

PropertyInterface offline wiki

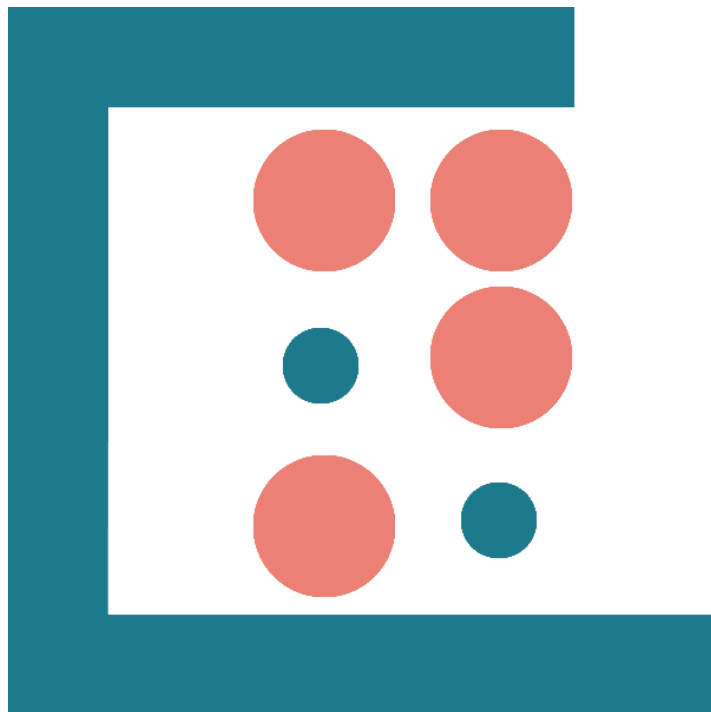


Table des matières

Introduction.....	1
Required	1
Lets see what this asset can do for you.....	1
First Look	2
What does this Asset do ?	2
How does it work ?	2
What about performance ?	3
Tag Overview	4
What is it for ?	4
Opening the Tags window (<i>Ctrl+T by default</i>):.....	4
Setting Tags :	4
Export Tags.....	5
Import Tags.....	6
Property Overview	7
What is it for ?	7
Property's elements	7
Property's values	7
How does it works?	8
Add a Property.....	8
Setting a Property.....	9
Manually.....	9
Linked to another component.....	9
Property Updater	11
What is it for ?	11
How does it work ?	11
Where should it be created ?	11
How to change the PropertyUpdater location ?	11
Property inheritance	12
Example class.....	12
Find a Property	14
From a GameObject (<i>ex : myGameObject</i>)	14
In the whole scene	14
EditorGUI_PropertyInterface	15
What is it for ?	15

Wich fields it draws ?	15
EditorGUILayout_PropertyInterface	17
What is it for ?	17
Wich fields it draws ?	17
Tagged component overview	19
What is a good tagged component ?	19
Why inherit MonoBehaviourTagged ?	19
Built in tagged component	20
PropertyTriggerHandler	20
PropertyToSlider.....	20
Create my tagged component.....	21
Inherite MonoBehaviourTagged	21
Default Editor	22
Find a MonoBehaviourTagged	22
From a GameObject (<i>ex : myGameObject</i>)	22
In the whole scene	22
Null propagating operator.....	23

Introduction

Welcome to the PropertyInterface wiki wich will help you to use and extend this powerfull asset.

Required

To use this asset you need :

- At least Unity 2019.1.0f2
- Unity Player settings/Scripting Runtime Version => .NET 4.X

Lets see what this asset can do for you.



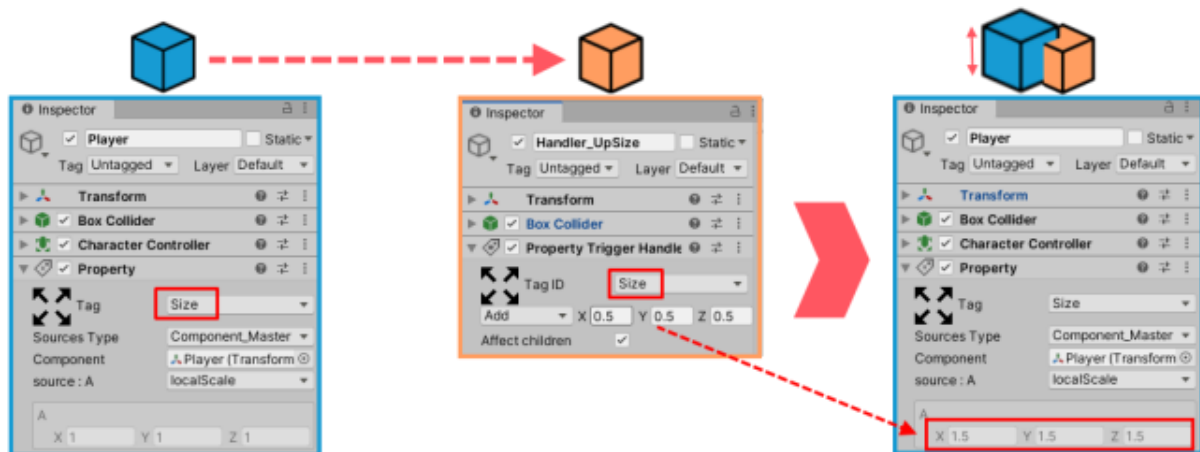
First Look

What does this Asset do ?

This asset allow you to easily add [Property](#) to GameObject.

- Each [Property](#) is a component so it can be linked to your other Components.
- Each [Property](#) can be found by script via a powerfull and easy to use custom [Tag](#) system.
- Each [Property](#) values can be filled with raw values or sourced from other Component.

VERY EASY TO USE



A complete and easy to use library allow you to handle [Property](#) by script as well.

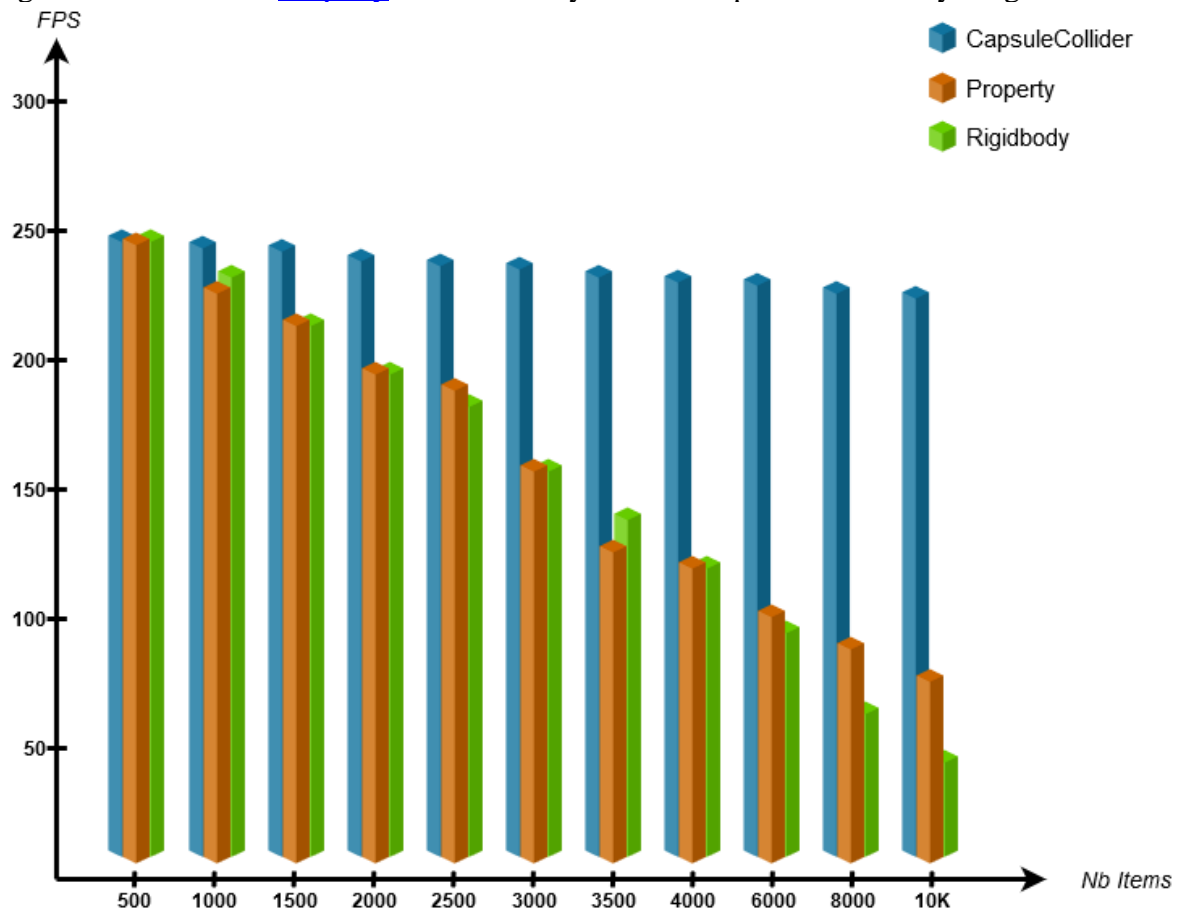
How does it work ?



1. Add [Property](#) to GameObject.
2. Select a [Tag](#) for the [Property](#).
3. [Make your own tagged component](#).
4. Select a [Tag](#) in your custom component. It will now be able to interact with all [Property](#) component in the scene wich have the same [Tag](#).

What about performance ?

As you can see below, [Property](#) is not the lightest component. However it will take a significant amount of [Property](#) before it really affects the performance of your game.



All tests have been done with the following configuration :

- OS : Windows 10 Family 64Bits
- UNITY : 2019.3.0f6
- CPU : Intel Core I7-4800MQ (4 x 2.7GHz)
- GPU : Nvidia GeForce GTX870m 3GB GDDR5
- RAM : 8GB DDR3 800Mhz
- HD : Sata 7200rpm

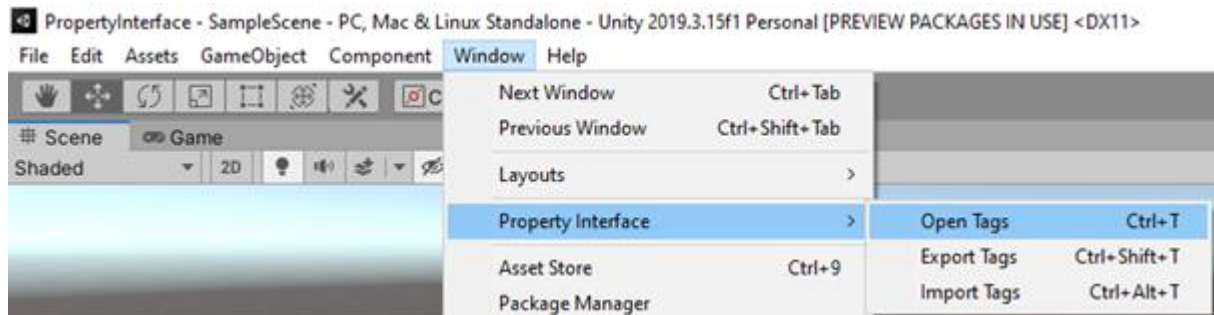
Tag Overview

What is it for ?

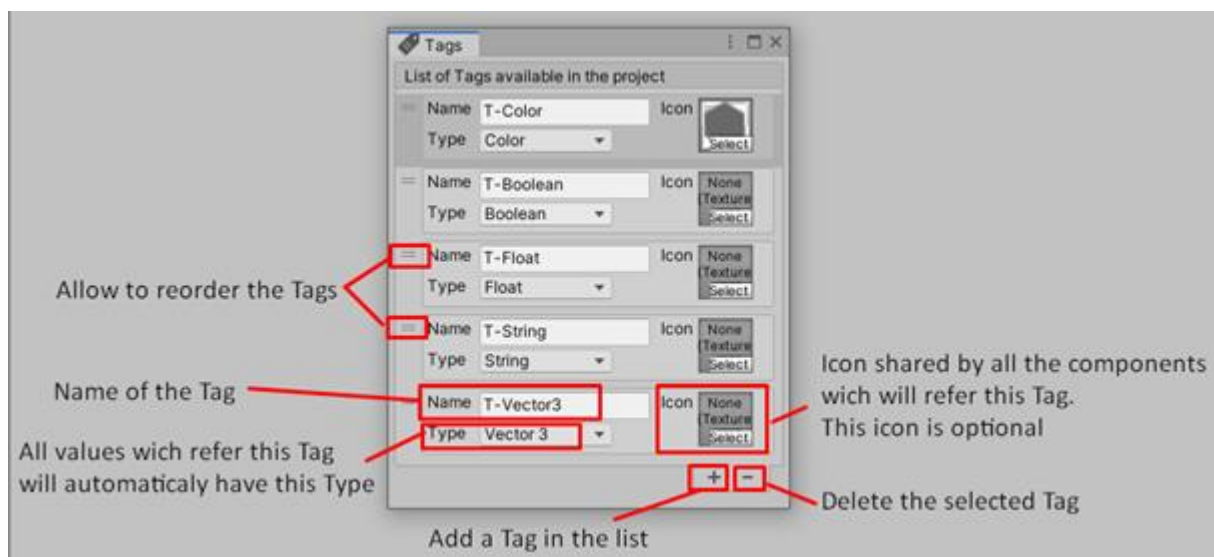
Tags are made to link your [tagged components](#) and the [Property](#) components that you will add to your GameObjects.

⚠ Tag are not built. You shouldn't access them in game play.

Opening the Tags window (*Ctrl+T by default*):



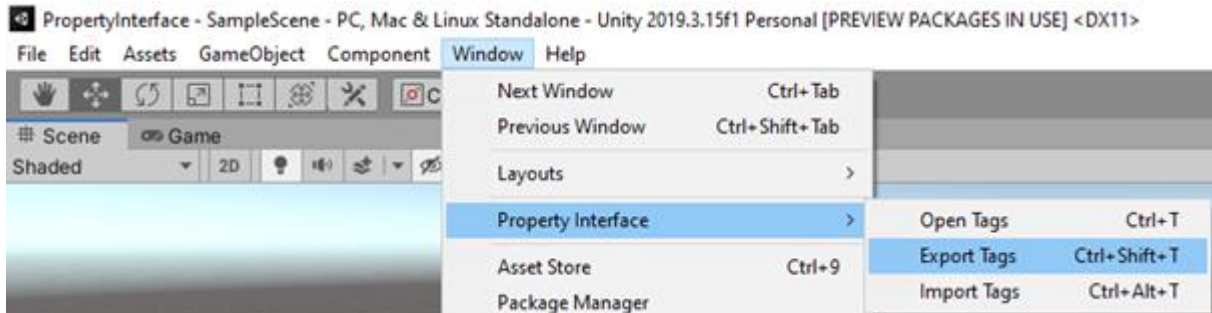
Setting Tags :



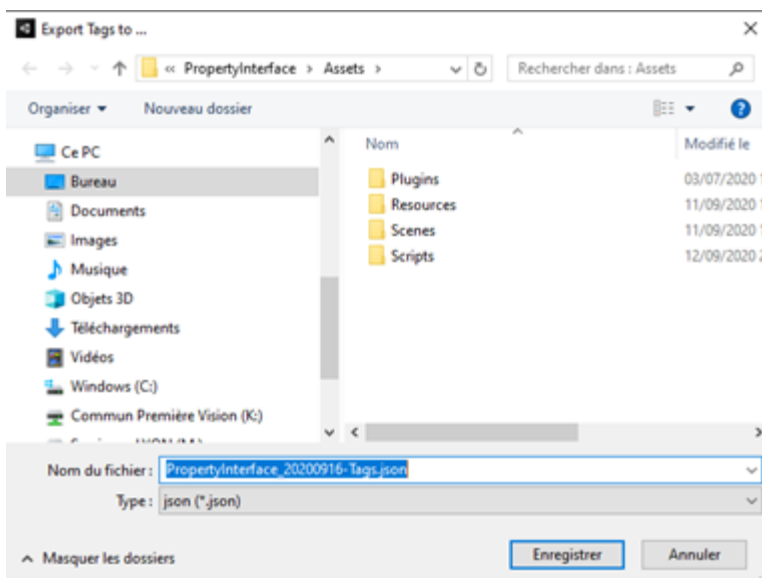
Export Tags

It's possible to export a Tag list to backup or share it.

In Unity editor, go to menu "Windows/PropertyInterface/Export Tags" (*Ctrl+Shift+T by default*):



Choose in your explorer where to export your Tags :



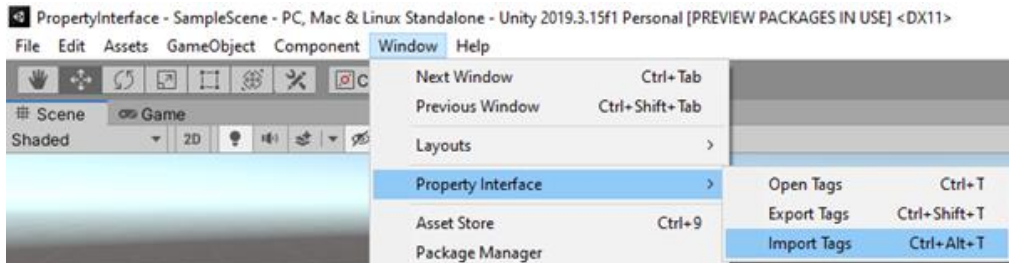
You can now share it to another project or use it as backup.

Import Tags

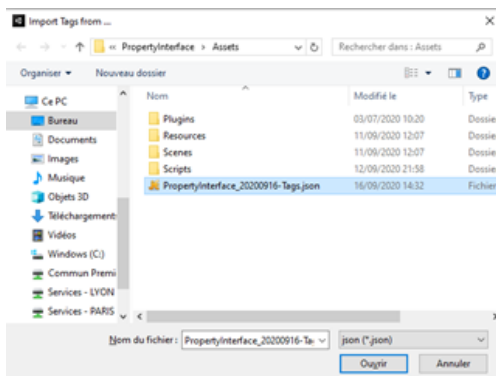
It's possible to import a Tag list from a backup or another project.

⚠ This action will replace all your Tags by those imported

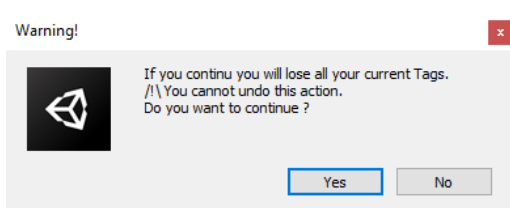
In Unity editor, go to menu "Windows/PropertyInterface/Import Tags" (*Ctrl+Alt+T by default*):



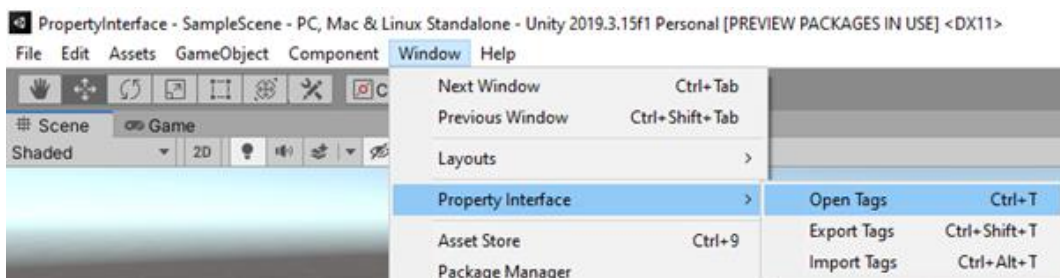
Choose a previous exported Tags file in your explorer (See [Export Tags](#)) :



⚠ This action can't be undone. If you are sure, click on the **Yes** button in the warning popup :



Open the Tags window to see your changes (*Ctrl+T by default*):



Property Overview

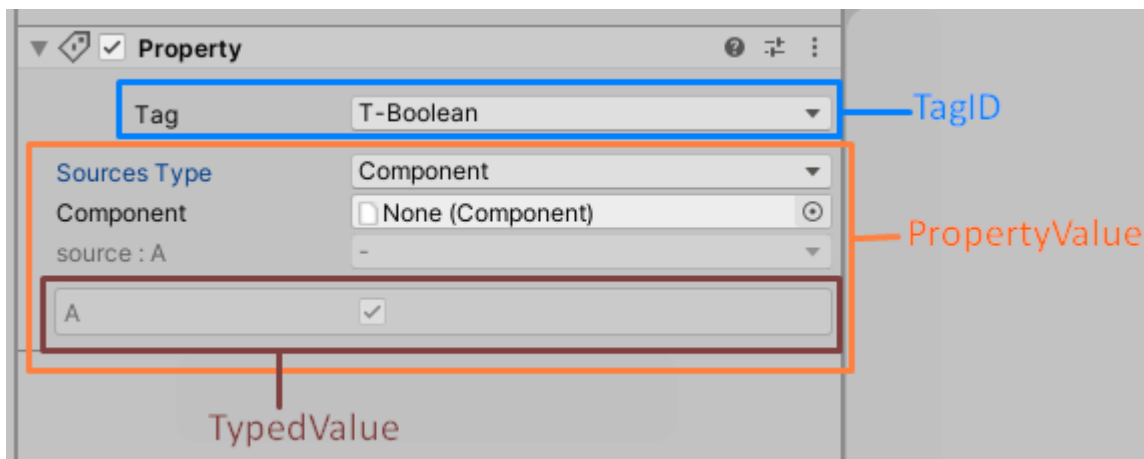
What is it for ?

A Property is a component which allow to add properties in GameObjects.

Property's elements

Property components are composed of 2 elements :

- A [TagID](#) (used to refer a Tag)
- A [PropertyValue](#) (where all the Property settings are stored)



The [TypedValue](#) is stored in the PropertyValue and will be define by the Type set in the Tag. In the above example, the Type is set to **Boolean** in the Tag named "T-Boolean". So the TypedValue is a checkbox.

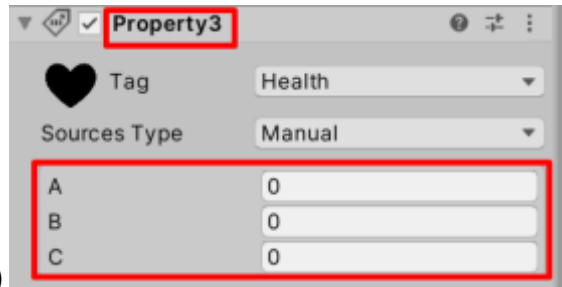


Property's values

As Vector fields, Property components can have several values. By default they can have 3 values maximum :

- Property => one value (A)
- Property2 => two values (A, B)

- Property3 => three values (A, B, C)



⚠ Property, Property2 and Property3 are different components.

Property2 and Property3 exist because it can be useful sometime to have distinct values for the same Property. For example, if you take the Health characteristic for your player. With Property2, you can set the current Health (A) and the max Health (B). With Property3, you can set the min Health (A), the current Health (B) and the max Health (C).

If you need more values for your Property, you can make your own Property component. See [Property inheritance](#).

How does it work?

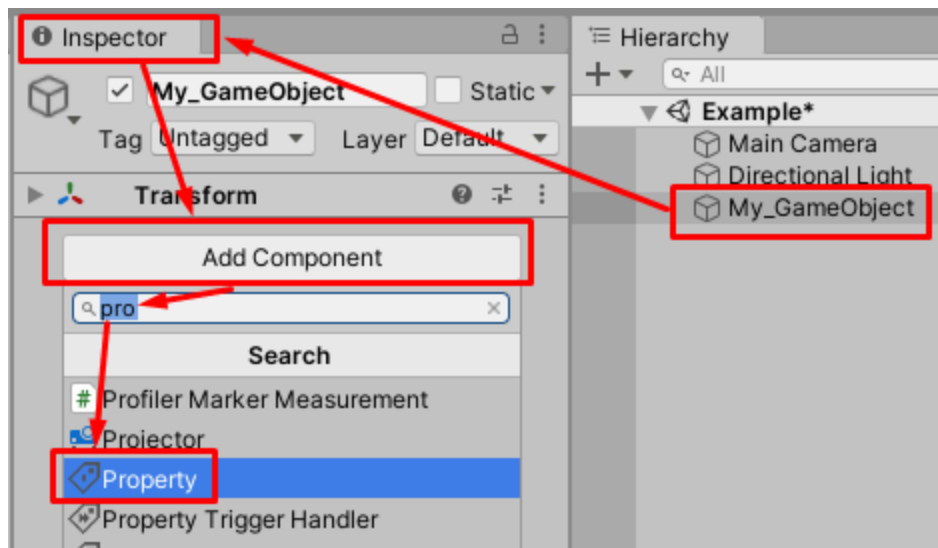
In editor mode, you will [setting the Property's value](#).

At runtime, the value will be updated by your custom [tagged components](#).

Add a Property

A Property is a component. So you can add it as any other component.

Select a GameObject in the scene view and click on "Add component" button in the inspector view :



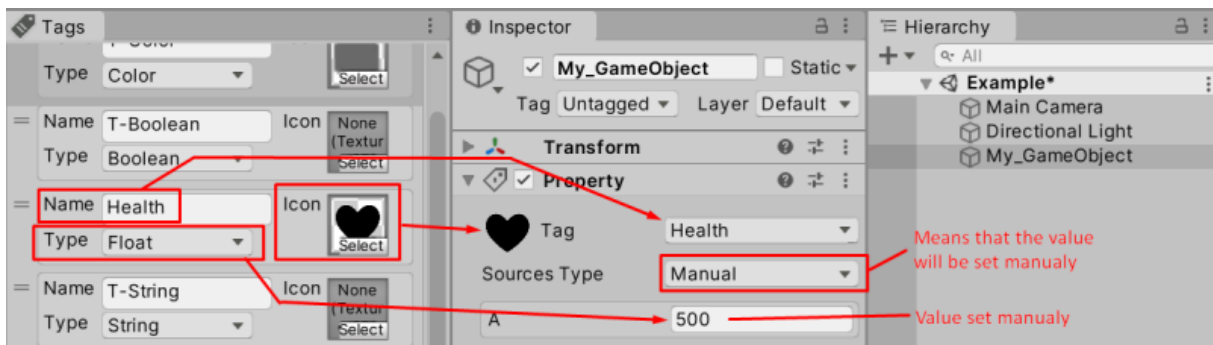
Setting a Property

⚠ To be able to set a Property you have to create at least one [Tag](#) before.

There are two ways to set Property component :

Manually

The initial Property's value is set manually and won't be synchronized with any other object.

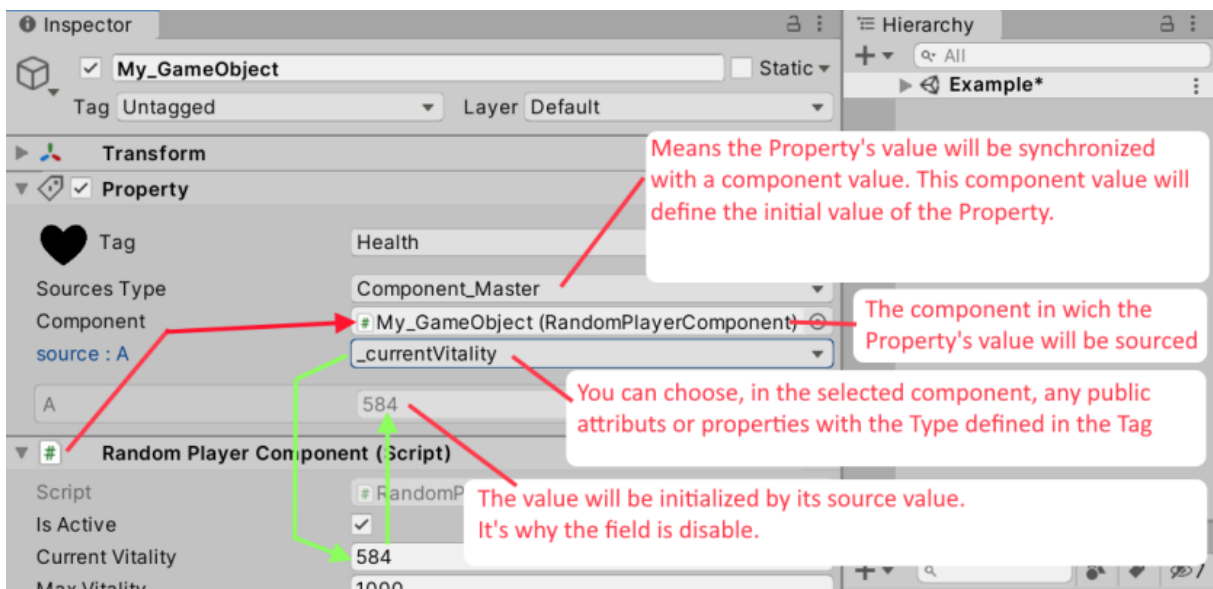


Linked to another component

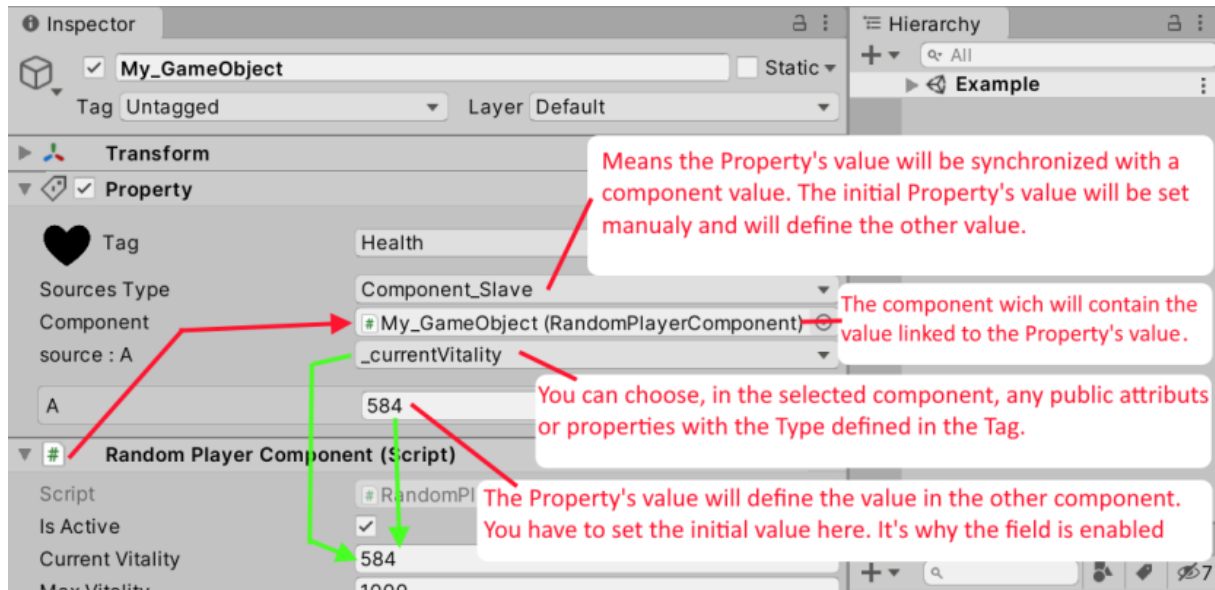
The Property's value is link to an existing value in another component. Both values will be synchronized at runtime.

In this case we have two possibilities to initialize the Property's value.

1. The Property's value is sourced from the other value => **Component_Master**



2. The Property's value is set manually and will define the other value => **Component_Slave**



In both case, the Property's value will be synchronized at runtime with the other value as follow :

- The Property's value has changed => Update the other value with the Property's value.
- The other value has changed => Update the Property's value with the other value.

Property Updater

⚠️ You should not take care of this component. You should just be aware that it exists.

What is it for ?

For performance reasons, all Property components in the scene will be update one after the other in a single place. *(The update order is initiate randomly)*
This place is the PropertyUpdater component.

How does it work ?

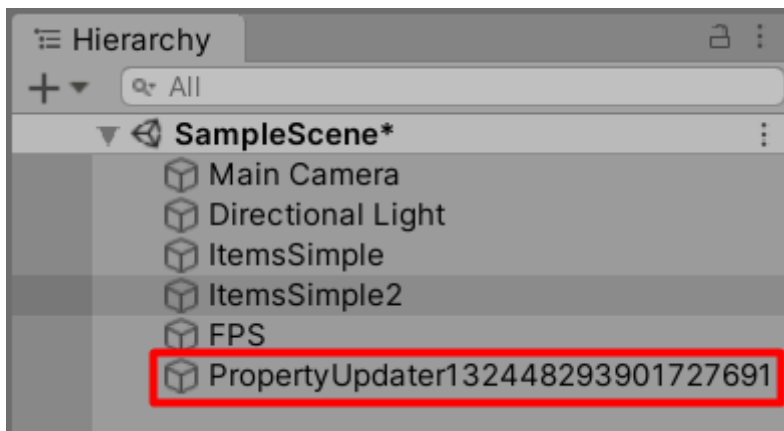
When a Property component is enabled in the scene, it is added in the PropertyUpdater.
When a Property component is disable in the scene, it is removed in the PropertyUpdater.

Each frame, the PropertyUpdater will parse all Property components it knows. For each of them, it will execute their [DelayedUpdated\(\)](#) method.

Where should it be created ?

You should not add this component by your own in the scene.

This component is created dynamically in the scene at runtime if none exists. It will be deleted in the same way at the game stop.



By default it is created in the scene's root folder.

How to change the PropertyUpdater location ?

This reason should be the only one wich make you add this component in the scene by your own.

You can add this component in any GameObject in the scene. If you do that, none will be created dynamically.

⚠️ Only one PropertyUpdater is need by scene.

Property inheritance

In some case you will need to make your own Property component. To understand how to do that, you can check the source code of [Property2](#) or [Property3](#) components. You can find them in your Unity Project tab : "Packages/Nectunia/Property Interface/Package/Scripts/Components/".

To create a component wich will be able to interact with Property, see [Create your Tagged component](#) instead.

Example class

Bellow you will find an example that do nothing more than Property component but implements all you need to make your own Property child:

```
using UnityEngine;
using System;
using Nectunia.PropertyInterface;

/// <summary>
/// My new Property component
/// </summary>
[Serializable]
public class MyProperty : Property {
    // Add your attributs here. Don't forget to add [SerializeField] for thoose wich need to be
    // drawn in Editor
    // ...

    /// <summary>
    /// Set the Tag.ValueType for all PropertyValue.
    /// </summary>
    public override void SetValuesType () {
        base.SetValuesType();
    }

    /// <summary>
    /// Set a new Property.SourceType for all PropertyValue.
    /// </summary>
    /// The new Property.SourceType.
    protected override void SetValuesSourceType (SourceType newType) {
        base.SetValuesSourceType(newType);
    }

    /// <summary>
    /// Set a new Component for all PropertyValue.
    /// </summary>
    /// The new Component.
    protected override void SetValuesComponent (Component newComponent) {
        base.SetValuesComponent(newComponent);
    }
}
```

```

/// <summary>
/// Refresh all object values with the serializedValues.
/// </summary>
public override void RefreshValuesObject () {
    base.RefreshValuesObject();
}

/// <summary>
/// Synchronize the values with their sources if need.
/// </summary>
public override void SynchronizeValues () {
    base.SynchronizeValues();
}

/// <summary>
/// Check if the sources of the values are set or not.
/// </summary>
///
/// True if at least one value.Source is set. False otherwise.
///
public override bool AtLeastOneSourceIsSet () {
    return base.AtLeastOneSourceIsSet();
}
}

```


Find a Property

From a GameObject (*ex : myGameObject*)

myGameObject.[GetProperties\(\)](#) : Get all Property components in the GameObject

myGameObject.[GetPropertiesInChildren\(\)](#) : Get all Property components in the GameObject and its childrens

myGameObject.[GetProperty\(\)](#) : Get the Property component in the GameObject

myGameObject.[GetPropertyInChildren\(\)](#) : Get the Property component in the GameObject and its childrens

In the whole scene

[Property.GetProperties\(\)](#) : Get all Property components in the scene or in a specific GameObject

[Property.GetPropertiesInChildren\(\)](#) : Get all Property components in the scene or in a specific GameObject and its childrens

[Property.GetProperty\(\)](#) : Get the Property component in the scene or in a specific GameObject

[Property.GetPropertyInChildren\(\)](#) : Get the Property component in the scene or in a specific GameObject and its childrens

[Property.Exists\(\)](#) : Does a Property component exist in the scene or in a specific GameObject ?

[Property.ExistsInChildren\(\)](#) : Does a Property component exist in the scene or in a specific GameObject and its childrens ?

EditorGUI_PropertyInterface

What is it for ?

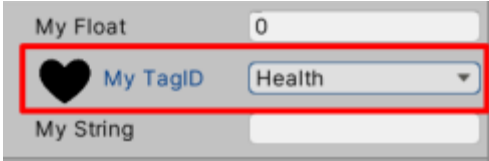

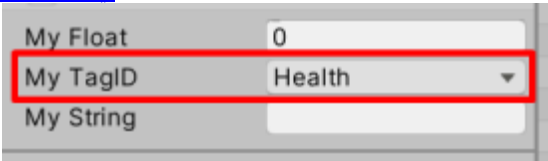

It's a GUI class to help you to draw in Unity editor the new fields implemented by PropertyInterface asset with a manual positioning.

For more information about this classe implementation, see [EditorGUI_PropertyInterface scripting desrcption](#)

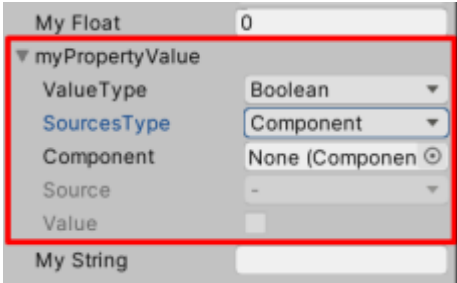
See also [EditorGUILayout_PropertyInterface](#) for auto laid version.

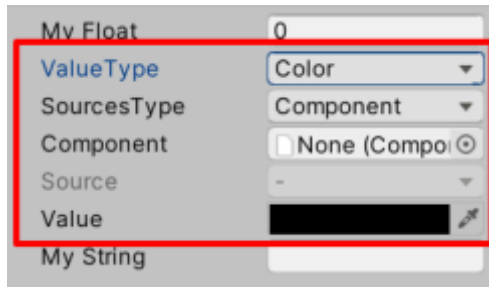
Wich fields it draws ?

TagID fields :

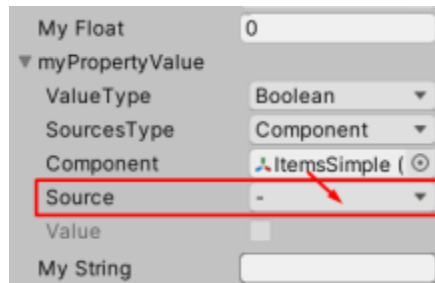
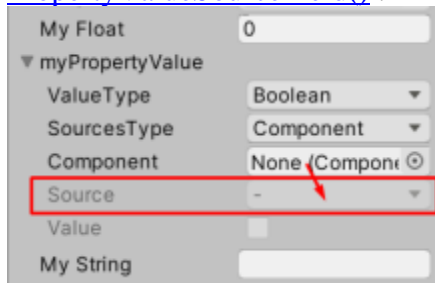
- Default drawer : 
- [TagIdField WithIcon\(\)](#) : 
- [TagIdField\(\)](#) : 
- [TagIconField\(\)](#) : 

PropertyValue fields :

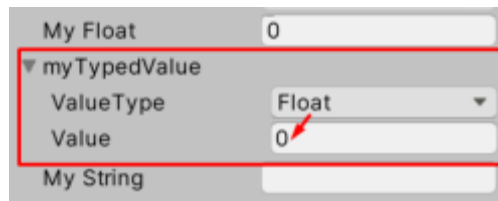
- Default drawer : 



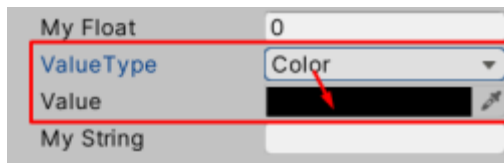
- [Property ValueFields\(\)](#) :
- [Property ValueSourceField\(\)](#) :



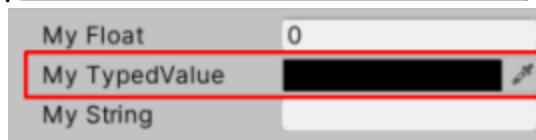
TypedValue fields :



- Default drawer :



- [TypedValueFields\(\)](#) :

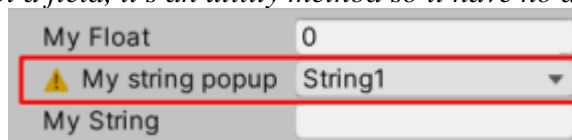


- [TypedValueField\(\)](#) :

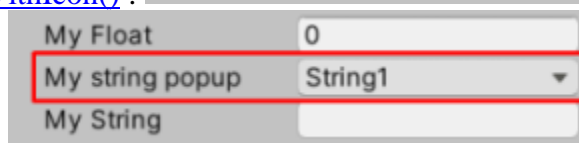
StringPopup :

As opposed to a [EditorGUI.Popup](#), a *StringPopup* works with its values instead of its index.

- Default drawer : *This is not a field, it's an utility method so it have no drawer.*



- [StringPopup WithIcon\(\)](#) :



- [StringPopup\(\)](#) :

EditorGUILayout_PropertyInterface

What is it for ?

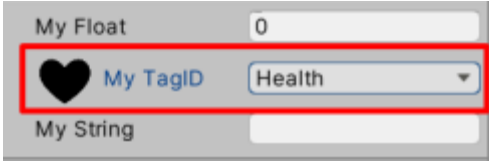

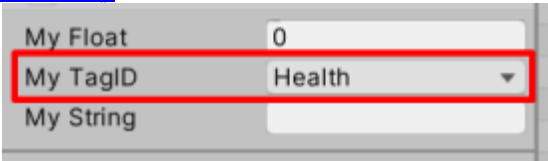

It's a GUI class to help you to draw in Unity editor the new fields implemented by PropertyInterface asset with an auto laid positioning.

For more information about this classe implementation, see [EditorGUILayout_PropertyInterface scripting description](#)

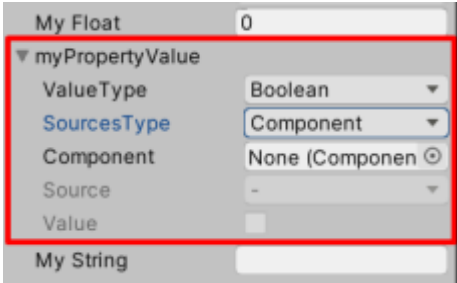
See also [EditorGUI_PropertyInterface](#) for manual positioning version.

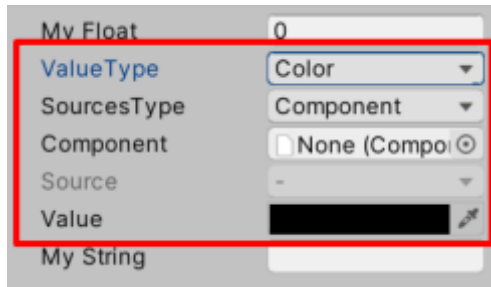
Wich fields it draws ?

TagID fields :

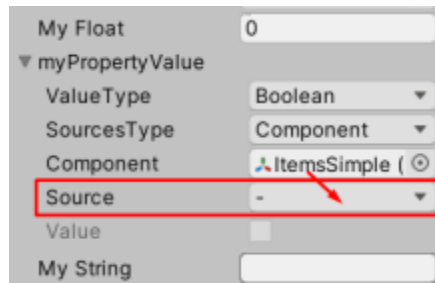
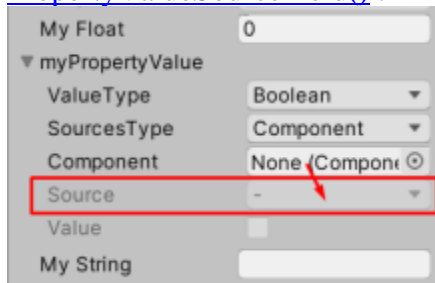
- Default drawer : 
- [TagIdField WithIcon\(\)](#) : 
- [TagIdField\(\)](#) : 
- [TagIconField\(\)](#) : 

PropertyValue fields :

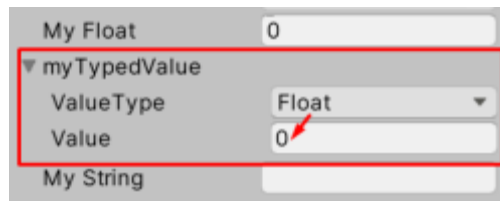
- Default drawer : 



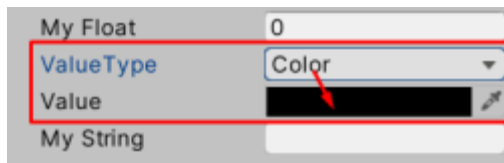
- [Property ValueFields\(\)](#) :
- [Property ValueSourceField\(\)](#) :



TypedValue fields :



- Default drawer :



- [TypedValueFields\(\)](#) :

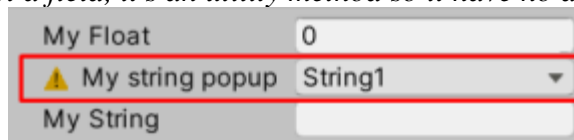


- [TypedValueField\(\)](#) :

StringPopup :

As opposed to a [EditorGUILayout.Popup](#), a *StringPopup* works with its values instead of its index.

- Default drawer : *This is not a field, it's an utility method so it have no drawer.*



- [StringPopup WithIcon\(\)](#) :



- [StringPopup\(\)](#) :

Tagged component overview

Tagged components are the main elements that will interact with [Property](#) components in your scene.

They are components with a [TagID](#) field. In the Unity editor, the tagged components will select a [Tag](#). By the [Tag](#) they have selected, they will be able to interact with all [Property](#) components in the scene which referred the same [Tag](#).

They should all inherit the class [MonoBehaviourTagged](#).

What is a good tagged component ?

- It can work with all [Tag.ValueType](#).
- It inherit the class [MonoBehaviourTagged](#).

Why inherit [MonoBehaviourTagged](#) ?

Easier maintenance

The [Tags](#) in the Tag window are link to [MonoBehaviourTagged](#) components. When a scene is opening in the editor or when a [Tag](#) is modified, all [MonoBehaviourTagged](#) components in the scene will check if they have to be updated to match their [Tag](#) settings. If yes and if they can't be updated automatically, a warning will be sent in the Unity console with a link to the given [MonoBehaviourTagged](#) component.

Easier accessibility

[MonoBehaviourTagged](#) component provide methods to find them in scene or GameObject.

See [Find MonoBehaviourTagged](#)

Built in tagged component

Few built in tagged components are available in the asset.

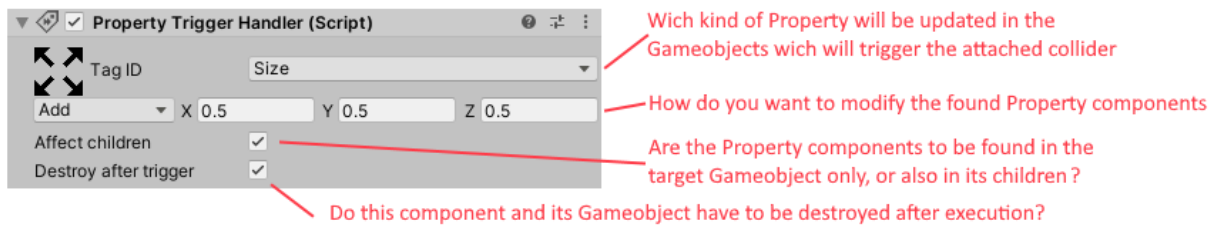
In simple projects they should be enough.

For more advance use, you can [create your own tagged components](#).

PropertyTriggerHandler

⚠ This component need a collider set as trigger.

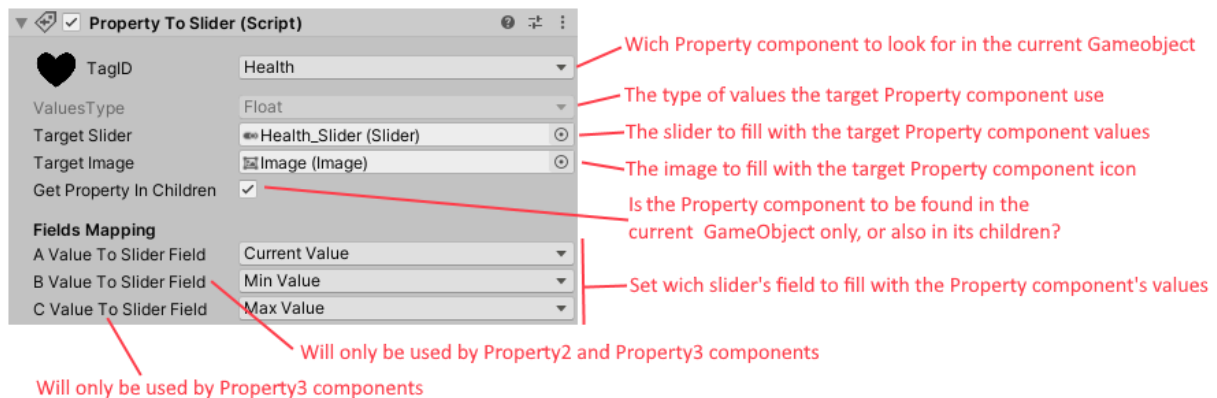
When the attached collider is triggered, this component will apply changes to targeted Property components in the target GameObject.



PropertyToSlider

⚠ This component will target Property component in the same GameObject or in its children.
This component is used to fill a UI Slider with a Property component.

A UI Image component can be linked as well to fill it with the targeted Property component icon.



Create my tagged component

It's pretty simple to create a tagged component :

1. Create you own class wich inherit from [MonoBehaviourTagged](#).
2. Make sure that your component can [find Property components](#) or [find MonoBehaviourTagged components](#) it needs to interact with.
3. Script the mechanic you want to implement.
4. (Optional) Make a custom Editor if need.

As an advanced example you can have a look at the **PropertyTriggerHandler** component available in this asset. It can be find in the Unity editor Project tab : "Packages/Property Interface/Package/Scripts/Components/"

Inherit [MonoBehaviourTagged](#)

When inherit from [MonoBehaviourTagged](#), you can catch the triggers [onValuesTypeChanged\(\)](#) and [OnTagIDChanged\(\)](#). Below an example class :

```
using Nectunia.PropertyInterface;
using System;
using UnityEngine;

/// <summary>
/// My custom tagged component
/// </summary>
public class MyCustomTaggedComponent : MonoBehaviourTagged{
    public string _tagTypeName;
    [SerializeField]
    private Property _attachedProperty;

    public void Update () {
        // Property component have only 1 value => it's name "A"
        // Here we log its value in the console
        Debug.Log("The value of the attached Property is : " +
this._attachedProperty?.A.Value.ToString());
    }

    /// <summary>
    /// Event triggered when this.TagID changed.
    /// </summary>
    protected override void OnTagIDChanged () {
        // Get in the current GameObject the first Property component wich refer the same
TagID
        this._attachedProperty = this.gameObject.GetProperty(this.TagID);
    }

    /// <summary>
    /// Event triggered when this.ValuesType changed.
    /// </summary>
    protected override void OnValuesTypeChanged () {
```



```

// Get the Type name of the refered Tag's ValueType
this._tagTypeName = this.ValuesType.GetSystemType().Name;

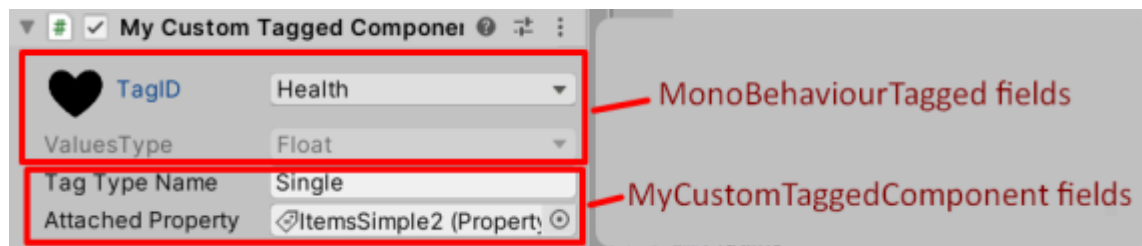
// Another solution
//this._tagTypeName = Enum.GetName(typeof(Tag.ValueType), this.ValuesType);
}
}

```

Default Editor

MonoBehaviourTagged provide a default Editor wich draw its fields and those of its children.

Here the default Editor for the above custom tagged component example :



Find a MonoBehaviourTagged

From a GameObject (ex : *myGameObject*)

myGameObject.[GetMonoBehavioursTagged\(\)](#) : Get all MonoBehaviourTagged components in the GameObject

myGameObject.[GetMonoBehavioursTaggedInChildren\(\)](#) : Get all MonoBehaviourTagged components in the GameObject and its childrens

myGameObject.[GetMonoBehaviourTagged\(\)](#) : Get the MonoBehaviourTagged component in the GameObject

myGameObject.[GetMonoBehaviourTaggedInChildren\(\)](#) : Get the MonoBehaviourTagged component in the GameObject and its childrens

In the whole scene

[MonoBehaviourTagged.GetMonoBehavioursTagged\(\)](#) : Get all MonoBehaviourTagged components in the scene or in a specific GameObject

[MonoBehaviourTagged.GetMonoBehavioursTaggedInChildren\(\)](#) : Get all MonoBehaviourTagged components in the scene or in a specific GameObject and its childrens

[MonoBehaviourTagged.GetMonoBehaviourTagged\(\)](#) : Get the MonoBehaviourTagged component in the scene or in a specific GameObject

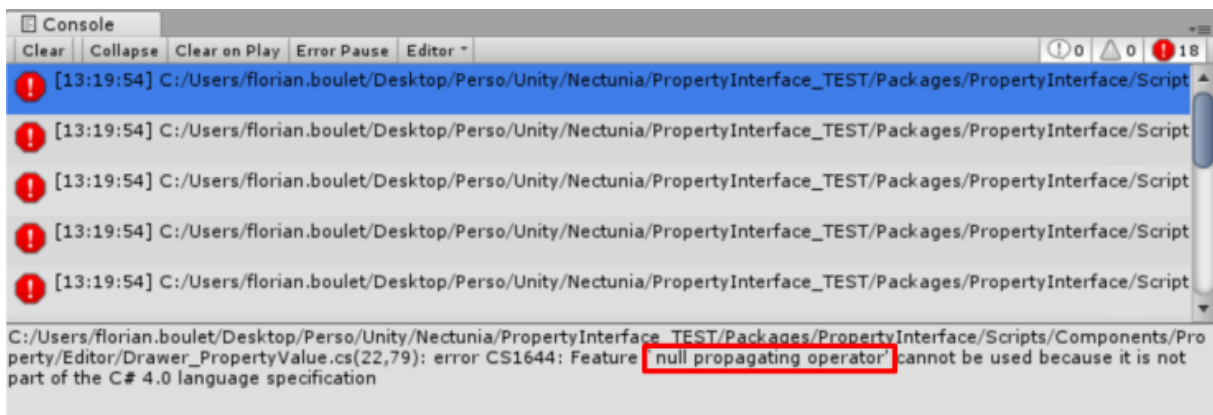
[MonoBehaviourTagged.GetMonoBehaviourTaggedInChildren\(\)](#) : Get the MonoBehaviourTagged component in the scene or in a specific GameObject and its childrens

[MonoBehaviourTagged.Exists\(\)](#) : Does a MonoBehaviourTagged component exist in the scene or in a specific GameObject ?

[MonoBehaviourTagged.ExistsInChildren\(\)](#) : Does a MonoBehaviourTagged component exist in the scene or in a specific GameObject and its childrens ?

Null propagating operator

After downloading this asset, you could have the following errors in your project :



To solve this issue, you just need to change the Player settings/Scripting Runtime Version to **.NET 4.x**

