Inventory Framework Documentation

v0.1

Introduction

The purpose of this package is to provide a versatile and extensible solution for creating inventories in Unity. Its design allows developers to quickly set up functional inventories while offering the flexibility to adapt to custom use cases. The framework supports a variety of inventory types, making it suitable for a wide range of games.

The predefined inventory types are:

- **List/Grid Inventory**: A straightforward collection of items arranged as a list or a grid.
- **Equipment Inventory**: Ideal for RPGs or games with wearable gear.
- **Advanced Grid Inventory**: Supports items of varying sizes (*Not available in Lite*).
- **Puzzle Inventory**: A grid system for rotatable shapes (*Not available in Lite*).

Built on **UI Toolkit**, it offers advanced styling and layout capabilities while avoiding the overhead of GameObjects and Prefabs. Learn more in the UI Toolkit documentation.
It also comes bundled with an abstract yet powerful **item system** that minimizes setup while supporting diverse use cases.

This framework is targeted at **programmers**. It currently lacks designer-friendly features such as custom editors or Scriptable Objects, so some coding is required to implement custom functionality.

Getting started

To get started, check out the 2 demos included in the *Lite* version - "Minimal" and "Basic".

The **Minimal** demo has the least amount of code needed to setup a working inventory. It is a great starting point to familiarize yourself with the fundamentals of the framework. It consists of one inventory, some items and the simplest behavior of moving items around.

The **Basic** demo is a *Minecraft*-inspired inventory featuring the following:

- A `main inventory` window consisting of equipment, inventory and a consumables hot-bar
- A `vendor` window where you can buy pre-defined items and sell items from your inventory.
- A `treasure chest` window, which has some items that you can pick but cannot place anything back. You can also pick all at once.
- A `crafting bench` window, which has slots for materials and an outcome slot. Placing materials in the correct order will populate the outcome slot. Picking the item from outcome slot will "craft" the item.
- A `stash` window, which is an inventory you can store items in.
- A way to drop items on the "ground", by dragging them over a designated area. This will place the item "on the ground". Items on the ground can be then picked up by opening yet another window.
- A way to destroy items, by dragging them over a designated area.

Installation

You can install the package from Unity Package Manager or from git (*only Lite version*).
There are no external dependencies.

Core Concepts

The framework is built around several key concepts that define its structure and behavior. Understanding these will help you make the most of its features.

Item

An **Item** is a data type that represents an individual object in an inventory. It is highly flexible and comprises two main components:

- **Base**: Contains fixed properties like ID, name, and icon path.
- **Data**: Holds dynamic properties such as quantity or rarity.

```
// Item.cs
public record Item(int Id, ItemBase ItemBase, ItemData ItemData)
public record ItemBase(string Id, string Name, string IconPath, bool Stackable, Size Size);
public record ItemData(int Quant = 1);
```

This architecture is heavily inspired from RPGs like Diablo and Path of Exile but the separation ensures adaptability to fit various gameplay scenarios.

Slot

A **Slot** is a container that can either hold an **Item** or remain empty. Slots can enforce **restrictions**, such as allowing only specific types of items, by providing a function (`Accepts`) that operates on `Item` type.

```
// Inventory.cs
public delegate bool FilterFn(Item item);
public record Slot(Item Item) {
```

```
        public FilterFn Accepts = (_) => true;
};
```

The base `Slot` type is extended into specialized slot types:

- `ListSlot`: Index-based storage.
- `SetSlot`: Named slots for equipment/wearable items.
- `GridSlot` (*not available in Lite*): Grid-based position storage.

Inventory

An **Inventory** manages a collection of **Slots**. Each inventory type comes with its own behavior and can enforce restrictions on accepted items. The base inventory type is `Bag` , a short and descriptive name. A `Bag` can notify its subscribers of state changes through an **Observable** mechanism, making it easy to update views or other systems.

```
// Inventory.cs
public abstract record Bag(string Id) : INotifyChange {
        public FilterFn Accepts = (_) => true;
        public event Action<object> OnChange = (_) => { };
        public void Notify() => OnChange(this);
}
```

The base `Bag` type is extended into specialized types:

- `ListBag`: Contains a list of `ListSlots`
- `SetBag`: Contains a list of `SetSlots`
- `GridBag` (*not available in Lite*): Contains a 2D array of `GridSlots` as well as a list of `GridItems`

Event Bus and Events

The framework uses an [Event Bus] to decouple its components. **Events** are data containers describing actions, such as picking up an item. For example, a `PickItem` event includes the item itself, the source inventory and slot.

```
// Events.cs
public record PickItemEvent(Bag Bag, Item Item, Slot Slot);
```

An event does not perform any actions itself, that falls under the responsibility of the **Store**.

Store

Inspired by web technologies like [Redux], the **Store** serves as the central hub for managing system state. It subscribes to events on event channels and performs necessary state updates. All changes to the system state flow through the **Store**, ensuring consistency and predictability.

```
// BasicStore.cs
public Store() {
        Bus.Subscribe<PickItemEvent>(e => OnPickItem(e as PickItemEvent));
        // ...
}

void OnPickItem(PickItemEvent e) {
        LogEvent(e);
        // ...
}
```

Item Database

This is your repository of item bases and all the types that fully describe and categorize them.
For example an *Apple* is defined as a **stackable**, **consumable basic item**, while a *Dagger* is a **1-handed weapon**, with *Attack Damage* and *Attack Speed*

```
// BasicDB.cs
public record BasicItemBase(BaseId BaseId, string Name, string IconPath, bool Stackable,
ItemClass Class);

public record WeaponItemBase(BaseId BaseId, string Name, string IconPath, ItemClass Class, int
Attack, float AttackSpeed)
```

```
public static List<BasicItemBase> AllBases = new() {
        new WeaponItemBase(Dagger, "Dagger", "Shared/Icons/sword-blue", ItemClass.Weapon1H,
80, 1.5f),

        new BasicItemBase(Apple, "Apple", "Shared/Icons/apple", true,  ItemClass.Consumable),
        // ...
}
```

Technical Details

The framework takes a **functional programming approach**, emphasizing simplicity and separation of concerns. It uses the a common "good" practice of *Composition* over *Inheritance* and not so common convention of *Rules* over *Conditions* (pattern matching).

```
// InventoryExtensions.cs
public static (bool, Item) PlaceItem(this Bag bag, Item item, Slot slot){
        return (bag, slot) switch {
                (GridBag b, GridSlot s) ⇒ PlaceItem(b, item, s),
                (SetBag b, SetSlot s) ⇒ PlaceItem(b, item, s),
                (ListBag b, ListSlot s) ⇒ PlaceItem(b, item, s),
                _ ⇒ (false, Item.NoItem)
        };
}
```

C# Records are used to model data types. They offer concise syntax, value-like behavior and are immutable by default. The immutability aspect is particularly useful for **View** Components, as it simplifies determining whether a re-render is necessary.

```
// Item.cs
public record Item(int Id, ItemBase ItemBase, ItemData ItemData)
```

A type's behavior is implemented via **extension methods** to keep the data structures clean and focused. This prevents behavior inheritance, which is a big pain-point in classic OOP design.

```
// InventoryExtensions.cs
public static Slot Clear(this Slot slot) ⇒ slot with { Item = Item.NoItem };
```

To ensure proper input validation when creating new instances, each data type comes with one or more **factory** functions. For example, an **Item** can be created by providing both an `ItemBase` and an `ItemData` or just an `ItemBase`. In either case, the item `Id` will be automatically generated.

```
// Item.cs
public static class ItemFactory {
        public static Item Create(ItemBase itemBase, ItemData itemData) ⇒ new(Id(), itemBase,
itemData);

        public static Item Create(ItemBase itemBase) ⇒ new(Id(), itemBase, NoItemData);
        // ...
}
```

To represent the absence of an object the **null object pattern** is employed in favor of the classic *null* reference. This means that most types have an associated void type: `Item ⇒ NoItem` which acts like the identity element from algebra. One of the main advantages of this approach is that an operation can be performed on a an item or a collection of items without the need of a null check beforehand. An operation like this will have no effect on the item, if it is a `NoItem`.

UI Toolkit Integration

The framework leverages **UI Toolkit** for rendering inventory views. It provides a declarative way to define elements, their data, and their styles using C# and USS, while completely avoiding **UXML**. The reason for bypassing UXML is that data binding, referencing elements and working with the UXML asset, all come with levels of verbosity and complexity that drastically slow down productivity. This means, however, that the included visual components (inventories, windows, slots) cannot be viewed in Unity's **UI Builder**. You can still use the UI Builder to design own custom components if you prefer so.
Since UI Toolkit is heavily based on web technologies, the framework proposes web-inspired nomenclature, such as:

- **DOM**: Utility functions for managing elements.
- **DIV**: A shorthand alias for `VisualElement`.

How it works

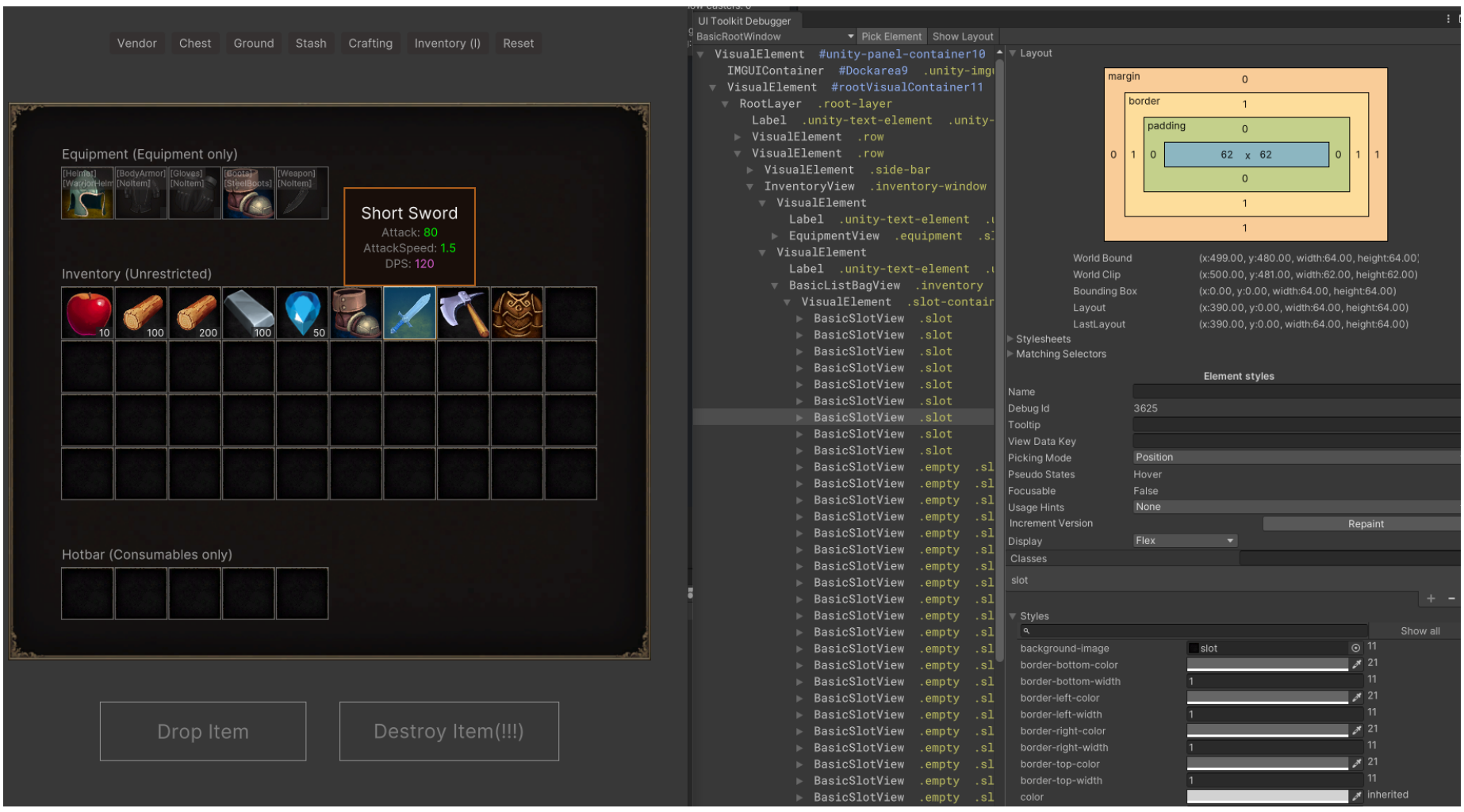
`coming soon`

How to Use the Framework

1. **Set up an inventory**: Start by modifying or cloning one of the provided demos. Cloning ensures you don’t lose your changes when updating the package.
2. **Add new items**:
   - Define a new item base in the `AllBases` dictionary.
   - Use the `ItemFactory.Create` function to create the item.
   - Add the item to an inventory using `AddItem` or `SetItems` in the *Store*'s `Reset` method.
3. **Reference examples**: Use the provided samples for guidance on extending functionality.

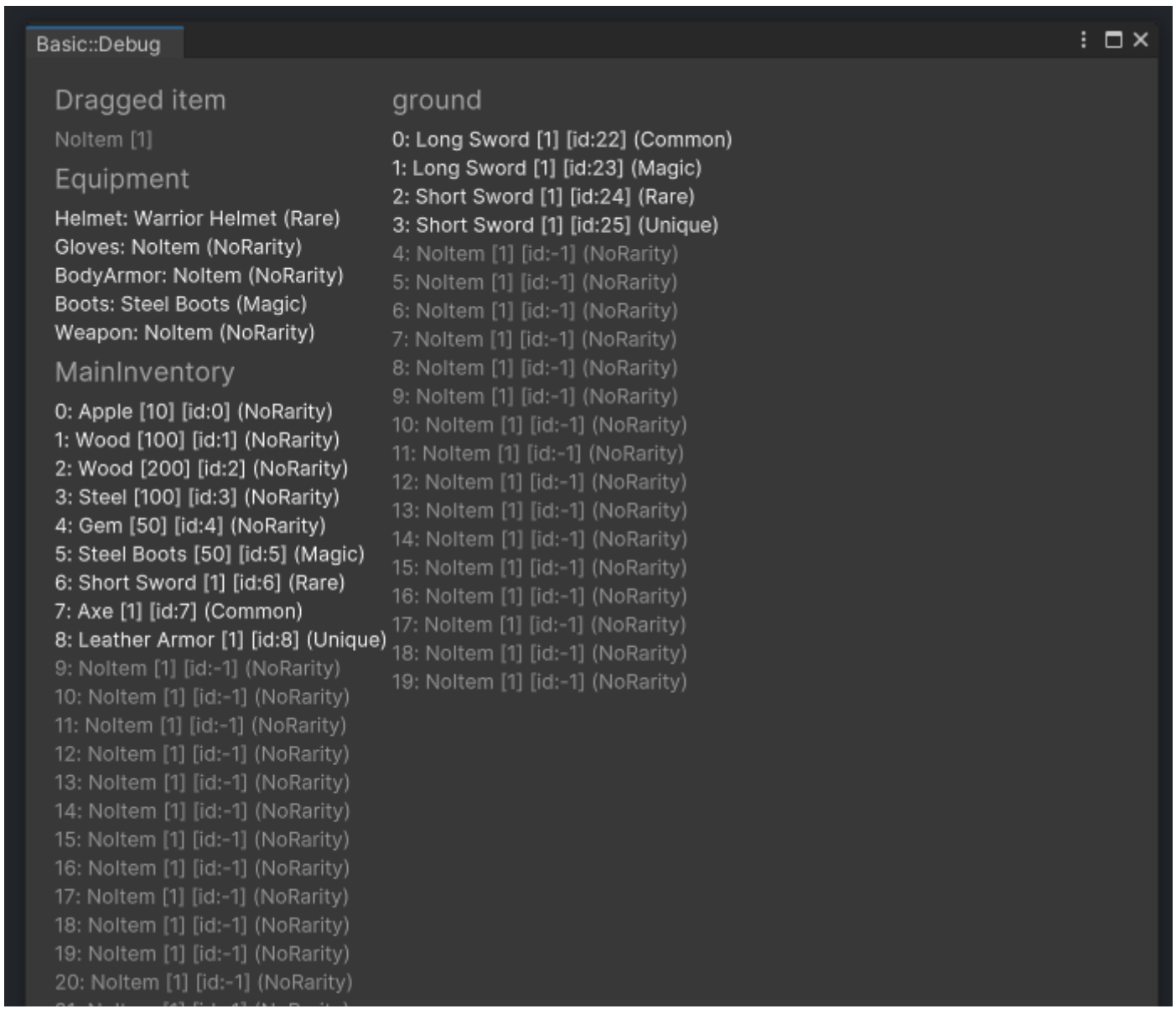
Troubleshooting

Use the UI Toolkit Debugger (`Window > UI Toolkit > Debugger`) to inspect the visual elements tree and its styles and classes.



Since UI Toolkit uses a form of CSS, it also inherits all the downsides of the cascading aspect - always check for unintended style propagation affecting your UI.

Use the included *Debug* editor windows to view the raw state of the system. These tools help identify whether issues originate from the data layer or the view layer.



Need Help?

Join our [Discord server] to ask questions and get support from the community.

Roadmap

- **Persistence**: Save/load functionality using JSON and/or CSV.
- **Designer Tools**: Investigating support for Scriptable Objects and custom editors.
- An **advanced version** (paid) is coming "soon". It will include
  - **Grid inventory** with varying item sizes.
  - **Puzzle inventory** with varying item shapes that can also be rotated.


Inventory
[Weapon]
[NoItem]
[Helmet]
[WarriorHelmet]
[BodyArmor]
[NoItem]
[Gloves]
[NoItem]
[Boots]
[NoItem]
SteelBoots(33)
ShortSword(
LeatherArmor(35)
Apple(36)
Apple(37)
10
20
[2,0]
[3,0]
[4,0]
[7,0]
[8,0]
[9,0]
[2,1]
[5,1]
[6,1]
[7,1]
[8,1]
[9,1]
[0,2]
[1,2]
[2,2]
[3,2]
[5,2]
[6,2]
[7,2]
[8,2]
[9,2]
[0,3]
[1,3]
[2,3]
[3,3]
[4,3]
[5,3]
[6,3]
[7,3]
[8,3]
[9,3]