Runtime Asset Database for Unity

The **Runtime Asset Database** is a library designed to simplify the implementation of a runtime save and load subsystem in your Unity application. This library replicates and extends the familiar concepts of prefabs, prefab variants, and assets found within the Unity Editor, making it easier than ever to manage and manipulate game assets at runtime and implement workflows similar to those of the Unity Editor dynamically during runtime.

> **Note**
> The repository containing the project used to create the above video can be found here

> **Code Companion** https://chat.openai.com/g/g-1UCDubUwr-your-code-companion-don-t-trust-me-blindly

Introduction

Unity developers often rely on the convenience and flexibility of the Editor's asset management system when designing their games. However, when it comes to implementing a save and load system at runtime, this process can become more complex. The Runtime Asset Database bridges this gap by bringing the essential asset management functionalities you're accustomed to into the runtime environment.

Features

- **Runtime Asset Management API:** Provides functionality to create, load, and manage assets during runtime.
- **Built on Unity Editor Prefab Concepts:** Utilizes familiar concepts from the Unity Editor's prefab workflow.
- **Asset and Asset Variant Support:** Supports assets and their variants.

- **Extensibility with new types and components:** Allows for the extension with new serializable types.
- **Pluggable External Asset Importers:** Offers the ability to integrate external asset importers seamlessly.

Getting Started

1. Unpack **StarterKit** Unity Package
2. Click Tools > Runtime Asset Database > **Build All**
3. Click Tools > Runtime Asset Database > **Create Host**
4. Create a new C# script named **GettingStarted.cs** in your Unity project.
5. Add the Following Code to Your Script:

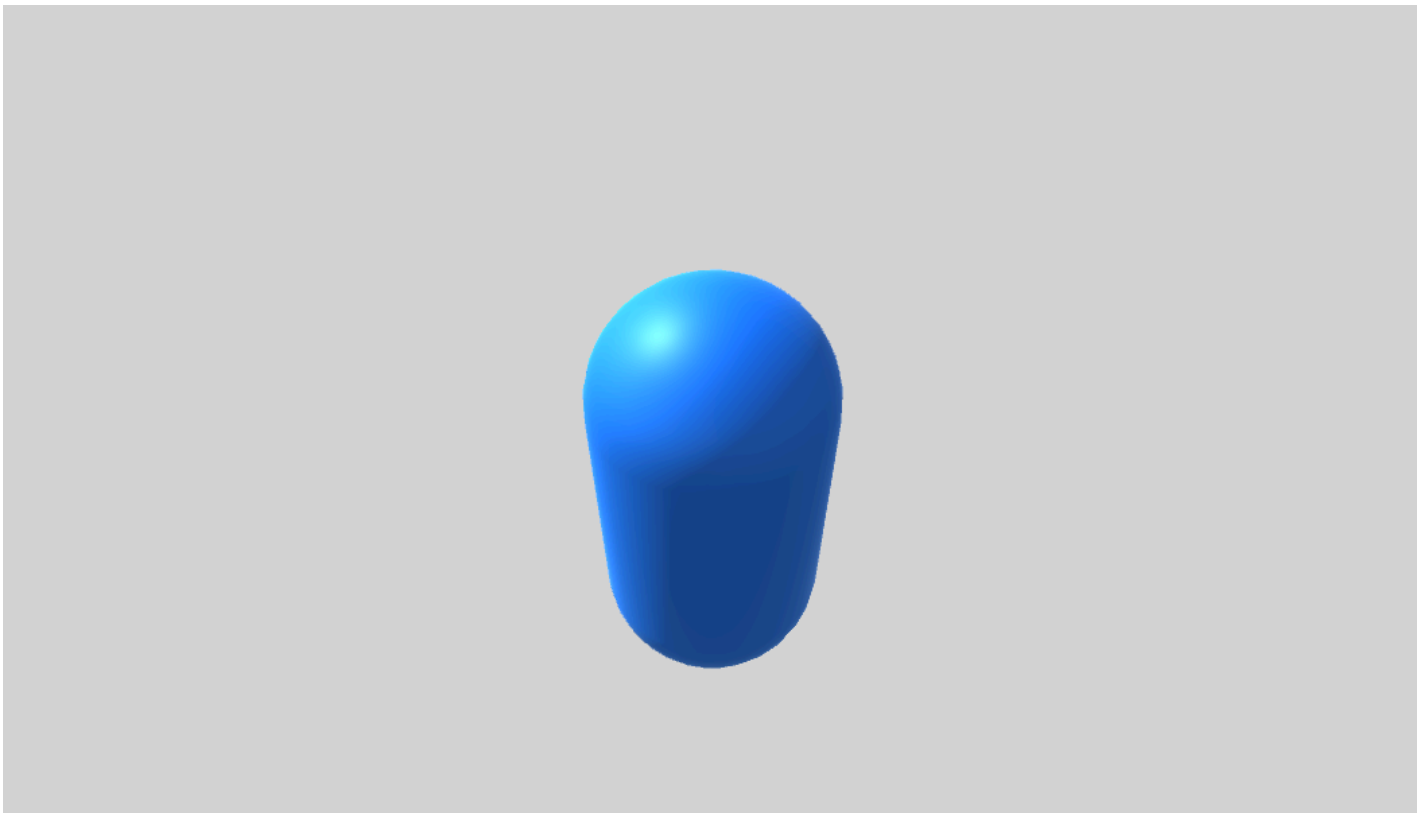
```
using UnityEngine;
using Battlehub.Storage;

public class GettingStarted : MonoBehaviour
{
    private IAssetDatabase m_assetDatabase;

    async void Start()
    {
        // Define your project path
        string projectPath = $"MyProject";

        // Obtain a reference to the asset database
        IAssetDatabase m_assetDatabase = RuntimeAssetDatabase.Instance;

        // Load the project
        await m_assetDatabase.LoadProjectAsync(projectPath);
    }
}
```

6. Modify Your Script as Follows:

```
using UnityEngine;
using Battlehub.Storage;

public class GettingStarted : MonoBehaviour
{
    async void Start()
    {
        string projectPath = $"MyProject";  // Define your project path

        // Obtain a reference to the asset database
        IAssetDatabase m_assetDatabase = RuntimeAssetDatabase.Instance;

        // Load the project
        await m_assetDatabase.LoadProjectAsync(projectPath);

        // Delete the "Assets" folder if it exists
        if (m_assetDatabase.Exists("Assets"))
            await m_assetDatabase.DeleteFolderAsync("Assets");

        // Create a new "Assets" folder
        await m_assetDatabase.CreateFolderAsync("Assets");

        // Create a primitive object (capsule) and make some modifications
        var go = GameObject.CreatePrimitive(PrimitiveType.Capsule);
        var filter = go.GetComponent<MeshFilter>();
        var renderer = go.GetComponent<Renderer>();
        var mesh = filter.mesh;
        var material = renderer.material;
        material.color = new Color32(0x0, 0x74, 0xFF, 0x0);

        // Create a mesh asset
        await m_assetDatabase.CreateAssetAsync(mesh, "Assets/Mesh.asset");

        // Create a material asset
        await m_assetDatabase.CreateAssetAsync(material, "Assets/Material.asset");

        // Create a "prefab" asset
        await m_assetDatabase.CreateAssetAsync(go, "Assets/Capsule.prefab");

        // Unload the project and destroy all assets to free up memory
        await m_assetDatabase.UnloadProjectAsync(destroy: true);

        // Load the project again
        await m_assetDatabase.LoadProjectAsync(projectPath);

        // Instantiate the prefab.
        await m_assetDatabase.InstantiateAssetAsync("Assets/Capsule.prefab");
```

```
    }
}
```

7. Press the "Play" button in Unity. **You should now see an instance of the object loaded from the Runtime Asset Database in your Unity scene**



Definitions

Folder

A folder simply refers to a directory on disk within the Runtime Asset Database project directory.

Asset

An asset is any object derived from UnityEngine.Object that can be serialized and deserialized. It is represented by three files on disk: the meta file, data file, and thumbnail. Assets fall into two categories:

- **Instantiable Assets**: These assets are analogous to prefabs in the Unity Editor. They can be instantiated and used directly in your project.
- **Non-instantiable Assets**: Examples of non-instantiable assets include materials and meshes.

Additionally, there is the concept of a Root Asset and a regular Asset.

- **Root Asset**: A GameObject is an example of a Root Asset.
- **Asset**: Components or meshes are examples of regular Assets. Their data is embedded in the same data file as the Root Asset.

Asset Variant

An **Asset Variant** is the equivalent of a Prefab Variant in the Unity Editor. It can only be created from an **Instantiable Asset**. Asset Variants become valuable when you need to define a set of predetermined variations of an Asset.

External Asset

An External Asset is an asset imported into the project using a specific importer, such as Addressable importers, importer that load asset from the Resources folder, GLB importer, or any other third-party importer. External Assets are read-only and contain only identifiers for parts within the data file. If you need to make edits to an External Asset, you can create an Asset Variant of it.

Instance

You can only instantiate an Asset, Asset Variant, or External Asset. The runtime asset database maintains mappings between instance parts and their corresponding asset parts

Dirty Instance

A "dirty" instance is one that has been marked to notify the runtime asset database that a change has occurred within an instance of an Asset Variant. This change needs to be saved to disk. When you load the Asset Variant next time, the asset database will read and apply this change to the base asset

Detached Instance

A detached instance is an instance that has no connection to the actual asset it originated from. You can convert an existing instance into a detached instance using the DetachAsync method, which will be discussed in more detail below.

Meta File

The Meta File contains asset metadata, which includes identifiers of dependencies, the asset's name, file ID. To get the metafile path, combine the file ID with the **.meta** extension; to get the thumbnail path, combine the file ID with the **.thumb** extension.

Data File

The Data File contains the binary serialized data of the asset. Runtime Asset Database uses protobuf-net as serializer.

Thumbnail File

The Thumbnail File contains image data of the asset's thumbnail texture.

Surrogate

Surrogates are classes with which the Serializer works. They act as intermediary classes that facilitate the reading and writing of data to the target Unity object. While these classes are often auto-generated, you have the flexibility to edit or create them from scratch.

Enumerator

Enumerators are classes used to retrieve Unity object dependencies in a structured manner. They enable the serialization of an entire object tree in a way that ensures dependencies are deserialized before dependent objects during the deserialization process. Similar to surrogates, enumerators are often auto-generated, but users also have the flexibility to create or edit them.

Examples

Load project

```
using System;
using UnityEngine;
namespace Battlehub.Storage.Samples
{
    public class LoadProjectExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;

            string fullpath = $"{Application.persistentDataPath}/Example Project";

            // load the project (creates a project folder if it does not exist)
            await m_assetDatabase.LoadProjectAsync(fullpath);

            // get root folder id
            Guid rootID = m_assetDatabase.RootID;

            // get child id by root id
            foreach (var childID in m_assetDatabase.GetChildren(rootID, sortByName: true))
            {
                // get asset metadata by id
                var meta = m_assetDatabase.GetMeta(childID);

                if (m_assetDatabase.IsFolder(childID))
                {
                    Debug.Log($"Folder {meta.Name} {meta.FileID}");
                }
                else
                {
                    Debug.Log($"{meta.Name} {meta.FileID}");
                }
            }
        }
    }
}
```

Unload project

```
using UnityEngine;
namespace Battlehub.Storage.Samples
{
    public class UnloadProjectExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;

            string projectPath = $"{Application.persistentDataPath}/Example Project";

            await m_assetDatabase.LoadProjectAsync(projectPath);
        }

        private async void OnDestroy()
        {
            if (m_assetDatabase != null)
            {
                if (m_assetDatabase.IsProjectLoaded)
                {
                    // unload the project and all assets

                    // destroy: true -> destroy the corresponding objects and game objects

                    await m_assetDatabase.UnloadProjectAsync(destroy: true);
                }
            }
        }
    }
}
```

Import external asset

```
using UnityEngine;
namespace Battlehub.Storage.Samples
{
    /// <summary>
    /// ---------------------------------------------------------------------------
    /// First register an external asset loader. This should only be done once,
    /// after that you can import multiple asses using this loader.
    /// ---------------------------------------------------------------------------
    /// In this example, I'm using the built-in ResourcesLoader for simplicity,
    /// but it could be any loader which implements the IExternalAssetLoader
    /// interface (AddressablesLoader, glTFLoader, FBXLoader, etc.)
    /// ---------------------------------------------------------------------------
    /// The loader in this example loads an asset from the Resources folder.
    /// In this particular example, the asset with the key "Hellephant" is in
    /// Assets/Battlehub/Storage.Samples.ProjectBrowser/Content/Resources
    /// ---------------------------------------------------------------------------
    /// </summary>
    public class ImportExternalAssetExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;

            string projectPath = $"{Application.persistentDataPath}/Example Project";
            await m_assetDatabase.LoadProjectAsync(projectPath);

            var rootID = m_assetDatabase.RootID;
            string key = "Hellephant";
            string loaderID = nameof(ResourcesLoader);

            IExternalAssetLoader loader = new ResourcesLoader();
            await m_assetDatabase.RegisterExternalAssetLoaderAsync(loaderID, loader);

            // convert externalAssetKey to unique file id
            var targetFileID = m_assetDatabase.GetUniqueFileID(rootID, $"{key}");

            // import external asset
            await m_assetDatabase.ImportExternalAssetAsync(key, loaderID, targetFileID);

            // instantiate imported asset
            await m_assetDatabase.InstantiateAssetAsync(targetFileID);
        }
```

```
    }
}
```

> **Note**
> To use the AddressablesLoader, make sure to import the Addressables package

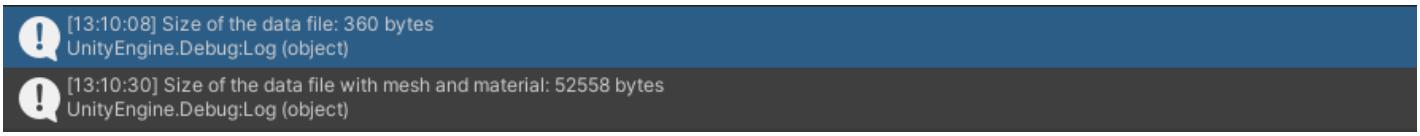
> **Note**
> You can also create your own external asset loader. To do this, create a new class and implement the IExternalAssetLoader interface:

```
using System;
using System.Threading.Tasks;
using UnityEngine;
namespace Battlehub.Storage.Samples
{
    public class MyLoader : IExternalAssetLoader
    {
        public Task<object> LoadAsync(string key, object root, IProgress<float> progress)
        {
            return Task.FromResult<object>(new GameObject(key));
        }

        public void Release(object obj)
        {
            GameObject go = obj as GameObject;
            if (go != null)
            {
                UnityEngine.Object.Destroy(go);
            }
        }
    }
}
```

Register external asset

```
using System;
using System.Collections.Generic;
using System.IO;
using UnityEngine;
namespace Battlehub.Storage.Samples
{
    /// <summary>
    /// ---------------------------------------------------------------------------
    /// Sometimes you don't want certain assets to appear in your runtime project
    /// managed by the RuntimeAssetDatabase (or you can't serialize/deserialize them),
    /// but you still want other assets to be able to reference them.
    /// ---------------------------------------------------------------------------
    /// A good example of an external asset is the default materials or meshes that
    /// exist in your Unity editor project.
    /// ---------------------------------------------------------------------------
    /// The RegisterExternalAssetsAsync method solves this problem.
    /// ---------------------------------------------------------------------------
    /// You should generate some guids to use them as an external asset identifiers
    /// https://guidgenerator.com/
    /// ---------------------------------------------------------------------------
    /// Assets passed to RegisterExternalAssetsAsync are never stored in data files
    /// and do not have a corresponding metadata file in the runtime project.
    /// ---------------------------------------------------------------------------
    /// </summary>
    public class RegisterExternalAssetExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;

            string projectPath = $"{Application.persistentDataPath}/Example Project";
            await m_assetDatabase.LoadProjectAsync(projectPath);
            var rootID = m_assetDatabase.RootID;

            GameObject capsule = GameObject.CreatePrimitive(PrimitiveType.Capsule);
            Material material = capsule.GetComponent<MeshRenderer>().sharedMaterial;
            Mesh mesh = capsule.GetComponent<MeshFilter>().sharedMesh;

            /// https://guidgenerator.com/
            var externalAssets = new Dictionary<Guid, object>()
            {
                {  new Guid("c872b08a-8b5e-41df-bf89-3522b8219dd6"), material },
```

```
                {  new Guid("3bad1a26-d851-49b5-a11c-6dfe74ee5341"), mesh }
            };

            // ------------------------------------------------------------------
            // Comment out the following line and you will notice that the
            // size of the data file written to the console becomes larger.
            // This is because without registering as external assets,
            // the mesh and material are serialized into the data file.
            // ------------------------------------------------------------------
            await m_assetDatabase.RegisterExternalAssetsAsync(externalAssets);

            var fileID = m_assetDatabase.GetUniqueFileID(rootID, $"Capsule");
            await m_assetDatabase.CreateAssetAsync(capsule, fileID);

            Debug.Log($"Size of the data file: {new FileInfo(fileID).Length} bytes");
        }
    }
}
```



Create asset

```
using Battlehub.Storage.EditorAttributes;
using UnityEngine;
namespace Battlehub.Storage.Samples
{
    public class CreateAssetExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;
        private IThumbnailUtil m_thumbnailUtil;

        [Layer]
        public LayerMask ThumbnailLayer;

        private void Awake()
        {
            var thumbnailUtil = gameObject.AddComponent<ThumbnailUtil>();
            thumbnailUtil.ThumbnailLayer = ThumbnailLayer;
            m_thumbnailUtil = thumbnailUtil;
        }

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;

            string projectPath = $"{Application.persistentDataPath}/Example Project";
            await m_assetDatabase.LoadProjectAsync(projectPath);
            var rootID = m_assetDatabase.RootID;

            GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
            // ---------------------------------------------------------------------
            // Asset can be created with or without thumbnail.
            // To generate thumbnail data you can use ThumbnailUtil.
            // ---------------------------------------------------------------------
            var thumbnailTexture = await m_thumbnailUtil.CreateThumbnailAsync(cube);
            var thumbnailBytes = await m_thumbnailUtil.EncodeToPngAsync(thumbnailTexture);

            var fileID = m_assetDatabase.GetUniqueFileID(rootID, $"Capsule");
            await m_assetDatabase.CreateAssetAsync(cube, fileID, thumbnailBytes);
        }
    }
}
```

Create thumbnail

```
using Battlehub.Storage.EditorAttributes;
using UnityEngine;
using UnityEngine.UI;

namespace Battlehub.Storage.Samples
{
    public class CreateAndSaveThumbnailExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;
        private IThumbnailUtil m_thumbnailUtil;

        [Layer]
        public LayerMask ThumbnailLayer;

        [SerializeField]
        private RawImage m_thumbnailImage;

        private void Awake()
        {
            var thumbnailUtil = new
GameObject("ThumbnailUtil").AddComponent<ThumbnailUtil>();

            // rotate thumbnail camera
            thumbnailUtil.transform.LookAt(-Vector3.one);

            // set thumbnail camera layer
            thumbnailUtil.ThumbnailLayer = ThumbnailLayer;

            // set desired thumbnail res
            thumbnailUtil.SnapshotTextureWidth = 512;
            thumbnailUtil.SnapshotTextureHeight = 512;

            m_thumbnailUtil = thumbnailUtil;

            if (m_thumbnailImage == null)
            {
                Debug.LogWarning("Set thumbnail image");
            }
        }

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;
```

```
            string projectPath = $"{Application.persistentDataPath}/Example Project";
            await m_assetDatabase.LoadProjectAsync(projectPath);
            var rootID = m_assetDatabase.RootID;

            GameObject capsule = GameObject.CreatePrimitive(PrimitiveType.Capsule);
            var fileID = m_assetDatabase.GetUniqueFileID(rootID, $"Capsule");

             // Create Asset (capsule becomes instance of an asset)
            await m_assetDatabase.CreateAssetAsync(capsule, fileID);

            // Create, encode and save thumbnail
            var thumbnailTexture = await m_thumbnailUtil.CreateThumbnailAsync(capsule);

            // Release asset instance
            await m_assetDatabase.ReleaseAsync(capsule);

            // Encode and save thumbnail
            var thumbnailBytes = await m_thumbnailUtil.EncodeToPngAsync(thumbnailTexture);
            await m_assetDatabase.SaveThumbnailAsync(fileID, thumbnailBytes);

            if (m_thumbnailImage != null)
            {
                // Show thumbnail texture
                m_thumbnailImage.texture = thumbnailTexture;
            }
        }
    }
}
```

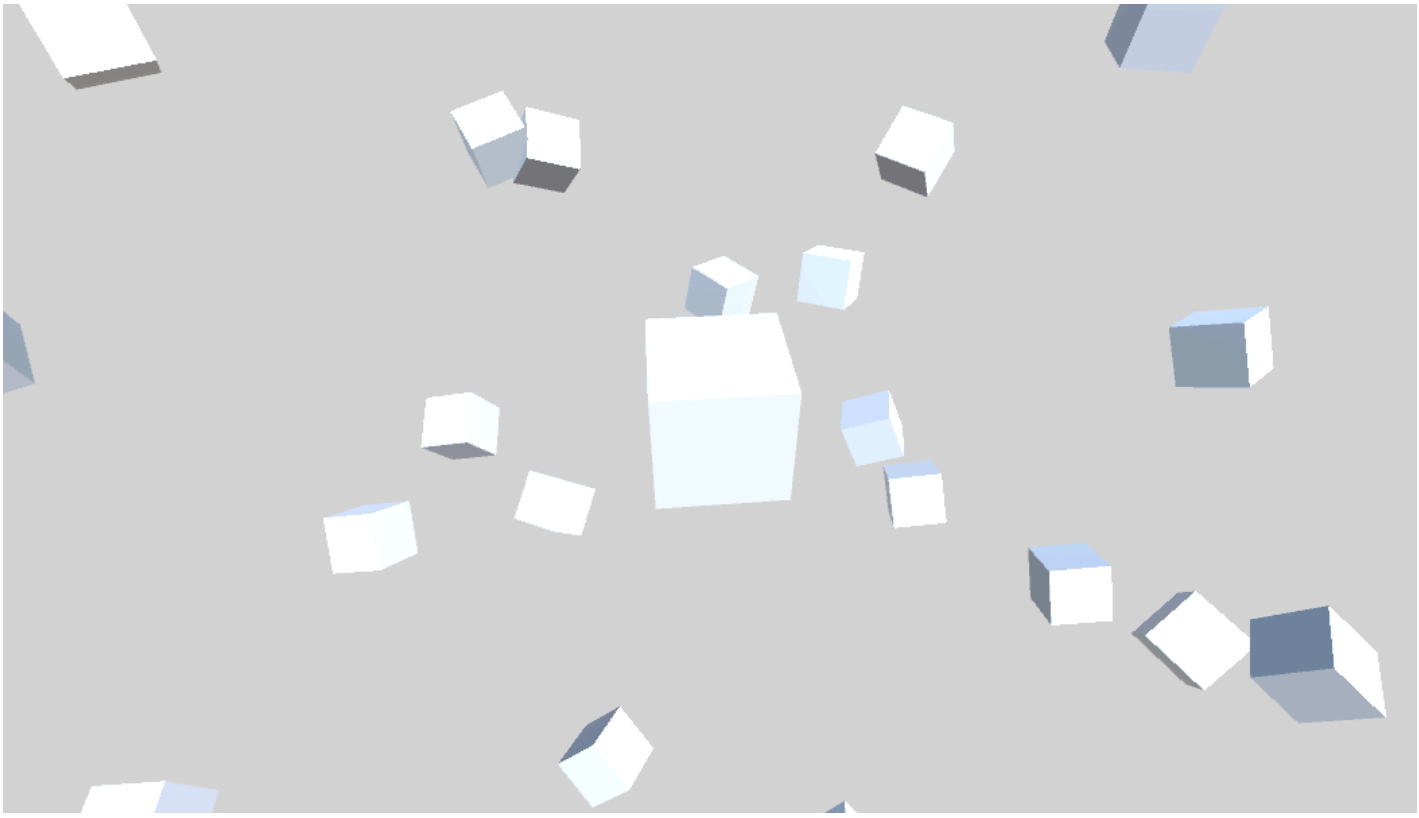
Load asset

```
using System.Linq;
using UnityEngine;
namespace Battlehub.Storage.Samples
{
    public class LoadAssetExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;

            string projectPath = $"{Application.persistentDataPath}/Example Project";
            await m_assetDatabase.LoadProjectAsync(projectPath);

            var rootID = m_assetDatabase.RootID;
            var fileID = m_assetDatabase.GetUniqueFileID(rootID, $"Capsule");

            // Create GameObject
            var capsule = GameObject.CreatePrimitive(PrimitiveType.Capsule);

            // Make some changes
            ModifyGameObject(capsule);

            // Create Asset
            await m_assetDatabase.CreateAssetAsync(capsule, fileID);

            // Unload All assets and free up memory
            await m_assetDatabase.UnloadAllAssetsAsync(destroy: true);

            // Load asset by fileID
            await m_assetDatabase.LoadAssetAsync(fileID);

            // Instantiate loaded asset
            await m_assetDatabase.InstantiateAssetAsync(fileID);
        }

        private static void ModifyGameObject(GameObject capsule)
        {
            var meshRenderer = capsule.GetComponent<MeshRenderer>();
            meshRenderer.sharedMaterial = new Material(Shader.Find("Unlit/Color"));
            meshRenderer.sharedMaterial.color = Color.blue;

            var meshFilter = capsule.GetComponent<MeshFilter>();
```

```
            meshFilter.sharedMesh = meshFilter.mesh;
            meshFilter.sharedMesh.vertices = meshFilter.sharedMesh.vertices.
                Zip(meshFilter.sharedMesh.normals, (v,n) => (v, n)).
                Select(vn => vn.v + vn.n).ToArray();
        }
    }
}
```

Instantiate asset

```
using UnityEngine;
namespace Battlehub.Storage.Samples
{
    public class InstantiateAssetExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;

            string projectPath = $"{Application.persistentDataPath}/Example Project";
            await m_assetDatabase.LoadProjectAsync(projectPath);

            var rootID = m_assetDatabase.RootID;
            var fileID = m_assetDatabase.GetUniqueFileID(rootID, $"Cube");

            // Create GameObject
            var cube = GameObject.CreatePrimitive(PrimitiveType.Cube);

            // Create Asset
            await m_assetDatabase.CreateAssetAsync(cube, fileID);

            // Create 100 Asset Instances
            for (int i = 0; i < 100; ++i)
            {
                GameObject instance =
                    await m_assetDatabase.InstantiateAssetAsync<GameObject>(fileID);
                instance.transform.position = Random.onUnitSphere * 10;
                instance.transform.rotation = Random.rotation;
            }
        }
    }
}
```



Detach asset instance

```
// Detaching an instance means breaking the links between the instance and the
// corresponding asset. Once you detach an instance, you will no longer be able
// to apply changes to the underlying asset or create an asset variant from that
// specific instance.

// the "completely" parameter set to true means that all child instances attached
// to this instance as child transforms will also be detached. Otherwise, only the
// instance passed as a parameter  to the DetachAsync method will be detached.

await m_assetDatabase.DetachAsync(instance, completely: true)
```

Unload asset

```
// Unloads asset, optionally destroying attached instances (destroy: true)
await UnloadAssetAsync(assetID, destroy: true);

// Unloads all assets, optionally destroying attached instances (destroy: true)
await UnloadAllAssetsAsync(destroy: true);
```

Create asset variant

```
using UnityEngine;
namespace Battlehub.Storage.Samples
{
    public class CreateAssetVariantExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;

            string projectPath = $"{Application.persistentDataPath}/Example Project";
            await m_assetDatabase.LoadProjectAsync(projectPath);

            var rootID = m_assetDatabase.RootID;
            string key = "Hellephant";
            string loaderID = nameof(ResourcesLoader);

            IExternalAssetLoader loader = new ResourcesLoader();
            await m_assetDatabase.RegisterExternalAssetLoaderAsync(loaderID, loader);

            var fileID = m_assetDatabase.GetUniqueFileID(rootID, $"{key}");
            var variantFileID = m_assetDatabase.GetUniqueFileID(rootID, $"{key} Variant");

            // Import external asset
            await m_assetDatabase.ImportExternalAssetAsync(key, loaderID, fileID);

            // Instantiate external asset
            var hellephantVar =
                await m_assetDatabase.InstantiateAssetAsync<GameObject>(fileID);

            // Modify its materials
            var renderer = hellephantVar.GetComponentInChildren<SkinnedMeshRenderer>();
            ModifyMaterials(renderer);

            // Mark the rendering component as "dirty".
            // This will let the CreateAssetAsync method know that this component
            // has changed and should be stored in the data file, thus creating a variant
            // of the asset that differs from the base only in that component
            await m_assetDatabase.SetDirtyAsync(renderer);
            await m_assetDatabase.CreateAssetAsync(hellephantVar, variantFileID);

            // Instantiate base asset
            var hellephant =
```

```
                await m_assetDatabase.InstantiateAssetAsync<GameObject>(fileID);
            hellephant.transform.position = Vector3.right * 3;

            // Instantiate asset variant
            var hellephantVariant2 =
                await m_assetDatabase.InstantiateAssetAsync<GameObject>(variantFileID);
            hellephantVariant2.transform.position = Vector3.left * 3;
        }

        private static void ModifyMaterials(SkinnedMeshRenderer renderer)
        {
            var materials = renderer.materials;
            materials[0].SetColor("_EmissionColor", Color.green);
            materials[1].SetColor("_EmissionColor", Color.red);
            renderer.sharedMaterials = materials;
        }
    }
}
```

Modify instance and apply changes

```
using Battlehub.Storage.EditorAttributes;
using UnityEngine;
using UnityEngine.UI;

namespace Battlehub.Storage.Samples
{
    public class ModifyInstanceAndApplyChangesExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;
        private IThumbnailUtil m_thumbnailUtil;

        [Layer]
        public LayerMask ThumbnailLayer;

        [SerializeField]
        private RawImage m_thumbnailImage;

        private void Awake()
        {
            var thumbnailUtil = gameObject.AddComponent<ThumbnailUtil>();

            // rotate thumbnail camera
            thumbnailUtil.transform.LookAt(-Vector3.one);
            thumbnailUtil.ThumbnailLayer = ThumbnailLayer;

            m_thumbnailUtil = thumbnailUtil;
        }

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;

            string projectPath = $"{Application.persistentDataPath}/Example Project";
            await m_assetDatabase.LoadProjectAsync(projectPath);
            var rootID = m_assetDatabase.RootID;

            GameObject capsule = GameObject.CreatePrimitive(PrimitiveType.Capsule);
            var rend = capsule.GetComponent<MeshRenderer>();
            rend.material.color = Color.red;

            // Create an asset (the capsule becomes an instance attached to the asset)
            var fileID = m_assetDatabase.GetUniqueFileID(rootID, $"Capsule");
            await m_assetDatabase.CreateAssetAsync(capsule, fileID);
```

```
            // Modify instance transform
            capsule.transform.Rotate(45, 0, 45);
            await m_assetDatabase.SetDirtyAsync(capsule.transform);

            // Modify instance renderer
            rend.sharedMaterial.color = Color.blue;
            await m_assetDatabase.SetDirtyAsync(rend);

            // Apply the changes to the asset and save it.
            // This method also updates the thumbnails.
            var ctx = new ThumbnailCreatorContext(m_thumbnailUtil);
            await m_assetDatabase.ApplyChangesAndSaveAsync(capsule, ctx);

            if (m_thumbnailImage != null)
            {
                // Load thumbnail data
                await m_assetDatabase.LoadThumbnailAsync(fileID);
                var thumbnailBytes = m_assetDatabase.GetThumbnail(fileID);

                // Load thumbnail texture
                var texture = new Texture2D(1, 1);
                texture.LoadImage(thumbnailBytes);

                m_thumbnailImage.gameObject.SetActive(true);
                m_thumbnailImage.texture = texture;
            }
        }
    }
}
```

Modify instance and apply changes to base asset

```
using UnityEngine;

namespace Battlehub.Storage.Samples
{
    public class ModifyInstanceAndApplyChangesToBaseExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;

            string projectPath = $"{Application.persistentDataPath}/Example Project";
            await m_assetDatabase.LoadProjectAsync(projectPath);
            var rootID = m_assetDatabase.RootID;

            GameObject capsule = GameObject.CreatePrimitive(PrimitiveType.Capsule);
            var rend = capsule.GetComponent<MeshRenderer>();
            rend.material.color = Color.red;

            var fileID = m_assetDatabase.GetUniqueFileID(rootID, $"Capsule");
            var fileVariantID = m_assetDatabase.GetUniqueFileID(rootID, $"Capsule
Variant");
            var materialFileID = m_assetDatabase.GetUniqueFileID(rootID, "Material");

            // Create an asset (the capsule becomes an instance attached to the asset)
            await m_assetDatabase.CreateAssetAsync(capsule, fileID);

            // Modify variant instance transform
            capsule.transform.Rotate(45, 0, 45);
            await m_assetDatabase.SetDirtyAsync(capsule.transform);

            // Modify variant instance renderer
            rend.sharedMaterial = Instantiate(rend.sharedMaterial);
            rend.sharedMaterial.color = Color.green;
            await m_assetDatabase.SetDirtyAsync(rend);

            // Create material asset
            await m_assetDatabase.CreateAssetAsync(rend.sharedMaterial, materialFileID);

            // Create variant of the asset
            await m_assetDatabase.CreateAssetAsync(capsule, fileVariantID);

            GameObject instance = await m_assetDatabase.InstantiateAssetAsync<GameObject>
```

```
(fileID);
            instance.transform.position = Vector3.right * 2;

            // Mark the base asset instance's transformation as dirty
            // to prevent ApplyChangesToBase from overriding it.
            await m_assetDatabase.SetDirtyAsync(instance.transform);

            if (m_assetDatabase.CanApplyChangesToBaseAndSaveAsync(capsule))
            {
                // Propagate the changes to the base asset and save it.
                await m_assetDatabase.ApplyChangesToBaseAndSaveAsync(capsule);
            }
        }
    }
}
```

Revert changes to base

```
using UnityEngine;

namespace Battlehub.Storage.Samples
{
    public class ModifyInstanceAndRevertChangesToBaseExample : MonoBehaviour
    {
        private IAssetDatabase m_assetDatabase;

        private async void Start()
        {
            m_assetDatabase = RuntimeAssetDatabase.Instance;

            string projectPath = $"{Application.persistentDataPath}/Example Project";
            await m_assetDatabase.LoadProjectAsync(projectPath);
            var rootID = m_assetDatabase.RootID;

            GameObject capsule = GameObject.CreatePrimitive(PrimitiveType.Capsule);
            var rend = capsule.GetComponent<MeshRenderer>();
            rend.material.color = Color.red;

            var fileID = m_assetDatabase.GetUniqueFileID(rootID, $"Capsule");
            var fileVariantID = m_assetDatabase.GetUniqueFileID(rootID, $"Capsule
Variant");
            var materialFileID = m_assetDatabase.GetUniqueFileID(rootID, "Material");

            // Create an asset (the capsule becomes an instance attached to the asset)
            await m_assetDatabase.CreateAssetAsync(capsule, fileID);

            // Modify variant instance transform
            capsule.transform.Rotate(45, 0, 45);
            await m_assetDatabase.SetDirtyAsync(capsule.transform);

            // Modify variant instance renderer
            rend.sharedMaterial = Instantiate(rend.sharedMaterial);
            rend.sharedMaterial.color = Color.green;
            await m_assetDatabase.SetDirtyAsync(rend);

            // Create material asset
            await m_assetDatabase.CreateAssetAsync(rend.sharedMaterial, materialFileID);

            // Create variant of the asset
            await m_assetDatabase.CreateAssetAsync(capsule, fileVariantID);

            GameObject instance = await m_assetDatabase.InstantiateAssetAsync<GameObject>
```

```
(fileID);
            instance.transform.position = Vector3.right * 2;

            // Mark the base asset instance's transformation as dirty
            //to prevent ApplyChangesToBase from overriding it.
            await m_assetDatabase.SetDirtyAsync(instance.transform);

            if (m_assetDatabase.CanRevertChangesToBaseAndSaveAsync(capsule))
            {
                // Propagate the changes from the base asset and save dependent variants
                await m_assetDatabase.RevertChangesToBaseAndSaveAsync(capsule);
            }
        }
    }
}
```

Web Storage Sample

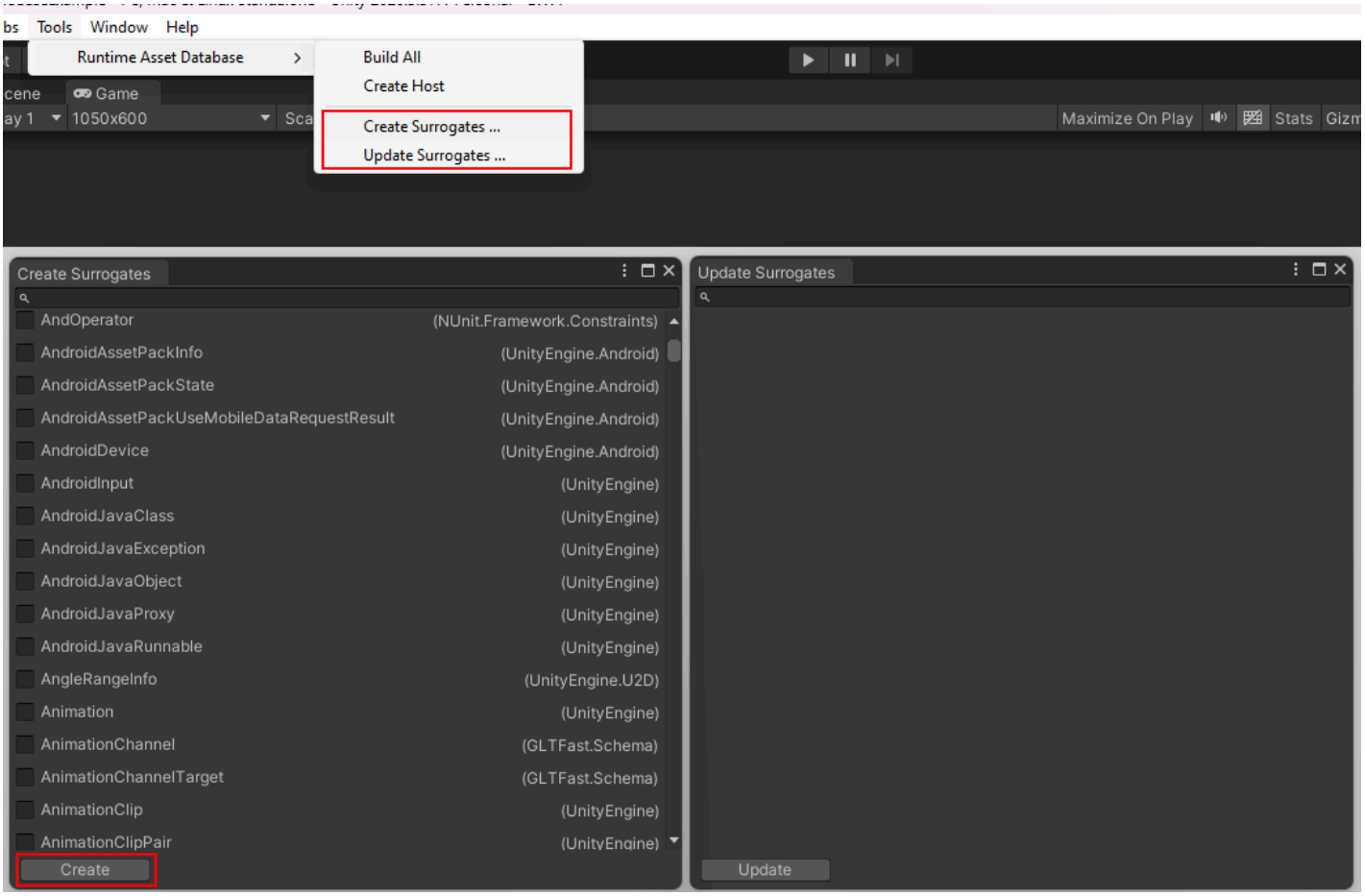
You can use the runtime asset database in conjunction with an HTTP web server, in WebGL and Standalone builds.

To start the web sample, follow these steps:

1. Install the Newtonsoft Json package. In Package Manager, click on "+ Add package by name" and enter "com.unity.nuget.newtonsoft-json"
2. Unpack the `Asset/Battlehub/Storage.Web` Unity package.
3. Open the `Asset/Battlehub.Extensions/Storage.Web/WebProjectBrowser` Unity scene.
4. Extract `Asset/Battlehub.Extensions/Storage.Web/SampleHttpServer.zip` to a folder (e.g., `C:\SampleHttpWebServer`).
5. Install Node.js from Node.js.
6. Open a terminal and navigate to `C:\SampleHttpWebServer`.
7. Run the command `npm install`.
8. Run the command `node app.js`.
9. Enter play mode in Unity.

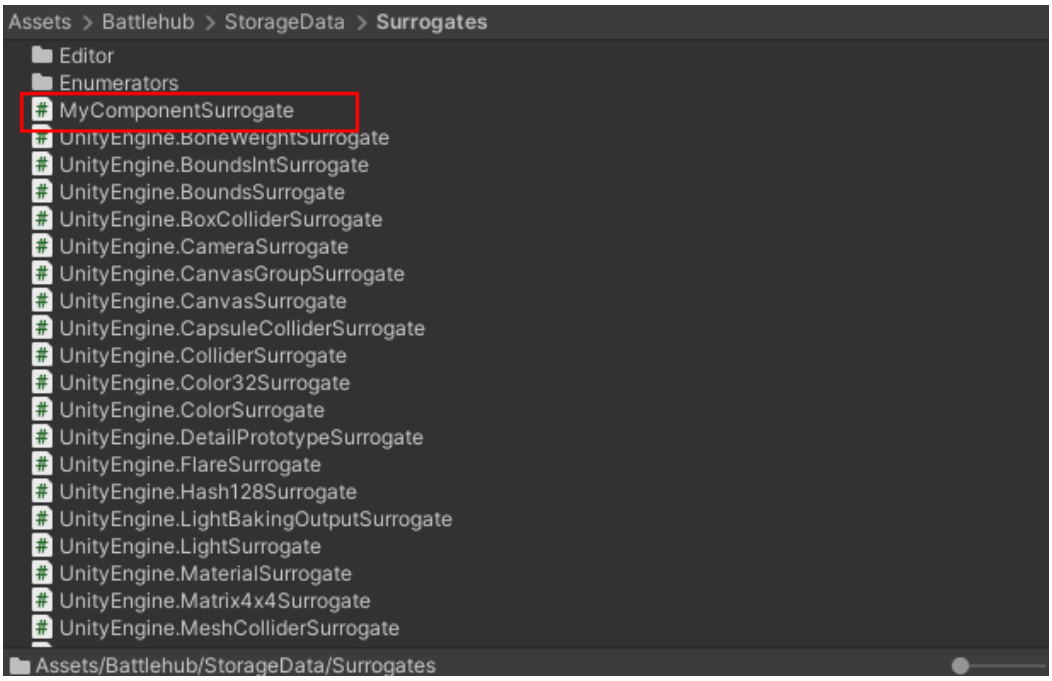
Surrogates

Surrogates are intermediary classes used by the Serializer to facilitate the reading and writing of data to Unity objects during serialization. **To enable the serialization of a specific class, you must create a surrogate for it**. These surrogates can be generated automatically or created from scratch. To generate a Surrogate class, you can use the "Create Surrogates" window.



Sample component:

```
using UnityEngine;

public class MyComponent : MonoBehaviour
{
    public Material Material;

    public GameObject Target;

    public int IntValue;

    public string StringValue;
}
```


Assets > Battlehub > StorageData > Surrogates
Editor
Enumerators
MyComponentSurrogate
UnityEngine.BoneWeightSurrogate
UnityEngine.BoundsIntSurrogate
UnityEngine.BoundsSurrogate
UnityEngine.BoxColliderSurrogate
UnityEngine.CameraSurrogate
UnityEngine.CanvasGroupSurrogate
UnityEngine.CanvasSurrogate
UnityEngine.CapsuleColliderSurrogate
UnityEngine.ColliderSurrogate
UnityEngine.Color32Surrogate
UnityEngine.ColorSurrogate
UnityEngine.DetailPrototypeSurrogate
UnityEngine.FlareSurrogate
UnityEngine.Hash128Surrogate
UnityEngine.LightBakingOutputSurrogate
UnityEngine.LightSurrogate
UnityEngine.MaterialSurrogate
UnityEngine.Matrix4x4Surrogate
UnityEngine.MeshColliderSurrogate
Assets/Battlehub/StorageData/Surrogates

```
using MessagePack;
using ProtoBuf;
using System;
using System.Threading.Tasks;

namespace Battlehub.Storage.Surrogates
{
    [ProtoContract]
    [MessagePackObject]
    [Surrogate(typeof(global::MyComponent), _PROPERTY_INDEX, _TYPE_INDEX)]
    public class MyComponentSurrogate<TID> : ISurrogate<TID> where TID : IEquatable<TID>
    {
        const int _PROPERTY_INDEX = 7;
        const int _TYPE_INDEX = 153;
        //_PLACEHOLDER_FOR_EXTENSIONS_DO_NOT_DELETE_OR_CHANGE_THIS_LINE_PLEASE

        [ProtoMember(2), Key(2)]
        public TID id { get; set; }

        [ProtoMember(3), Key(3)]
        public TID gameObjectId { get; set; }

        [ProtoMember(4), Key(4)]
        public global::System.Boolean enabled { get; set; }

        [ProtoMember(5), Key(5)]
        public TID Material { get; set; }

        [ProtoMember(6), Key(6)]
        public TID Target { get; set; }

        [ProtoMember(7), Key(7)]
        public global::System.Int32 IntValue { get; set; }
        //_PLACEHOLDER_FOR_NEW_PROPERTIES_DO_NOT_DELETE_OR_CHANGE_THIS_LINE_PLEASE

        public ValueTask Serialize(object obj, ISerializationContext<TID> ctx)
        {
            var idmap = ctx.IDMap;

            var o = (global::MyComponent)obj;
            id = idmap.GetOrCreateID(o);
            gameObjectId = idmap.GetOrCreateID(o.gameObject);
            enabled = o.enabled;
            Material = idmap.GetOrCreateID(o.Material);
            Target = idmap.GetOrCreateID(o.Target);
            IntValue = o.IntValue;
```

```
//_PLACEHOLDER_FOR_SERIALIZE_METHOD_BODY_DO_NOT_DELETE_OR_CHANGE_THIS_LINE_PLEASE

            return default;
        }

        public ValueTask<object> Deserialize(ISerializationContext<TID> ctx)
        {
            var idmap = ctx.IDMap;

            var o = idmap.GetComponent<global::MyComponent, TID>(id, gameObjectId);
            o.enabled = enabled;
            o.Material = idmap.GetObject<global::UnityEngine.Material>(Material);
            o.Target = idmap.GetObject<global::UnityEngine.GameObject>(Target);
            o.IntValue = IntValue;

//_PLACEHOLDER_FOR_DESERIALIZE_METHOD_BODY_DO_NOT_DELETE_OR_CHANGE_THIS_LINE_PLEASE

            return new ValueTask<object>(o);
        }
    }
}
```

After successfully generating surrogates, you have the flexibility to make various customizations within your surrogate class. You can remove properties that you don't want to be serialized, add new properties, or perform other operations as needed.

To prevent changes you make to surrogates from being tracked and displayed in the "Update Surrogates" window, you can set the enableUpdates attribute to false using the following syntax:

```
[Surrogate(typeof(global::MyComponent), _PROPERTY_INDEX, _TYPE_INDEX, enableUpdates:
false)]
```

For value types, make sure to set enabled to false like this:

```
[Surrogate(typeof(global::MyComponent), _PROPERTY_INDEX, _TYPE_INDEX, enabled: false)]
```

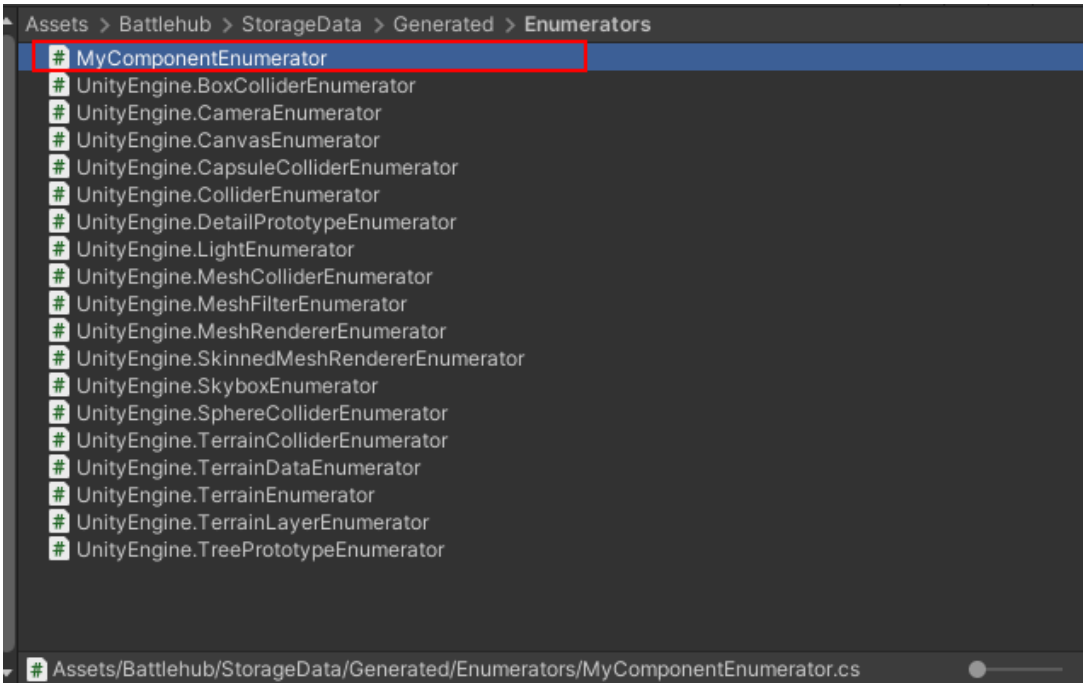
Two constants, **int _PROPERTY_INDEX** and **int _TYPE_INDEX**, have following purpose:

- int _PROPERTY_INDEX: This constant assists the surrogate updater in determining the index of the next property to be generated.
- int _TYPE_INDEX: This constant acts as a unique type index.

Please note that references to other types with surrogates are replaced with their identifier (TID). You can use an "idmap" to generate unique IDs for objects and retrieve objects using their corresponding IDs.

Enumerators

The enumerator is created along with the surrogate. Enumerators are specialized classes used to retrieve Unity object dependencies in a structured manner. These enumerators enable the serialization of an entire object tree, ensuring that dependencies are deserialized before dependent objects during the deserialization process.



```
namespace Battlehub.Storage.Enumerators
{
    [ObjectEnumerator(typeof(global::MyComponent))]
    public class MyComponentEnumerator : ObjectEnumerator<global::MyComponent>
    {
        public override bool MoveNext()
        {
            do
            {
                switch (Index)
                {

                    case 0:
                        if (MoveNext(TypedObject.Material, 5))
                            return true;
                        break;
                    case 1:
                        if (MoveNext(TypedObject.Target, 6))
                            return true;
                        break;
                    case 2:
                        if (MoveNext(Object, -1))
                            return true;
                        break;
                    default:
                        return false;
                }
            }
            while (true);
        }
    }
}
```

Note that the second parameter of the MoveNext method is the property index, which should be equal to the argument of the ProtoMember attribute assigned to that property in the surrogate class.

You can also use the following simplified syntax when editing an enumerator:

```
namespace Battlehub.Storage.Enumerators
{
    [ObjectEnumerator(typeof(global::MyComponent))]
    public class MyComponentEnumerator : ObjectEnumerator<global::MyComponent>
    {
       protected override IEnumerator<(object Object, int Key)> GetNext()
       {
           yield return (TypedObject.Material, 5);
           yield return (TypedObject.Target, 6);
           yield return (TypedObject, -1);
       }
    }
}
```

Dynamic Surrogates (Preview)

It is possible to avoid creating surrogates by hand, instead letting the runtime asset database serialize types using reflection at runtime. This process is roughly equivalent to Unity serialization rules. It works with fields and not with properties.

To enable Dynamic Surrogate for a type, use the following code:

```
var assetDatabase = RuntimeAssetDatabase.Instance;
assetDatabase.RegisterDynamicType(typeof(MyMonoBehaviour));
```

To use field serialization, ensure that the field:

- Is public, or has a `SerializeField` attribute.
- Isn't static.
- Isn't const.
- Isn't readonly.
- Has a field type that can be serialized:
  - Primitive data types (int, float, double, bool, string, etc.)
  - Enum types (32 bits or smaller)
  - Fixed-size buffers
  - Unity built-in types, for example, Vector2, Vector3, Rect, Matrix4x4, Color. *Some types like AnimationCurve will not be serialized.*
  - Custom structs with the `Serializable` attribute
  - References to objects that derive from `UnityEngine.Object`
  - Custom classes with the `Serializable` attribute (see Serialization of custom classes).
  - An array of a field type mentioned above.
  - A `List<T>` of a field type mentioned above.

Serialization of Custom Classes

For Unity to serialize a custom class, ensure the class:

- Has the `Serializable` attribute.
- Isn't static.

The `[SerializeReference]` attribute is not supported and serialization is inline.

Custom Serialization

Sometimes you might want to serialize something that the serializer doesn't support (for example, a C# Dictionary). The best approach is to implement the `ISerializationCallbackReceiver` interface in your class. This allows you to implement callbacks that are invoked at key points during serialization and deserialization:

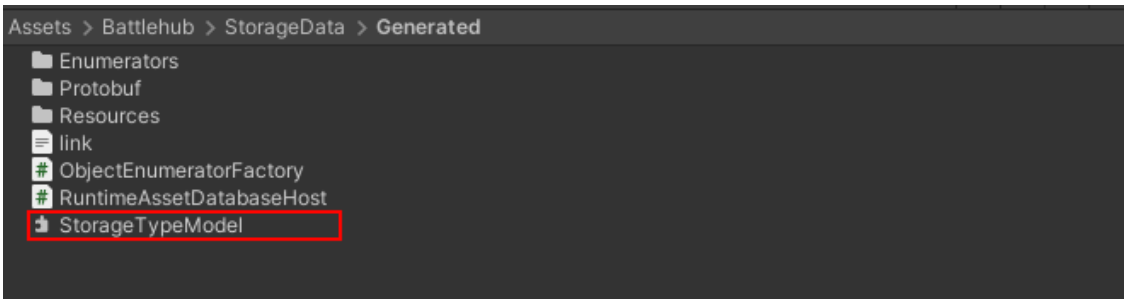
- When an object is about to be serialized, Unity invokes the `OnBeforeSerialize()` callback. Inside this callback is where you can transform your data into something Unity understands. For example, to serialize a C# Dictionary,

copy the data from the Dictionary into an array of keys and an array of values.

- After the `OnBeforeSerialize()` callback is complete, Unity serializes the arrays.
- Later, when the object is deserialized, Unity invokes the `OnAfterDeserialize()` callback. Inside this callback is where you can transform the data back into a form that's convenient for the object in memory. For example, use the key and value arrays to repopulate the C# Dictionary.

Build All

After finishing creating or updating surrogates, make sure to click **"Tools"** > **"Runtime Asset Library"** > **"Build All"** from the main menu. This command will build the type model and serializer.



Support

If you cannot find something in the documentation or have any questions, please feel free to send an email to Battlehub@outlook.com or ask directly in this support group. Keep up the great work in your development journey! 😊