Application Manager

Index

Overview

Application Manager is a **Unity Editor tool** that easily manages different configurations for different **deployment environments** and executes code at application startup.
It makes it possible to change a lot of information between builds **without editing manually the Project Settings**, helping you avoid mistakes or forgetfulness.

It is a very useful tool in combination with **Unity Cloud Build** with its Pre- and post-export methods.

Installation

It is available on the **Unity Asset Store** (**Using the Asset Store**).

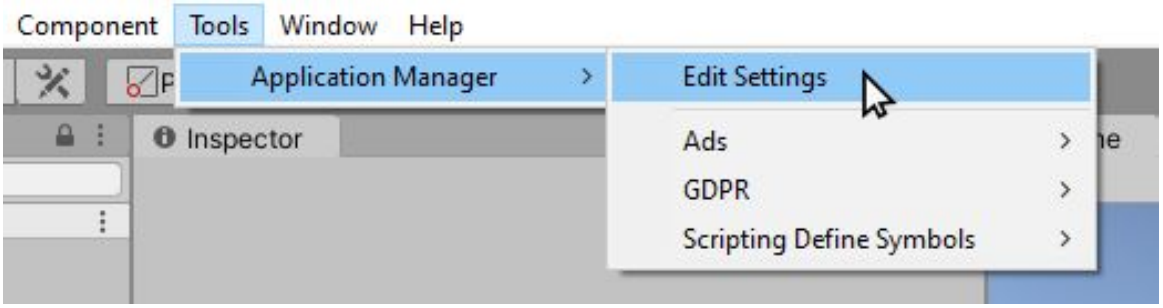
It doesn't require particular installations or configurations, to work it just has to be present in the Asset folder.

How to access scripts

All Application Manager scripts are placed under the **namespace** "**DragonkinStudios.ApplicationManagement**", so if you want to access them you have to add a **using** directive.

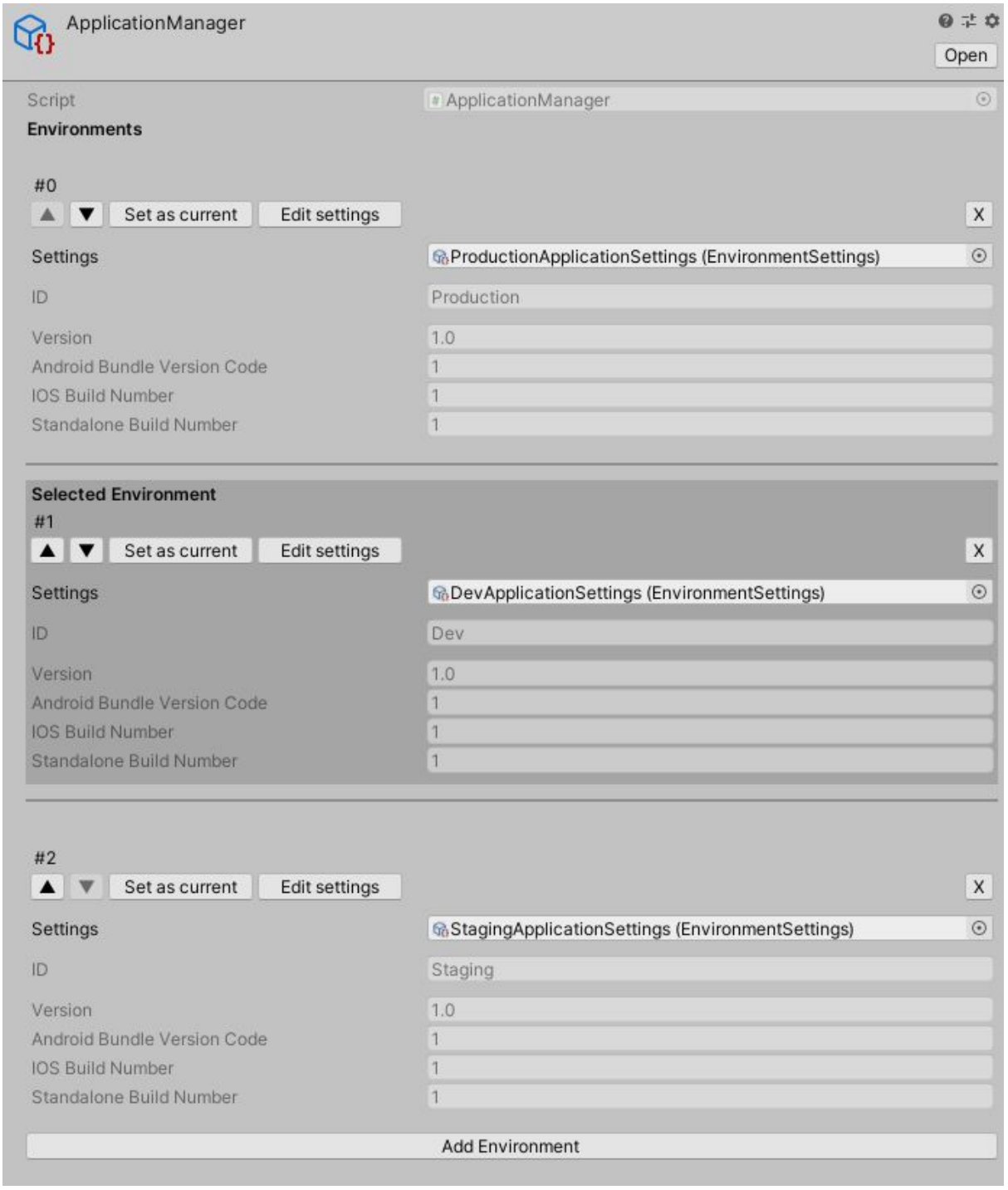
Edit Application Manager Settings

To start editing the settings go to "**Tools/Application Manager/Edit Settings**".
It will open the ApplicationManager settings contained in the Resources folder, if present, otherwise it creates new settings.



Application Manager settings window

Here you can manage all your **environments** adding, removing or editing each configuration.



You can set the current environment in the editor by just clicking on “**Set as current**” button in the list of all the environments.

When you **set an environment as current** the Application Manager will copy all the information within it and set them in the **Player Settings**.

You can also change it via scripts using the property “**ApplicationManager.CurrentEnvironment;**” or calling “**ApplicationManager.SetCurrentEnvironment(“environmentID”);**” and passing as parameter the ID of the environment you want to set.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using DragonkinStudios.ApplicationManagement;

Script Unity | 0 riferimenti
public class MyCustomManager : MonoBehaviour
{
    [SerializeField]
    private string myDevEnvironmentID;

    0 riferimenti
    public void SetMyDevEnvironment()
    {
        ApplicationManager.SetEnvironment(myDevEnvironmentID);
    }
}
```

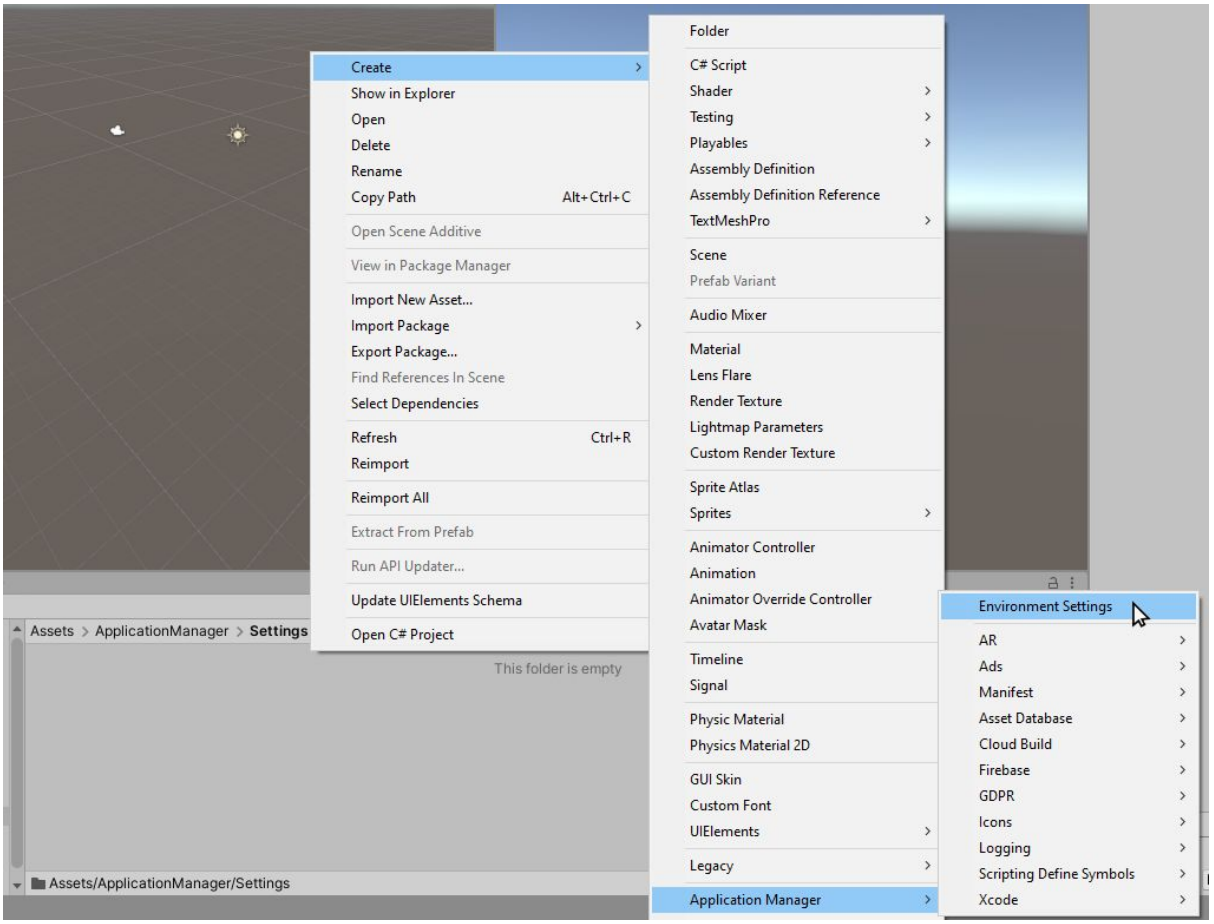
The Application Manager overrides by default the Unity **build process** executed with the "Build" and "Build and Run" buttons in the "Build Settings" window in order to invoke the **BuildPreExport** and **BuildPostExport** methods of Current Environment.

If you want you can disable this behaviour and invoke them manually from your editor scripts calling “**ApplicationManager.SetCurrentEnvironmentPreExportValues();**” and “**ApplicationManager.SetCurrentEnvironmentPostExportValues(“pathToBuiltProject”);**”.

Create new Environment Settings

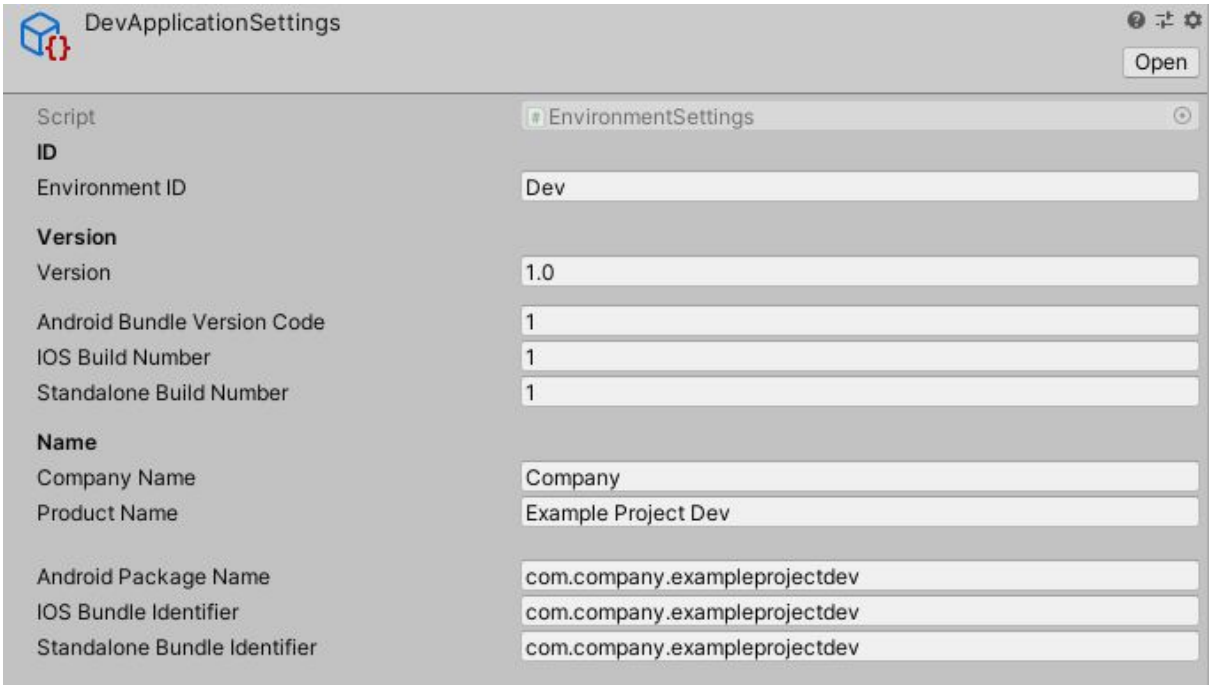
To create new **Environment Settings** go to “**Assets/Create/Application Manager/Environment Settings**”.
Alternatively, you can right-click within the Project window and go to “**Create/Application Manager/Environment Settings**”.

It is recommended to store settings files outside the Resources folder to avoid sensitive or useless data in builds.



Environment Settings

An **Environment Settings** is a **Scriptable Object** containing the basic information of the app and a list of **custom settings** useful to manage information that may vary between the different environments.



ID

Environment ID: the string that identifies the environment. It is used to set the current environment, so it must be unique and different from any other environment ID.

Version

Version: the string containing the version of the app.

Android Bundle Version Code: the bundle version code of the Android app.

IOS Build Number: the build number of the bundle of the iOS app.

Standalone Build Number: the build number of the bundle of the Mac App Store app.

Name

Company Name: the name of your company.

Product Name: the name of your game.

Android Package name: the application ID which identifies your Android app on the device and in the Google Play Store

IOS Bundle Identifier: the Bundle Identifier which identifies your iOS app on the device and in the App Store.

Standalone Bundle Identifier: the Bundle Identifier which identifies your Mac app on the device and in the Mac App Store.

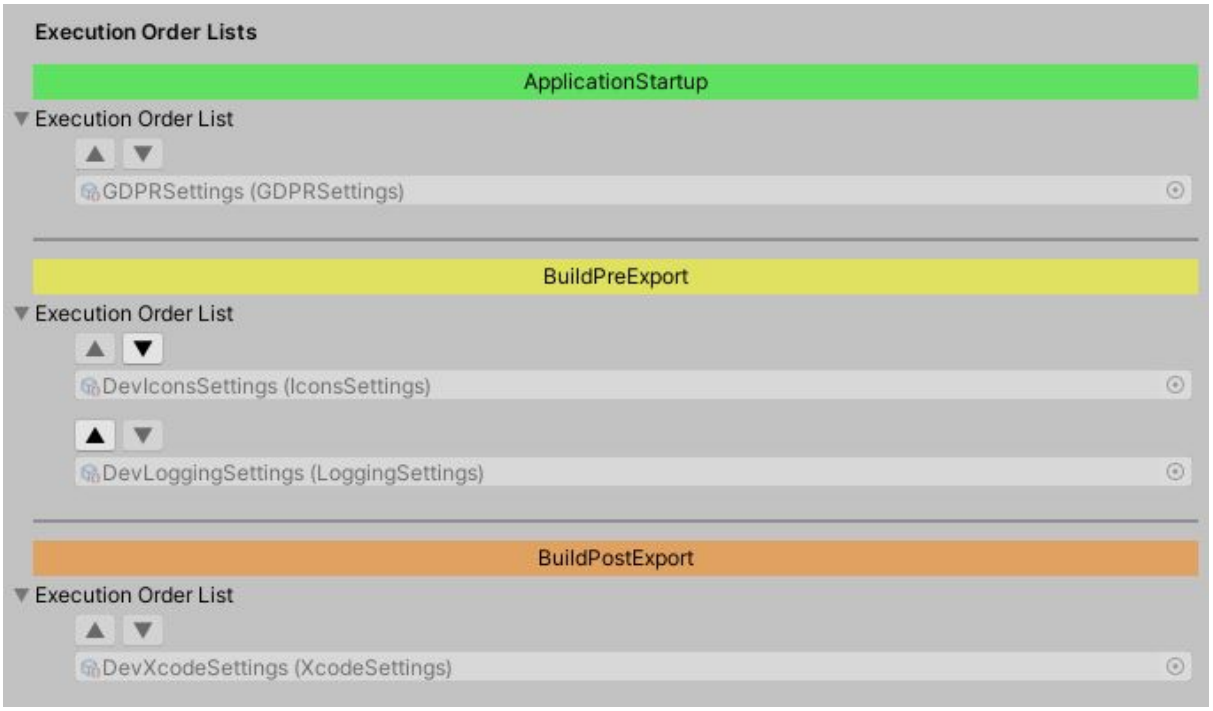
Custom Settings List

The **Custom Settings List** contains a collection of **settings**, identified by a **tag**.

Note: You can insert only one settings of the same type.

Execution Order Lists

The **Execution Order Lists** are used **to order** the execution of the methods of the **custom settings** each divided by **tag**.



Every time you insert or remove elements from the **Custom Settings List**, all these lists are automatically updated.

Settings with multiple tags will appear in multiple lists and have to be ordered individually for each tag.

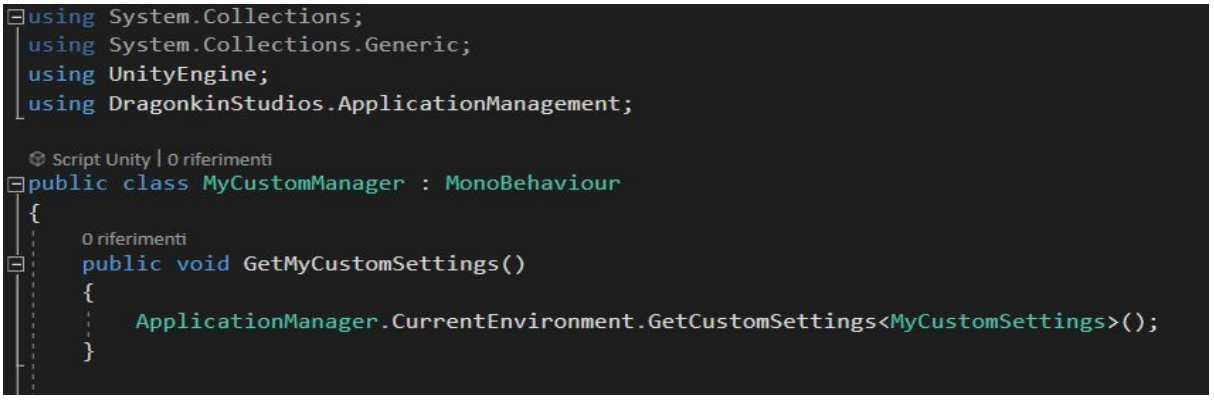
Custom settings

Custom settings are **Scriptable Objects** containing additional information for your environment settings, for example GDPR settings, Cloud Build settings, or other settings you decide to create (see **examples** if you want to create your own custom settings).

Access Custom Settings inside your code

In order to access a Custom Settings of type T from any **Environment Settings** call "**GetCustomSettings<T>();**".

Instead, if you want to retrieve the element of type T from the current environment you can use "**ApplicationManager.CurrentEnvironment.GetCustomSettings<T>();**".



Tags

The custom settings can have one or more tags. Under the hood tags are merely interfaces that a custom settings class implements containing definitions for a group of related functionalities.

In this package, there are three different **tags**: **ApplicationStartup**, **BuildPreExport** and **BuildPreExport**.

Each tag ensures a different behavior for the settings.

Note: Tags are not required for custom settings, so for example, settings that are only data containers may not have any tag.

ApplicationStartup

This tag is assigned when the Scriptable Object implements the **IApplicationStartup** interface with the declared method "**ExecuteStartupAction(System.Action callback);**".

```
Script Unity | 0 riferimenti
public class MyApplicationStartup : ScriptableObject, IApplicationStartup
{
    8 riferimenti
    public void ExecuteStartupAction(Action callback)
    {
        // my startup code
    }
}
```

All these methods are invoked at the app startup by the **Application Starting Manager** only if their own custom settings are present on **Custom Settings List**.

It may be used to check permissions, GDPR acceptance, initialize SDKs and other managers.

The "**callback**" parameter of the method **must be invoked** when the action is ended and the **Application Starting Manager** can invoke the next settings method.

In the example below, the "**callback**" is invoked if the user has already accepted the GDPR policies or when the "accept" button in the instanced UI canvas is clicked.

```
7 riferimenti
public void ExecuteStartupAction(Action callback)
{
    Debug.Log("Check GDPR Acceptance");

    if (CheckValidExistingGDPRAcceptance())
    {
        callback();
    }
    else
    {
        ShowGDPRCanvas(callback);
    }
}
```

BuildPreExport

This tag is assigned when the Scriptable Object implements the **IPreExportSettings** interface with the declared method “**SetPreExportValues();**”.

It is an **Editor** only method, so you should implement it inside a “#if UNITY_EDITOR” block or in a script located under an Editor folder.

```
Script Unity | 0 riferimenti
public class MyPreExportSettings : ScriptableObject, IPreExportSettings
{

#if UNITY_EDITOR
    10 riferimenti
    public void SetPreExportValues()
    {
        // my pre export code
    }
#endif

}
```

All these methods are invoked by Application Manager every time the current environment is changed and before any build process is started but only if their own custom settings are present on **Custom Settings List**.

It may be used to edit Project Settings or any other settings values, delete or override Assets files and modify everything you want in the project before the build process is started.

In the example below, it is used to change the icons of the application.

```
9 riferimenti
public void SetPreExportValues()
{
    Debug.Log("Set application icons");

    // set Android icons

    BuildTargetGroup buildTargetGroup = BuildTargetGroup.Android;
    PlatformIconKind[] supportedIconKinds = PlayerSettings.GetSupportedIconKindsForPlatform(buildTargetGroup);

    SetIcons(buildTargetGroup, supportedIconKinds[0], AndroidAdaptiveBackgroundIcon, AndroidAdaptiveForegroundIcon); // adaptive
    SetIcons(buildTargetGroup, supportedIconKinds[1], AndroidRoundIcon); // round
    SetIcons(buildTargetGroup, supportedIconKinds[2], AndroidLegacyIcon); // legacy

    // set iOS icons

    buildTargetGroup = BuildTargetGroup.iOS;
    supportedIconKinds = PlayerSettings.GetSupportedIconKindsForPlatform(buildTargetGroup);
```

Note: The Application Manager overrides the default build process executed with the "Build" and "Build and Run" buttons in the "Build Settings" window in order to invoke all the PreExport methods before any build process.

BuildPostExport

It implements the **IPostExportSettings** interface with the declared method "**SetPostExportValues(string pathToBuiltProject);**".

It is an **Editor** only method, so you should implement it inside a "#if UNITY_EDITOR" block or in a script located under an Editor folder.

```
Script Unity | 0 riferimenti
public class MyPostExportSettings : ScriptableObject, IPostExportSettings
{

#if UNITY_EDITOR
    3 riferimenti
    public void SetPostExportValues(string pathToBuiltProject)
    {
        // my post export code
    }
#endif

}
```

All these methods are invoked by Application Manager after any build process is ended but only if their own custom settings are present on **Custom Settings List**.

It may be used to manipulate the project files after the project is built.

In the example below, it is used to edit the Xcode Settings after an iOS build.

```
        2 riferimenti
        public void SetPostExportValues(string pathToBuiltProject)
        {
#if UNITY_IOS
        SetXcodeValues(pathToBuiltProject);
#endif
        }

        0 riferimenti
        private void SetXcodeValues(string pathToBuiltProject)
        {
            string projPath = PBXProject.GetPBXProjectPath(pathToBuiltProject);
            PBXProject pbxProject = new PBXProject();
            pbxProject.ReadFromFile(projPath);
            string targetName = "Unity-iPhone";
            string plistPath = Path.Combine(pathToBuiltProject, "Info.plist");
```

Custom settings examples

A collection of built-in examples of the custom settings provided with the package.

AdsSettings (ApplicationStartup)

It allows you to check if users have already given **consent** to show **personalized ads**, otherwise it shows a UI canvas[1] to allow them to choose it.

Note: It is not related to any Ads SDK, so you need to replace some lines with the API of the chosen SDK.

AndroidManifestSettings (BuildPreExport)

It allows you to **edit** the **AndroidManifest.xml** file inside the Assets folder before an Android build.

You can replace or remove some strings and add new XML elements.

ARSettings (ApplicationStartup)

It shows at startup a UI canvas[2] with the **safety warning** for **Augmented Reality** applications.

AssetDatabaseSettings (BuildPreExport)

It allows you to **clone files and directories** inside the Assets folder and overwrite the destination.

CloudBuildSettings (BuildPostExport)

It allows you to easily interface your app with the **Unity Cloud Build** service.

You can use **pre- and post-export methods of Unity Cloud Build** to automatically change the environment of the build.

You can also add files necessary only to the Cloud Build process, such as specific plugins or libraries.

FacebookSettings (ApplicationStartup, BuildPreExport)

It allows you to initialize the **Facebook SDK** at startup.

---

[1] You have to provide your own canvas prefab in the settings attached with script *AdsConsentCanvas.cs*.
[2] You have to provide your own canvas prefab in the settings attached with script *ApplicationStartingCanvas.cs*.

You can use it also to edit the **App Index** in the Facebook settings. In this way you can use different Facebook apps inside your Unity project.

Note: You need to replace some lines with the API of Facebook SDK for the initialization and the settings changes.

FirebaseSettings (ApplicationStartup, BuildPreExport)

It allows you to initialize the **Firebase SDK** at startup.

You can use it also to replace the **Google Service Files** in the Assets folder with the correct one. In this way you can use different Firebase projects inside your Unity project.

Note: You need to replace some lines with the API of the Firebase SDK for the initialization.

GDPRSettings (ApplicationStartup)

It allows you to check if users have already accepted the **GDPR policies**, otherwise it shows a UI canvas[3] with policies URLs and “accept” button.

URLs are compatible with multilanguage.
You can set URLs for **Terms and Conditions**, **Privacy Policy** and **Cookie Policy**.
It contains a **GDPR version data** that must be updated when your policies are subjected to some changes. In this way the UI canvas for the acceptance will be shown again.

IconsSettings (BuildPreExport)

It allows you to set the **icons** of the app.

You can set the icons for **Android** (legacy, round and adaptive), **iOS** and **Standalone** platforms.

LoggingSettings (BuildPreExport)

It allows you to edit the base logging settings of the app.

You can choose if **enable logs** and which **type of message** should be logged.

PrefabSpawnerSettings (ApplicationStartup)

It allows you to **spawn prefabs** at app **startup**.

You can use it to spawn other managers or game elements.

---

[3]You have to provide your own canvas prefab in the settings attached with script “GDPRCanvas”.

ScriptingDefineSymbolsSettings (BuildPreExport)

It allows you to enter the names of the **symbols** you want to define for **Android**, **iOS** and **Standalone** platforms, separated by semicolons.

You can use these symbols as the conditions for #if directives, just like the built-in ones.

XcodeSettings (BuildPostExport)

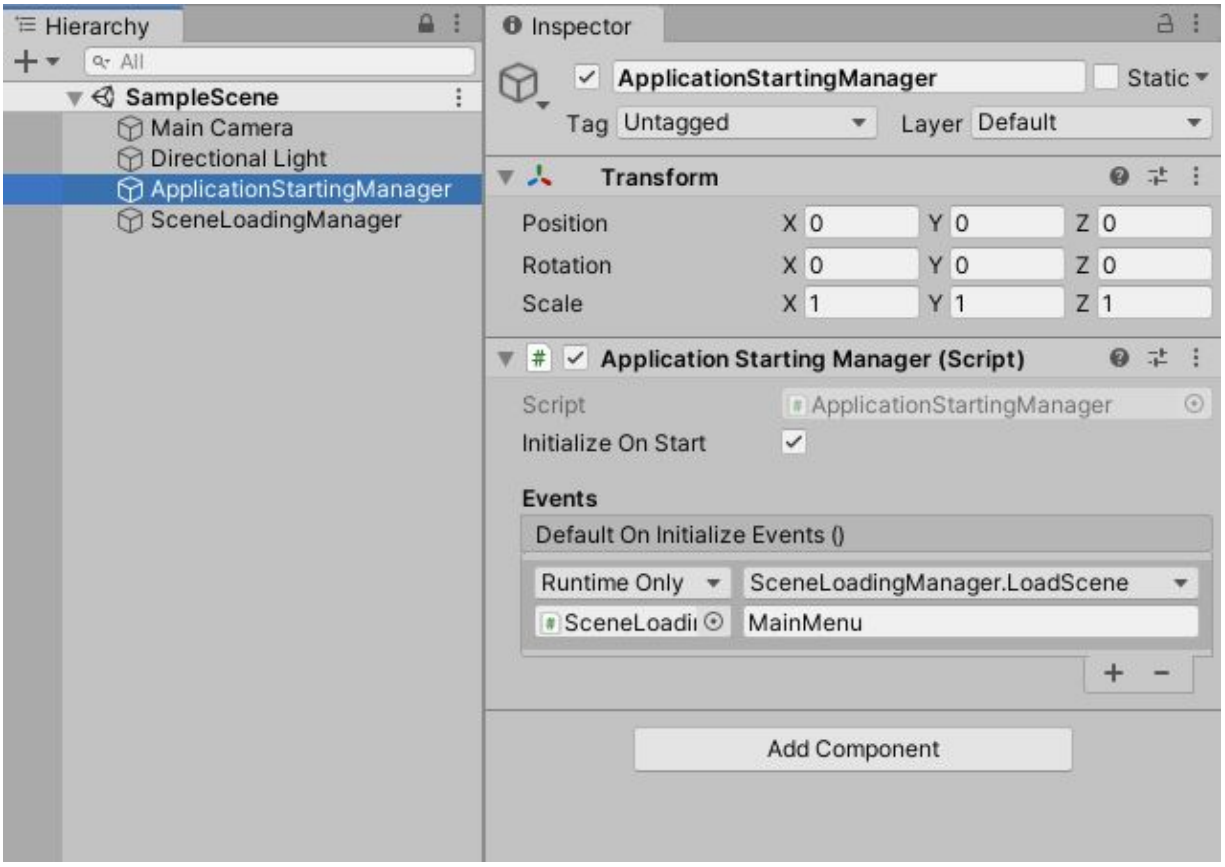
It allows you to edit the **Xcode** project values after an iOS build.

You can edit the **project languages**, enable the **push notifications**, enable **capabilities** and remove strings from the **Info.plist** file.

Application Starting Manager

Application Starting Manager is a MonoBehaviour class used to invoke the "**ExecuteStartupAction(System.Action callback);**" methods of all the **Application Startup** settings in the current environment.

Remember that the invocation order is specified in the **Execution Order Lists**.



Initialization

To works properly, it has to be placed in a Game Object of the first scene loaded at startup.

It can initialize itself automatically on Start() if the "**Initialize On Start**" is selected, otherwise you can manually initialize it calling the "**Initialize();**" method.
The manual initialization can be useful if you want to execute it after a custom splash screen or other.

On Initialize Events

You can add some "**On Initialize Events"** that will be invoked when all startup methods are called and the initialization is ended.

It can be done in the editor using **Unity Events** or in scripts using the **System.Action delegate**.

Using them, after the initialization of your settings, you can load a new scene (for example the Main Menu) or you can start your app components and menus.

Support

If you have any bugs or feature requests, don't hesitate to email us at **info@dragonkinstudios.com**