Scriptable Object Architecture Pattern

User Guide

Version 2.7.1

OBVIOUS
game

Table of Content

What is SOAP?

First of all, thank you for purchasing Soap 😊. Soap was made to simplify game development. It aims to be easy to understand and to use, making it useful to both junior and senior game developers.

Soap leverages the power of **ScriptableObjects** which are native objects from the Unity Engine, to create a set of tools that enables you to build your game in a simple, modular and reusable manner.

ScriptableObjects are assets that can contain code. They can be referenced by other assets and are accessible at runtime and authoring time. ScriptableObjects are commonly used only for data storage, however they can do much more. Soap builds on top of them to exploit their full potential.

If you want to know more about this idea, I strongly suggest the talk of Ryan Hipple.

Soap was created mostly while developing casual games. Because of the nature of these games, Soap focuses on **speed** and **simplicity**. Nevertheless, Soap is great at forcing you to decouple your code (avoiding dependencies) and helping you to create modular and reusable systems. Therefore, it can be used in any type of games, simple or complex, small or big.

The package contains **5 example scenes** to quickly show how to use this framework. Each scene has its own specific documentation (in the same folder as the scene). On top of this, there are **6 step-by-step tutorial videos** to help you to create a Roguelike game with Soap: https://www.youtube.com/@obviousgame/

Why use SOAP?

1. Solve dependencies through the editor

By utilizing scriptable objects at its core, it prevents your code from becoming coupled (spaghetti code). This is especially true in the casual mobile market for several reasons:

- Many features are enabled or disabled through independent A/B Tests. This can lead to hardcoded or messy code within the core of our game. Having these features 'hooked' into an independent architecture makes it easy to add, remove, or toggle them on and off.
- Soap enables you to reuse features more easily. By using Soap, you can create features as 'drag and drop,' which can be a huge time saver in the long run for elements that don't change much across games (like meta features, ability systems, etc.).

2. Increase Development Speed

- Avoid creating new classes for straightforward behaviors.
- Directly access key variables using a simple reference.
- Modify global variables from the editor or within the code.
- Maintain persistent data across scenes or play sessions.
- Effortlessly debug at runtime.

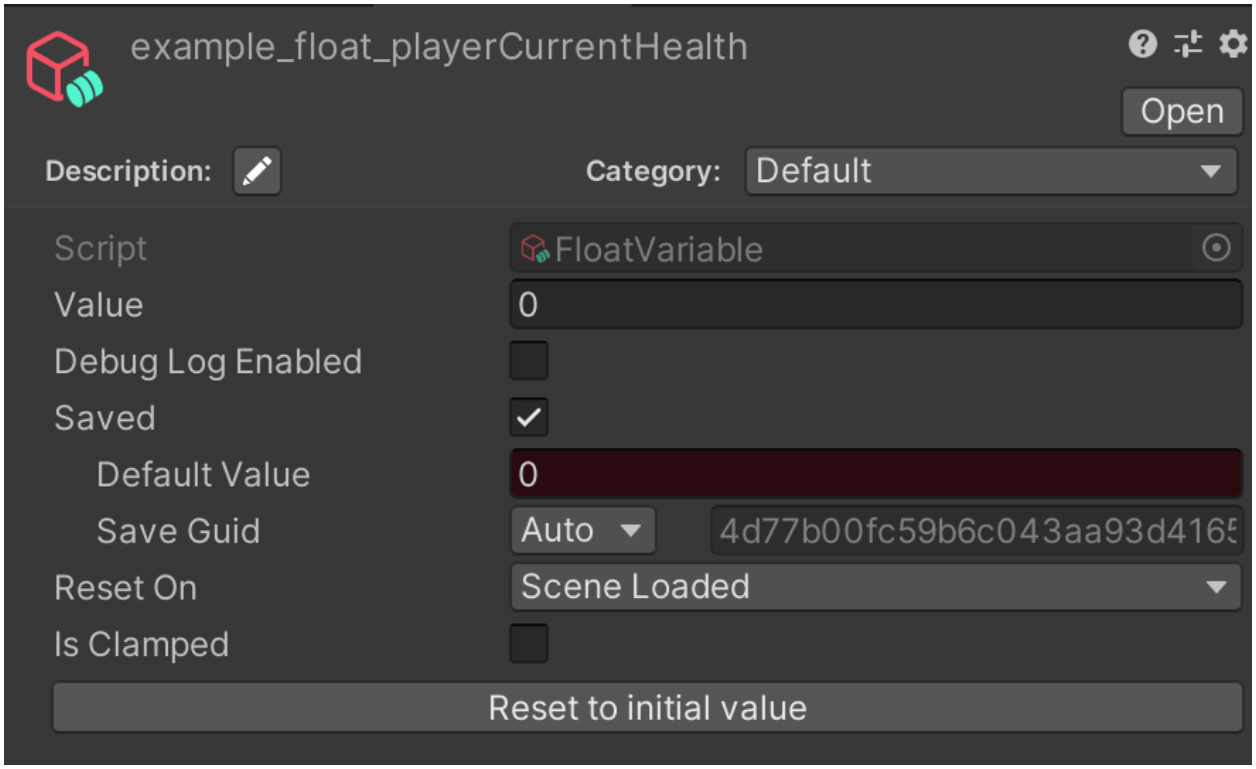
3. Code Efficiency / Clarity

- Subscribe to what you need and have each class handle itself.
- Avoid useless managers.
- Reduce code complexity (by preventing spaghetti code)
- Performant

Finally, because it's **fun**. Once you start developing with Soap, you might realize that the workflow is more enjoyable. It is not for everyone, but people who like this kind of workflow will love Soap.

Scriptable Variables

A Scriptable Variable (SV) is a scriptable object of a specific type containing a value and other parameters. Here is an example of a FloatVariable:



Let's go through its properties, starting from the top:

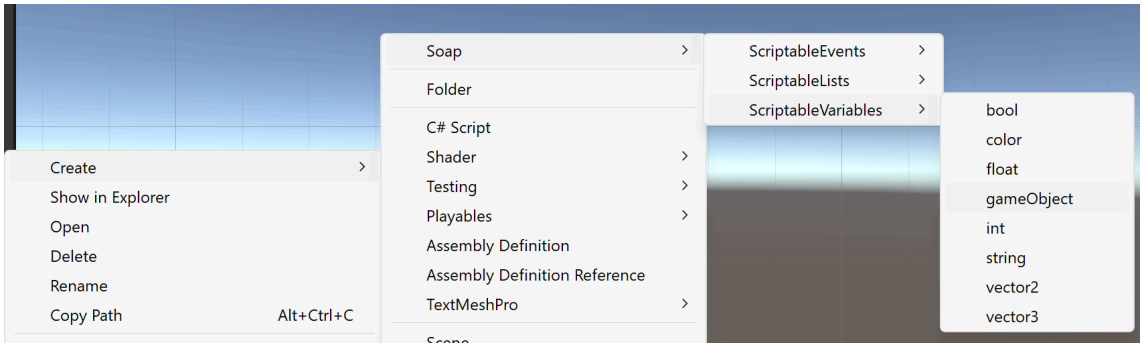
- **Description:** Add/edit an optional description to your SV
- **Category**: similar to a tag. Makes it easier to filter in the Soap Wizard.
- **Value**: the current value of the variable. Can be changed in inspector, at runtime, by code or in unity events. Changing the value will trigger an event "OnValueChanged" that can be registered to by code. See examples for practical usage.
- **PreviousValue:** the previous value of the variable. Accessible only via code. (not visible in the inspector)
- **Debug Log Enabled**: if true, will log in the console whenever this value is changed.

- **Saved**: If true, the value of the variable will be saved to Player Prefs when it changes. (See 5_Save_Example scene & documentation).
    - **Default Value**: Default value used the first time you load from PlayerPrefs.
    - **Save Guid**: by default, a unique Guid is created for the variable. In case you want to override it, change this setting to Manual and assign your own custom Guid.
- **Reset On:** when is this variable reset (or loaded, if saved)?
    - *Scene Loaded:* whenever a scene is loaded. Use this if you want the variable to be reset if it is only used in a single scene. Note: does not reset when loading additive scenes.
    - *Application Start:* reset once when the game starts. Useful if you want changes made to the variable to persist across scenes.
- **Is Clamped:** only for FloatVariable and IntVariable, gives you the ability to clamp its value. (See 1_ScriptableVariables Documentation).

In the Editor, non-saved ScriptableVariables automatically revert to their initial values (the values shown in the inspector before entering play mode) upon exiting play mode.

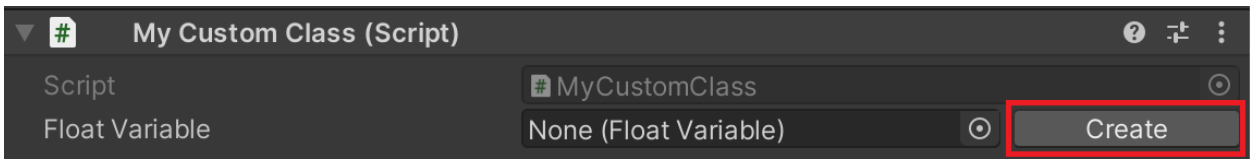
Additionally, there is a convenient utility button labeled 'Reset to Initial Value.' This allows you to quickly reset the variable to its initial value from the inspector.

To create a new variable, simply right-click in the project window and locate the ScriptableVariable you need: *Create/Soap/ScriptableVariables/*

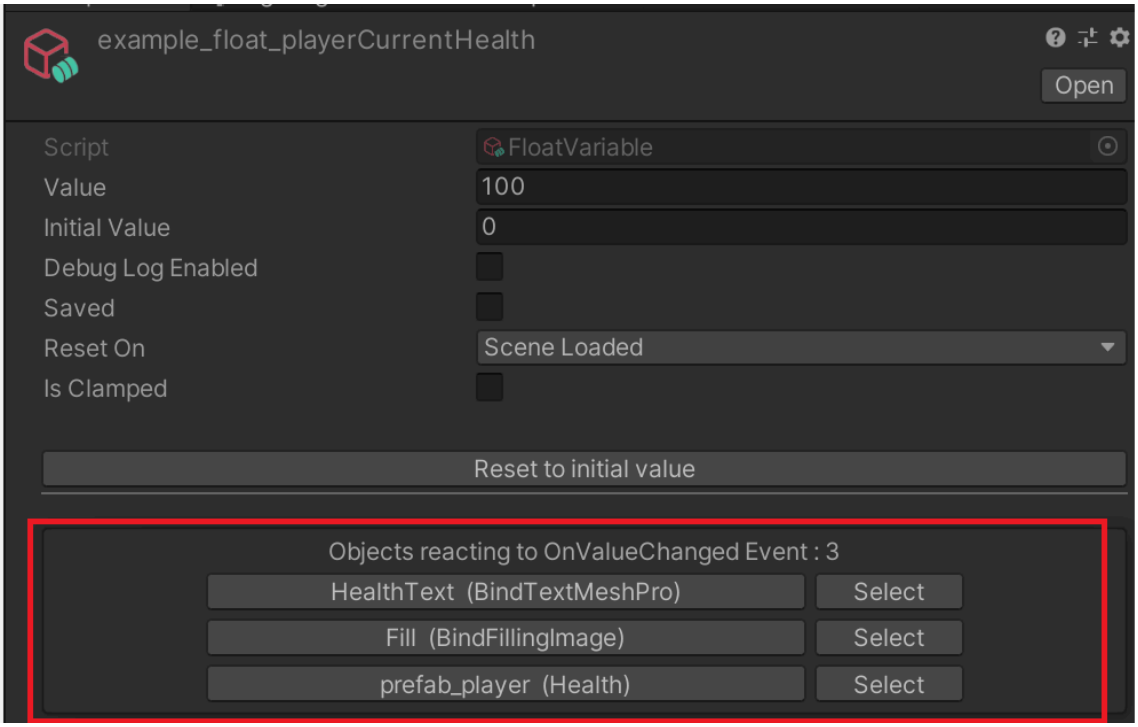


Alternatively, you can expose a reference to a Scriptable Variable directly in your class. Then, by clicking on the 'Create' button in the inspector, you can generate a new instance of that Scriptable Variable in the folder currently selected in the project window (you can change folder path in the settings). For example:

```
public class MyCustomClass : MonoBehaviour
{
    [SerializeField] private FloatVariable _floatVariable;
}
```



When you are in play mode, the objects (and their components) that have registered for the **OnValueChanged** Event of a Scriptable Variable become visible in the inspector



As seen in the screenshot above, three objects are registered to this variable. Examining the first element that responds to this variable, we find the GameObject named 'HealthText' in the hierarchy, specifically its component 'BindTextMeshPro.'

By clicking on the first button, the object is pinged in the hierarchy. Clicking on the 'Select' button will select the object in the hierarchy. This visual debugging is useful for tracking what responds to the changes of your variable.

For more detailed explanation and examples of Scriptable Variables, please refer to the example scene '1_ScriptableVariables_Example_Scene'.

Question: it seems like you might need to have every individual variable be its own scriptable object, so for 50 enemies, each enemy's HP would be 50 different scriptable variables?

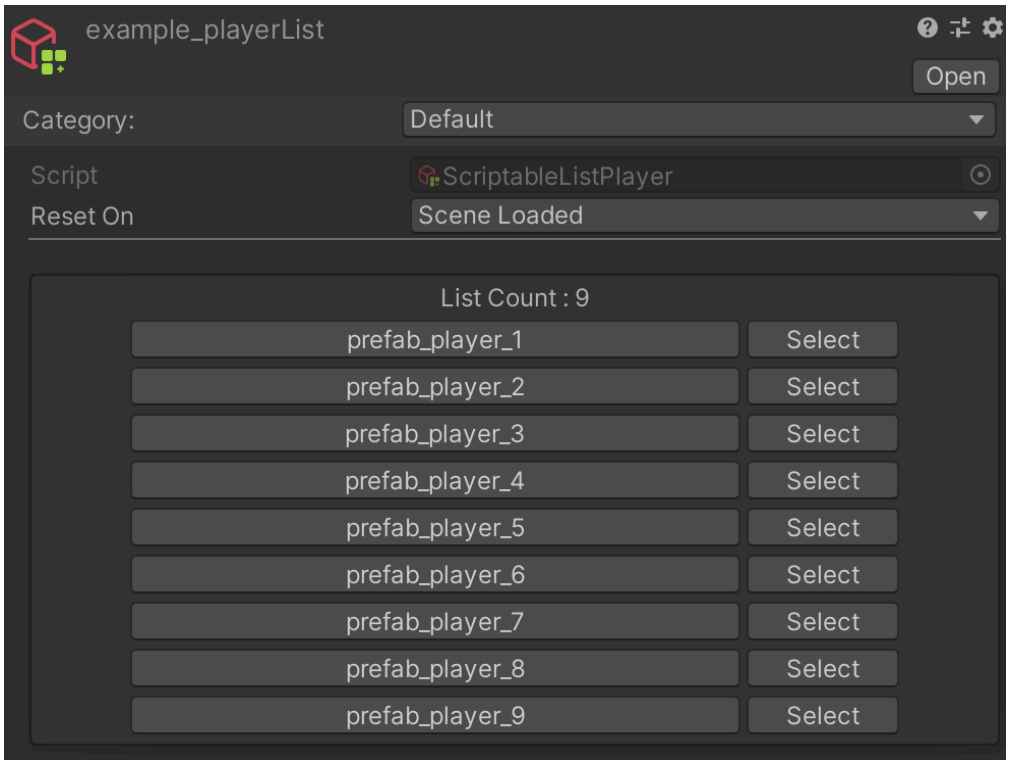
Yes. If you don't want to create the variables beforehand, you can instantiate the Scriptable variable containing the HP at runtime (on Awake for example) and pass it as a reference to your different systems - using events or another way.

However, maybe that is not the best approach. Scriptable variables are best for things that have dependencies with other systems but are singular. It suits more for things like Player HP or a Boss HP as well. For many enemies maybe it's overkill.

Most of the time, if there are a lot of enemies, you don't really want or need to solve dependency with their individual stats. Instead, the most common use case is to create a Scriptable List of enemies that you can reference anywhere (solving most of your dependencies) and access their stats there.

Scriptable Lists

Lists can be useful to solve dependencies by avoiding the need to have a "manager" in between your classes.



Let's go through its properties:

- **Reset On:** when is this list cleared?
  - *Scene Loaded:* whenever a scene is loaded. Ignore Additive scene loading.
  - *Application Start:* when the game starts.
- **List content**: in play mode, you can see all the elements that populate the list. By clicking on the first button (with the name of the object), it pings the object in the hierarchy. By clicking on the "Select" button, it will select the object in the hierarchy.

The ScriptableList has several events that you can subscribe to by code.

1. **OnItemAdded/ OnItemRemoved**

Sends the item added/removed as an argument.

2. **OnItemCountChanged**

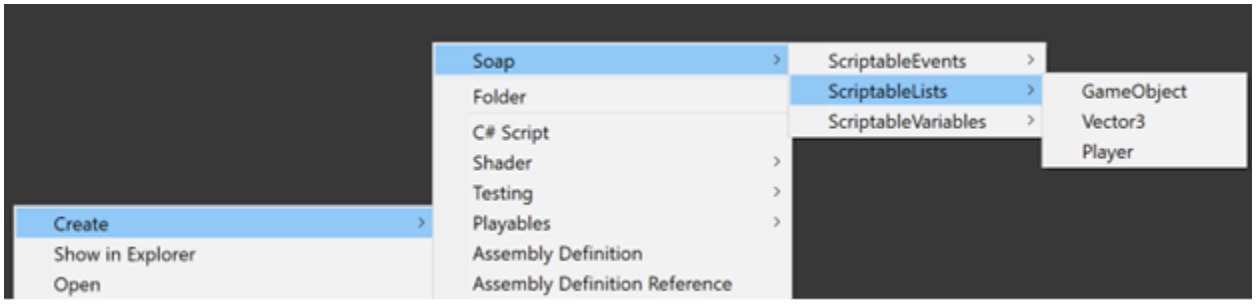
Sends the item added/removed as an argument.

3. **OnItemsAdded / OnItemsRemoved**

This event is useful if you don't want the OnItemAdded/OnItemRemoved event triggered every time you add/remove an item. This will be called once after a range of items have been added/removed.

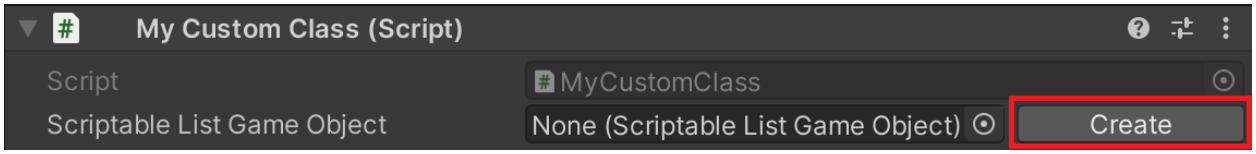
To create a new list, simply right click in the project window and locate the scriptable list:

Create/Soap/ScriptableLists/



Alternatively, you can expose a reference to a Scriptable List directly in your class. Then, by clicking on the 'Create' button in the inspector, you can generate a new instance of that Scriptable List in the folder currently selected in the project window. For example:

```
public class MyCustomClass : MonoBehaviour
{
    [SerializeField] private ScriptableListGameObject _scriptableListGameObject;
}
```



For more detailed explanation and examples of Scriptable Lists, please refer to the example scene '3_ScriptableLists_Example_Scene'.

Question: Can I use a ScriptableList to store data? Meaning setting data in the editor, then play the game and read this data.

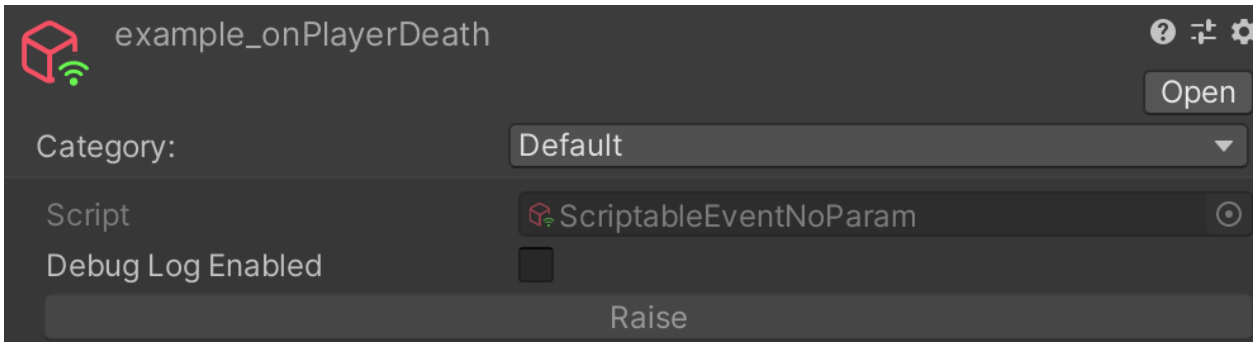
No. Scriptable lists should only be used at runtime. They will clear when exiting play mode, so it's not a solution to store data. They are very useful and they can contain anything. For example, you can have a script that stores positions (in a Scriptable list of Vector3) of a moving object, then another object would spawn coins on those positions.

If you want to store data, just make a ScriptableObject that has a reference to a List<T>.

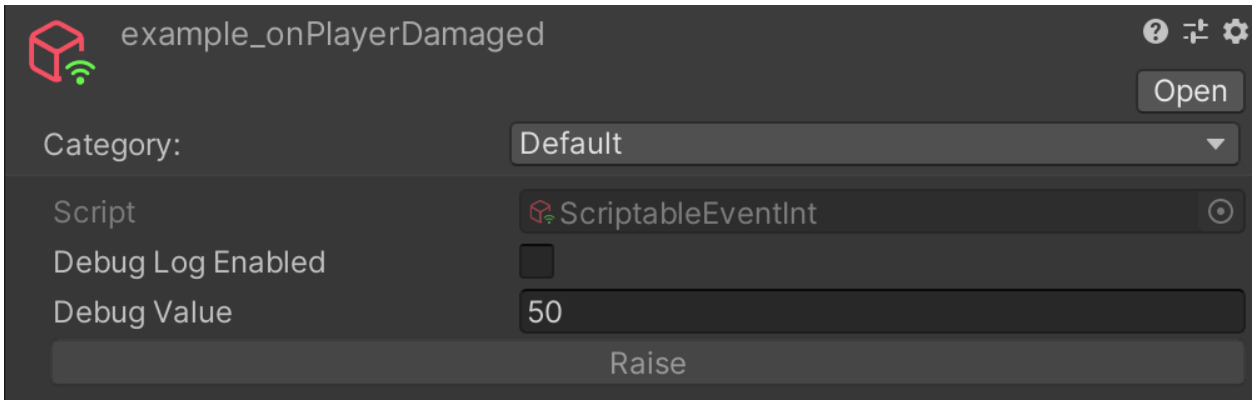
Scriptable Events

There are two types of scriptable events:

1. Events without parameters



2. Events with parameters



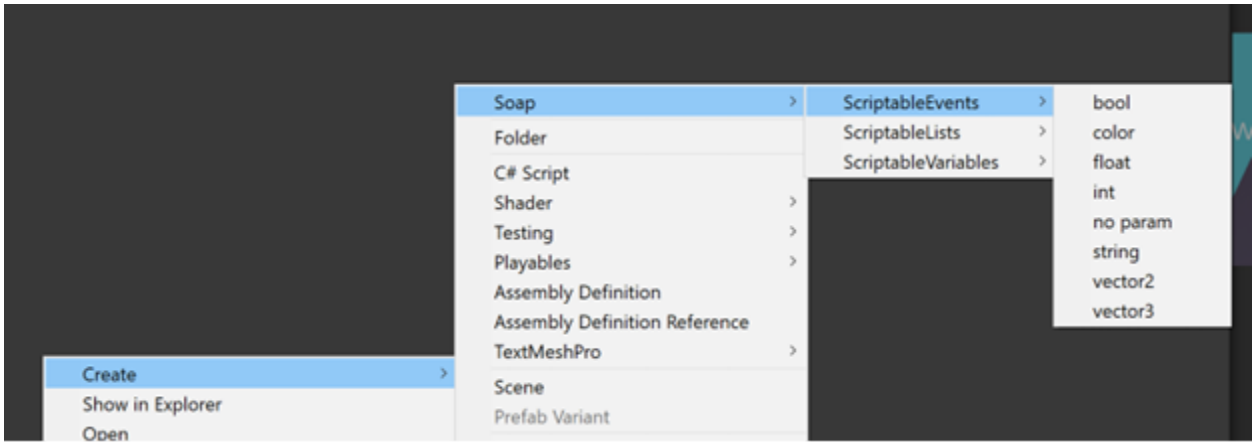
Let's go through its properties:

- **Debug Log Enabled**: If true, will log in the console the methods called when this event is raised (and the gameObject concerned).
- **Debug Value**: a value you can input to debug from the inspector
- **Raise:** this button is only active during play mode. It enables you to raise the event from the inspector.

Events can be raised by code, unity actions or from the inspector(via the raise button). Raising events in the inspector can be useful to quickly debug your game.

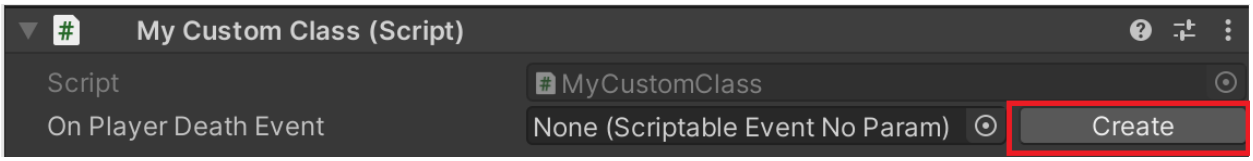
To create an event:

Create/Soap/ScriptableEvents/

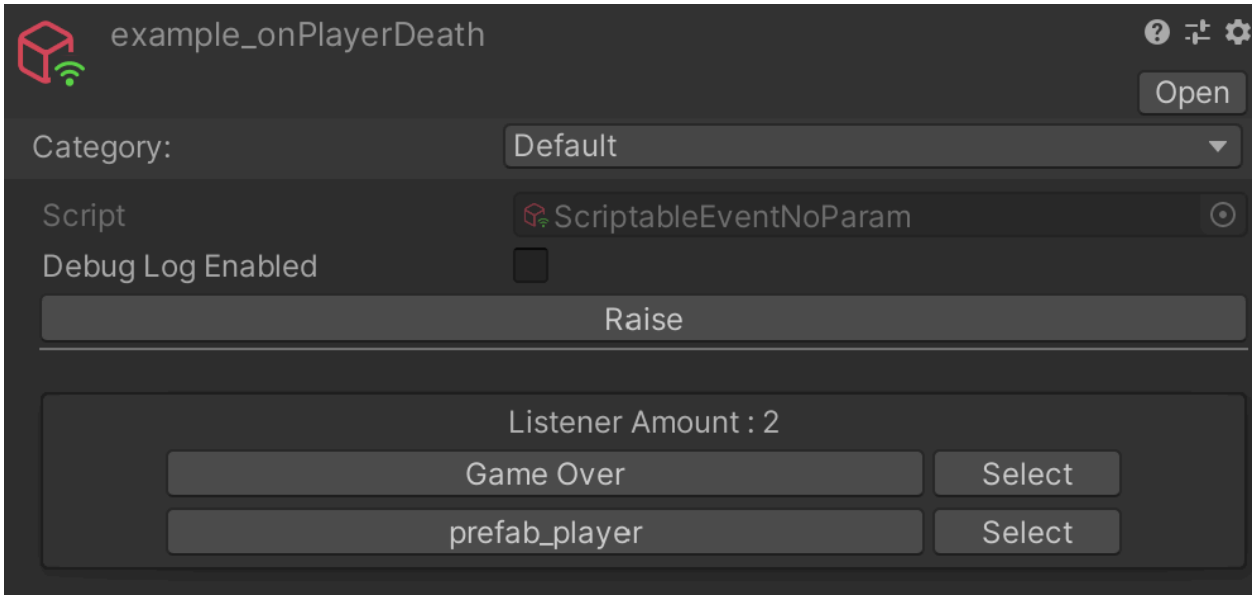


Alternatively, you can expose a reference to a Scriptable Event directly in your class. Then, by clicking on the 'Create' button in the inspector, you can generate a new instance of that Scriptable Event in the folder currently selected in the project window. For example:

```
public class MyCustomClass : MonoBehaviour
{
    [SerializeField] private ScriptableEventNoParam _onPlayerDeathEvent;
}
```

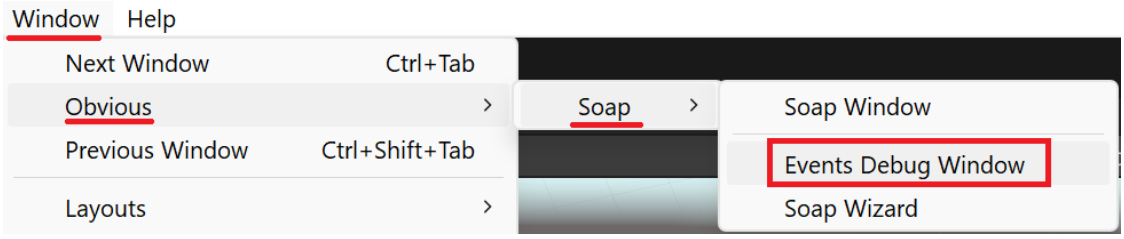


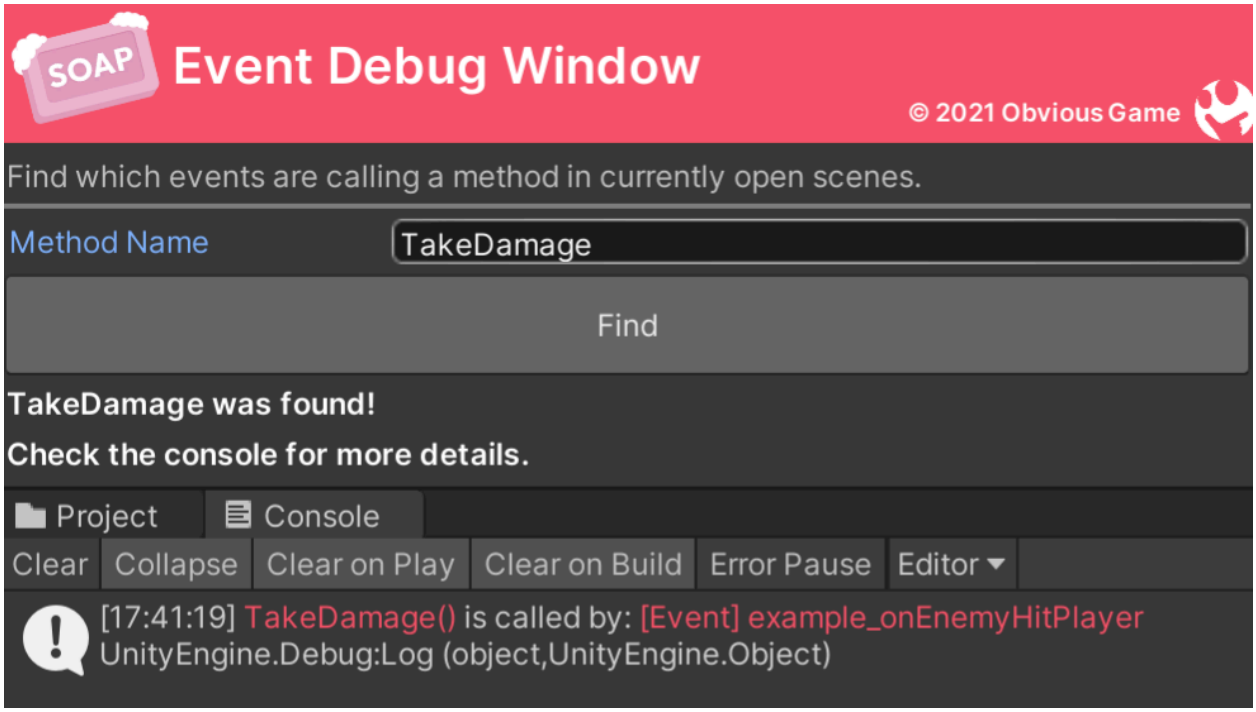
Scriptable events also have their own play mode custom inspector. During play mode, you can inspect all GameObjects that are registered to that event:



In editor, if you don't remember which event calls a method in one of your scripts, you can use the **Events Debug Window.** You can access it through the menu:

Window/Obvious/Soap/Event Debug Window





By typing the name of the method in the event debug window, it will search all objects in your scene and find which event calls that method.

Note: it is case sensitive.

Once the method is found (or not), a message is logged in the console. Clicking on this message will ping the corresponding object in the hierarchy. Additionally, you can click on the debug message to view the stack trace for more detailed information.

Please refer to the 4_ScriptableEvents_Example_Scene for a practical understanding of Scriptable Events and how to use them.

Question: When should I use a Scriptable Event or a Scriptable Variable?

It's not a black or white answer. The more you use SOAP, the clearer it becomes when to choose one approach over the other. Until then, I can provide a few examples for each use case. These examples are helpful, but they are not the only way. Everyone has their own preferences, and sometimes you might feel like using a different solution for a specific task.

Scriptable Variables are mostly useful for solving dependencies of a single element (or global variable) across different parts of the game:

- Player (and anything related to it, like its stats)

- Settings (volume, quality, options, etc.)

- Score, Coins, Level Index

- Boss (and anything related to it, like its stats)

- Game States (TutorialComplete, QuestXCompleted)

Scriptable Events are mostly useful for communicating indirectly with other components:

- OnPlayerDamaged(int damageAmount), OnPlayerDeath

- OnLevelCompleted(int levelIndex)

- OnCoinsCollected(int amount of Coins)

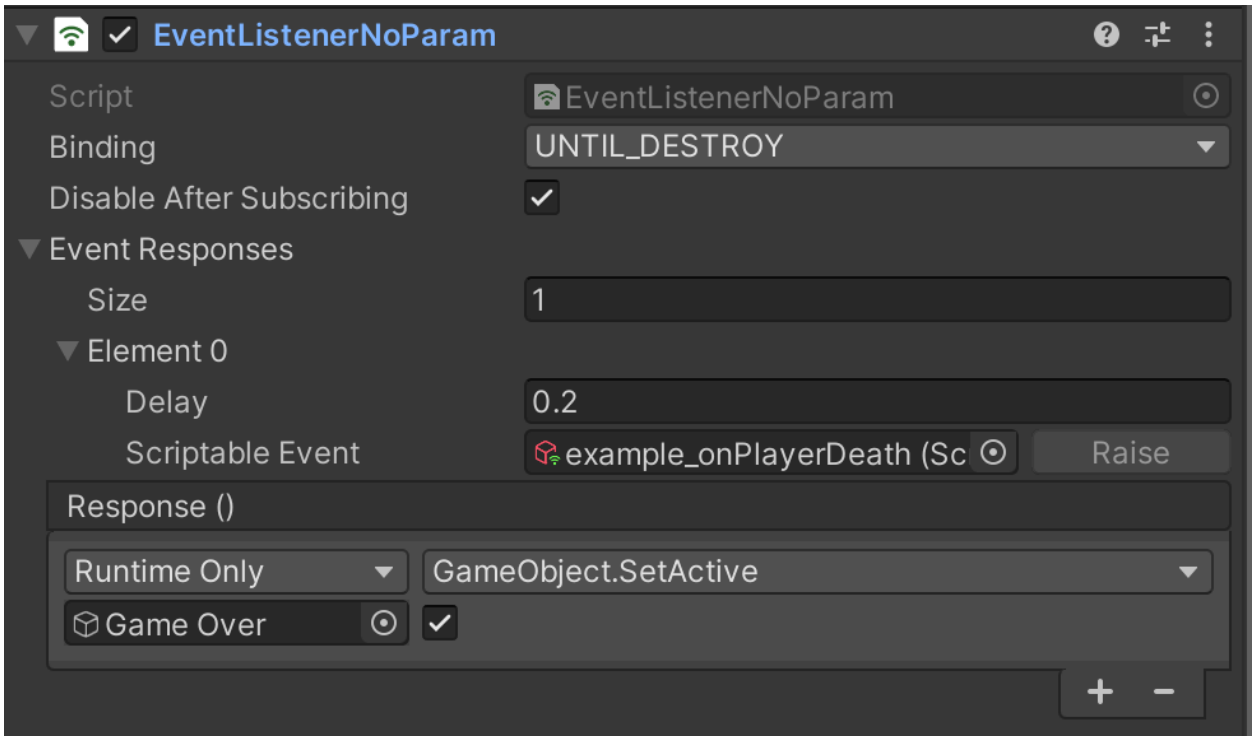
- OnPowerUpCollected(Vector3 position) - to spawn a VFX at that location

Personally, I use Scriptable Events for UI and for visual feedback (camera shake, VFX, sound, etc.). Many developers use them for game logic as well, but I prefer to keep that in code and use static events or an event bus because I find it easier to track down game flow through code.

Another example: If you want to show the score in the UI (using TextMeshPro). Should you use a Scriptable variable or a scriptable event? You can use both but I would choose to bind to a scriptable variable. For this, use the BindTextMeshPro Component. Attach that to the TextMeshPro GameObject and the text will refresh every time the value of the variable changes.

Event Listeners

To listen to these events when they are fired, you need to attach an **Event Listener** component (of the same type) to your GameObjects.

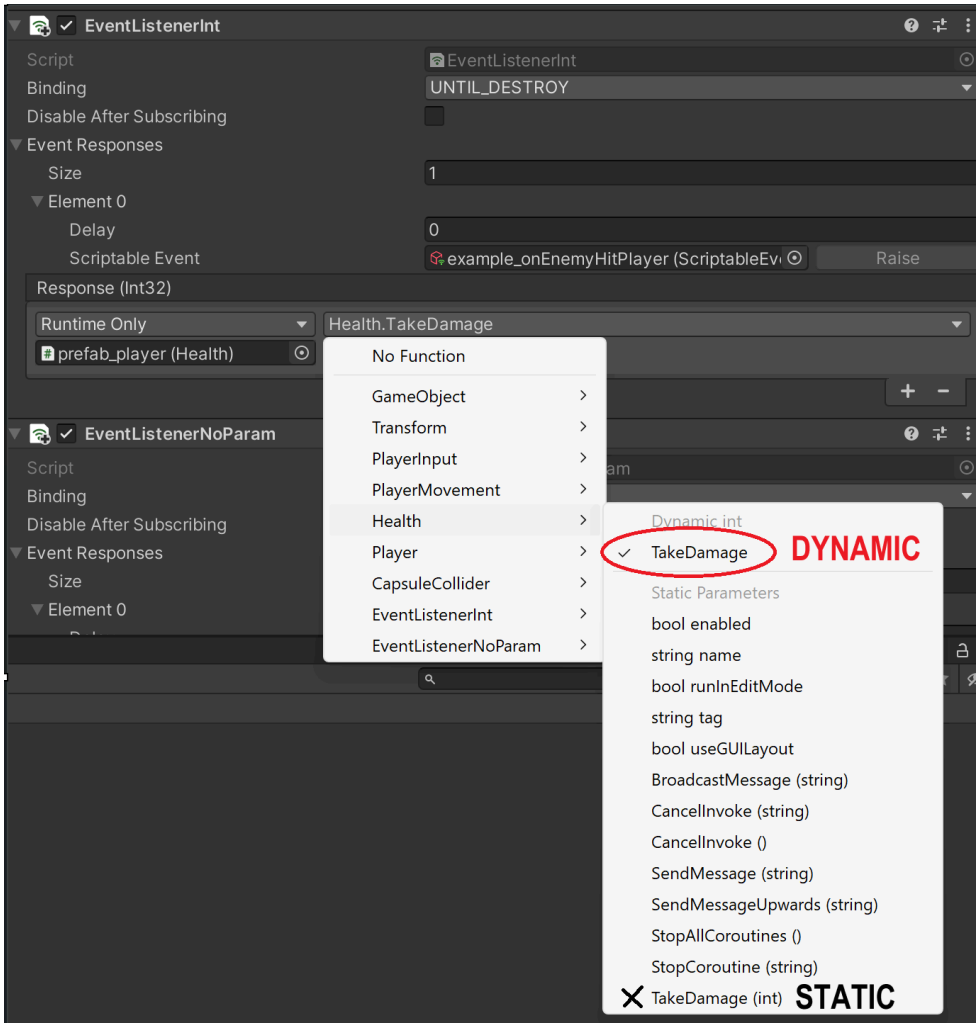


Let's go through its properties:

- **Binding**:
  - *Until_Destroy:* will register in Awake() and unsubscribe OnDestroy().
  - *Until_Disable:* will register OnEnable() and unsubscribe OnDisable().
- **Disable after subscribing**: if true, will deactivate the GameObject after registering to the event. Useful for UI elements.
- **Event responses:** each response invoked after the event was raised.
  - *Delay:* a delay in seconds before the response is invoked
  - *Scriptable Event*: the event to listen to
  - *Response*: unity actions triggered

You can also register to events directly from code (via the OnRaised action).

When using an EventListener with a parameter, make sure you bind it to the **Dynamic** method and not the static method ! Otherwise you won't be able to receive the parameters.



For more details and examples, please refer to documentation and the scene 4_ScriptableEvents_Example_Scene.

Question: Do I need to use an EventListener to receive messages from ScriptableEvents?

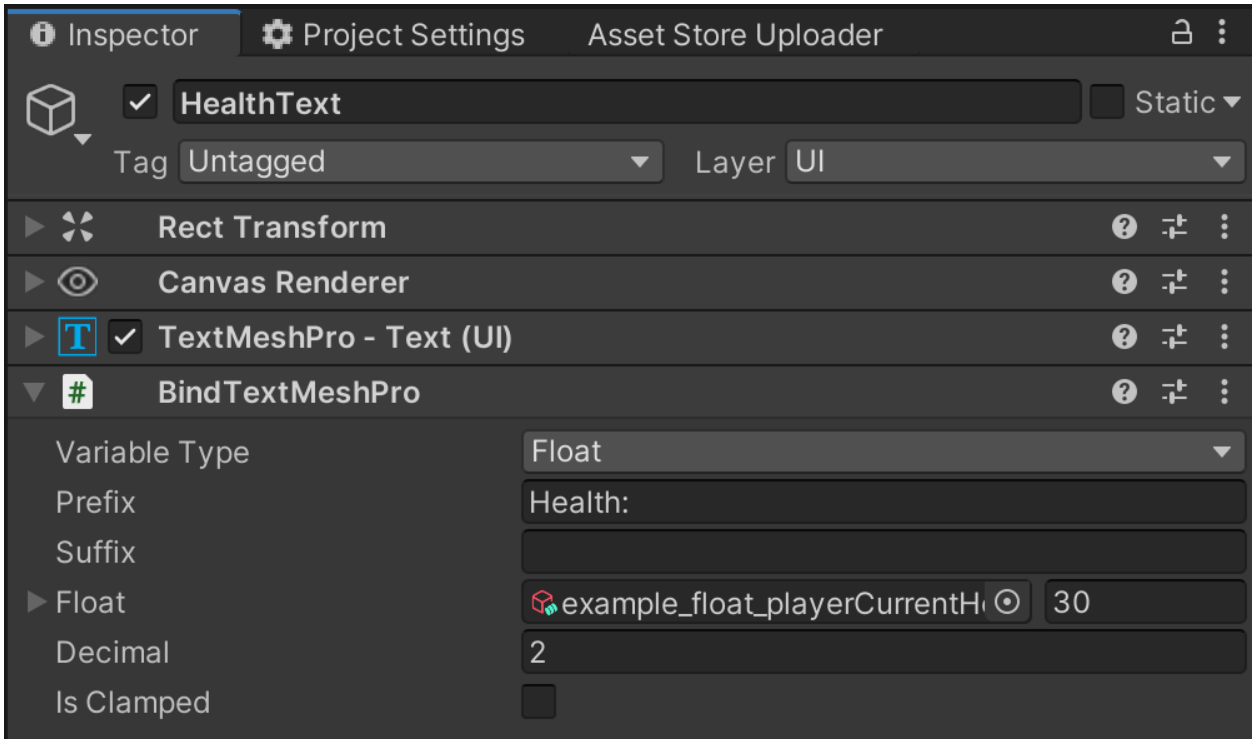
No, you can but you don't need to. If you like, you can register to the OnRaise method of ScriptableEvents from code, bypassing the EventListener.

Bindings

Bindings are components attached to GameObjects, enabling them to bind to a scriptable variable and execute a simple behavior when the variable's value changes. They are designed to save time, eliminating the need to create a script for minor adjustments like changing color, text, image, and other elements.

For a comprehensive understanding of how to use Bindings and their benefits, please refer to the '2_Bindings_Examples_Scene' in the Example scene and its accompanying documentation.

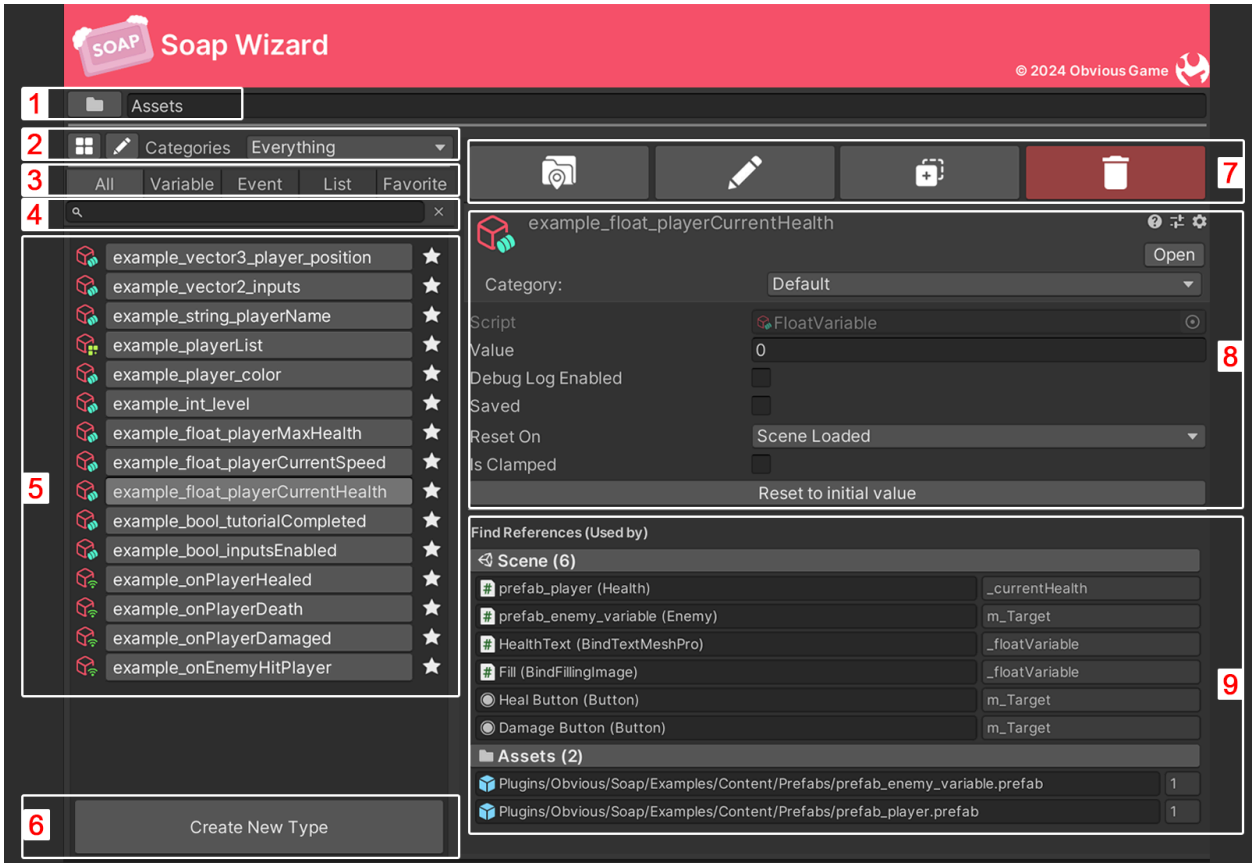
Example: BindTextMeshPro



Soap Wizard

The Soap Wizard is a custom window designed to manage all the Scriptable Objects from Soap in one location. It also offers convenient quality-of-life features. You can access this wizard from the menu:

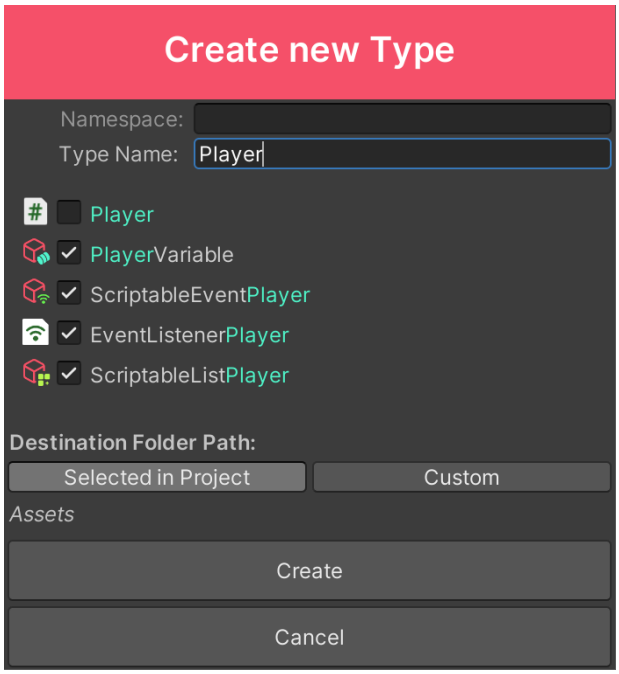
Window/Obvious/Soap/Soap Wizard



1. **Change Folder**: This is where you can specify the root folder from which the wizard will locate all the scriptable variables, events, and lists. By default, it is set to 'Assets', meaning it will populate the wizard with all the Soap scriptable objects in your project
2. **Categories**: filter by categories. Works like a layer mask (you can select multiples). You can change the layout of this filter and edit categories.

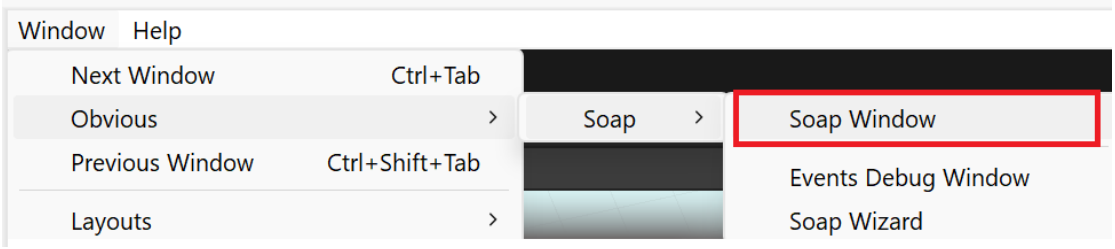
3. **Type filter**: filter by the type of soap scriptable objects (all, variables, event, list and favorite).
4. **Search Bar**: filter and search for specific scriptable objects. Use the clear button to clear your search quickly.
5. **Content**: display the Soap scriptable objects according to filters and selected folder. You can favorite any SO by clicking on the star button.
6. **Create New Type**: This opens the Create new type popup where you can choose to create a new type of Scriptable Object.
7. **Utilities**: (from left to right) select the SO in the project window, rename the SO, duplicate the SO (similar to CTRL+D in project window), delete the SO.
8. **Selected SO view**: inspector displaying the selected scriptable object.
9. **Find References:** See where this particular SO is used by components in the scene and by other assets in your project.

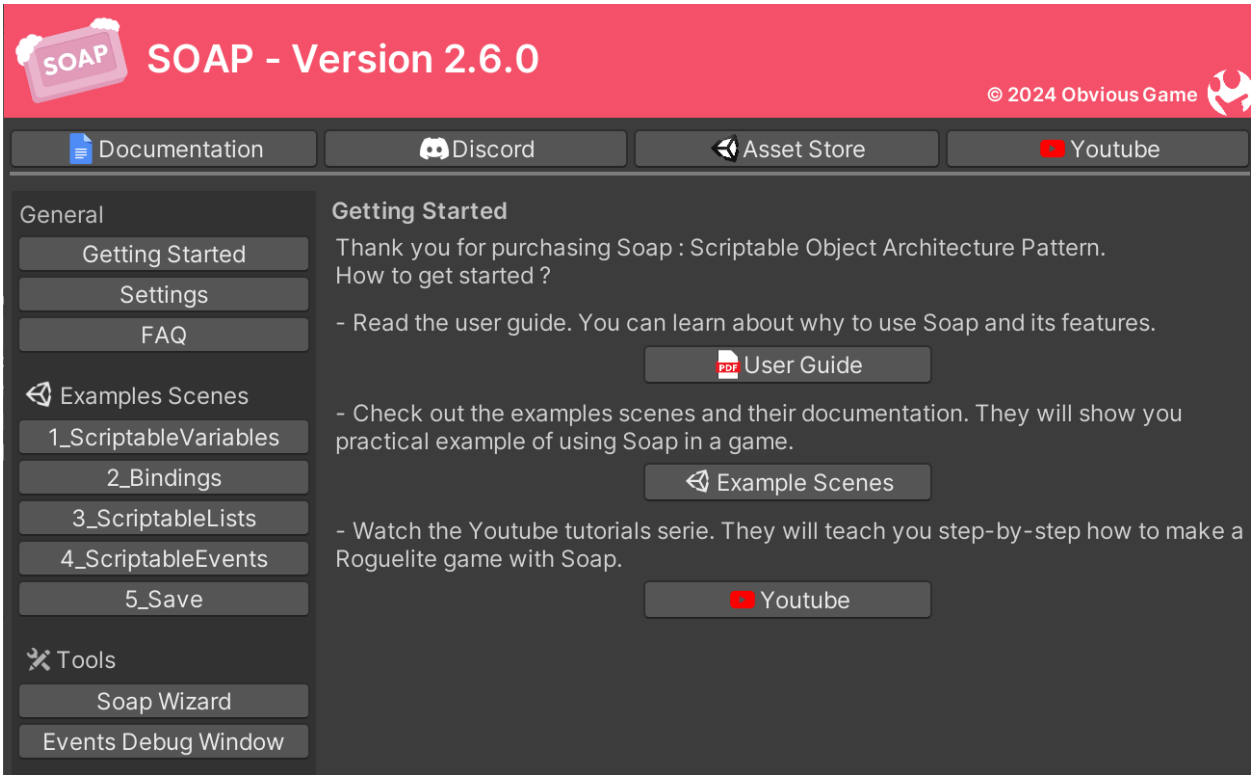
The Create new type popup allows you to create new types of Soap Scriptable Objects. For instance, if you wish to create a custom variable of type 'Player', this feature will generate the C# classes for you and save them in the selected folder or a custom folder of your choice.



Soap Window

The Soap Window appears the first time you import Soap. You can always open it via: *Window/Obvious/Soap/Soap Window*





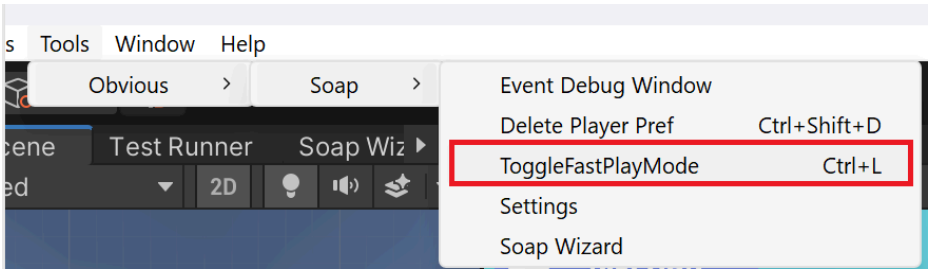
The Soap Window serves as a comprehensive hub, providing all the essential information you need to know about Soap. It's designed to simplify finding everything in one place. From here, you can access and modify the **settings**.

Enabling Editor fast play mode

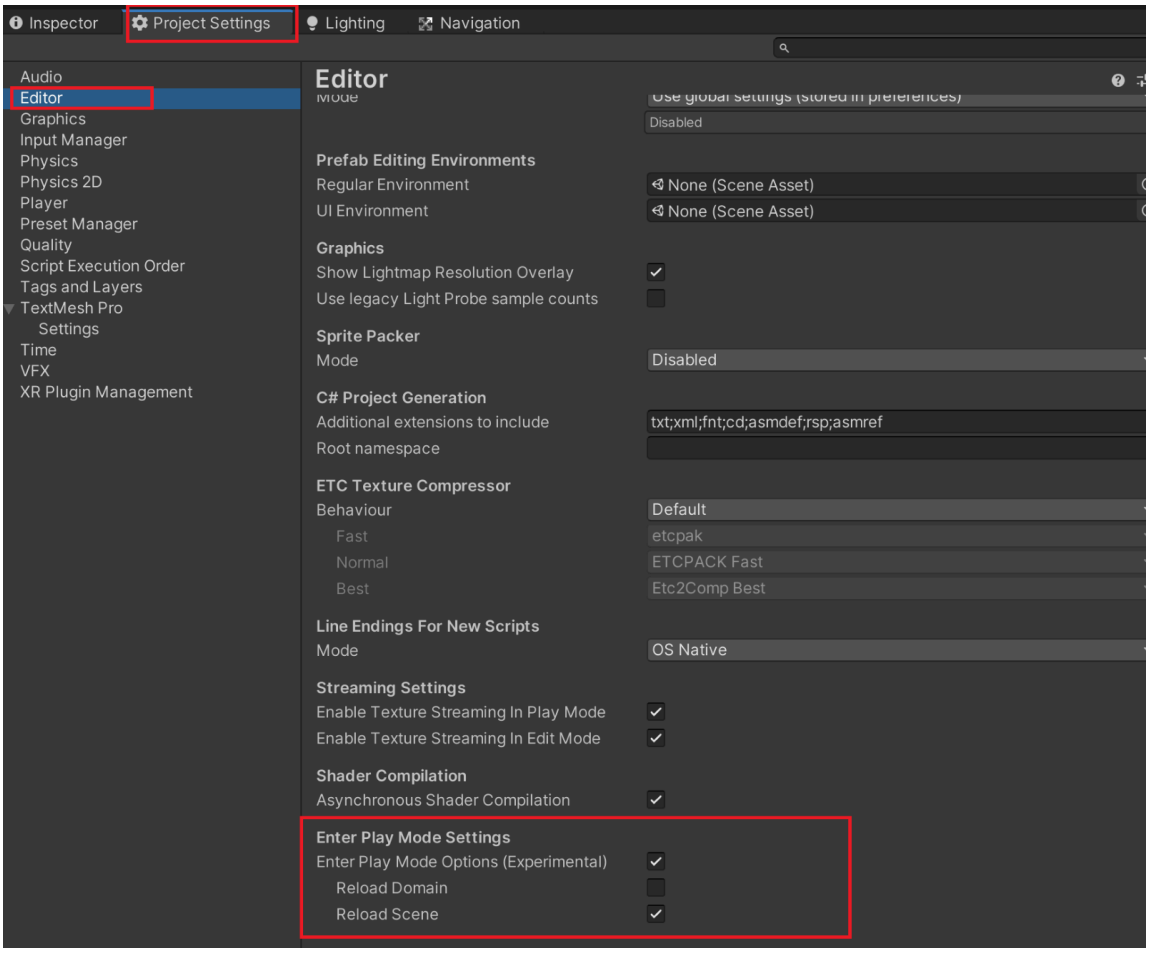
To enable / disable fast play mode. You can either:

- use the shortcut (CTRL+ L) or Tools menu: *Tools/Obvious/Soap/ToggleFastPlayMode*

A message in the console will tell you if its enabled or disabled.



- Manually set it (*Project Settings/Editor/EnterPlayMode Settings*)



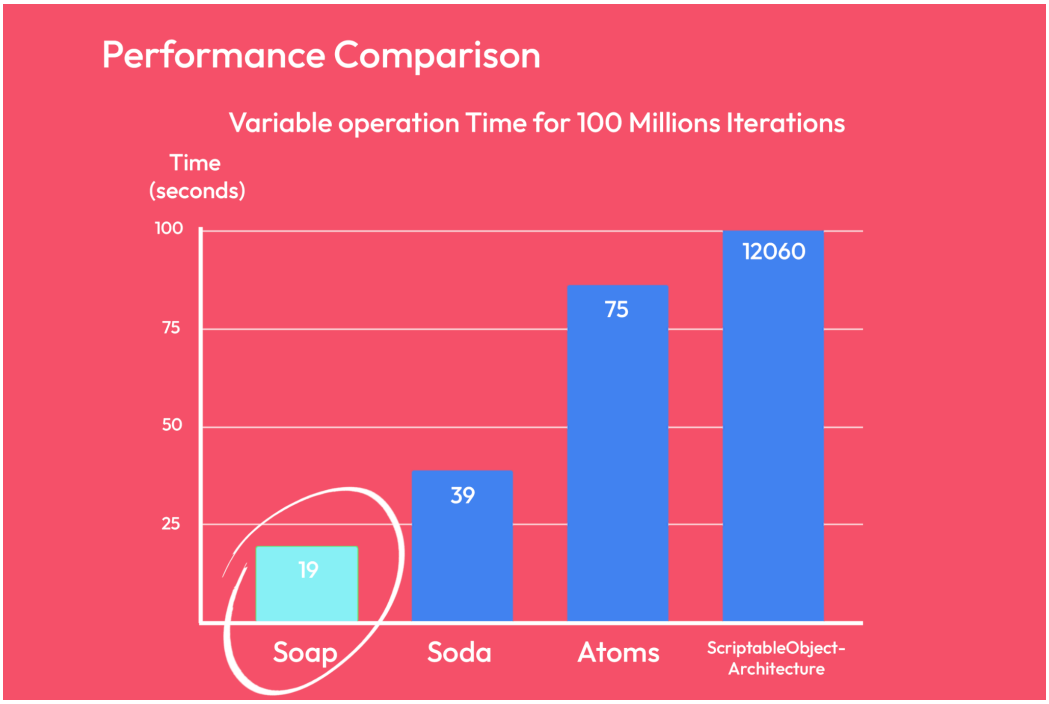
Ensure that 'Reload Scene' is enabled. Our goal is to quickly reload the scene but not the domain, which is the slower part.

Note: The option to toggle and have a shortcut for switching between fast and normal play mode exists because sometimes, certain plugins or components may cause errors (usually due to improper reset). Therefore, by quickly shifting to default play mode and playing once, you can resolve occasional issues. If something isn't working, most of the time, simply reloading the domain fixes it. Once resolved, I typically switch back to fast play mode .

Performance

Some people asked me about the performance cost of using Soap. Soap has been used in many commercial games (most on mobile/web GL), and performance issues were never related to using Soap. For those who are still curious, please read the explanations below.

Soap variables, lists and events rely on native C# events for callbacks. Soap custom editors and features are kept simple and minimal to stay efficient and performant (as opposed to some other ScriptableObject architecture frameworks). For example, scriptable variable operation is the fastest in Soap compared to other popular frameworks.



Setting the value of a Scriptable Variable every frame will create some garbage (few bytes) due to boxing and unboxing. Unfortunately, all the scriptable object architectures have the same issue. In most cases though, it is not common to have a lot of variables that have their value set every frame. Furthermore, it is rare that this amount of garbage becomes an issue, but I still want to mention it.

By assigning references in the editor with scriptable objects you don't need to solve the references at runtime (as opposed to a traditional Dependency injection framework).

How to extend SOAP?

You can expand this plugin by creating your own scriptable variables, lists, and events. You can either:

1. Use the Soap Wizard to automatically generate the code for the classes (refer to 'Create New Type' above),
2. Write the classes yourself. If you choose the latter, simply inherit from the following classes and replace 'T' with your type:

- ScriptableVariable<T>
- ScriptableList<T>
- ScriptableEvent<T>
- EventListener<T>
- VariableReferences<V, T>

Ensure your class is Serializable by adding the [System.Serializable] attribute to it.

Additionally, you can create new 'Binding' components. While I have developed a few, you are encouraged to be creative and design bindings that will be beneficial in your games. Examples of how to extend the package can be found in the scripts of various examples.

Compatibility

Soap is compatible with **Odin**, **Fast Script Reload** and most editor specific assets.

Soap integrates well with any other plugins. Currently, Soap has integration with:

- **Playmaker**: 🗀 Soap-Playmaker

Note: if you are using NaughtyAttributes, make sure you delete the ObjectEditor.cs script from Soap. NaughtyAttributes has its own script, and it conflicts with the one from Soap. Things should work fine after that 😊.

Question: Is it safe to move the "Obvious" folder into a "Plugins" folder?

Yes. If you import the Soap-Playmaker actions, make sure the Obvious folder is in the same **Parent** folder as Playmaker, otherwise you will get a compile error.

Contact

Any questions, suggestions, or feedback?

Check out first our FAQ: SOAP: FAQ

Feel free to reach me at: obviousgame.contact@gmail.com

And join the discord server: https://discord.gg/CVhCNDbxF5

Please leave a review on the asset store, as it really helps us to improve the package and to gain visibility.