BA INVENTORY - PROCESSES

Target of this Document

This is the second process map for starting a small RPG/Survival like game on top of Unreal Lyra

The first focused on base processes and entities found in Lyra and how to integrate them. It can be found here

This document will focus on higher level processes, mainly
- Item and inventory management
- Melee Fighting
- World Building

It will extend Lyra concepts as these aspects are not covered in the Unreal Lyra demo

Videos

Inventory 01 - covering inventory strategy, **Global Inventory Manager** and **Inventory Component**

Inventory 02 - data driven ingest of **Dungeons&Dragons weapons**, anatomy of the **BP_Item** class and how to create and use fragments

Inventory 03 - Common UI considerations, building of a scrollable inventory grid

ITEMS AND INVENTORIES

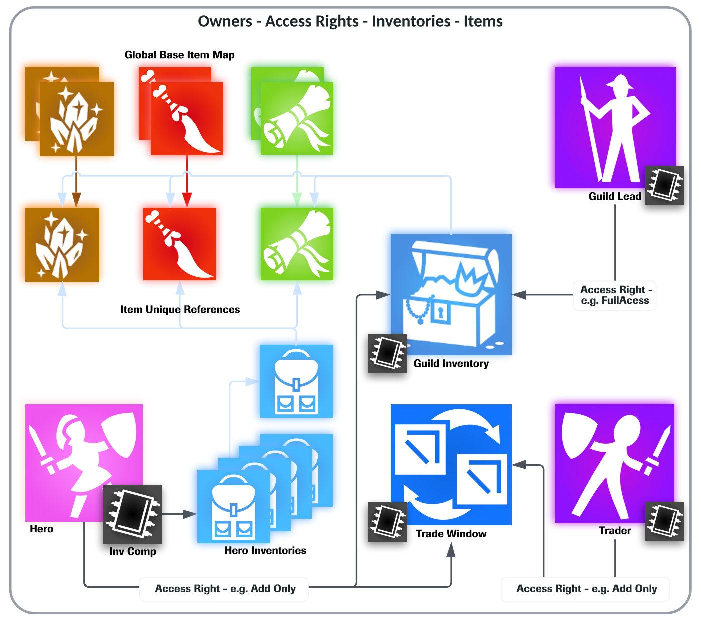
Principles

- General principles - security
  - All Item and Inventory processes are managed on Authority only
  - Remote client only receives UI related information
- Scalability
  - All components are not directly coupled but message based
  - High load data access is based on cached (Data Registries) or in-memory storage
- Owner ◂ ▸ Inventory Relations
  - Owner can have multiple inventories - permanent (equipment) or temporary (chest)
  - Owner can have different access rights per inventory - Add Only, Full Access, Read Only, ...
  - Inventory can have multiple owner with different access levels
- Items
  - Items are always data driven via databases, data registries or data tables
  - Items are composed of fragments detailing different aspects (name, weight, durability, ...)

Multiple inventories, multiple users, different access rights

Base and Instanced
- Items and Inventories are loaded from Data Tables as their Base Template version
- Both get initialized on usage and a unique Guid is added
- This allows for unlimited variations of a base version

Inventory Component
- The owner of an inventory component can be a hero, a chest, a stone you look under
- Each Inventory component can have relation with unlimited inventories throughout the world

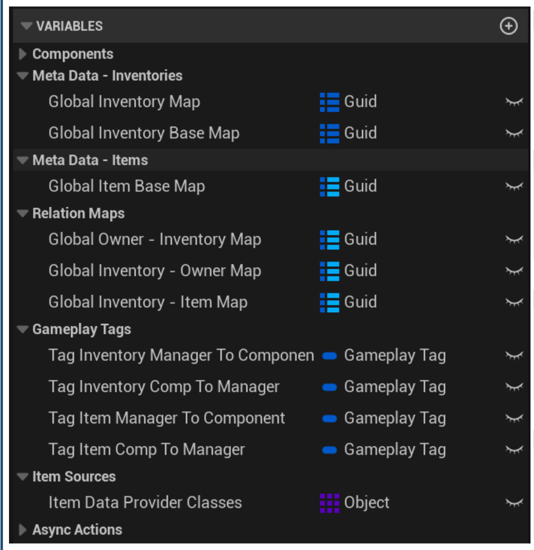


Owners - Access Rights - Inventories - Items

Right Management
- Each owner-inventory relation can have right as ReadOnly, AddOnly, RemoveOnly, FullAccess or None (to delete relation)

Flexible Assignments
- The list of inventories stay throughout the game
- Relations bweten owners and inventories are fluent and can be added or removed depending on game situation

Global Inventory Manager - single instance to manage items and inventory relations



VARIABLES
- Components
- Meta Data - Inventories
  - Global Inventory Map — Guid
  - Global Inventory Base Map — Guid
- Meta Data - Items
  - Global Item Base Map — Guid
- Relation Maps
  - Global Owner - Inventory Map — Guid
  - Global Inventory - Owner Map — Guid
  - Global Inventory - Item Map — Guid
- Gameplay Tags
  - Tag Inventory Manager To Componen — Gameplay Tag
  - Tag Inventory Comp To Manager — Gameplay Tag
  - Tag Item Manager To Component — Gameplay Tag
  - Tag Item Comp To Manager — Gameplay Tag
- Item Sources
  - Item Data Provider Classes — Object
- Async Actions

Global Inventory Map
- Map of Inventory Guid ◂ ▸ Base Inventory Guid
- Map to show relation between a unique inventory and a base inventory template
- On initialization a base inventory template gets an inventory guid and this relation is stored here

Global Base Inventory Map
- Map of Base Guid ◂ ▸ Inventory Struct
- List of all inventory templates loaded from Data Table
- On Request, a Base Inventory is initialized with a unique Inventory Guid and assigne to the requesting owner - normally an Inventory Component

Global Item Base
- Item Base Guid ◂ ▸ BP Item
- This is a list of all base items existing inthe game
- Base Guid is defined in the data driven item sources and will be initialized into many instances of said base item with different Item Instance Guids

Global Owner - Inventory Map
- Owner Guid ◂ ▸ Inventories Map
- A unique owner (=inventory component guid) is mapped to 1..n inventories
- This allows a direct lookup "Get all inventories for a defined owner"

Global Inventory - Owner Map
- Inventory Guid ◂ ▸ Owner Map
- A unique inventory guid is mapped to 1..n owners
- This allows a direct lookup "Get all owners for a defined directory"

Global Inventory - Item Map
- Inventory Instance Guid ◂ ▸ Inventory-Items-Relationship
- Global Map to host relations between one inventory and one BP_Item. The guid key is the inventory instance guid, the values a Inventory- Items-Relationship instance

Tag Inv Manager ▸ Inv Component
- Gameplay Tag
- Defines channel to send inventory related information from Manager to Component
- Uses struct *InventoryExchangeInfo*

Tag Inv Component ▸ Inv Manager
- Gameplay Tag
- Defines channel to send inventory related information from Component to Manager
- Uses struct *InventoryExchangeInfo*

Tag Item Manager ▸ Inv Component
- Gameplay Tag
- Defines channel to send item related information from Manager to Component
- Uses struct *ItemExchangeInfo*

Tag Item Component ▸ Inv manager
- Gameplay Tag
- Defines channel to send item related information from Component to Manager
- Uses struct *ItemExchangeInfo*

Item Data Provider Classes
- Data Provider classes used to load Data Tables
- Standard pattern should be only using data driven item generations. All base items are loaded into Global Item Base Map at start.
- All initialized items are stored then in *Global Inventory - Item Map*

Publisher-Subscriper pattern: Lyra Gameplay Message Router

Publisher - Subscriber Pattern

An example for this pattern could be a widget listening to a game channel to get updates for its content - without knowing how that content update is produced.

So the pattern can be characterized as:

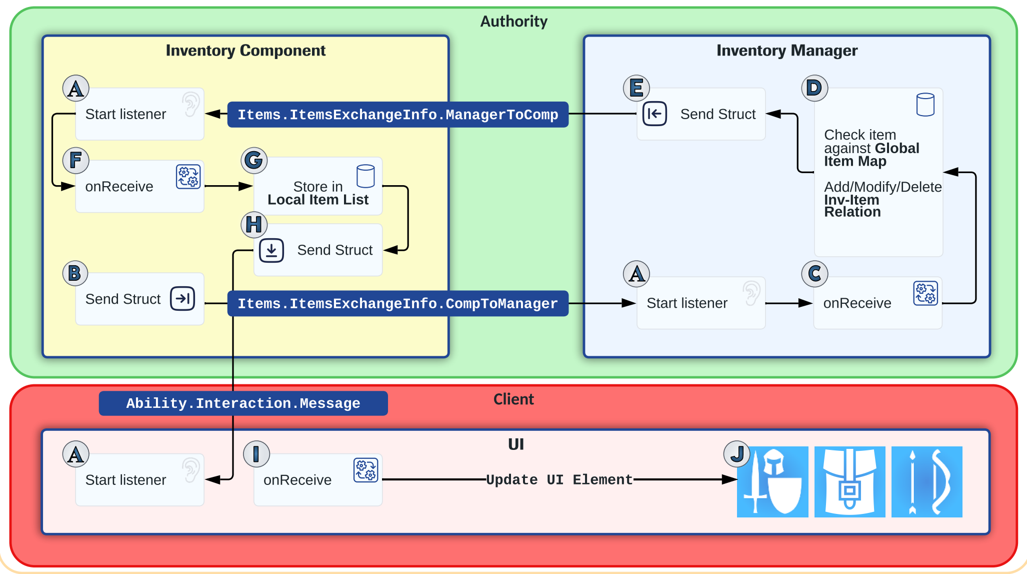
- a messaging pattern where **publishers** categorize messages into structures that are received by **subscribers**
- this is contrasted to the *observer pattern*, where e.g. delegates are used to watch certain processes and receive updates
  - This is Unrealls standard pattern for BluePrint-to-BluePrint communication - but lacks decoupeling between information producer and consumer
- advantages are
  - decoupling between producer and consumer
  - extreme scalability - can easily leverage cloud based messaging systems
- disadvantages are
  - decoupling makes it harder to track who consumes and reacts to messages
  - this tracking is needed if message structures are changing to make sure all subscriber can react to the changed data structures

Lyra Gameplay Message Router

"A subsystem that allows registering for and sending messages between unconnected gameplay objects"

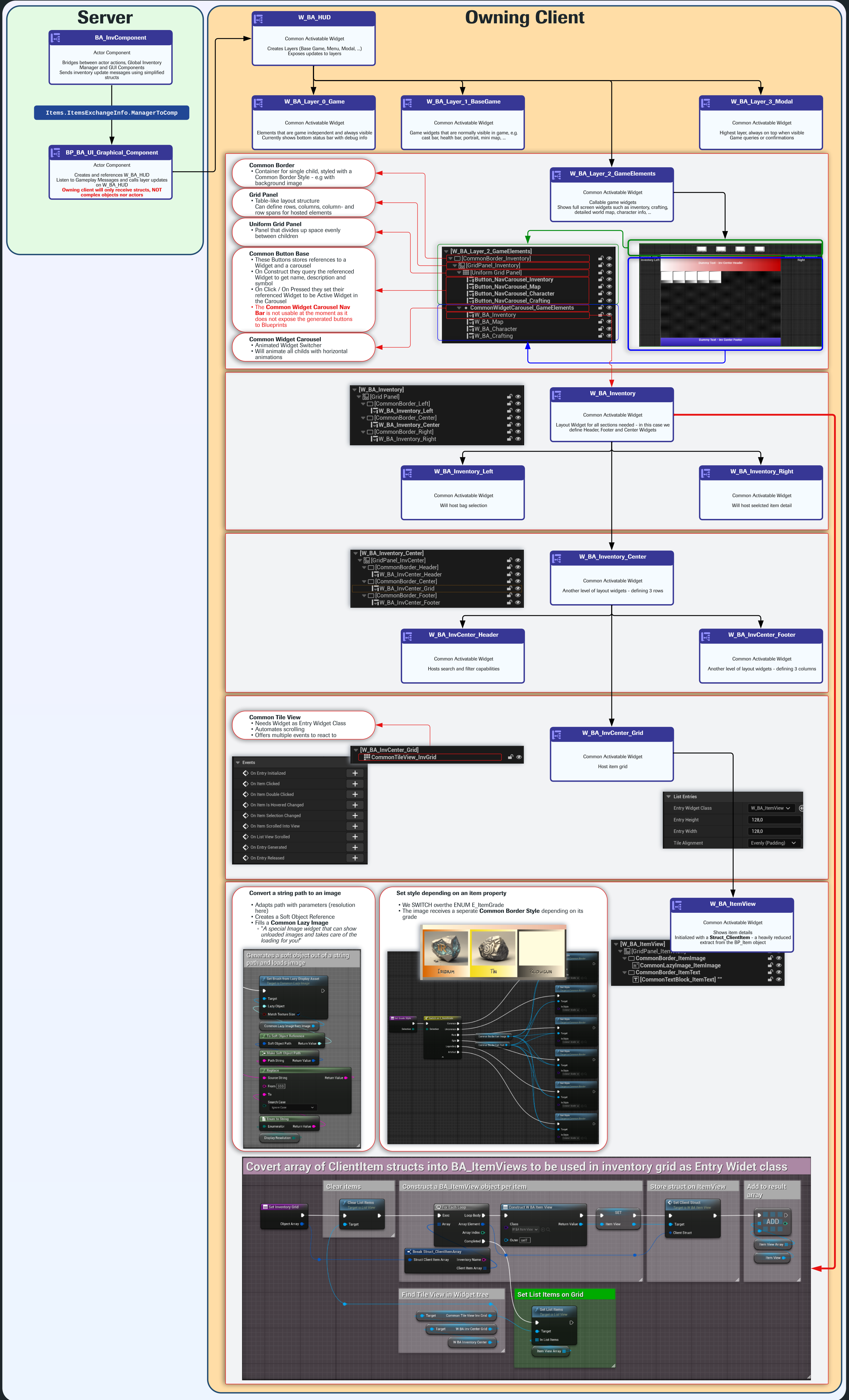
- **Lyra Plugin**
  - can be found in every Lyra Installation at ***LyraInstallDir\Plugins\GameplayMessageRouter***
  - recommendation is to copy it to your engine plugin folder to have it available for all projects

Example: Communication with Inventory Manager and GUI for adding an item that was harvested



Authority
Inventory Component
A Start listener
F onReceive
G Store in Local Item List
H Send Struct
B Send Struct
Items.ItemsExchangeInfo.ManagerToComp
Items.ItemsExchangeInfo.CompToManager
Inventory Manager
E Send Struct
D Check item against Global Item Map
Add/Modify/Delete Inv-Item Relation
A Start listener
C onReceive
Client
Ability.Interaction.Message
UI
A Start listener
I onReceive
Update UI Element
J

COMMON UI



Server

BA_InvComponent — Actor Component — Bridges between actor actions, Global Inventory Manager and GUI Components. Sends inventory update messages using simplified structs

Items.ItemsExchangeInfo.ManagerToComp

BP_BA_UI_Graphical_Component — Actor Component — Creates and references W_BA_HUD. Listen to Gameplay Messages and calls layer updates on W_BA_HUD. Owning client will only receive structs, NOT complex objects nor actors

Owning Client

W_BA_HUD — Common Activatable Widget — Creates Layers (Base Game, Menu, Modal, ...) Exposes updates to layers

W_BA_Layer_0_Game — Common Activatable Widget — Elements that are game independent and always visible. Currently shows bottom status bar with debug info

W_BA_Layer_1_BaseGame — Common Activatable Widget — Game widgets that are normally visible in game, e.g. cast bar, health bar, portrait, mini map, ...

W_BA_Layer_3_Modal — Common Activatable Widget — Highest layer, always on top when visible. Game queries or confirmations

W_BA_Layer_2_GameElements — Common Activatable Widget — Callable game widgets. Shows full screen widgets such as inventory, crafting, detailed world map, character info, ...

Common Border
- Container for single child, styled with a Common Border Style - e.g with background image

Grid Panel
- Table-like layout structure. Can define rows, columns, column- and row spans for hosted elements

Uniform Grid Panel
- Panel that divides up space evenly between children

Common Button Base
- These Buttons stores references to a Widget and a carousel
- On Construct they query the referenced Widget to get name, description and symbol
- On Click / On Pressed they set their referenced Widget to be Active Widget in the Carousel
- The **Common Widget Carousel Nav Bar** is not usable at the moment as it does not expose the generated buttons to Blueprints

Common Widget Carousel
- Animated Widget Switcher
- Will animate all childs with horizontal animations

```
[W_BA_Layer_2_GameElements]
  [CommonBorder_Inventory]
    [GridPanel_Inventory]
      [Uniform Grid Panel]
        Button_NavCarousel_Inventory
        Button_NavCarousel_Map
        Button_NavCarousel_Character
        Button_NavCarousel_Crafting
    CommonWidgetCarousel_GameElements
      W_BA_Inventory
      W_BA_Map
      W_BA_Character
      W_BA_Crafting
```

```
[W_BA_Inventory]
  [Grid Panel]
    [CommonBorder_Left]
      W_BA_Inventory_Left
    [CommonBorder_Center]
      W_BA_Inventory_Center
    [CommonBorder_Right]
      W_BA_Inventory_Right
```

W_BA_Inventory — Common Activatable Widget — Layout Widget for all sections needed - in this case we define Header, Footer and Center Widgets

W_BA_Inventory_Left — Common Activatable Widget — Will host bag selection

W_BA_Inventory_Right — Common Activatable Widget — Will host seelcted item detail

```
[W_BA_Inventory_Center]
  [GridPanel_InvCenter]
    [CommonBorder_Header]
      W_BA_InvCenter_Header
    [CommonBorder_Center]
      W_BA_InvCenter_Grid
    [CommonBorder_Footer]
      W_BA_InvCenter_Footer
```

W_BA_Inventory_Center — Common Activatable Widget — Another level of layout widgets - defining 3 rows

W_BA_InvCenter_Header — Common Activatable Widget — Hosts search and filter capabilities

W_BA_InvCenter_Footer — Common Activatable Widget — Another level of layout widgets - defining 3 columns

Common Tile View
- Needs Widget as Entry Widget Class
- Automates scrolling
- Offers multiple events to react to

```
[W_BA_InvCenter_Grid]
  CommonTileView_InvGrid
```

Events
- On Entry Initialized
- On Item Clicked
- On Item Double Clicked
- On Item Is Hovered Changed
- On Item Selection Changed
- On Item Scrolled Into View
- On List View Scrolled
- On Entry Generated
- On Entry Released

W_BA_InvCenter_Grid — Common Activatable Widget — Host item grid

List Entries	
Entry Widget Class	W_BA_ItemView
Entry Height	128,0
Entry Width	128,0
Tile Alignment	Evenly (Padding)

Convert a string path to an image
- Adapts path with parameters (resolution here)
- Creates a Soft Object Reference
- Fills a **Common Lazy Image**
  - *"A special Image widget that can show unloaded images and takes care of the loading for you!"*

Generates a soft object out of a string path and loads image

Set style depending on an item property
- We SWITCH overthe ENUM E_ItemGrade
- The image receives a seperate **Common Border Style** depending on its grade

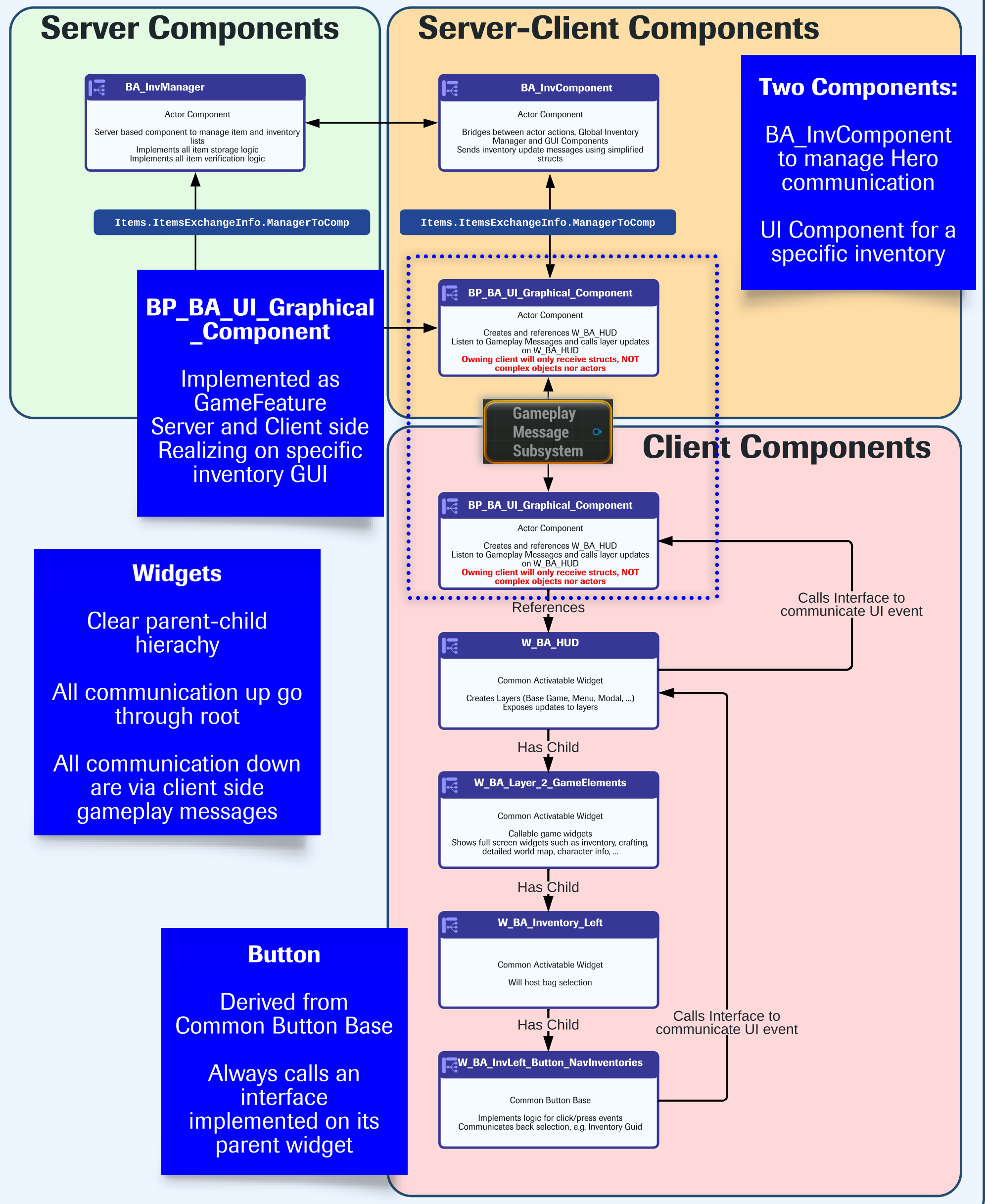
W_BA_ItemView — Common Activatable Widget — Shows item details. Initialized with a **Struct_ClientItem** - a heavily reduced extract from the BP_Item object

```
[W_BA_ItemView]
  [GridPanel_ItemView]
    CommonBorder_ItemImage
      CommonLazyImage_ItemImage
    CommonBorder_ItemText
      [CommonTextBlock_ItemText] ""
```

Covert array of ClientItem structs into BA_ItemViews to be used in inventory grid as Entry Widet class

Communication between Widgets and Server

- General principles
  - Widgets are client based and cannot directly broadcast to Server nor can they replicated variables to server
  - We need one bridge component to collect all widget requests and post all server responses to widgets
- Where to place the server connection logic?
  - I often see the recommendation to use e.g. the PlayerState, but don't agree - this will clutter it with a lot of references - quickly creating a player of Gigabytes to load
  - My recommendation is to use an **ActorComponent**
- Ease of use
  - Try to keep game logic as far as possible out of Widgets - they really should only manage their components, but nothing outside
  - I use a central root Widget (W_BA_HUD) as the funnel component to communicate with the Actor Component. Both, the HUD widget and the Actor Component are implementing the same interface



Server Components

BA_InvManager — Actor Component — Server based component to manage item and inventory lists. Implements all item storage logic. Implements all item verification logic

Server-Client Components

BA_InvComponent — Actor Component — Bridges between actor actions, Global Inventory Manager and GUI Components. Sends inventory update messages using simplified structs

Items.ItemsExchangeInfo.ManagerToComp

Two Components:

BA_InvComponent to manage Hero communication

UI Component for a specific inventory

BP_BA_UI_Graphical_Component

Implemented as GameFeature Server and Client side Realizing on specific inventory GUI

BP_BA_UI_Graphical_Component — Actor Component — Creates and references W_BA_HUD. Listen to Gameplay Messages and calls layer updates on W_BA_HUD. Owning client will only receive structs, NOT complex objects nor actors

Gameplay Message Subsystem

Client Components

References

W_BA_HUD — Common Activatable Widget — Creates Layers (Base Game, Menu, Modal, ...) Exposes updates to layers

Has Child

W_BA_Layer_2_GameElements — Common Activatable Widget — Callable game widgets. Shows full screen widgets such as inventory, crafting, detailed world map, character info, ...

Has Child

W_BA_Inventory_Left — Common Activatable Widget — Will host bag selection

Has Child

W_BA_InvLeft_Button_NavInventories — Common Button Base — Implements logic for click/press events. Communicates back selection, e.g. Inventory Guid

Calls Interface to communicate UI event

Calls Interface to communicate UI event

Widgets

Clear parent-child hierachy

All communication up go through root

All communication down are via client side gameplay messages

Button

Derived from Common Button Base

Always calls an interface implemented on its parent widget

Links

www	https://developerbastian.github.io/www
YouTube	https://www.youtube.com/@bastiandev/videos
GitHub	https://github.com/DeveloperBastian
UNREAL ENGINE	Unreal Forum: http://bit.ly/3LDHnjJ
cc BY	License: Creative Commons Attribution 4.0

How to use this plugin

Get the code

- Use a Github client (e.g. TortoiseGit) to Clone the repository **https://github.com/DeveloperBastian/BA_Inventory**
  - Alternatively, you can just downlod the code as ZIP file if you don't want to keep it updated
- This is a full demo project. The BA_Inventory plugin itself is a GameFeature plugin, so make sure to copy the folder /Plugins/GameFeatures/BA_InventoryAndItems to your new project if you want to reuse
- I recommend to treat this plugin as mostly read-only to not mix your test work with the latest updates, but create a parallel one and share copy the content you need

This sign shows that the tutorial is released and will link to the regarding video. All unsigned tutorials are work in progress and most probably will not work. Yet