Full Inspector Guide

Overview

Congratulations! Full Inspector is a powerful editor extension that will simplify your workflow as a game developer.

The inspector now supports interfaces and abstract types, structs, arbitrary generic types, properties, and has a better list/array editor. Of course, dictionaries are naturally supported too – they go through the normal generic type editing system.

Full Inspector also provides a natural extension to the Unity serialization system. All of the above types will now be serialized properly within Unity; if Json.NET can serialize it, then you're good to go. Serialization integration is seamless; it goes directly through Unity serialization.

Support

Support is freely available for Full Inspector. Bugs and general issues can be reported on the GitHub page. Email contact is also available here.

Quick Start

The following list is what you need to keep in mind when working with Full Inspector. More information on these bullet points follows in later sections.

1. (day-to-day) Derive from `BaseBehavior`, not `MonoBehaviour`
2. (day-to-day) If you override `Awake`, call `base.Awake()`
3. (day-to-day) Annotate your object with `[JsonObject(MemberSerialization.OptIn)]` and members with `[JsonProperty]`
4. (day-to-day) Wrap `Component` references with `Ref<>`
5. (once) Call `FullInspectorSaveManager.SaveAll()` before custom save-game logic

Setup

The only requirement to use Full Inspector is that instead of deriving from `MonoBehaviour`, you derive from `BaseBehavior`. Additionally, if you override Awake, ensure that you call `base.Awake()` as the first line of your override. You also need to annotate your types with `[JsonObject(MemberSerialization.OptIn)]` attributes and annotate properties that you want serialized with `[JsonProperty]`.

Stated differently, here's a bullet list of how to use Full Inspector.

1. Derive from `BaseBehavior`, not `MonoBehaviour`
2. If you override `Awake`, call `base.Awake()`
3. Annotate your object with `[JsonObject(MemberSerialization.OptIn)]` and members with `[JsonProperty]`

Important: If you have a reference to a `BaseBehavior` in your serialized code, ensure that you wrap it in `Ref<>`.

`Ref<>` ensures that the referenced `Component` serializes as a reference instead of as another full instance of the behavior. Please follow either *Right (1)* or *Right (2)*. Writing code similar to *Wrong* will cause your object to be serialize incorrectly.

Right (1):

```
    [JsonObject(MemberSerialization.OptIn)]
    public class SampleRefBehavior : BaseBehavior {
        [JsonProperty]
        public Ref<BaseBehavior> BehaviorReference;
    }
```

Right (2):

```
    [JsonObject(MemberSerialization.OptIn)]
    public class SampleRefBehavior : BaseBehavior {
        [JsonProperty]
        [JsonConverter(typeof(ComponentConverter))]
        public BaseBehavior BehaviorReference;
    }
```

Wrong:

```
    [JsonObject(MemberSerialization.OptIn)]
    public class SampleRefBehavior : BaseBehavior {
        [JsonProperty]
        public BaseBehavior BehaviorReference;
    }
```

`Ref<>` instances serialize identically to how Unity serializes `UnityEngine.Objects`; they exhibit the same behavior when the containing GameObject becomes a prefab, when it is instantiated, etc.

Important: If you have custom save-game logic, make sure that you run `FullInspectorSaveManager.SaveAll()` before your save logic; it will ensure that every `BaseBehavior` instance is ready to go through Unity serialization. Saves can be detected automatically in the editor but not in a published build.

There are lots of samples inside of the sample folder that contain more examples of how to use Full Inspector; however, it should be extremely straightforward. You have been provided with the full source code; it is highly commented. This document also provides a higher-level overview of key concepts and extension points in Full Inspector.

Simple Inspector Customization

There are a couple of special attributes that you can apply to your class members to provide some easy inspector customization.

CommentAttribute	Add a comment below the given field/property/type
TooltipAttribute	Add a tooltip viewable after hovering over the field/property
MarginAttribute	Add space above the given field/property
HiddenAttribute	Don’t show this attribute in the inspector (by default, every member is shown, even private ones)

Full Inspector Customization

Please see `FullInspectorSettings` to customize how Full Inspector operates. It is located at `FullInspector/FullInspector/FullInspectorSettings.cs`

Extra Editor Features

You can right-click on any component which derives from `BaseBehavior` to manually save its current state or restore its last saved state. Further, you can select “FullInspector/Show Serialized State” in the Unity top-menu to view the currently serialized state of the object directly below the inspector content. This JSON is modifiable and the state of the behavior will update in real time to the serialized state modifications.

Custom Property Editors

Full Inspector works its magic via a fully rewritten editing system inspired by `PropertyDrawer`; however, Full Inspector continues where `PropertyDrawer` stops. You only need to read this section if you’re interested in writing a custom property editor.

Writing a property editor is similar to writing a custom `PropertyDrawer`. We’ll go through how to write a `PropertyEditor` through a couple of real examples that are being used in Full Inspector. You can view all of the `PropertyEditors` in `FullInspector/FullInspector/Editor/PropertyEditors/Common`.

Special note: If you want to completely replace the editor for a component, simply write a `PropertyEditor` for that component type.

Simple (non-generic) Property Editors

Lets look at an extremely simple case: the property editor that gets invoked for ints.

```
[CustomPropertyEditor(typeof(int))]
public class IntPropertyEditor : PropertyEditor<int> {
    public override int Edit(Rect region, GUIContent label, int element) {
        return EditorGUI.IntField(region, label, element);
    }
    public override float GetElementHeight(GUIContent label, int element) {
        return EditorStyles.numberField.CalcHeight(label, 1000);
    }
}
```

Notice that this property editor is a public type that derives from `PropertyEditor<int>`. `PropertyEditor<int>` (which derives from `IPropertyEditor`) provides a type-safe version of `IPropertyEditor`. `IPropertyEditor` provides the core API that Full Inspector uses to interact with property editors.

Next notice that this type has an attribute `[CustomPropertyEditor(typeof(int))]`; this notifies the property editing system that this type can be used to edit ints.

The `Edit` callback simply provides the actual Unity editing experience; we just forward the call to `EditorGUI`; `GetElementHeight` returns how tall this property should be for the given label and property element.

Generic Property Editors

The previous property editor is also writable using a `PropertyDrawer`. However, `PropertyDrawer` lacks support for generic types; let's see how the `PropertyEditor` for `Ref<>` is written.

```
[CustomPropertyEditor(typeof(Ref<>))]
public class RefPropertyEditor<ComponentType> : PropertyEditor<Ref<ComponentType>>
    where ComponentType : Component {

    private IPropertyEditor _componentPropertyEditor =
       PropertyEditor.Get(typeof(ComponentType));

    public override Ref<ComponentType> Edit(Rect region, GUIContent label,
        Ref<ComponentType> element) {

        ComponentType component =
          (ComponentType)_componentPropertyEditor.Edit(region, label, element.Value);
        return new Ref<ComponentType> {
            Value = component
        };
    }

    public override float GetElementHeight(GUIContent label, Ref<ComponentType>
        element) {

        return _componentPropertyEditor.GetElementHeight(label, element.Value);
    }

}
```

This property editor looks very similar to the previous non-generic one, except that it is a generic type (`class RefPropertyEditor<ComponentType>`) and its attribute references an open generic type `Ref<>` (`[CustomPropertyEditor(typeof(Ref<>))]`).

The only special part of generic property editors is that they have matching generic arguments for the generic property type that they edit.

Here's another example of how to define generic property editors:

```
public class Pair<T1, T2> { }

[CustomPropertyEditor(typeof(Pair<,>))]
public class PairPropertyEditor<T1, T2> : PropertyEditor<Pair<T1, T2>> {/*omitted*/}
```

Again notice that the pattern holds.

Let's get back to the `RefPropertyEditor`. It doesn't look like Edit and GetElementHeight do much except forward calls to some weird value called `_componentPropertyEditor`. This `_componentPropertyEditor` is actually the property editor for the component type that the `RefPropertyEditor` is editing. This is one of the key patterns for writing generic property editors: we defer editing the actual generic parameters to some other property editor. More complex generic property editors (for example, the dictionary or list ones) do more work before dispatching to other property editors, but the core idea remains the same.

Inherited Property Editors

So you've gone digging through the property editors and have noticed that there is no `ListPropertyEditor`! How does Full Inspector provide editing for `List`, `LinkedList`, … types? Full Inspector also provides property editors which are inherited to their child types. So, if you look closely, there is actually an `IListPropertyEditor` and an `IDictionaryPropertyEditor`. Why don't we take a closer look at the `IListPropertyEditor`?

```
[CustomPropertyEditor(typeof(IList<>), Inherit = true)]
public class IListPropertyEditor<TList, TData> : PropertyEditor<TList>
        /* constraints omitted */ {
    /* implementation omitted */
}
```

The code and constraints behind the property editor has been omitted; they are not relevant to inherited property editors.

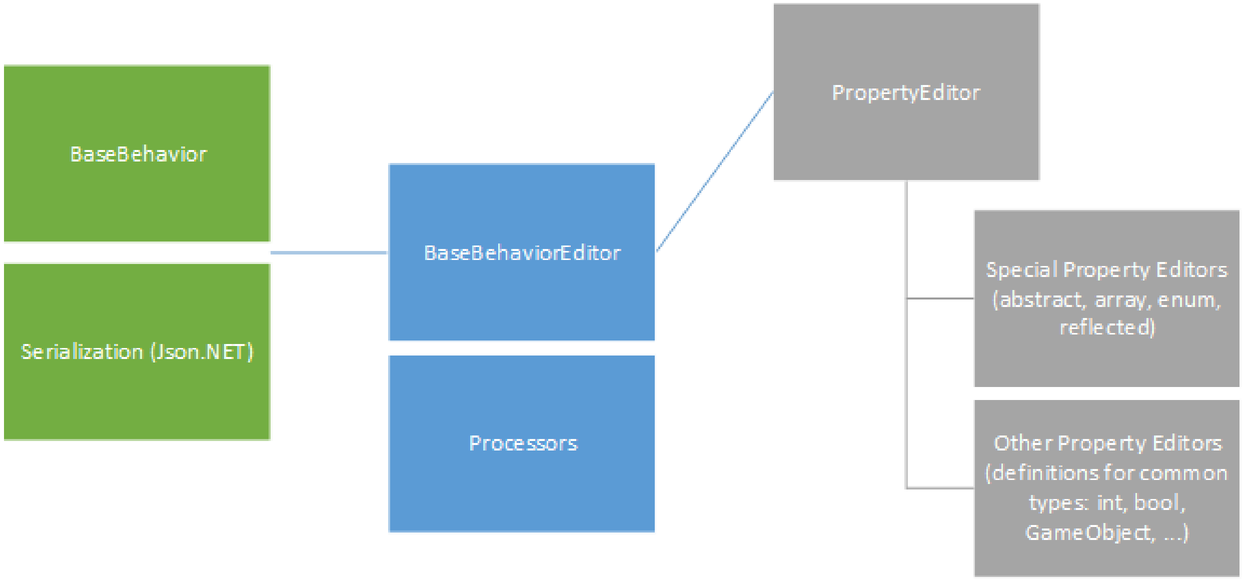
Notice that `IListPropertyEditor` takes *two* generic arguments, despite the fact that `IList` takes only one! This is because for every inherited property editor, the first generic argument is always the derived type that the property editor is editing. So for `List<int>`, the property editor will be an instance of `IListPropertyEditor<List<int>, int>`.

Also notice that for the attribute `[CustomPropertyEditor(typeof(IList<>), Inherit = true)]`, inherit has been set to true; by default, it is false.

Inherited property editors also work with non-generic types. The property editor for non-generic types can have either zero or one generic arguments; if it has one, it will be the actual property type the property editor is editing.

Architecture

This section contains some brief information on the internal architecture of Full Inspector. Internally the code is highly commented.


BaseBehavior
Serialization (Json.NET)
BaseBehaviorEditor
Processors
PropertyEditor
Special Property Editors (abstract, array, enum, reflected)
Other Property Editors (definitions for common types: int, bool, GameObject, ...)

This is a simplified diagram, but there are three primary moving parts.

BaseBehavior ensures that the component is always in a valid serialized state when data is requested from it. Json.NET is used as the serialization library.

BaseBehaviorEditor provides the integration between BaseBehavior and PropertyEditor. The Processors identify points in the editor when serialization save/restores need to occur, such as when code is being compiled or the user has entered play-mode.

PropertyEditor provides the core editing experience. It is a flexible and highly extensible system.