DTT
APPS WEB GAMES

SINCE 2010

DTT Managed Singletons

Documentation | V3.1.0 |27-01-22

Table of Contents

1. Changelog

1. version 1.0.0 - initial release
2. version 1.1.0 - Bug fixes and small additions
    a. Fix
        i. Update **BootModeAttributeWatcher** to not create the profile when it can't find it.
        ii. Add version define for unity test framework to prevent compile errors from occuring for developers that use visual studio code.
    b. Add
        i. Add more warnings when the singleton core asset is not used correctly.
        ii. Expose more of the API from **SingletonValidator**.
3. Version 2.0.0 - Update of Editor Utilities dependency
    a. Update
        i. Updated the Editor Utilities dependency to a more stable version.
    b. Fixed
        i. Fixed issue with script files not being openable in IDE's other than Visual Studio
        ii. Fixed issue with asset database saving sometimes causing a null reference exception.
4. Version 3.0.0 - Bug fixes and minor changes to formatting/spelling
    a. Fixed
        i. Fixed issue with **BootModeAttributeWatcher** where it would sometimes throw an exception when loading the project.
        ii. Fixed an issue with the **SingletonProfileManager** where it would return false values on singletons using the boot mode attribute.

iii. Fixed an issue with the **SingletonProfile** not being updated correctly after usage of the **BootModeType.Disabled** with a **BootMode** attribute.

iv. Fixed an issue with the **BootMode** attribute not being applied when directly starting playmode after updating the singleton script with it.

v. Fixed spelling in some summaries and log messages.

vi. Updated the way the configuration and profile are imported from the asset database to make the asset more stable.

5. Version 3.1.0 - Bug fixes and dependencies.

    a. Updated

        i. Dependencies to runtime and editor utilities.

        ii. Debug import error message.

2. Get started quickly

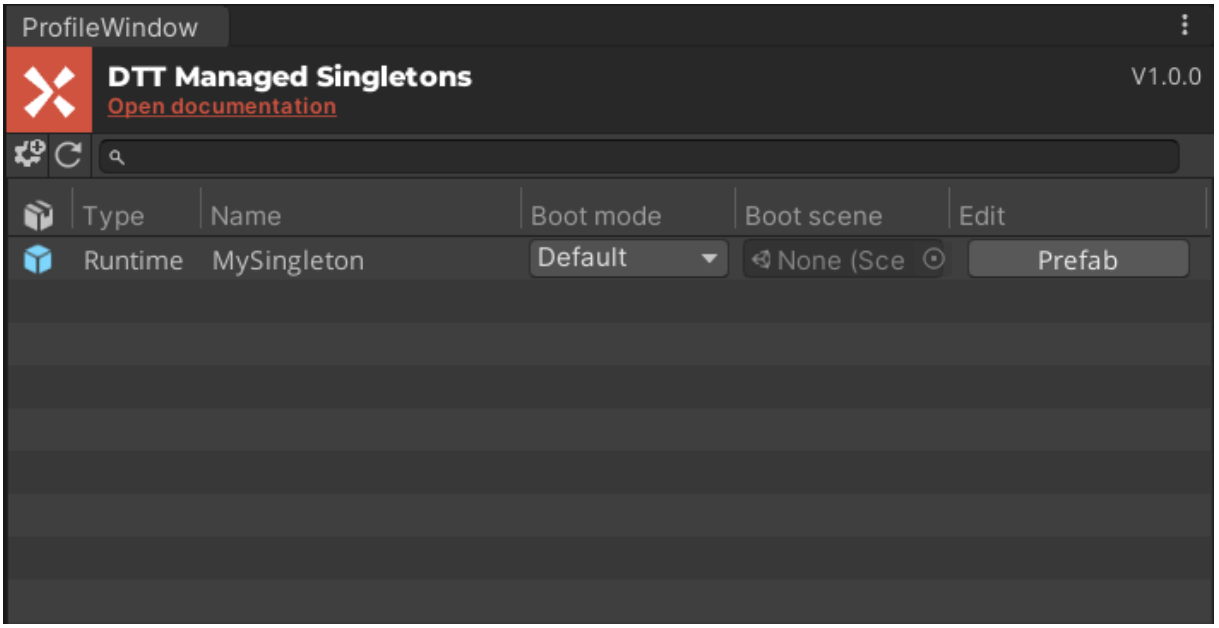
This is a summary to help you get started quickly. More details are provided below.

To get started, create a new script and add the **DTT.Singletons** namespace. Then make sure it inherits from **SingletonBehaviour<T>** using your class as **T**.

```
using DTT.Singletons;

public class MySingleton : SingletonBehaviour<MySingleton>
{

}
```

After this, create a prefab and add the new Singleton. You can do this by creating a new game object in the scene, adding your new singleton component to it and dragging it into the project. It is also possible to use the create menu: Create/Prefab.

Once you have done this, you can open the **SingletonProfile** window by navigating to 'Tools/DTT/SingletonCore/Profile'.



Here you can set your singleton up as needed. Important here is the **Boot mode** column. Here you can set the boot mode type of your singleton. For now, set it to **Default**.

- **Default** mode loads the singleton when runtime begins (before any scene is loaded), so it will always be active and available.

Alright. Your Singleton will be loaded up when you start your game. To use it, create a new script to access the Singleton. Your singleton will have a static **Instance** property. This means it can be accessed using your class name (e.g. **MySingleton.Instance**).

```
private void Start()
{
    MySingleton.Instance.DoSomethingAwesome();
}
```

You now know everything you need to start using the Singleton Core package. For more details on the asset's extensive features, please keep reading.

3. Introduction

3.1 What is a Singleton

The singleton object, as defined by the Singleton design pattern, is an object that has only one instance and is globally available. In comparison to a static class, which is also globally available, a singleton object is not static.

This attribute gives the singleton the benefit of being able to inherit from other classes and implement interfaces unlike the static class.

Its global availability comes from a static property. Convention is to call this property 'Instance', but names like 'Current' or 'Main' are also sometimes used. In **DTT Managed Singletons** the property is called 'Instance.'

Assignment of this Instance property is to be determined by the developer but most singletons use lazy instantiation. This creates the instance when the Instance property is first accessed.

In the Unity Game Engine, a singleton can also be initialized by using the Awake method to assign the Instance property when the singleton game object is created.

If you want to learn more about the Singleton design pattern, you can find more information on **this wikipedia page.**

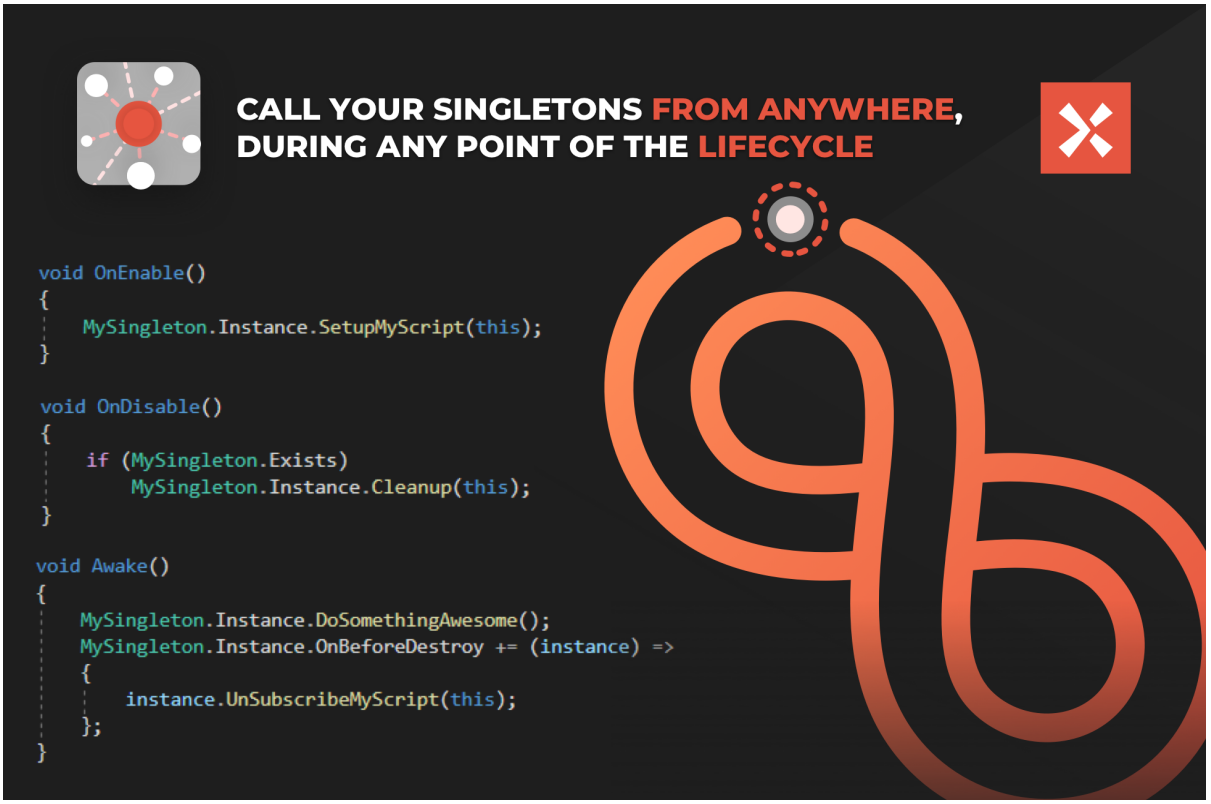
Some useful places in your game where you could use a Singleton include:

- Any 'game management' objects such as your GameManager, SceneManagers, Audiomanagers, etc.
- Objects that need global access, but also need to wait on coroutines, such as API management systems, or save systems.

3.2 Managed Singletons

DTT Managed Singletons is a Singleton code suite designed to eliminate common problems DTT encountered while working with Singletons. The features include, but are not limited to:

- Various types of Singletons, such as: Classic, Lazy (only created when called), Scene (always active in their given scene, never outside of it), and Editor.
- A clear editor window that provides an overview of singletons you have created in the project and allows you to configure the way your singletons are created during the lifetime of your application.
- Flexibility of singleton creation by making them available before the first scene is loaded. This ensures your singletons are created independent of which scene you start in.
- Availability of singletons at any point during the application's lifecycle.



DTT APPS WEB GAMES

4. Set-Up

SingletonBehaviour

The SingletonBehaviour class uses its Awake Method to set its instance and should be attached to a prefab before it can be used. Because it is part of a prefab it can use inspector fields unlike the Lazy version which is instantiated from script and will be discussed later in this document.

```
using DTT.Singletons

public class Singleton : SingletonBehaviour<Singleton>
{
    // This singleton can make use of inspector fields.
    [SerializeField]
    private bool _myBoolean = false;

    protected override void Awake()
    {
        base.Awake();
        // You can use Awake to initialize your singleton.
    }
}
```

The SingletonBehaviour class can be configured to 4 different boot mode types:

- Disabled: The singleton will not be created during the application's lifecycle.
- Default: The singleton will be created before the first scene is loaded.
- Lazy: The singleton will be created when it is needed (e.g. when its Instance is first accessed).
- Scene: The singleton will be created when a specified scene is loaded and destroyed when the specified scene is unloaded.

Using the **profile window** , you can configure your singleton's boot mode. A profile asset stores the singleton prefabs that have been configured for bootstrap (e.g. that are not disabled) and is used by the **SingletonBootstrapService** class to instantiate the singletons based on their boot mode when your game starts.

BootMode attribute

A **BootMode** attribute can be used together with your SingletonBehaviour implementation to set a prefered boot mode through script. This attribute ensures your singleton always uses the defined boot mode type. Its value overrides the value in the profile window and disables its configurability.

```
[BootMode(BootModeType.DEFAULT)]
public class AudioManager : SingletonBehaviour<AudioManager>
{

}
```

Lazy SingletonBehaviour

The **LazySingletonBehaviour** class uses lazy initialization to create your singleton on demand. You need only to implement this class and write out its functionality to make use of it.

Important to note however is that this type of singleton can't use serialized inspector fields. Since it will be added as a component at runtime, inspector field references are not copied but default values are used.

This singleton is the best option when your monobehaviour doesn't require any inspector fields to be set.

```
using DTT.Singletons;

public class LazySingleton : LazySingletonBehaviour<LazySingleton>
{
    // This singleton cannot make use of inspector fields.

    private void Awake()
    {
        // You can use Awake to initialize your lazy singleton.
    }
}
```

Lazy Editor SingletonBehaviour

The **LazyEditorSingletonBehaviour** class can be used outside of playmode. When its **Instance** property is first used, it will create itself in the currently active scene.

Like with the LazySingletonBehaviour, the developer only needs to implement this class and write out its functionality to make use of it. It also can't use inspector fields.

This singleton is your best option if you want a globally accessible object that can operate in the scene outside of play mode.[1]

Lazy Editor Prefab SingletonBehaviour

The **LazyEditorPrefabSingletonBehaviour** class is a variant on the LazyEditorSingletonBehaviour that can use serialized inspector fields. It has to be added to a prefab and stored in the project.

When it is first used (e.g. its **Instance** is first accessed) it will find the prefab in the project and Instantiate it in the scene.[1]

This singleton is your best option if you want a globally accessible object that can operate in the scene and has editable inspector fields.

Warning
If you intend to use editor singletons to start coroutines outside of playmode, we recommend you import Unity's 'EditorCoroutines' package to use together with this singleton as original coroutines don't work outside of playmode even with MonoBehaviours.

[1] Keep in mind when using Editor Singletons that switching scene will destroy them until the next time they are accessed. This means that 'live' data you are using will be lost.

Using your singleton

Your singleton is defined by its type but also its boot mode. Referencing a singleton that has been disabled will cause an exception, but referencing one that is set to lazy will create it the first time.

During the application's lifecycle there is practically only one way to interact with the singleton. This is by using the static Instance property. Make sure to use the instance property correctly during the different stages of your object's lifecycle.

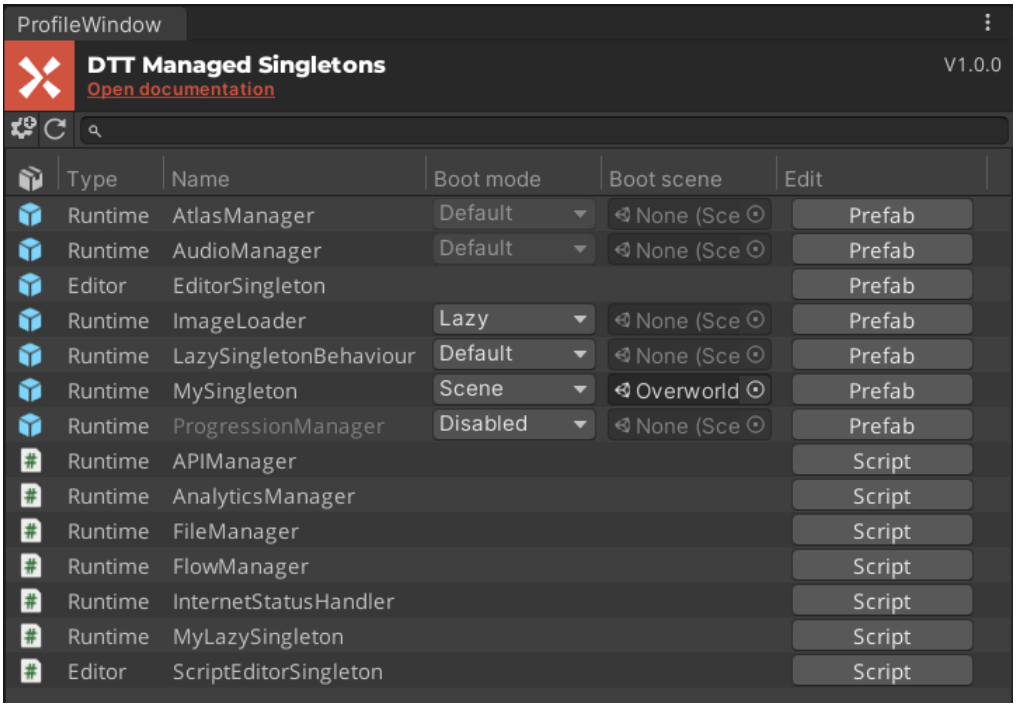
```
using DTT.Singletons;
using UnityEngine;

public class SingletonUser: MonoBehaviour
{
    private void Awake()
    {
        // Your singleton behaviour can be referenced in the Awake method.
        // Make sure it is not disabled because it will throw an exception
        // otherwise.
        MySingleton.Instance.DoSomethingAwesome(this);
    }

    private void Start()
    {
        // Because we can't guarantee the singleton will be destroyed before
        // our script, we can subscribe to an event that returns the singleton
        // instance right before it is destroyed. This means we don't have to
        // use the OnDestroy method which might be called after the singleton
        // is already destroyed.
        MySingleton.Instance.OnBeforeDestroy += (instance) =>
        {
            instance.DoAwesomeUnsubscribe(this);
        };
    }

    private void OnDestroy()
    {
        // If you find your script really needs to interact with the
        // singleton when it is being destroyed or maybe in OnDisable
        // you can check its existance before trying to access it. This
        // prevents a Null Reference Exception when the application is
        // shutting down inside the Editor and the singleton is destroyed
        // before your script can access it.
        if (MySingleton.Exists)
              MySingleton.Instance.AwesomeCleanup(this);
    }
}
```

5. Editor

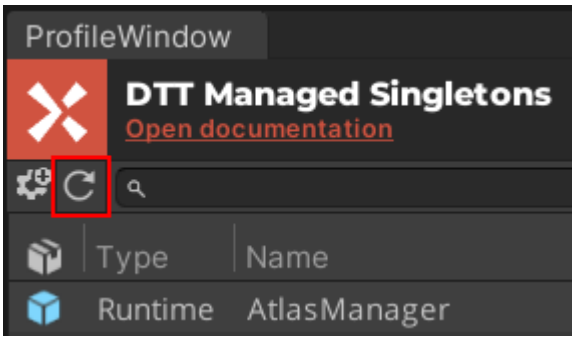
5.1 Singleton Profile Window

The singleton profile window is used to manage the singletons you have created in your project. In the profile your singletons will be displayed in the form of four different types:

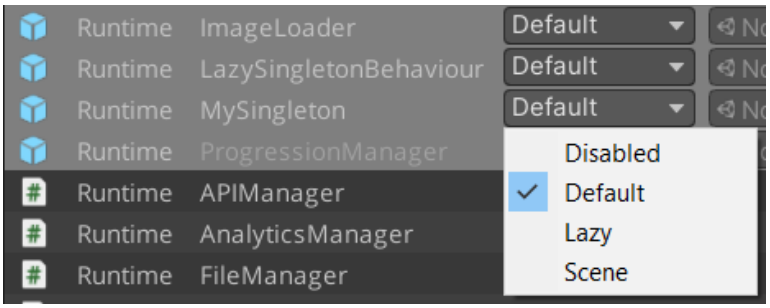
- **Runtime prefab:** Classes that have implemented the SingletonBehaviour<T> class and have added them to a prefab in the project.
- **Editor prefab:** Classes that have implemented the LazyEditorPrefabSingletonBehaviour<T> class and have added them to a prefab in the project.
- **Runtime script:** Classes that have implemented the LazySingletonBehaviour<T> class.
- **Editor script:** Classes that have implemented the LazyEditorSingletonBehaviour<T> class.



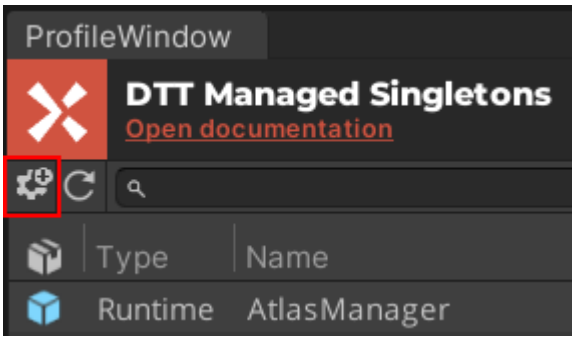
The **refresh button** can be used to re-initialize the window, searching the project for new singleton scripts and/or prefabs. This can be useful when you have just created a new singleton script or prefab and want to view it inside the window.



The boot mode select menu supports **multiselect** so if you want to update multiple singletons at once you can select all to apply your action.

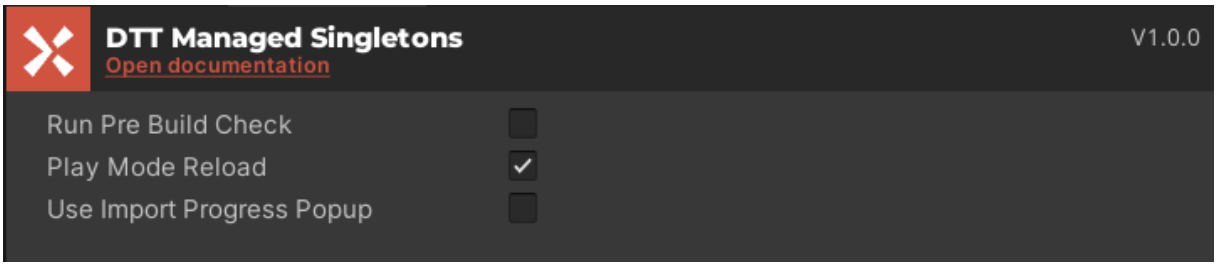


You can click the **settings button** to open the configuration asset which holds editable options for how Managed Singletons should behave.



5.2 Singleton Configuration

Currently, the configuration of Managed Singletons is limited to three flags:



- **RunPreBuildCheck:** This flag defines whether there should be a check on the correct usage of the singleton profile during the build process. Setting this to true might stop the build if the profile isn't used correctly.
    - The rules for the check are defined as follows:
        - If you have a profile you should have singletons in it.
        - If you don't have a profile there should not be any singletons in the project.
- **PlayModeReload:** This flag defines whether the singleton profile window can refresh itself after the play mode assembly reload has finished. The reload is a heavy operation that might delay the play mode start in larger projects. If this is set to false, no info will be shown in the profile during play mode.
- **UseImportProgressPopup:** This flag defines whether to show a progress popup when importing singletons for the profile to display. This can help when the import progress is delayed when having a bigger project.

6. Known Limitations

- Because of the way the singletons are set up using generics, inheriting from a concrete singleton implementation is not possible. So for example, if your **APIManager** singleton class inherits from **SingletonBehaviour<T>**. It won't be possible to create a **SpecialAPIManager** class that inherits from **APIManager**.
- If not used correctly, an implementation of **SingletonBehaviour<T>** could throw a **SingletonInstanceException** when you exit playmode. When playmode ends, **all** scene objects are destroyed, including your singletons. The order in which they are destroyed can't be determined. This is why it can be dangerous to call upon your singleton in an **OnDestroy** or **OnDisable** method without checking its existence first. Make sure to always use the static **Exists** property when you want to call upon your singleton in one of these two methods.

7. Support and feedback

If you have any questions regarding the use of this asset, we are happy to help you out. Always feel free to contact us at:

unity-support@d-tt.nl

(We typically respond within 1-2 business days)

We are actively developing this asset, with many future updates and extensions already planned. We are eager to include feedback from our users in future updates, be they 'quality of life' improvements, new features, bug fixes or anything else that can help you improve your experience with this asset. You can reach us at the email above.

Reviews and ratings are very much appreciated as they help us raise awareness and to improve our assets.

DTT stands for Doing Things Together

DTT is an app, web and game development agency based in the centre of Amsterdam. Established in 2010, DTT has over a decade of experience in mobile, game, and web based technology.

Our game department primarily works in Unity where we put significant emphasis on the development of internal packages, allowing us to efficiently reuse code between projects. To support the Unity community, we are publishing a selection of our internal packages on the Asset Store, including this one.

More information about DTT (including our clients, projects and vacancies) can be found here:

https://www.d-tt.nl/en/