BTagged

Live Version: [BTagged Documentation]

Box Of Clicks
boxofclicks.com

Introduction

Having a unique global ID associated with a GameObject makes it easy to find what you're looking for no matter how it's instantiated and without creating a mess of inter-dependencies.

BTagged strives to add a clean, native-feeling UI to make this as simple as possible.
Add to that a reactive system, DOTS/ECS support and reference finding and you have a professional solution that scales with your project.

Don’t Panic! - ECS/DOTS

Unity have introduced a new set of tools to help simplify code and make it super fast to execute across all your CPU cores. BTagged utilises this under-the-hood if you have installed Unity.Entities but it works completely fine without. Work the same way you always have with GameObjects & MonoBehaviours (often called Hybrid) or start moving things over to Entities & Systems (Pure).

Requirements

Requires Unity 2019.4 or greater.

2020.1 or greater recommended if using DOTS.

Video Tutorials

Latest Updates (features as of v1.4.3)

- Added **'Parent'** methods to api. E.g. **BTagged.For(someGameObject).ParentWith(aTag)**;
  This is useful when e.g. the child doesn't have a tag and you want to find a parent of it that has one.

- Added PlayMaker Action support (unverified) thanks to ch1ky3n on the forums
- Added option in Settings to have BTaggedSO inherit from **Odin's SerializedScriptableObject** (instead of ScriptableObject). This option is in BTagged's settings in Unity's Preferences menu.

- Added Ungroup for Sub-assets (cmd+shift+G)
- Improved light skin UI
- Option to show group names
- Option to show hashes
- Reference finding under foldout - when closed, no work is done to find references
- Improved runtime tag support

- Added **BTagged.HasTag(gameObj, tag);** or **gameObj.HasTag(tag);**
  Checks to see if a gameobject has been tagged with the specified tag.

- Added **BTagged.FindInactiveTags();**
  When called this will search the entire hierarchy for Tags both active and inactive. Also works at Edit time. Call when the hierarchy has changed, before executing a query.

- Added **BTagged.ByName(string name)**
  Finds a Tag by name. Does not support Groups / group names. Recommended to use only when needed as it comes with some additional overhead. Any call to this method will cause GameObjects to track the Tags - by default they only track the hashes of the Tags.

- Added **MultiTag** - functionally the same as adding multiple 'Tags' to a GameObject but with just a single component.

- Much improved management of Tags and ScriptableObjects in general. **Group through Cmd+G** and rename from dropdown, groups or the assets themselves.

- **Improved performance** throughout. From runtime querying through to finding all uses of assets, a major optimisation pass has improved things considerably (not that anyone was asking but who doesn't love better performance 😂)

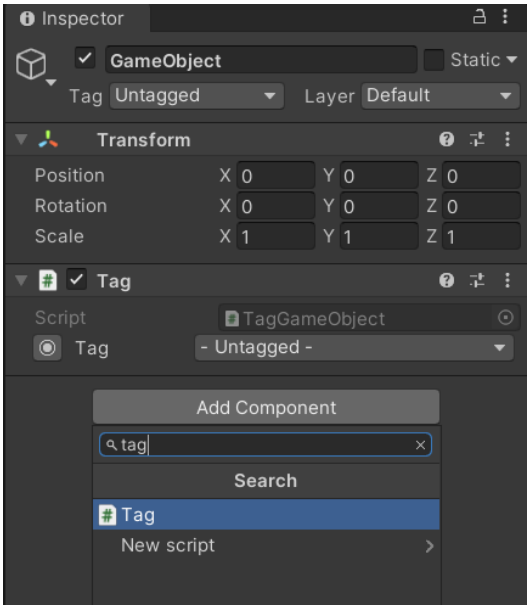
- Additional experimental global event listeners. E.g.
  var query = BTagged.FindAll(new Tag[] { aTag, bTag });
  BTagged.AddListenerOnEnabled(query, go => GOEnabled(go); });
  Callbacks similar to e.g. aTag.OnGameObjectEnabled but adding the power of queries to receive callbacks conditional upon multiple tags.

- More subtle improvements under the hood. Previously, a user was warned if they attempted to change the group of an asset. This is because it would be altering the guid of the asset, potentially causing references to be lost in prefabs or other scenes. Now in such instances all uses of these assets are detected and their **guids** are **automatically updated**. This should provide a hassle-free user experience however it's good to be mindful that this may cause some unexpected assets to change in source control. You should always be sure to check in such changes.

Tag Example

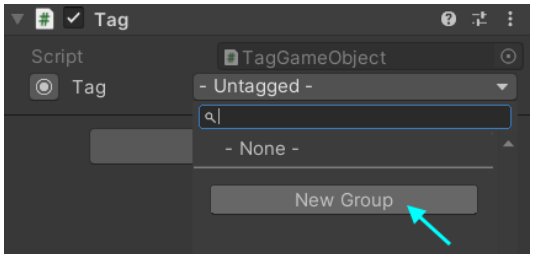
Tagging

1. To Tag a GameObject, first select the GameObject you wish to tag and add the 'BTagged->Tag' component:

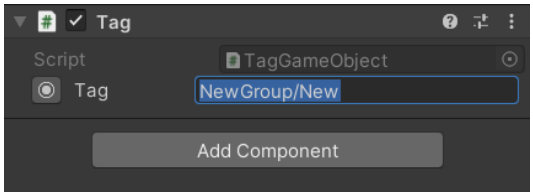


Let's start by creating a new Tag Group for all our tags:

2. Click the dropdown lable then 'New Group' to create our first Tag Group



3. By default it has created a new Tag Group in the Assets folder.
   The first part of the name (before the first '/') is the name of the Tag Group, the next part is the name of the Tag. Hit Enter when you've chosen something.



Great, the GameObject now has a unique hash associated with it. Keep reading to see how we can use this in our code.

How To Use A Tag

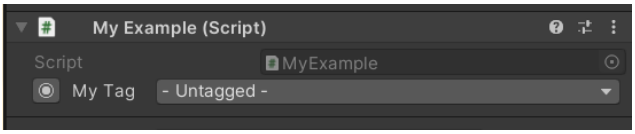
Let's quickly create a new MonoBehaviour called 'MyExample.cs' and add it to another GameObject in the scene.

Let's replace the code with the following:

```
using BOC.BTagged;
using UnityEngine;

public class MyExample : MonoBehaviour
{
    public Tag myTag;
}
```

If you add it to a GameObject in your project it should look like this:



Select the Tag you named before and let's add a bit more code:

```
using BOC.BTagged;
using UnityEngine;

public class MyExample : MonoBehaviour
{
    public Tag myTag;

    void Start()
    {
        var taggedGameObject = BTagged.Find(myTag).GetFirstGameObject();
        if(taggedGameObject != null) tagGamedObject.SetActive(false);
    }
}
```

BTagged.Find(myTag) creates a search query to find any object with the specified tag. **GetFirstGameObject()** executes the query and returns the first GameObject that has the tag you set above. For safety, we check if the GameObject is null in case there's no object in the scene with the tag or in case the tag hasn't been set. When you hit Play you should see your GameObject disappear / turn inactive.

At this point you may be thinking what happens when I have multiple GameObjects with the same Tag?
In that case there are several options. You can find all of them:

```
using BOC.BTagged;
using UnityEngine;
using System.Collections.Generic;

public class MyExample : MonoBehaviour
{
    public Tag myTag;

    void Start()
    {
        List<GameObject> allGOs = BTagged.Find(myTag).GetGameObjects();
    }
}
```

Or alternatively react to the tag's OnEnabled callback

```
using BOC.BTagged;
using UnityEngine;
using System.Collections.Generic;

public class MyExample : MonoBehaviour
{
    public Tag myTag;
    List<GameObject> allGOs = new List<GameObject>();

    void Start()
    {
        myTag.OnGameObjectEnabled += HandleTaggedGameObjectEnabled;
        myTag.OnGameObjectDisabled += HandleTaggedGameObjectDisabled;
    }
    private void OnDestroy()
    {
        myTag.OnGameObjectEnabled -= HandleTaggedGameObjectEnabled;
        myTag.OnGameObjectDisabled -= HandleTaggedGameObjectDisabled;
    }
    private void HandleTaggedGameObjectEnabled(GameObject go)
    {
        allGOs.Add(go);
    }
    private void HandleTaggedGameObjectDisabled(GameObject go)
    {
        allGOs.Remove(go);
    }
}
```

API Principles

Consistent

Transitioning between tagging GameObjects & Entities should ideally feel natural and the API should feel consistent across both.

Lightweight

Runtime performance should be a priority and use of the library should have minimal impact on existing project GameObjects and Entities. It should avoid cluttering the UI and components and generally keep out of the way.

Expressive

It should be possible to achieve all commonly desired behaviour such as finding GameObjects/Entities with combinations of tags. Ideally, all existing tagging functionality that Unity offers should be covered and exceeded.

Clear

The API should favour clarity over brevity, ensuring disambiguation wherever possible.

API Reference

Fluent Query API

The most flexible way to find a tagged object with BTagged is to use the fluent query API. Let's walk through some examples.

Find() - Search globally for any Tag

The basic form:

```
public Tag aTag;
var query = BTagged.Find(aTag);
List<GameObject> results = query.GetGameObjects();
```

This query will **Find all GameObjects** that have been tagged with whatever value **aTag** is set to.

If you only expect one matching GameObject you can use:

```
var query = BTagged.Find(aTag);
GameObject result = query.GetFirstGameObject();
```

You can also use the query to find matching components instead ('Image' components in this example)

```
var query = BTagged.Find(aTag);
List<Image> results = query.GetComponents<Image>();
```

Or again, just the first result

```
var query = BTagged.Find(aTag);
Image result = query.GetFirstComponent<Image>();
```

Find(Tag tag), FindAny(Tag[] tags), FindAll(Tag[] tags)

Any method that takes a single Tag can take multiple Tags. **Any** means that at least one Tag is required to be present. **All** means that they all must be present. The Any & All variants accept Lists or Arrays - anything that inherits IEnumerable.

For() - Search within a hierarchy

Often it is more useful to find a tagged object within the hierarchy, given a starting GameObject. For example, finding 'shoes' within a specific 'character' GameObject.

The basic form:

```
var query = BTagged.For(gameObject);
// With no filtering, 'gameObject' would be returned
GameObject result = query.GetFirstGameObject(); //gameObject
```

This doesn't look that useful by itself - let's introduce the concept of filtering using **With**.

With

```
public Tag aTag;
var query = BTagged.For(gameObject).With(aTag);
GameObject result = query.GetFirstGameObject(); //gameObject or null
```

If **gameObject** was tagged with the tag: **aTag,** result would be **gameObject**, otherwise it would be **null** - the gameObject doesn't have the tag. Still not that useful yet.

With

```
public Tag aTag;
public List<GameObject> someGameObjects;
var query = BTagged.For(someGameObjects).With(aTag);
List<GameObject> results = query.GetGameObjects();
```

You can also pass a List or Array of GameObjects to **For**(). In this case, the query will search through the list of GameObjects (someGameObjects) and return a list of any of them that have been tagged with the tag: **aTag**.

ChildrenWith

```
public Tag aTag;
var query = BTagged.For(gameObject).ChildrenWith(aTag);
List<GameObject> results = query.GetGameObjects();
```

Ok now it's starting to get more interesting - this query will find any children of **gameObject** that have the **aTag** tag.

ChildrenWith

```
public Tag aTag;
public List<GameObject> someGameObjects;
var query = BTagged.For(someGameObjects).ChildrenWith(aTag);
List<GameObject> results = query.GetGameObjects();
```

And this query will find *any* children in *any* of the **someGameObjects** that have the **aTag** tag.

With(Tag tag), WithAny(Tag[] tags), WithAll(Tag[] tags)

A reminder that any method that takes a single Tag can take multiple Tags. **Any** means that at least one Tag is required to be present. **All** means that they all must be present.

ChildrenWithAny

```
public List<Tag> someTags;
var query = BTagged.For(gameObject).ChildrenWithAny(someTags);
List<GameObject> results = query.GetGameObjects();
```

ChildrenWithAny searches for any children of **gameObject** that have at least one of the tags in the list **someTags**. Perhaps if children of a GameObject had been tagged with different colours, ChildrenWithAny(new Tag[] { blueTag, redTag } would find all children that were tagged blue or red.

Without

```
public Tag aTag;
public Tag bTag;
var query = BTagged.Find(aTag).Without(bTag);
```

This query will find anything tagged with **aTag** that isn't also tagged with **bTag**.

ChildrenWith(Tag tag), ChildrenWithAny(Tag[] tags), ChildrenWithAll(Tag[] tags)

Are also available.

These methods can be chained together to provide a natural way to search for your object.
For().With.ChildrenWithAny().Without().... etc

ECS & DOTS

Almost all the functionality described above also applies to queries for Entities.
The main exceptions are the Find methods. For Entities:

FindEntities(Tag tag)

FindAnyEntities(IEnumerable<Tag> tag)

FindAnyEntities(NativeArray<BHash128> tag)

FindAllEntities(IEnumerable<Tag> tag)

FindAllEntities(NativeArray<BHash128> tag)

BHash128

BHash128 is almost identical to Unity.Entities.Hash128 and can be safely cast between the two. BTagged's own Hash128 is used to avoid dependency on Unity.Entities & Unity.Mathematics for those purely working with GameObjects.

API Reference

Legacy Short API

In general the Query API is recommended for the power and flexibility it provides however there is still a place for the following convenience methods.

GameObjects

> tag.GetGameObjectForTag()

Returns the **first** GameObject in the scene matching the Tag. Note that if multiple GameObjects in the scene match this Tag, you should not rely on the result being deterministic.

> tag.GetGameObjectsForTag()

Returns **all** GameObjects (List<GameObject>) in the scene matching the Tag.

> gameObject.GetGameObjectForTag(Tag tag)

Returns the **first** GameObject with a matching tag that is a child of the passed gameObject or the gameObject itself. Note that if multiple GameObjects in the scene match this Tag, you should not rely on the result being deterministic.

> gameObject.GetGameObjectsForTag(Tag tag)

Returns **all** GameObjects (List<GameObject>) with a matching tag that is a child of the passed gameObject or the gameObject itself.

> IEnumerable<Tag>.GetGameObjectWithAllTags()

Returns the **first** GameObject in the scene matching **all** Tags passed in. Note that if multiple GameObjects in the scene match this Tag, you should not rely on the result being deterministic.

> IEnumerable<Tag>.GetGameObjectsWithAllTags()

Returns **all** GameObjects (List<GameObject>) in the scene matching **all** Tags passed in.

> gameObject.GetGameObjectWithAllTags(IEnumerable<Tag> tags)

Returns the **first** GameObject in the scene matching **all** Tags passed in that is also a child of the gameObject or the gameObject itself. Note that if multiple GameObjects in the scene match this Tag, you should not rely on the result being deterministic.

> gameObject.GetGameObjectsWithAllTags(IEnumerable<Tag> tags)

Returns **all** GameObjects (List<GameObject>) in the scene matching **all** Tags passed in that are also children of the gameObject or the gameObject itself.

> IEnumerable<Tag>.GetGameObjectWithAnyTags()

Returns the **first** GameObject in the scene matching **any** Tags passed in. Note that if multiple GameObjects in the scene match this Tag, you should not rely on the result being deterministic.

> IEnumerable<Tag>.GetGameObjectWithAnyTags()

Returns **all** GameObjects (List<GameObject>) in the scene matching **any** Tags passed in.

> gameObject.GetGameObjectWithAnyTags(IEnumerable<Tag> tags)

Returns the **first** GameObject in the scene matching **any** Tags passed in that is also a child of the gameObject or the gameObject itself. Note that if multiple GameObjects in the scene match this Tag, you should not rely on the result being deterministic.

> gameObject.GetGameObjectWithAnyTags(IEnumerable<Tag> tags)

Returns **all** GameObjects (List<GameObject>) in the scene matching **any** Tags passed in that are also children of the gameObject or the gameObject itself.

There are equivalent extensions for getting components too. For example:

> tag.GetComponentForTag<COMPONENT_TYPE>()

Where COMPONENT_TYPE is of type Component. E.g. Transform.
Method returns the **first** COMPONENT_TYPE in the scene matching the tag. Note that if multiple Components in the scene match this Tag, you should not rely on the result being deterministic.

Please check BTaggedExtensions.cs for a full list of extension methods and please post on the forums should you feel any essential functionality is missing.

ECS / Entities

> tag.GetEntityForTag(EntityManager em)

Returns the **first** Entity in the scene matching the Tag. Note that if multiple Entities in the scene match this Tag, you should not rely on the result being deterministic.

Please check BTaggedExtensions.cs for a full list of extension methods and please post on the forums should you feel any essential functionality is missing.

Architectural Reasoning/Justifications

This section is intended to give an overview of chosen implementation details and the reasoning behind them. These comments represent the last documented state, however implementation is subject to change.

Tags & Tag Groups are backed by ScriptableObjects (SOs)

Doing so (as opposed to JSON or another method) comes with some distinct advantages:

1) You can move and organise 'Tag Group' assets around your project as much as you like without disturbing functionality.
2) A custom inspector is easy to support and provides a nice way of inspecting all the occurrences of Tags and where they are used throughout the project.
3) Your own classes can easily reference these assets and gain the benefits of the custom property drawer.

Tags are serialized as references to ScriptableObjects (SOs) for GameObjects

An alternative approach explored was to serialize only the Hash on the MonoBehaviour. After all, this is really the minimal data required to find a GameObject at runtime and can be stored as a struct meaning good memory & gc characteristics.
Ultimately references to SOs were chosen instead for the following reasons:

1) A task as simple as logging the name of a Tag would require the construction of a lookup table which is both fragile to maintain and negates a lot of the aforementioned performance gains. A string could also be serialized with every tag instead but then we have a similar footprint at runtime.
2) Having references allows for a clean, easy to maintain, simple event callback API. Doing so without references would require the lookup table just mentioned and incur undesired runtime costs.
3) Serializing custom structs and supporting collections involves a lot of custom reflection code, fragile to Unity editor changes.
4) Ultimately, where performance is a high priority, the DOTS workflow excels and serializes only the necessary data.

The main con of this approach (vs the lookup) is the following issue:

Moving Tags between Groups causes GUIDs to change and references to be lost

For smooth sailing it's recommended to keep Tags within the group they were created in. Or else moved into another group before they are assigned to components in the project. Hopefully the reference finder helps somewhat mitigate this frustration.
If you find yourself strongly wanting to do this the best approach currently is instead of moving a tag to a different group, create a new tag in the destination group and then assign that one to all the objects that were referencing the old one. Then delete the old tag.
In the future perhaps a Replace With Tag feature could help with this issue.

Thank You!

A huge thank you for your support in purchasing this asset. Please reach out on the forums if you have any questions or feedback and of course, sparing 2 minutes to leave a review on the store would be *hugely* appreciated.

Good luck with your and your team's endeavours - you've got this!