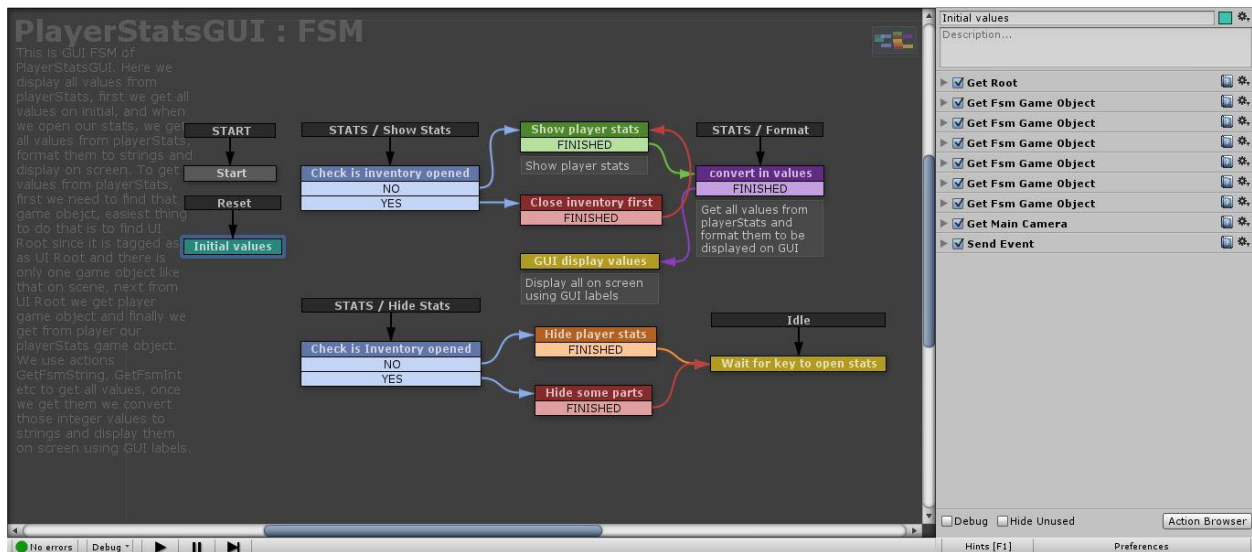
Energy FSM (look exactly the same as Health FSM) will monitor your energy and it will set values to your Energy bar so you can see how that value transition from last value to current value. In initial values state this FSM will get bars from UI Root and using GetChild action we can get Energy bar game object from Bars.

LevelSystem FSM is responsible for your player experience, levels, skill and stats points per level and It will calculate experience for next level using simple algorithm. You can set in the inspector experience per level, in this case this means that for first level you will need 500 experience, for second you need 1000 experience, for third 1500 etc.

For displaying Stats we have **PlayerStatsGUI** game object under our UI Root and GUI work similar to what we have in our Inventory system. It works by activating camera that will cull only GUI layer and displaying bunch of GUI labels which represent all attributes, skills... (Picture 40). Also if inventory is opened and you press C key it will first send event to inventory manager that inventory needs to close and then stats can be displayed. All values we get from playerStats and convert them into strings so they can be displayed on screen using GUI. There are few FSM components on this PlayerStatsGUI:

Display FSM will display any changes to your playerStats variables, for example you drink potion of experience and you get +220 experience, that change and value will be displayed on screen. Any change to playerStats variable will be displayed here in this FSM.

FSM on PlayerStatsGUI game object will wait for event **STATS / Format** which will be used to convert all integer values into strings so they can be displayed on screen using GUI label.



Picture 41

StatsUsage FSM will display your stats points to boost your attributes. It will check how many skill you have in playerStats and if 0, buttons will be hidden, when you level up you will get 1 stats and 1 skill to spent and buttons will be shown

SkillUsage FSM will display your skill points to boost your skills. Works same a stats usage where we get player stats and if you boost some skills you have to set that up in playerStats as well.

Sounds FSM works exactly the same as Sounds in inventory and we send event to this fsm based on sound we want to play.

FEW WORDS FOR THE END

I hope you can understand Squarebite inventory system better and that you set up your scene without problems. Don't forget to try all features that Squarebite inventory system has to offer and if you come on screen help you can just click on next instructions button and you will see some explanations about controls, movements and everything related to scene guide. I hope you understand Playmaker even better after using this package and reading this guide.

If you have any question, problem or maybe some suggestion / request you can visit my forum Squarebite on <http://squarebite.proboards.com/> and post your question problem, request there and I will respond as soon as I can.

Special thanks to Mike for making this free Unity Serializer that I used for saving and [Dredile](#) for this cool ambient music.

Enjoy with Squarebite, more systems and packages coming soon so stay tuned.

Cheers!!!

