EASY MOBILE Pro

User Guide

sglib

Table of Contents

GETTING STARTED

ADVERTISING

GAME SERVICES

GIF

IN-APP PURCHASING

NATIVE APIs

NOTIFICATIONS

PRIVACY

SHARING

UTILITIES

Easy Mobile Pro User Guide

This document is the official user guide for the Pro version of the Easy Mobile plugin for Unity by SgLib Games.

Easy Mobile Versions

- Easy Mobile Pro
- Easy Mobile Basic
- Easy Mobile Lite (Free)

Important Links

- Website
- API Reference
- Support Email
- Forum

Connect with SgLib Games

- Unity Asset Store
- Facebook
- Twitter
- YouTube

Introduction

Today mobile games have many "de facto standard" features ranging from advertising, in-app purchasing, game services, notifications to sharing and rating system. These are considered "necessary evils" by many developers though: required, but not necessarily fun to do. More importantly, it takes time (a lot) to implement them. That means developers would need to repeat the boring, time-consuming task of integrating these features in almost every mobile games they are going to build, instead of focusing their energy on doing what we're all in game development for: creating fun!

Easy Mobile is our attempt to solve this problem. It is a many-in-one Unity plugin that greatly simplifies the implementation of the de facto standard features every mobile game needs. It is packed with ready-to-use components that you can just "plug" into your project, and have the necessary evils taken care of. We've been spending thousands of hours creating this product, so you can save that same amount of time[1] and *spend your time on real game development*!

> Easy Mobile supports two major mobile platforms: iOS and Android.

Highlights

Usability is our number one priority when designing Easy Mobile, hence the name. Other top priorities including robustness, flexibility and versatility. With that in mind, we come up with a product that boasts the following highlights:

- **Cross-platform API:** Easy Mobile's scripting API allows you to accomplish most tasks with only one line of code. And it is cross-platform, which means you can write code once and deploy on both iOS and Android.
- **Friendly Editor:** Easy Mobile comes with a simple yet powerful (and beautiful!) built-in editor that allows you to easily control everything from one place.
- **Automation:** Much more than a mere collection of methods, Easy Mobile operates quietly in the background and automates many chores such as service initialization and ad loading.
- **Modular Design:** Easy Mobile's functionalities are grouped into modules that can be enabled or disabled separately, so you can use what you want and not be worried about redundancies or contradictions with existing code.
- **Leveraging Official SDKs:** Easy Mobile leverages official 3rd-party SDKs of interest, e.g. Google Play Games plugin for Unity, integrating them with its own code to form a coherent system that maximizes usability and reliability - without reinventing the wheel!
- **Visual Scripting**[2]**:** Easy Mobile is not only for coders - it is fully compatible with Unity's most popular visual scripting plugin Playmaker, thanks to more than 100 custom actions built-in to the package.

What's in the Box?

This plugin is currently packed with the following modules:

- **Advertising**
  - Supports most popular ad networks including AdColony, AdMob, Chartboost, Facebook Audience Network, Heyzap, ironSource, MoPub, Tapjoy and Unity Ads
  - Even more ad networks can be used via mediation service provided by AdMob, Heyzap, ironSource, MoPub or Tapjoy

    - Automatic ad loading
    - Allows using multiple ad networks in one build
    - Allows different ad configurations for different platforms
    - Allows having multiple placements of each ad type
    - User consent support (GDPR compliant)
- **Game Services**
    - Works with Game Center on iOS and Google Play Games on Android
    - Automatic authentication
    - Custom editor for easy management of leaderboards and achievements
    - Saved Games[3]
- **GIF**[4]
    - Records screen, plays recorded clips and exports GIF images
    - High-performance, mobile-friendly GIF encoder
    - Giphy upload API for sharing GIF to social networks
- **In-App Purchasing**
    - Leverages Unity In-App Purchasing service
    - Custom editor for easy management of product catalog
    - Auto initialization
    - Receipt validation
    - Apple's Ask To Buy
    - Apple's Promotional Purchases
- **Native APIs**
    - Access to mobile native UI elements including alerts and (Android) toasts
    - More native functionalities will be added soon
- **Notifications**
    - Fully-customizable local notifications
    - Compatible with OneSignal and Firebase Cloud Messaging, free and popular services for push notifications
    - Supports notification channels and channel groups on Android O and higher
    - User consent support (GDPR compliant)
- **Privacy**
    - Provides tools and resources to help getting compliant with user privacy regulations such as GDPR
    - Multi-purpose native consent dialog that can act as the common interface for collecting user consent for all relevant services, instead of having multiple interfaces for various services
    - Built-in system to manage consent and communicate consent to relevant services

    - API to check if the current device is in the European Economic Area region (EEA region, affected by GDPR)
- **Sharing**
    - Shares texts, URLs and images using the mobile native sharing functionality
- **Utilities**
    - Store Review: provides an effective way to ask for reviews and ratings using the system-provided rating prompt of iOS (10.3+) and a native, highly customizable popup on Android

Easy Mobile: Basic vs Pro

Easy Mobile comes in 2 different versions: Basic and Pro. Easy Mobile Basic is the lower-price version which contains most of the core features of the plugin, except a few advanced functionalities such as GIF and Saved Games. The Pro version is the premium one and have all features available. Below is a feature-comparison table of the Basic and Pro versions of Easy Mobile.

FEATURES	BASIC	PRO
Advertising	☑	☑
AdColony, AdMob, Chartboost, Facebook Audience Network, Heyzap, ironSource, MoPub, Tapjoy & UnityAds support	●	●
Banner, interstitial & rewarded ad formats	●	●
One unified API for all ad networks	●	●
Automatic ad loading	●	●
Using multiple ad networks in one build	●	●
Multiple ad placements	●	●
Built-in Remove-Ads function	●	●
User consent support (GDPR compliant)	●	●
Game Services	▬	☑
Game Center (iOS) & Play Game Services (Android) support	●	●
Automatic authentication/initialization	●	●
Leaderboards	●	●
Achievements	●	●
Saved Games (using iCloud & Google Drive)		●
GIF	☐	☑
Low-overhead screen recorder		●
Built-in players for recorded clip playback		●
Mobile-friendly GIF encoder		●
Full control on GIF parameters: sizes, length, quality, frame-rate, etc.		●
Advanced color quantization for high quality GIF images		●
Giphy upload API		●
In-App Purchasing	☑	☑
Multiple stores supported: App Store, Google Play, Amazon Apps, etc.	●	●
Consumable, non-consumable & subscription product types	●	●
Custom editor for easy management of product catalog	●	●
Auto initialization	●	●
Apple's Ask-To-Buy	●	●
Apple's Promotional Purchases	●	●
Local receipt validation	●	●
Native APIs	☑	☑
Native mobile UI elements (alerts, toast)	●	●
Notifications	☑	☑
Fully-customizable local notifications	●	●
Android 8.0 notification channel & channel group support	●	●
Push notifications (OneSignal & Firebase Cloud Messaging support)	●	●
User consent support (GDPR compliant)	●	●
Privacy (GDPR compliance support)	●	●
Native, multi-purpose consent dialog	●	●
Built-in flexible consent management system	●	●
EEA region detector	●	●
Sharing	☑	☑
Sharing of images, texts and URLs via native sharing functionality	●	●
Built-in screenshot capturing API	●	●
Utilities	☑	☑
Store Review (native rating dialog)	●	●
Visual Scripting Support	☐	☑
Playmaker support (100+ custom actions)		●

[1]. We consider ourselves average developers, so this should be true to most indie developers, the main audience of this product. ↩

[2]. Easy Mobile Pro only. ↩

[3]. Easy Mobile Pro only. ↩

[4]. Easy Mobile Pro only. ↩

Using Easy Mobile

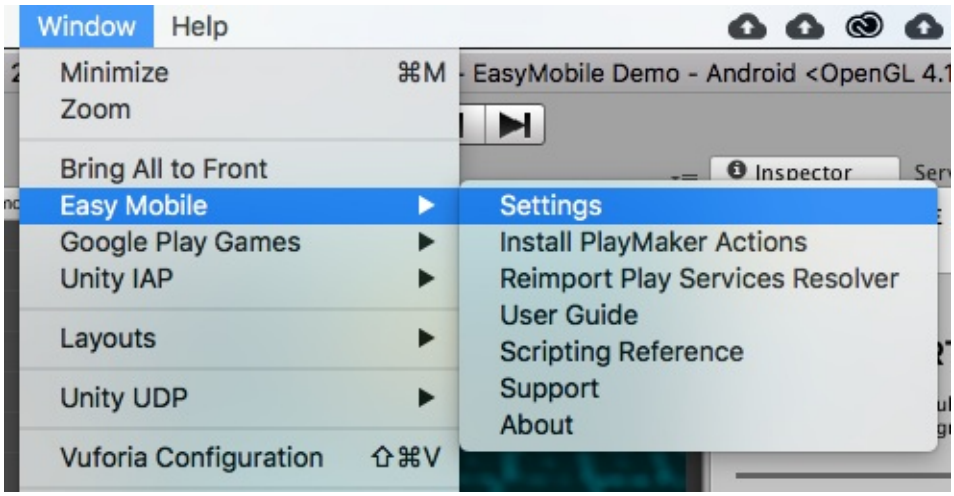
Overview

Using Easy Mobile involves 3 steps:

1. Configuring the plugin using the built-in Settings interface
2. Initializing the plugin runtime
3. Making appropriate API calls from script (or using visual scripting actions)

Main Menu

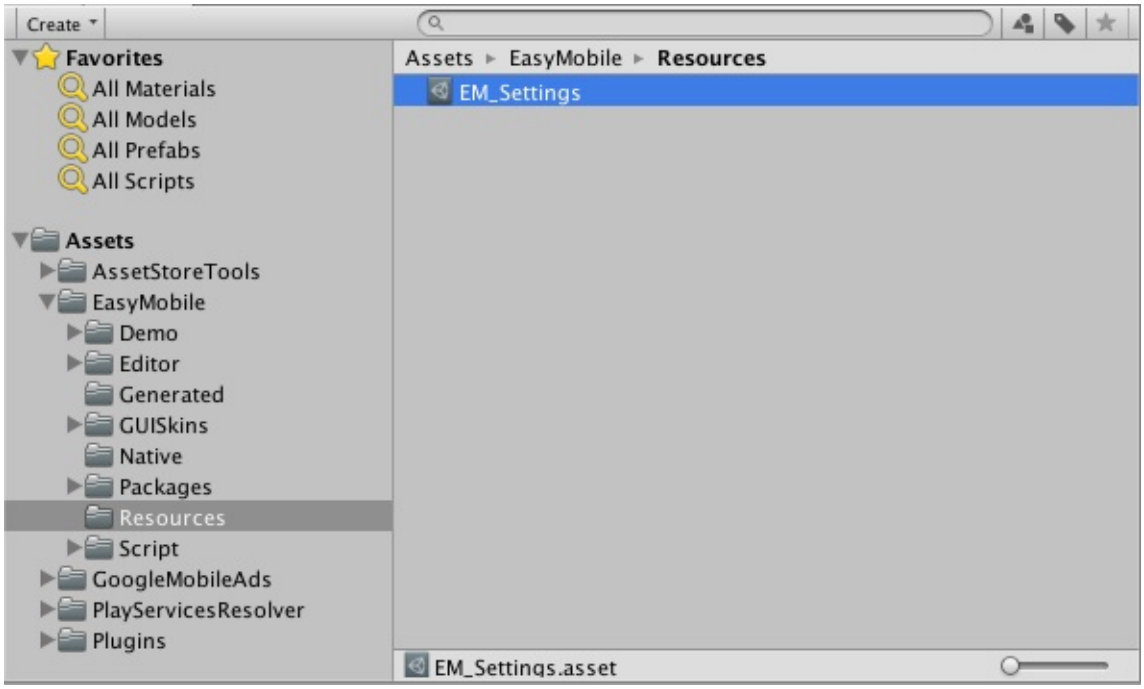
After importing Easy Mobile, there will be a new menu added at *Window* > *Easy Mobile* from which you can open the Global Settings interface to configure modules as well as access other resources.



Global Settings

The Global Settings interface is the only place you go to configure the plugin. Here you can control everything including toggling module on or off, providing ads credentials, adding leaderboards, creating a product catalog, etc.

All these settings are stored in the EM_Settings object, which is a ScriptableObject created automatically after importing the plugin and is located at *Assets/EasyMobile/Resources*. You can also access this EM_Settings class from script and via its properties accessing each module settings in runtime.



Initializing

> You must always initialize Easy Mobile before using its API.

Since the initialization must be done before the Easy Mobile API can be used, it is advisable to do it as soon as your app launches. This will also allow automatic services such as ad loading to start soon and the ads will be more likely available when needed. Practically, this means attaching the initializing script to some game object in the first scene of your app.

To initialize simply call the *Init* method of the *RuntimeManager* class. This method is a no-op if the initialization has been done so it's safe to be called multiple times. You can also check if Easy Mobile has been initialized using the *IsInitialized* method.

```
using UnityEngine;
using System.Collections;
using EasyMobile; // include the Easy Mobile namespace to use its scripting API

public class EasyMobileInitializer : MonoBehaviour
{
    // Checks if EM has been initialized and initialize it if not.
    // This must be done once before other EM APIs can be used.
    void Awake()
    {
        if (!RuntimeManager.IsInitialized())
            RuntimeManager.Init();
    }
}
```

Scripting API

After initializing Easy Mobile you can use its scripting API, which is written in C#. Note that the API is put under the namespace EasyMobile so you need to add the following statement to the top of your scripts to access it.

```
// Put this on top of your scripts to use Easy Mobile scripting API
using EasyMobile;
```

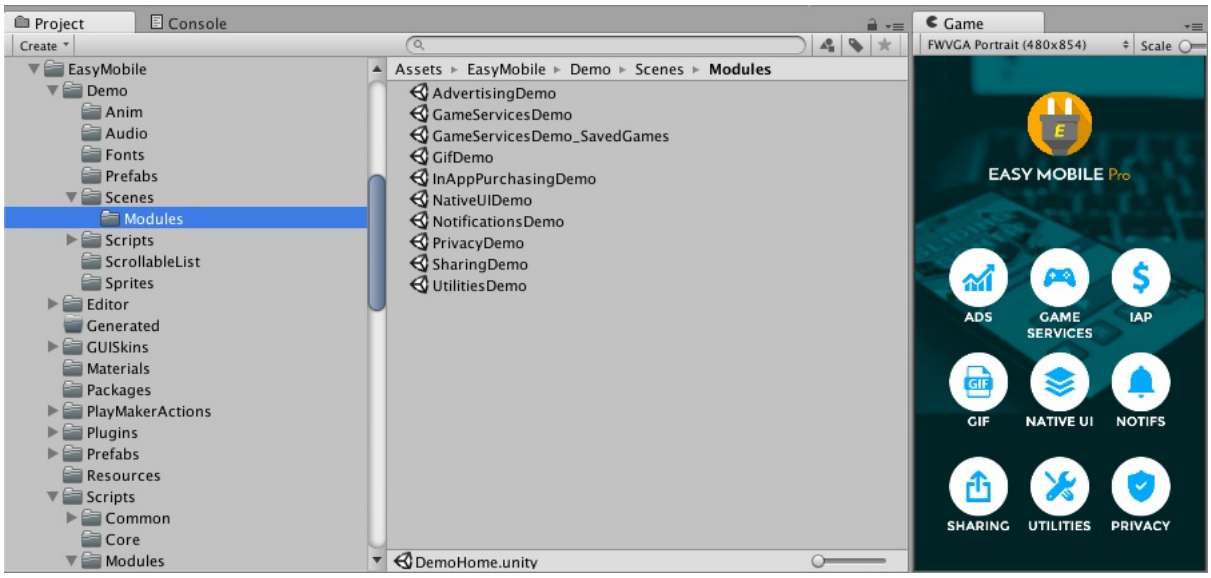
> Easy Mobile's API is cross-platform so you can use the same code for both iOS and Android. You can find the API reference here.

Testing

Most of Easy Mobile's functionalities are mobile-specific which may not work properly within the Unity editor. Therefore you should always perform testing on actual iOS or Android devices.

Using the Demo App

Easy Mobile comes with a demo app that you can use to quickly test each module's operation after configuring, even before writing any code! The demo app is located at folder *Assets/EasyMobile/Demo*. Its scenes are in the subfolder *Scenes*, you can build all scenes or only scenes that you want to test. Its scripts are in the subfolder *Scripts*, which you can refer to for an example of how Easy Mobile API can be used.



Using PlayMaker Actions

Easy Mobile is officially compatible with PlayMaker, with more than 100 custom actions ready to be used. You can install these actions from menu *Window* > *Easy Mobile* > *Install PlayMaker Actions*.

When installing the PlayMaker actions, a demo app will also be imported at *Assets/EasyMobile/Demo/PlayMakerDemo*. This demo is a copy of our main demo app, rebuilt using PlayMaker actions instead of C# scripts. You can take it as an example to get an insight into how Easy Mobile's PlayMaker actions can be used in practice. Apart from installing PlayMaker (obviously), you need to do a few more setup steps as described below before running this demo app.

Installing the Unity UI add-on for PlayMaker

Our PlayMaker demo app uses the Unity UI system, so you need to install the Unity UI add-on for PlayMaker.

Importing PlayMakerGlobals

The demo app uses global PlayMaker variables and events, so you need to import them before running the app.

If you project is new and doesn't have any PlayMakerGlobals (in the *Assets/PlayMaker/Resources* folder), simply copy our Globals over by these steps:

- Double-click the PlayMakerGlobals.unitypackage in the *Assets/EasyMobile/Demo/PlayMakerDemo* folder.
- Locate the newly created file PlayMakerGlobals_EXPORTED.asset right under the *Assets* folder.
- Rename the file to PlayMakerGlobals.asset and move it to the *Assets/PlayMaker/Resources* folder.

If you project already contains some global PlayMaker variables and events (a PlayMakerGlobals.asset file exists in the folder *Assets/PlayMaker/Resources)*, you can merge these with our demo's Globals using PlayMaker's Import Globals tool: go to menu *PlayMaker* > *Tools* > *Import Globals* and select the PlayMakerGlobals.unitypackage in the *Assets/EasyMobile/Demo/PlayMakerDemo* folder.

> With the PlayMaker demo app, we perform the Easy Mobile initialization using a FSM in the DemoHome_PlayMaker scene so you should always build this scene when using this demo app.

Requirements

Easy Mobile requires:

- Unity 5.5.5 or newer.
- iOS 8.0 or newer.
- Android 4.0 (API Level 14) or newer.

Advertising: Introduction

The Advertising module helps you quickly setup and show ads in your games. Here're some highlights of this module:

- **Supports multiple networks**
  - This module allows showing ads from most of top ad networks: AdColony, AdMob, Chartboost, Facebook Audience Network, Heyzap, ironSource, MoPub, Tapjoy and Unity Ads
  - Even more networks can be used via mediation service provided by AdMob, Heyzap, ironSource, MoPub or Tapjoy
- **Using multiple networks in one build**
  - It's possible to use multiple ad neworks at the same time, e.g. use AdMob for banner ads, while using Chartboost for interstitial ads and Unity Ads for rewarded ads
  - Different configurations for different platforms are allowed, e.g. use Unity Ads for rewarded ads on Android, while using Chartboost for that type of ads on iOS
- **Automatic ad loading**
  - Ads will be fetched automatically in the background; new ad will be loaded if the last one was shown

The table below summarizes the ad types supported by Easy Mobile for each ad network.

Ad Network	Banner Ad	Interstitial Ad	Rewarded Ad	Mediation
AdColony		◉	◉	
AdMob	◉	◉	◉	◉
Chartboost		◉	◉	
Audience Network	◉	◉	◉	
Heyzap	◉	◉	◉	◉
ironSource	◉	◉	◉	◉
MoPub	◉	◉	◉	◉
Tapjoy		◉	◉	◉
Unity Ads		◉	◉	

Ad Placements

An ad placement in Easy Mobile represents a specific "location" in your app where an ad is served. For example the "GameOver" placement can be defined as the "location" in your app where a game is over, and an ad is served. An ad placement is normally associated with an ad unit with a specific ad ID. Easy Mobile has several built-in ad placements including a default placement and other placements such as "Startup", "HomeScreen", "MainMenu", etc. You can also create more custom ad placements to suit your needs.

Grouping ads into placements provides an intuitive way to organize ads in your app and simplifies the implementation of flexible and sophisticated advertising strategies. It allows having more than one unit of a certain ad type (banner, interstitial or rewarded ad) of the same ad network. For example, you can use the default placement to show normal AdMob interstitial ads (and get paid!), while having a "Startup" placement to show AdMob interstitial house ads (at app launch), thus creating a *free* cross-promotion system for your apps!

> Ad placements don't complicate things though. If you only need a basic usage of advertising, you can simply ignore all placement stuff when working with the Advertising API, and Easy Mobile will automatically use the default placement. Therefore, it's only necessary to provide ad IDs associated with the default placement when setting up ad networks (some ad networks may not even require such IDs). All other ad placements are optional and you only need to configure if you want to use any of them.

Default vs Non-Default Ads

A default ad of a certain type is the ad unit that belongs to the default network for that ad type at the default placement. For example, if the default interstitial network for the current platform is AdColony, then the AdColony interstitial ad at the default placement is the *default interstitial ad*. The rest are considered non-default interstitial ads. The same is true for banner and rewarded ads.

GDPR Compliance

> We recommend you to read the Privacy chapter first to gain a comprehensive understanding of the tools and resources offered by Easy Mobile to help your app get compliant with GDPR, including the consent dialog and the consent management system.

Advertising is one of those services affected by the GDPR, because most ad providers collect user data to serve personalized ads. Most ad networks recommend requesting user consent for such data usage and serve personalized or non-personalized ads accordingly. Easy Mobile provides a native, multi-purpose dialog for collecting user consent for all ad networks or each individual ad network, as well as other relevant services, in a flexible manner. It also allows you to flexibily communicate the collected consent (apply the consent) to the Advertising module either at the module level or the vendor level.

> Allowing the user to provide and manage consent for all services via a single interface (dialog) is advisable in terms of user experience, because the user may find it irritating being presented multiple dialogs asking consent for various things.

Consent	Description	Priority
Module consent	Common consent applied to all supported ad networks in the Advertising module	Lower
Vendor consent	Consent applied to an individual ad network, e.g. AdMob	Higher

Consent is normally applied during the initialization of an ad network. Therefore it is important to collect consent before initializing ad networks. Practically this means collecting consent before initializing the Easy Mobile runtime. If there's a vendor consent specified for the current network, it will be used. Otherwise the module consent will be used. If neither was specified, the global consent will be used. In case no consent provided at all levels, the ad network carries out its initialization without applying any consent and will serve personalized ads (the "pre-GDPR" behavior). The table below summarizes how Easy Mobile configures each network according to the provided consent.

Ad Network	Consent Granted	Consent Revoked	Consent Unknown
AdColony	Setting GdprRequired to 'true' and GdprConsentString to "1"	Setting GdprRequired to 'true' and GdprConsentString to "0"	Do nothing
AdMob	Do nothing (keep serving personalized ads as normal)	Setting "npa" key to "1" when constructing AdRequest to serve non-personalized ads	Do nothing
Chartboost	Calling	Calling	Do

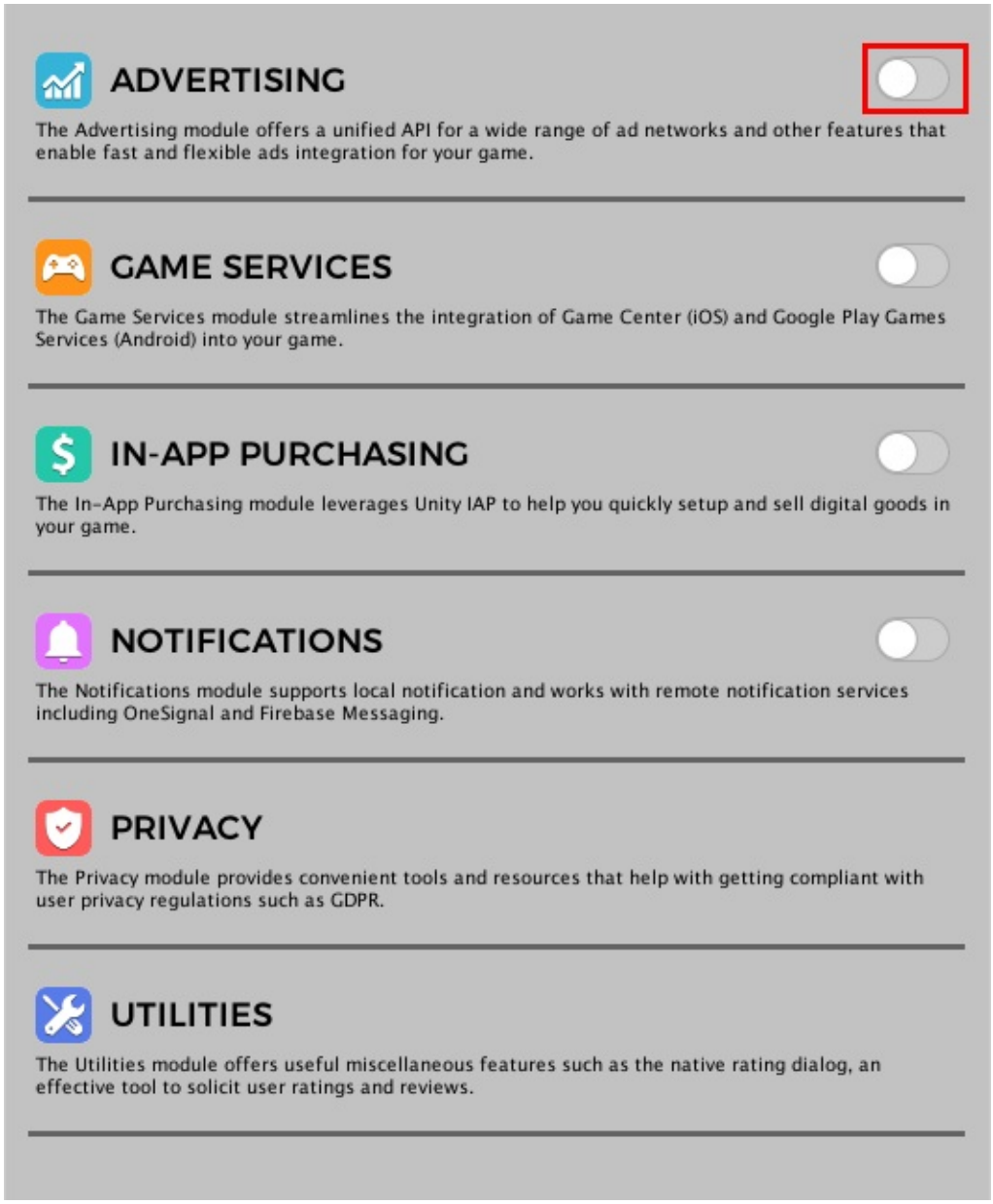
Chartboost	Chartboost.restrictDataCollection(false);	Chartboost.restrictDataCollection(true);	nothing
Audience Network	Do nothing	Do nothing	Do nothing
Heyzap	Calling HeyzapAds.SetGdprConsent(true);	Calling HeyzapAds.SetGdprConsent(false);	Do nothing
ironSource	Calling IronSource.setConsent(true);	Calling IronSource.setConsent(false);	Do nothing
MoPub	Calling MoPub.PartnerApi.GrantConsent();	Calling MoPub.PartnerApi.RevokeConsent();	Do nothing
Tapjoy	Calling Tapjoy.SetUserConsent("1");	Calling Tapjoy.SetUserConsent("0");	Do nothing
Unity Ads	Calling SetGdprMetadata(true);	Calling SetGdprMetadata(false);	Do nothing

Reference:

- AdColony: https://github.com/AdColony/AdColony-Unity-SDK-3/wiki/GDPR
- AdMob: https://developers.google.com/admob/unity/eu-consent
- Chartboost: the C# source code of the Chartboost SDK for Unity
- Facebook Audience Network: https://developers.facebook.com/docs/audience-network/unity/ (the documentation doesn't mention anything about consent)
- Heyzap: https://developers.heyzap.com/docs/unity_sdk_setup_and_requirements#step-7-adding-user-consent
- ironSource: https://developers.ironsrc.com/ironsource-mobile/android/advanced-settings/#step-1
- MoPub: https://developers.mopub.com/docs/unity/gdpr/
- Tapjoy: https://dev.tapjoy.com/sdk-integration/
- Unity Ads: https://unityads.unity3d.com/help/legal/gdpr

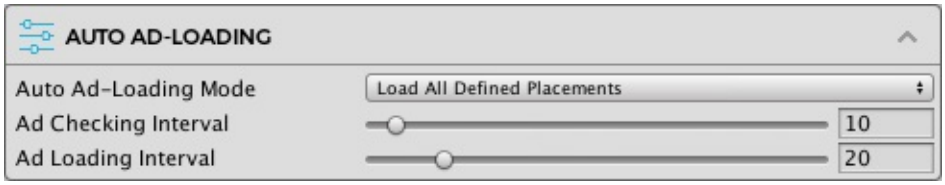
Advertising: Settings

To use the Advertising module you must first enable it. Go to *Window* > *Easy Mobile* > *Settings*, then click the toggle to the right-hand side of the Advertising tab to enable and start configuring the module.



Automatic Ad Loading

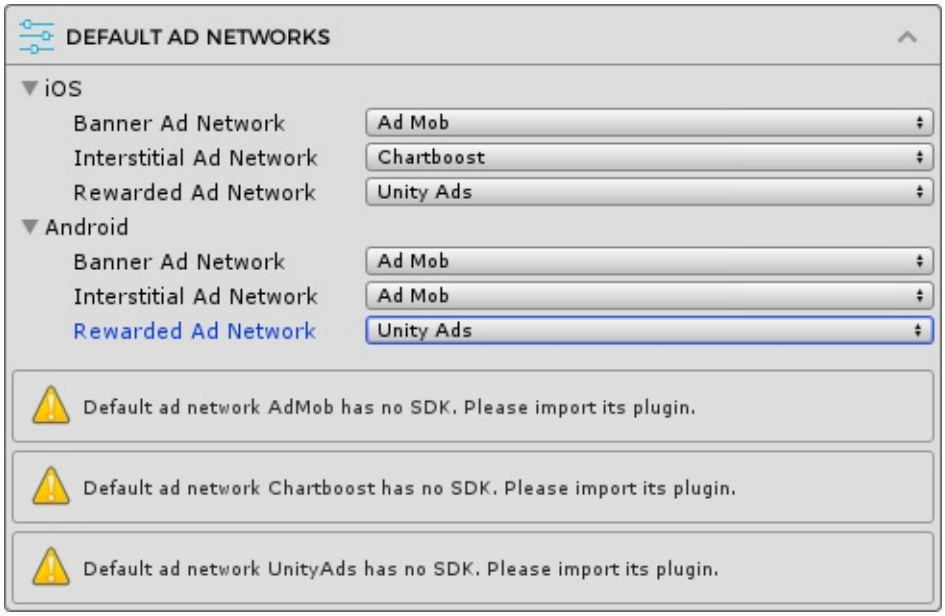
Automatic ad loading is a handy feature of the Advertising module. It regularly checks for the availability of ads, and performs loading if an ad hasn't been loaded or was consumed. With automatic ad loading, you can forget about loading ads manually and rest assure that ads are always ready whenever they are needed. You can configure this feature in the **AUTO AD-LOADING CONFIG** section.



- *Auto Ad-Loading Mode*:
  - *None*: disable the auto ad-loading, you can load ads manually from script, see Manual Ad Loading
  - *Load Default Ads*: only default ads are loaded automatically, non-default ads must be loaded manually
  - *Load All Defined Placements*: this mode allows loading of all placements (default and non-default) that are added in the module settings, provided that the placements are well defined, meaning that each has a valid associated ID (where applicable) and the corresponding SDK is imported; this is the recommended option
- *Ad Checking Interval*: change this value to determine how frequently the module should perform ads availability check, the smaller value the more frequently
- *Ad Loading Interval*: the minimum duration required between two ad loading requests

Default Ad Networks

You can select default ad networks for each platform in the **DEFAULT AD NETWORKS** section. You can have different networks for different ad types and different selections for different platforms. If you don't want to use a certain type of ad, simply set its network to *None*.



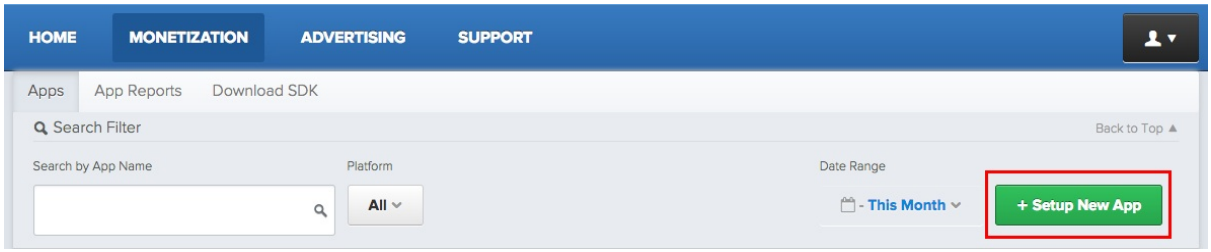
> Pay attention to the warnings and import the required plugins if you haven't already.

Advertising: Settings | Setup AdColony

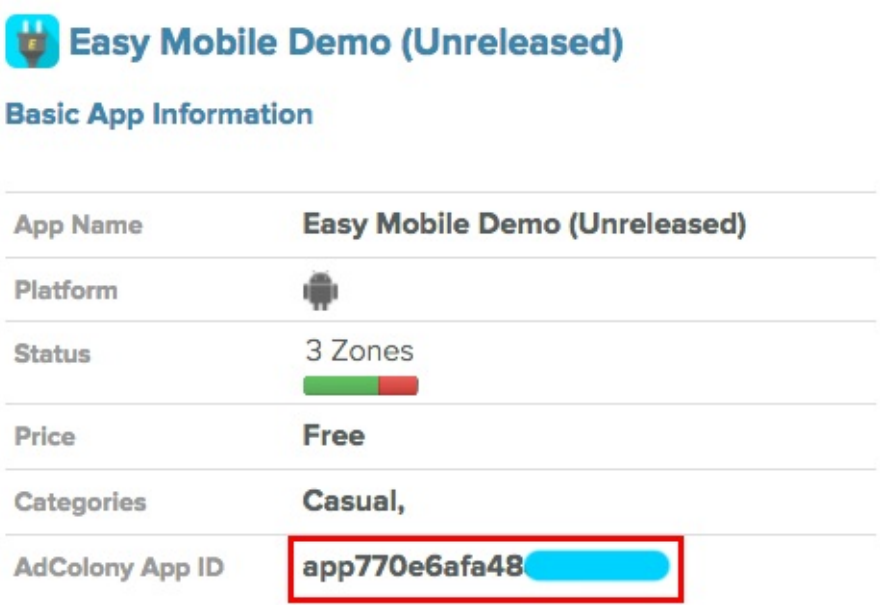
Creating AdColony Apps and Zone Ids

To show ads from AdColony you need to create apps and ad zones in its clients portal. To access the clients portal, create an account and login to AdColony page.

In the clients portal, select **MONETIZATION** tab, then select the **Apps** sub-tab and click the **Setup New App** button.



In the opened page enter the required information for your new app, e.g. app name, platform and location. You can also select the ad types that you would like to allow in your app. Hit **Create** when you're done, your app will be created and you'll be redirected back to the **Apps** page. Select your newly created app to reveal its information, which looks similar to the picture below. Note the **AdColony App ID** as we will use it later.



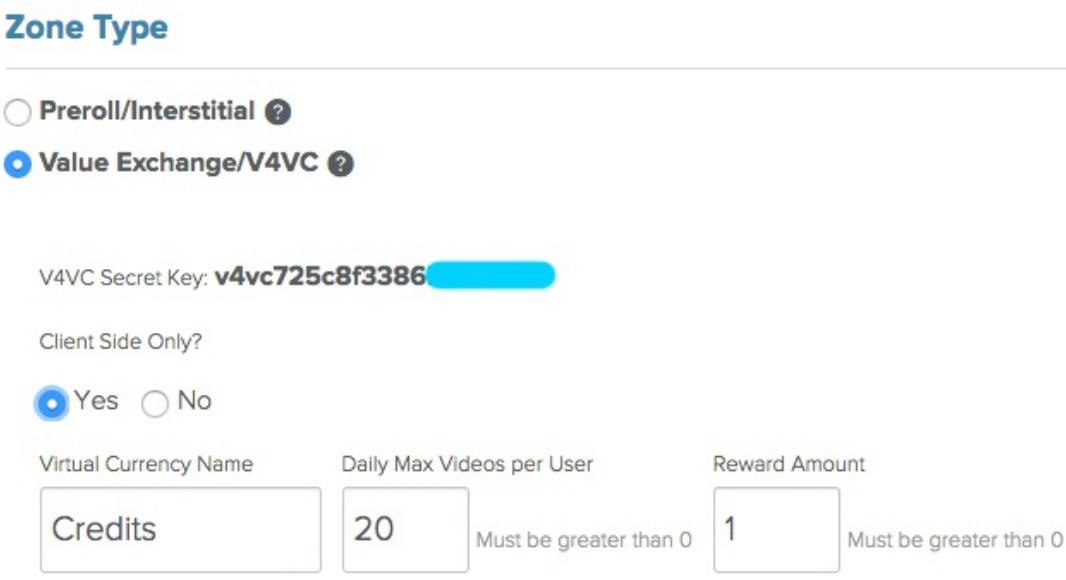
Now your app is ready, the next step is to create ad zones for it. Click the **Setup New Ad Zone** at the bottom of the app edit page to create a new ad zone.

In the **Integration** section, give your ad zone a name, optional notes and set its as active. Note the **Zone ID** as we'll use it later.

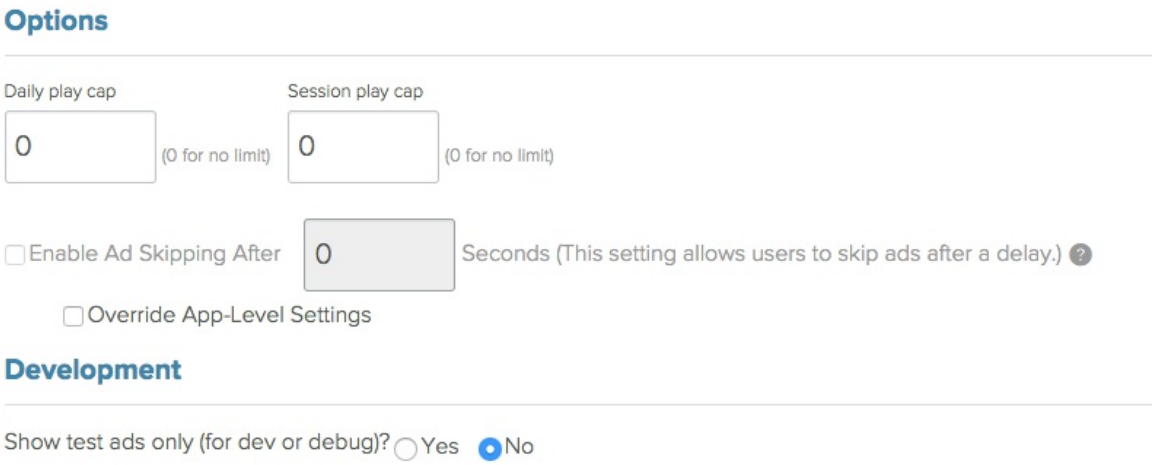
> The Zone ID will appear once you save your new ad zone.

In the **Creative Type** section, select the **Video** option.

In the **Zone Type** section, select **Preroll/Interstitial** if you want to use this zone for interstitial video ads. Otherwise, select **Value Exchange/V4VC** to use it for rewarded ads.



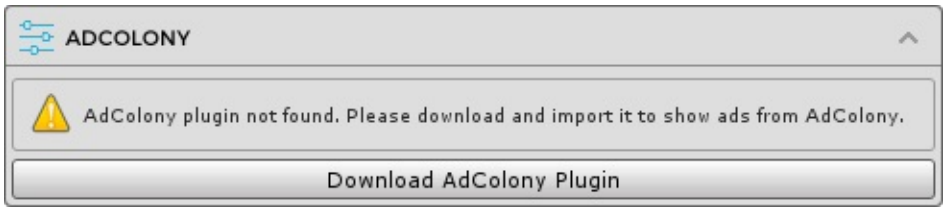
In the **Options** section, you can set a daily cap or a session cap to limit the number of ads served to a user per day or per session, respectively. In the **Development** section, you can choose to show test ads only (for debug purpose), don't forget to disable this option when your app is released.



Now your new ad zone is fully configured, click the **Save** button to save it. Repeat the process to create other ad zones to suit your needs. Typically, you'd want to have 2 ad zones, one for interstitial ads and one for rewarded ads. If you're targeting multiple platforms, create a new app for each platform, and for each app create the necessary ad zones.

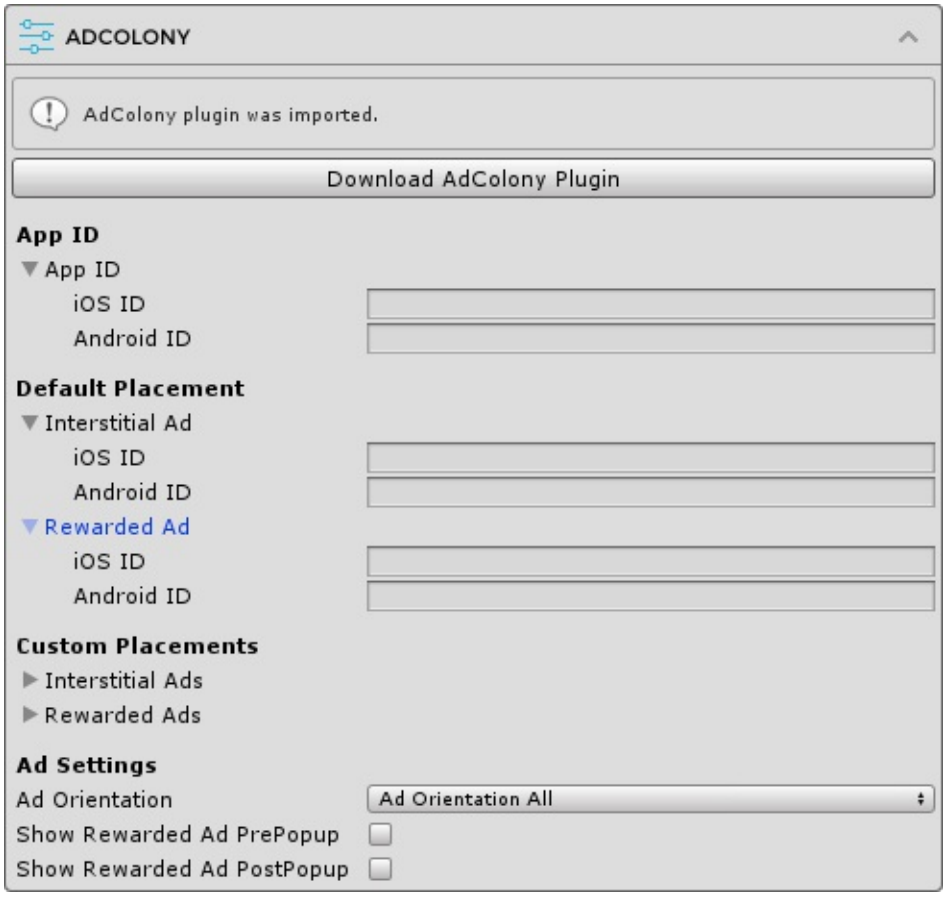
Importing AdColony Plugin

To have your Unity app work with AdColony you need to import the AdColony plugin for Unity. In the **ADCOLONY** section of the Advertising module, click the *Download AdColony Plugin* button to to open the download page. Download the plugin and import it to your project.



Configuring AdColony

After importing the AdColony plugin, the **ADCOLONY** section will be updated as below.



App ID

In this section you can enter the app ID created in the AdColony clients portal for each platform.

Default Placement

Here you can enter the ad IDs to be used with the default placement for each platform. These are the only ad IDs required if you are not using any custom placements in your app. Note that you only need to provide IDs for the ad types you want to use, e.g. if you only use AdColony interstitial ads you can leave the rewarded ad IDs empty.

Custom Placements

Here you can optionally enter the ad IDs associated with non-default ad placements to be used in your app. You can have an arbitrary number of custom placements and can use built-in placements or create new placements for your needs.

Ad Settings

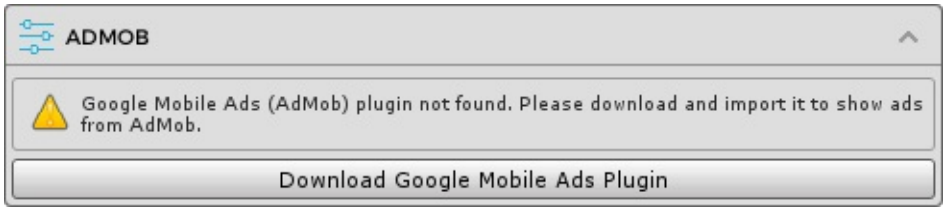
This section includes other settings including:

- *Ad Orientation*: the default ad orientation of your app.
- *Show Rewarded Ad PrePopup*: whether to show a popup before a rewarded ad starts.
- *Show Rewarded Ad PostPopup*: whether to show a popup after a rewarded ad has finished.

Advertising: Settings | Setup AdMob

Importing AdMob Plugin

To show ads from AdMob you need to import the Google Mobile Ads plugin. In the **ADMOB** section, click the *Download Google Mobile Ads Plugin* button to to open the download page. Download the plugin and import it to your project.



Configuring AdMob

After importing the Google Mobile Ads plugin, the **ADMOB** section will be updated as below.

App ID

First you need to enter the required AdMob app ID for each platform.

> To find the App ID for your app follow the instructions here.

Default Placement

Here you can enter the ad IDs to be used with the default placement for each platform. These are the only ad IDs required if you are not using any custom placements in your app. Note that you only need to provide IDs for the ad types you want to use, e.g. if you only use AdMob banner ads you can leave the interstitial and rewarded ad IDs empty.

> If you're not familiar with AdMob, follow the instructions here to create ad units and obtain the ad IDs; an ad ID should have the form of ca-app-pub-0664570763252260xxxxxxxxxx.

Custom Placements

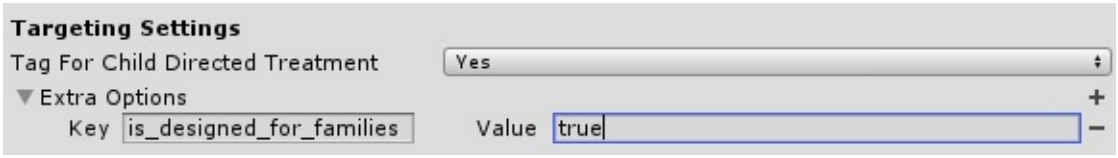
Here you can optionally enter the ad IDs associated with non-default ad placements to be used in your app. You can have an arbitrary number of custom placements and can use built-in placements or create new placements for your needs.

Targeting Settings

You can provide targeting information for your app in the **Targeting Settings** section. These settings will be applied to all AdMob ad requests in your app. You can learn more about AdMob ad targeting here.

- *Tag For Child Directed Treatment*: indicates whether you want Google to treat your content as child-directed when you make an ad request for the purposes of Children's Online Privacy Protection Act (COPPA)
- *Extra Options*: extra settings in form of key value pairs, e.g. for setting "max_ad_content_rating"

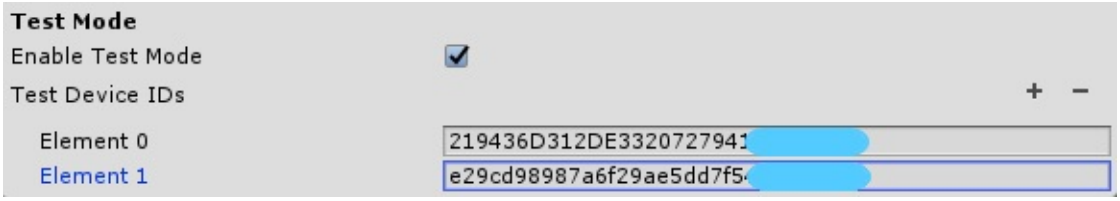
> **Using AdMob with the Designed for Families program**
>
> According to this article, apps that join in to Google Play's Designed for Families program can fall into two categories:
>
> - **Primarily child-directed apps:** if your app is admitted to the program as a primarily child-directed app, "AdMob will automatically begin serving Designed for Families-compliant ads for all ad requests coming from the app", which means you don't need to specify the child directed setting in your app.
> - **Mixed-audience apps:** if your app targets both child and adult audiences, you need to set the extra key "is_designed_for_families" to *true* and tag your app for child-directed treatment. You can do that in Easy Mobile settings as below.

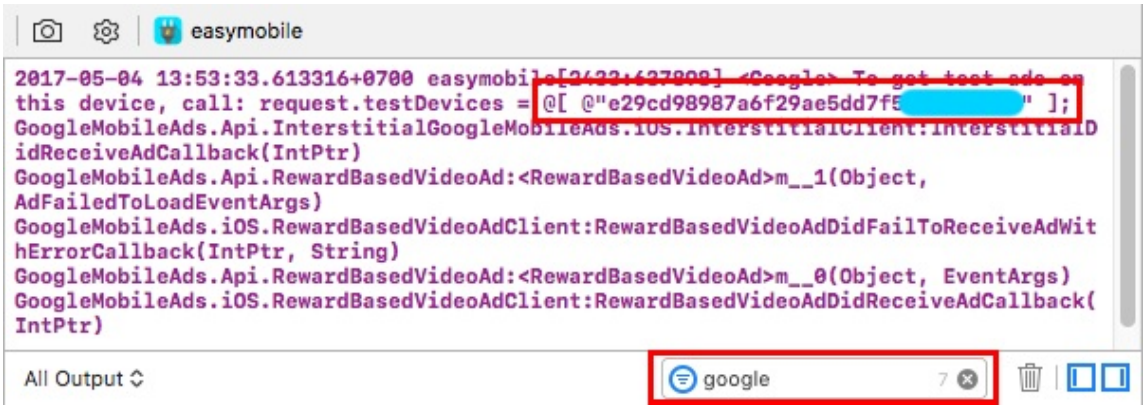


> **Overriding AdMob targeting settings in script**
>
> You can override AdMob targeting settings in script by setting the property *EM_Settings.Advertising.AdMob.TargetingSettings.* All subsequent ad requests will be sent with the new settings.

Test Mode

To enable AdMob's test mode, simply check the *Enable Test Mode* option and enter the IDs of your testing devices into the *Test Device Ids* array.



> You can find the ID of your test device by building and running the Easy Mobile demo app on that device. Remember to add the EasyMobile prefab to the DemoHome scene before starting the build.
>
> **Android device ID**
>
> - In Unity, build the Easy Mobile demo app for Android platform
> - Install and run the demo app on your testing device
> - Open Terminal (Mac) or Cmd (Windows) and type in
>
> ```
> adb logcat -s Ads
> ```
>
> (if you're on Windows, you may need to add the Android SDK path to the Windows System PATH)
>
> - In the demo app, select ADVERTISING and then click the SHOW BANNER AD button
> - Observe the output logcat in the Terminal/Cmd and locate a line similar to the one in the following image, the value between the double quotes is your device ID
>
> ```
> I Ads     : Starting ad request.
> I Ads     : Use AdRequest.Builder.addTestDevice("219436D312DE332072794") to get test ads on this device.
> ```
>
> **iOS device ID**
>
> - In Unity, build the Easy Mobile demo app for iOS platform
> - Open the generated project in Xcode and run it on your testing device
> - Type 'google' into the filter box of the Xcode Console, and find your device ID between the double quotes as highlighted in the following image
>
> 
>

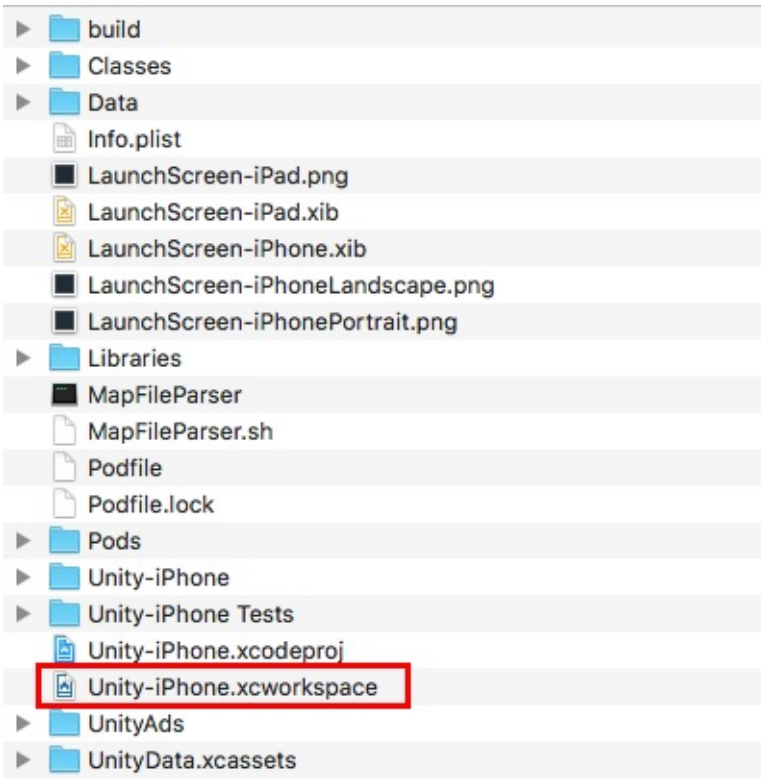
Notes on AdMob Rewarded Ads

AdMob only allows one rewarded ad to be loaded at a time. That means the loaded ad must be consumed before another ad can be loaded. If you register multiple placements for AdMob rewarded ad, only one of them can be used at a time. The automatic ad loading feature favors the rewarded ad at the default placement and will always load it first.

Building Notes

The Google Mobile Ads plugin for Unity employs CocoaPods to automatically import the necessary frameworks to the generated Xcode project when an iOS build is performed in Unity. Therefore you need to install CocoaPods to your Mac: please go to https://cocoapods.org/ for install instructions, as well as for more information about CocoaPods. Note that you only need to install CocoaPods, everything else will be done automatically by the Google Mobile Ads plugin.

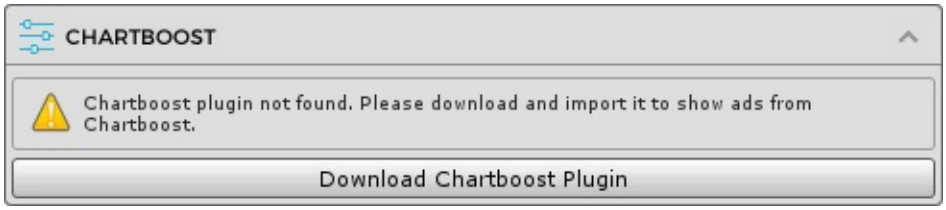
> When building for iOS in Unity, CocoaPods will automatically create an Xcode workspace (with .xcworkspace extension) in the generated Xcode project. You should always open this workspace instead of the normal project file (with .xcodeproj extension).



Advertising: Settings | Setup Chartboost

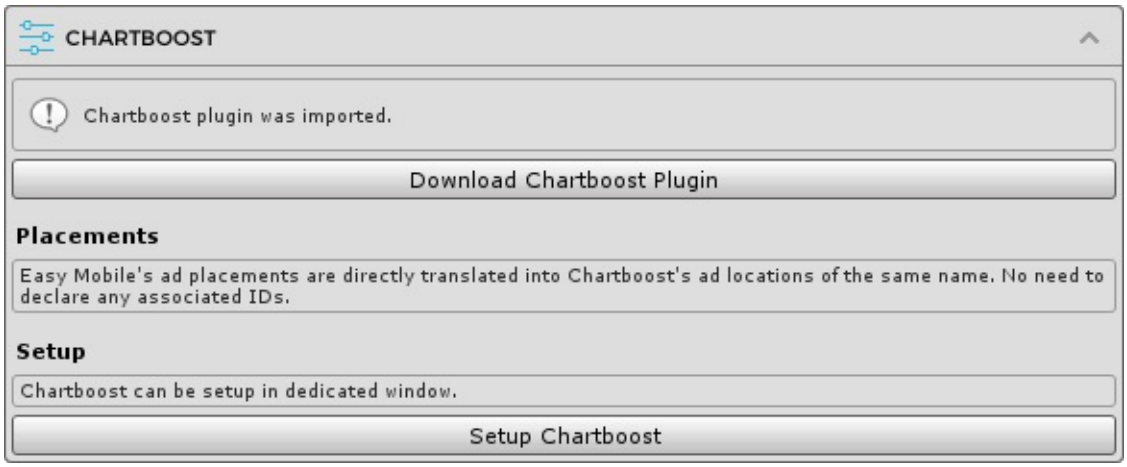
Importing Chartboost Plugin

To show ads from Chartboost you need to import the Chartboost plugin for Unity. In the **CHARTBOOST** section, click the *Download Chartboost Plugin* button to open the download page. Download the plugin and import it to your project.



Configuring Chartboost

After importing Chartboost plugin, the **CHARTBOOST** section will be updated as below.

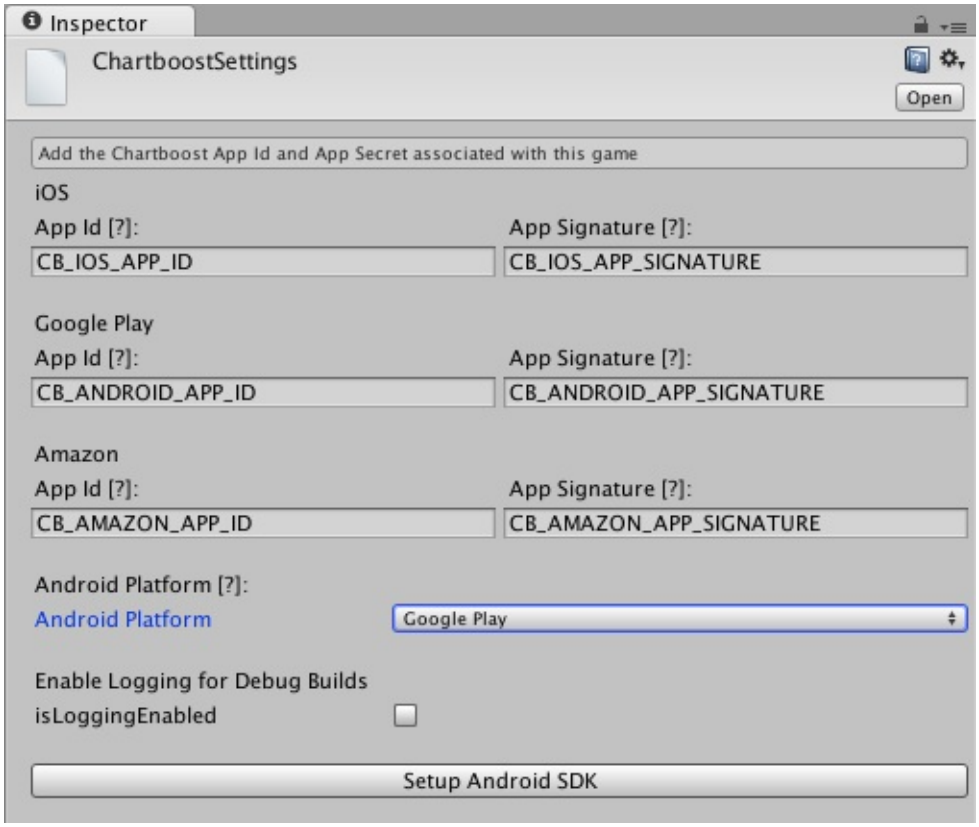


Placements

An Easy Mobile's ad placement will be directly translated into a Chartboost's ad location with the same name in runtime. Therefore if you want to specify a location when showing a Chartboost ad, simply show the ad with an ad placement of the same name. There's no need to declare any ad IDs to be associated with the placements.

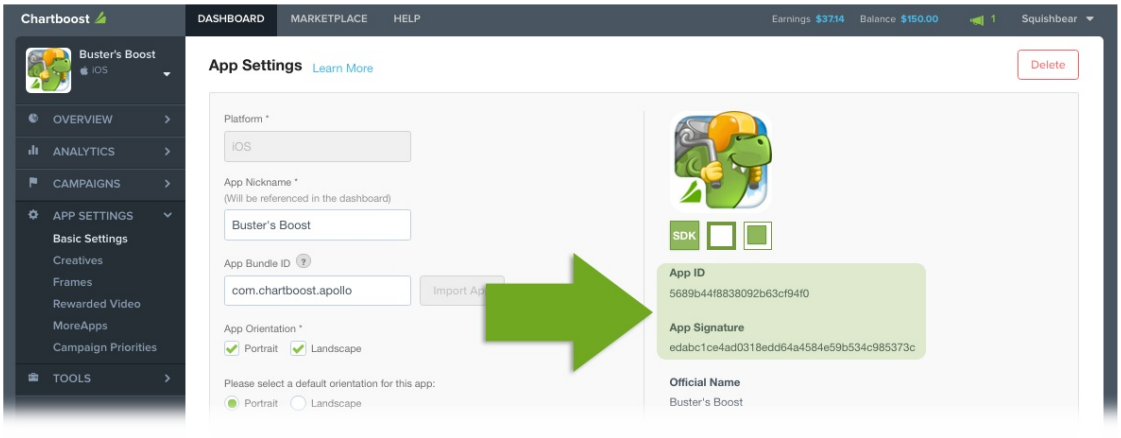
Setup

Click the *Setup Chartboost* button to open Chartboost's dedicated settings interface.



Provide the App IDs and App Signatures for your targeted platforms. Remember to click the *Setup Android SDK* button if you're building for Android.

> To obtain the App Id and App Signature you need to add your app to the Chartboost dashboard. If you're not familiar with the process please follow the instructions here.
>
> After adding the app, go to APP SETTINGS > Basic Settings to find its App ID and App Signature.



Android READ_PHONE_STATE Permissions

The Chartboost SDK includes the READ_PHONE_STATE permission on Android, to "handle video playback when interrupted by a call", as stated in its manifest. READ_PHONE_STATE permission requires your app to have a privacy policy when uploaded to Google Play. Since this permission is not mandatory to run the Chartboost SDK, you can safely remove it if you are not ready to provide the required privacy policy. To remove the permission, open the AndroidManifest.xml file located at *Assets/Plugins/Android/ChartboostSDK* folder, then delete the corresponding line (or comment it out as below).

```
<!-- Exclude the READ_PHONE_STATE permission because it requires a privacy policy -->
<!-- <uses-permission android:name="android.permission.android.permission.READ_PHONE_STATE" /> -->
```

Testing Notes

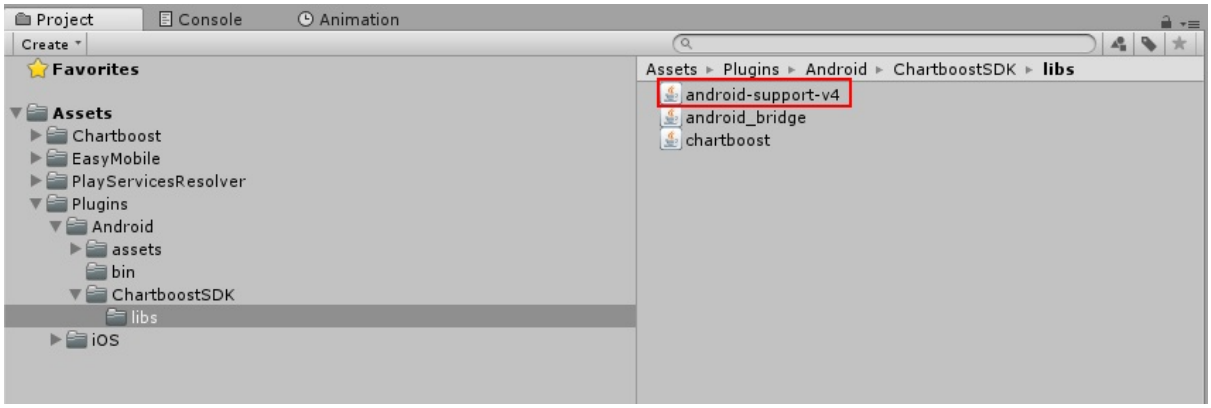
Please note that to show ads from Chartboost you need to either create a publishing campaign or enable the Test Mode for your app.

- To create a publishing campaign follow the instructions here
- To enable Test Mode follow the instruction here

Building Notes

If the version of the imported Chartboost plugin includes a copy of *android-support-v4.jar*, it may create build errors on Android platform due to conflict with the *com.android.support-v4* library that exists in the Assets/Plugins/Android folder.

To avoid this issue, remove the *android-support-v4.jar* in Assets/Plugins/Android/ChartboostSDK/libs/ folder.



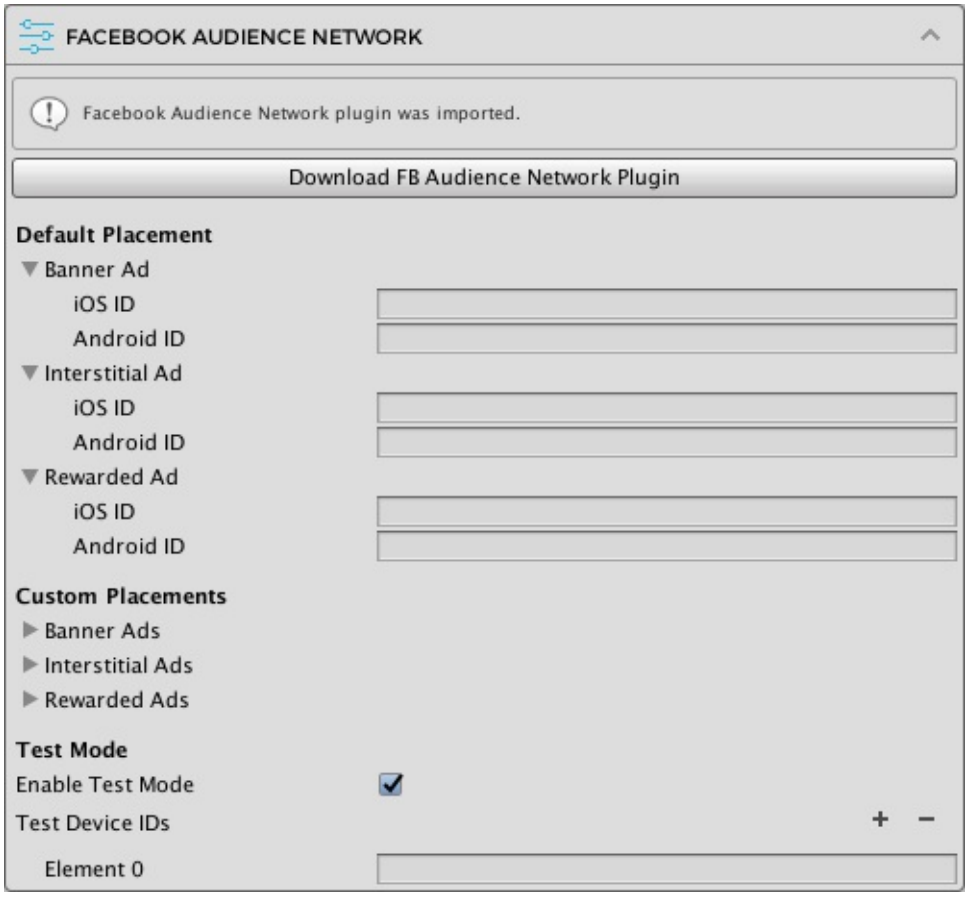
Advertising: Settings | Setup Audience Network

Importing Facebook Audience Network Plugin

To show ads from Facebook Audience Network you need to import the Facebook Audience Network plugin for Unity. In the **FACEBOOK AUDIENCE NETWORK** section, click the *Download FB Audience Plugin* button to open the download page. Download the plugin and import it to your project.

Configuring Facebook Audience Network

After importing Facebook Audience Network plugin, the **FACEBOOK AUDIENCE NETWORK** section will be updated as below.



Default Placement

Here you can enter the ad IDs to be used with the default placement for each platform. These are the only ad IDs required if you are not using any custom placements in your app. Note that you only need to provide IDs for the ad types you want to use, e.g. if you only use Audience Network banner ads you can leave the interstitial and rewarded

ad IDs empty.

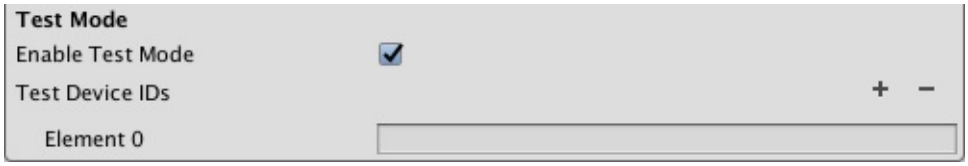
> To create Audience Network ad units and get the IDs, follow the instructions here.

Custom Placements

Here you can optionally enter the ad IDs associated with non-default ad placements to be used in your app. You can have an arbitrary number of custom placements and can use built-in placements or create new placements for your needs.

Test Mode

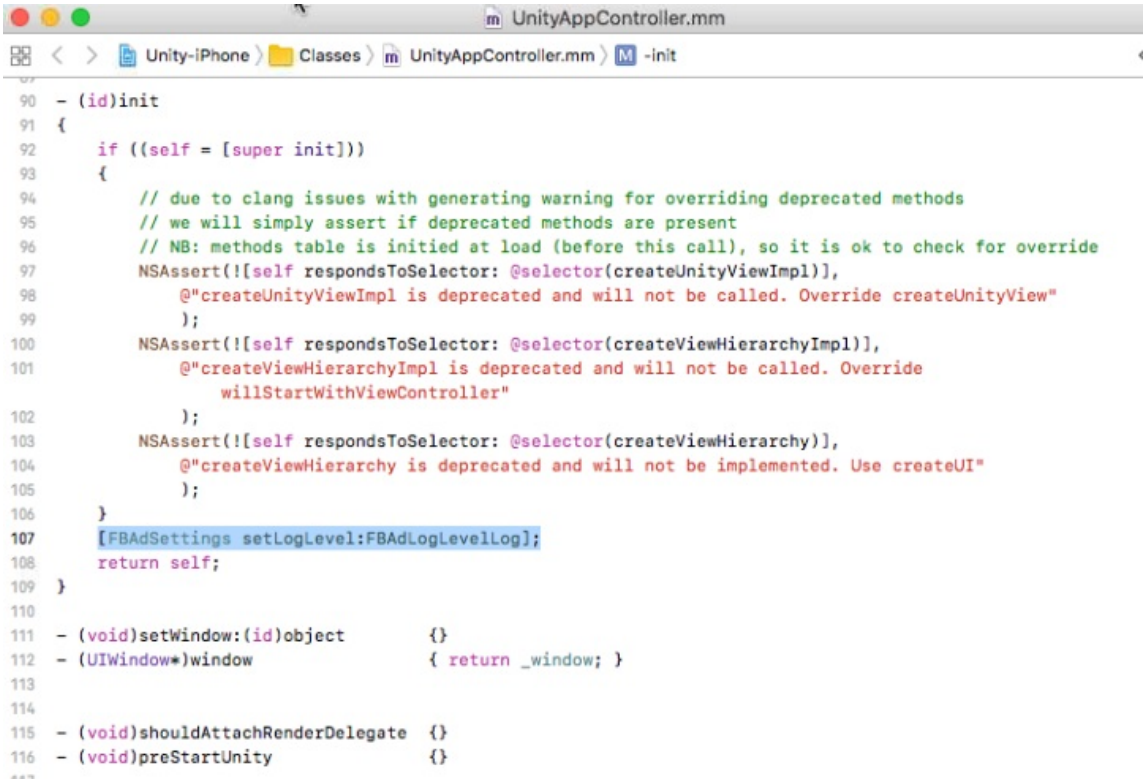
To enable test mode, simply check the *Enable Test Mode* option and enter the test device IDs.



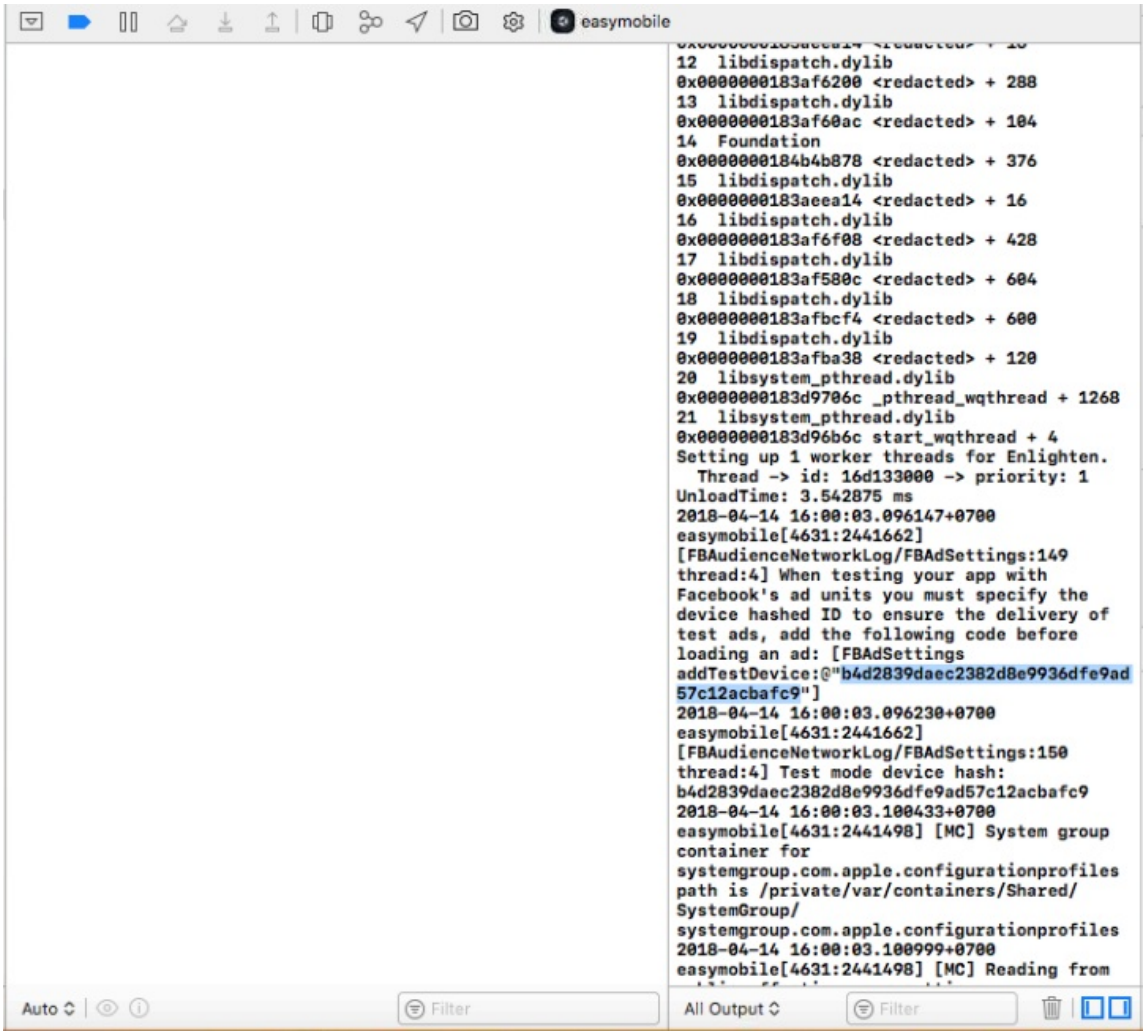
> Facebook Audience Network testing instructions can be found here.
>
> If your project is still in development, we strongly recommend setting up the testing mode properly to avoid "No fill" error, especially on iOS.
>
> **Android device ID**
>
> The device ID is printed in the device logcat, you can follow the instructions in the Setup AdMob/Test Mode section to find it. Note that you will need to import the Google Mobile Ads plugin, which is quite awkward if you don't use it in your project. However the Audience Network documentation is lacking on this detail and using the AdMob plugin is the simplest workaround we know for now.
>
> **iOS device ID**
>
> Follow these steps to find the device ID:
>
> - In Unity, build your project for IOS platform.
> - Open the generated project in XCode.
> - Open the UnityAppController.mm file.
> - Import the FBAudienceNetwork header into that file.

```
#import <CoreGraphics/CoreGraphics.h>
#import <QuartzCore/QuartzCore.h>
#import <QuartzCore/CADisplayLink.h>
#import <Availability.h>

#import <FBAudienceNetwork/FBAudienceNetwork.h>

#import <OpenGLES/EAGL.h>
#import <OpenGLES/EAGLDrawable.h>
#import <OpenGLES/ES2/gl.h>
#import <OpenGLES/ES2/glext.h>
```

> - Add this line [FBAdSettings setLogLevel:FBAdLogLevelLog]; into the *init* method, before the return statement.



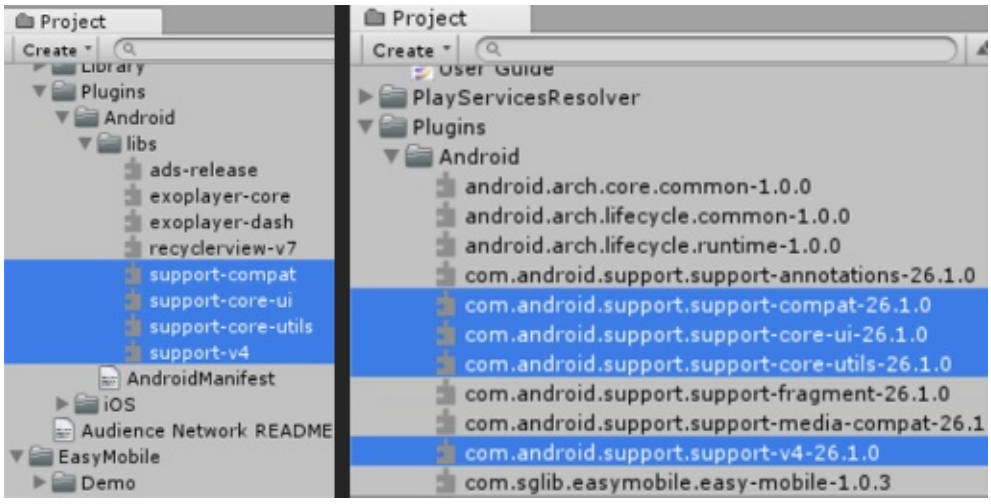
- Run the app on your device and find the ID in the XCode console.



Building Notes

Please follow build instructions here: Android, iOS.

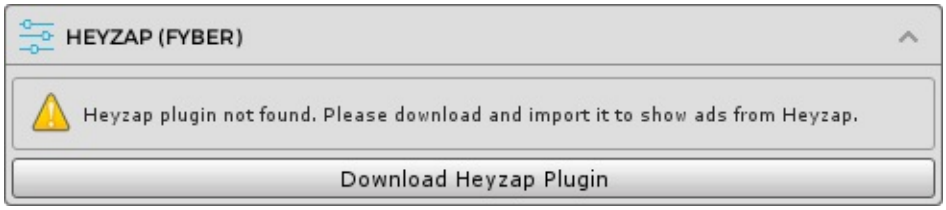
> On Android, if you encounter the "Unable to convert classes into dex format" issue when building your game, there might be duplicated files in your project and you need to delete the *support-xxx* files in the *AudienceNetwork* folder.



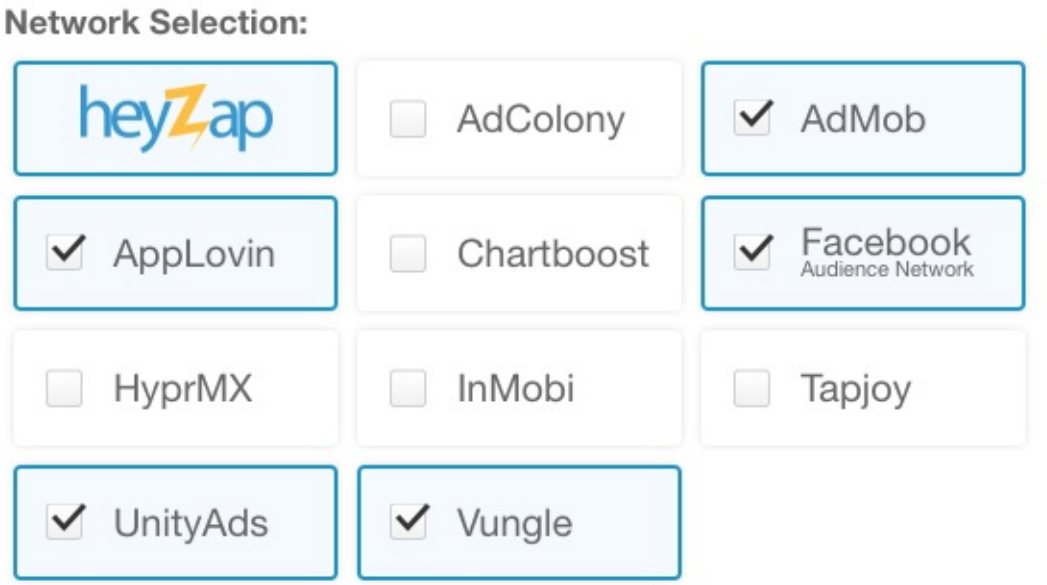
Advertising: Settings | Setup Heyzap

Importing Heyzap Plugin

To show ads from Heyzap you need to import the Heyzap plugin for Unity. In the **HEYZAP** section, click the *Download Heyzap Plugin* button to open the download page.



In the download page select your preferred networks to use with Heyzap mediation. The Heyzap dynamic documentation will update automatically to reflect your selections.

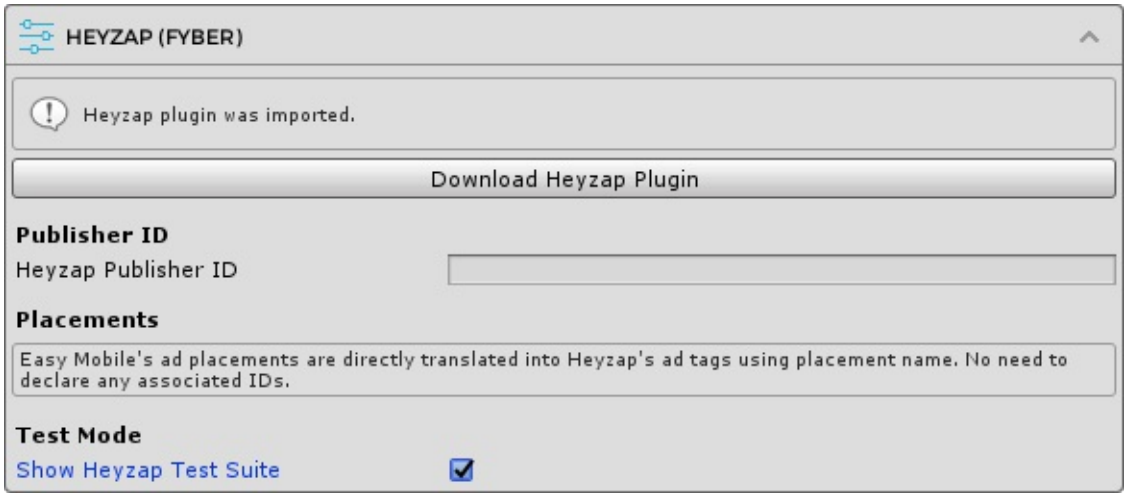


Follow the instructions provided by Heyzap to download and import its plugin as well as other required 3rd-party plugins. Also go through the **Integration Notes** section below to avoid problems that may occur during the integration of 3rd-party networks.

> If you haven't already, use Heyzap's Integration Wizard to setup the 3rd-party networks to use with mediation.

Configuring Heyzap

After importing Heyzap plugin, the **HEYZAP** section will be updated as below.



Publisher ID

Here you can enter the required publisher ID to the *Heyzap Publisher ID* field. This ID can be found in the *Account Details* page in the Heyzap dashboard.

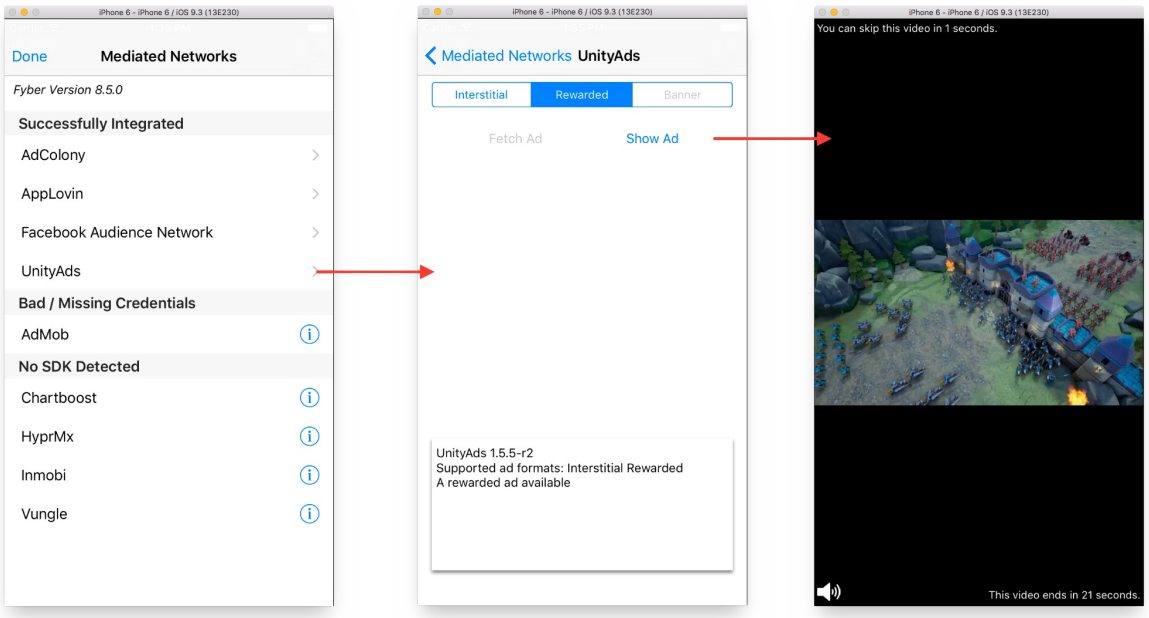
Placements

An Easy Mobile's ad placement will be directly translated into a Heyzap's ad tag that is same to the placement name in runtime. Therefore if you want to specify a tag when showing a Heyzap ad, simply show the ad with an ad placement whose name is the tag you want to use. There's no need to declare any ad IDs to be associated with the placements.

Test Mode

The Heyzap plugin comes with a convenient Test Suite that you can use to test the operation of each mediation network. To use this Test Suite, simply check the *Show Heyzap Test Suite* option in the **Test Mode** section.

Below is the Test Suite interface on iOS (it's similar on Android).



Mediation Notes

This section discusses some notes that you should take when using Heyzap mediation with various other networks.

> If you use Heyzap's mediation feature with other networks (AdColony, AdMob, Chartboost, etc.), you should not import the standalone-plugins of those networks, to avoid potential conflicts. Instead, import their corresponding adapter packages provided at the Heyzap download page.

Facebook Audience Network (Android-specific)

The Facebook Audience Network package contains an *android-support-v4.jar _file under Assets/Plugins/Android* folder. If you project already contains a *support-v4-xx.x.x.aar* file under that same folder, feel free to remove (or exclude it when importing) the jar file or it will cause the "Unable to convert dex..." error when building due to duplicate libraries.

AppLovin (Android-specific)

As instructed in the Heyzap documentation, you need to add the AppLovin SDK key to its AndroidManifest.xml file located at *Assets/Plugins/Android/AppLovin* folder. Simply add the following line inside the <application> tag in the manifest, replacing YOUR_SDK_KEY with your actual AppLovin SDK key.

```
<meta-data android:name="applovin.sdk.key" android:value="YOUR_SDK_KEY"/>
```

This manifest also includes the READ_PHONE_STATE permission, which requires your app to have a privacy policy when uploaded to Google Play. This permission is not mandatory to run the AppLovin SDK, therefore you can safely remove it if you are not ready to provide the required privacy policy. To remove the permission, simply delete the corresponding line from the manifest or comment it out as below.

```
<!-- Exclude the READ_PHONE_STATE permission because it requires a privacy policy -->
<!-- <uses-permission android:name="android.permission.READ_PHONE_STATE" /> -->
```

The minSdkVersion Problem (Android-specific)

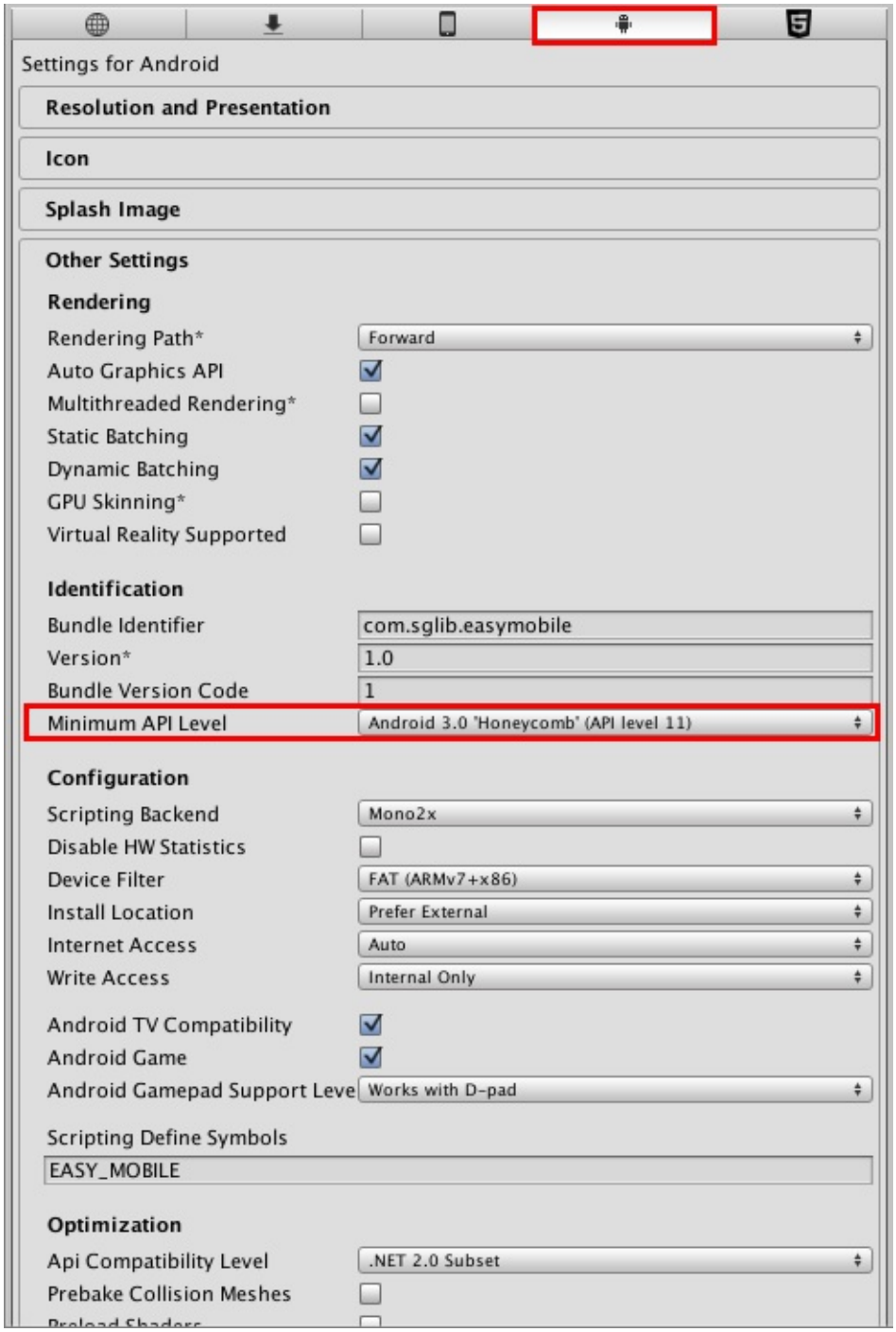
The current Heyzap SDK requires a minSdkVersion of 10, while some other 3rd-party plugins may require a version of 11 or above. If you get a build error including this line

```
Unable to merge android manifests...
```

and this line

```
Main manifest has <uses-sdk android:minSdkVersion='x'> but library uses minSdkVersion='y'
```

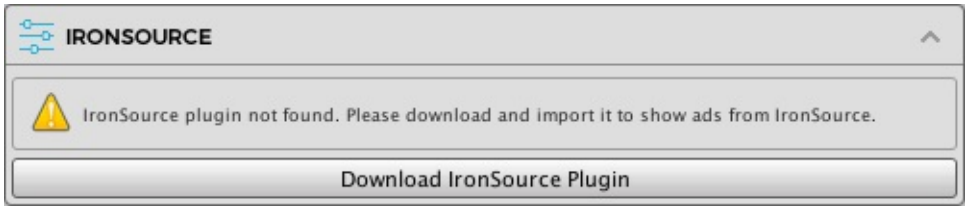
where x < y, it means you need to increase the minSdkVersion of the app. To do so go to *Edit* > *Project Settings* > *Player*, then select the *Android settings* tab and increase its *Minimum API Level* to the required one (which is 'y' in this example).


Settings for Android
Resolution and Presentation
Icon
Splash Image
Other Settings
Rendering
Rendering Path*
Forward
Auto Graphics API
Multithreaded Rendering*
Static Batching
Dynamic Batching
GPU Skinning*
Virtual Reality Supported
Identification
Bundle Identifier
com.sglib.easymobile
Version*
1.0
Bundle Version Code
1
Minimum API Level
Android 3.0 'Honeycomb' (API level 11)
Configuration
Scripting Backend
Mono2x
Disable HW Statistics
Device Filter
FAT (ARMv7+x86)
Install Location
Prefer External
Internet Access
Auto
Write Access
Internal Only
Android TV Compatibility
Android Game
Android Gamepad Support Leve
Works with D-pad
Scripting Define Symbols
EASY_MOBILE
Optimization
Api Compatibility Level
.NET 2.0 Subset
Prebake Collision Meshes

Advertising: Settings | Setup ironSource

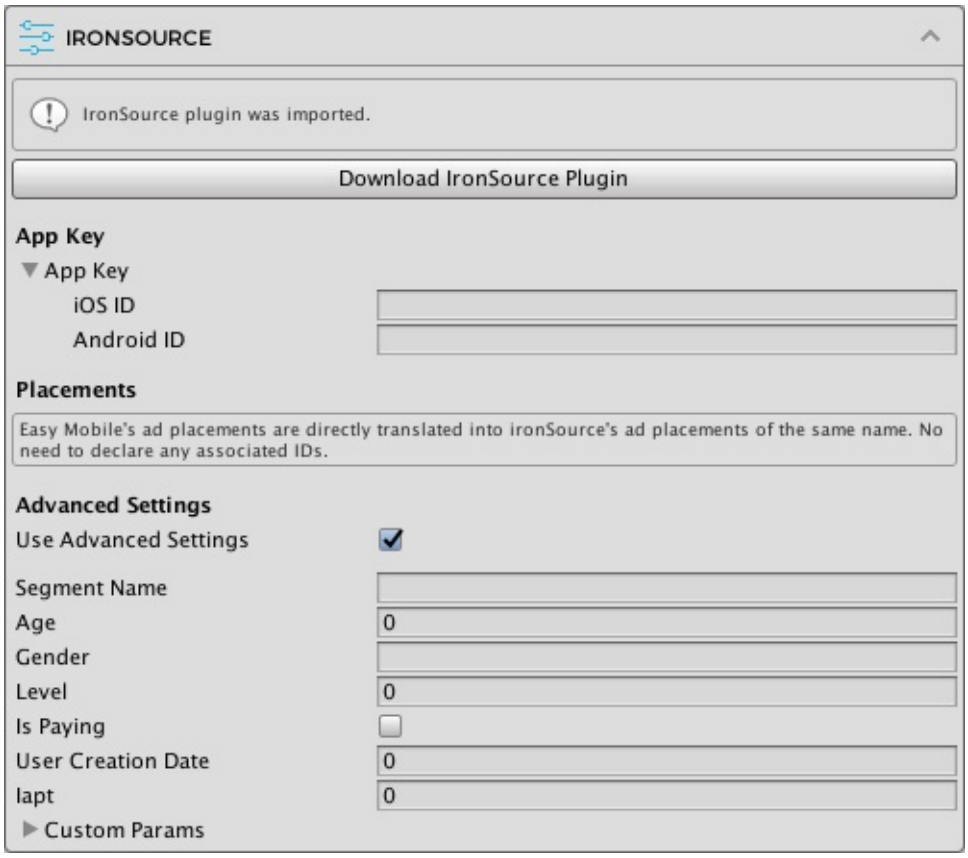
Importing ironSource Plugin

To show ads from IronSource you need to import the IronSource plugin. In the **IRONSOURCE** section, click the *Download IronSource Plugin* button to to open the download page. Download the plugin and import it to your project.



Configuring ironSource

After importing the ironSource plugin, the **IRONSOURCE SETUP** section will be updated as below.



App Key

First you need to provide the app key for each platform.

> Go to the ironSource dashboard to setup your app and get the key. Then follow these instructions to setup mediation networks for your project.

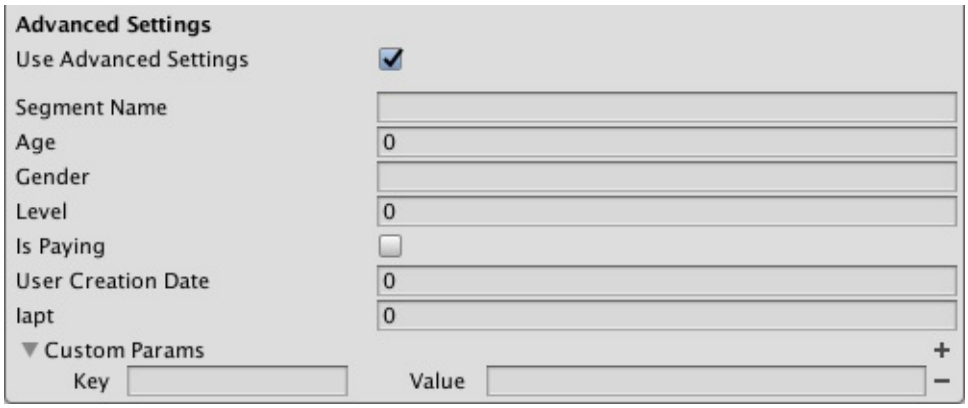
Placements

Easy Mobile's ad placements will be directly translated into ironSource's placements of the same name in runtime. If you want to specify a non-default placement when showing an ironSource ad, simply show the ad with one of Easy Mobile's built-in placements or create a new placement for your needs. There's no need to declare any ad IDs to be associated with the placements.

> Note that the placements you use with Easy Mobile must also exist on ironSource dashboard, otherwise you have to create them on the dashboard before using. If you show an ad with an undefined placement (the placement doesn't exist on ironSource dashboard) the ad will be shown *with the default placement* but corresponding ad event handlers will receive the undefined placement as argument, which may cause unexpected behaviors.

Advanced Settings

ironSource has an additional feature called segment to help you serve ads that target a specific audience, further information can be found here. To enable the advanced settings, simply check the *Use advanced settings* option and enter required the fields.



Adding IronSourceEventsPrefab

It's important to add the *IronSourceEventsPrefab* found at folder *Asset/IronSource/Prefabs* to the **first** scene in your app, otherwise ironSource ad events won't be received properly.

Mediation Setup Notes

In order to integrate 3rd-party mediation networks you need to perform two steps.

1. Set up the networks on the ironSource dashboard, follow these instructions.
2. Go to this page and download the mediation adapters that you want to integrate and import them into your Unity Project.

> The code to work with ironSource ads remains the same whether you setup mediation or not. All mediating works are done by the ironSource SDK and the adapters.

AdMob Network (Android-specific)

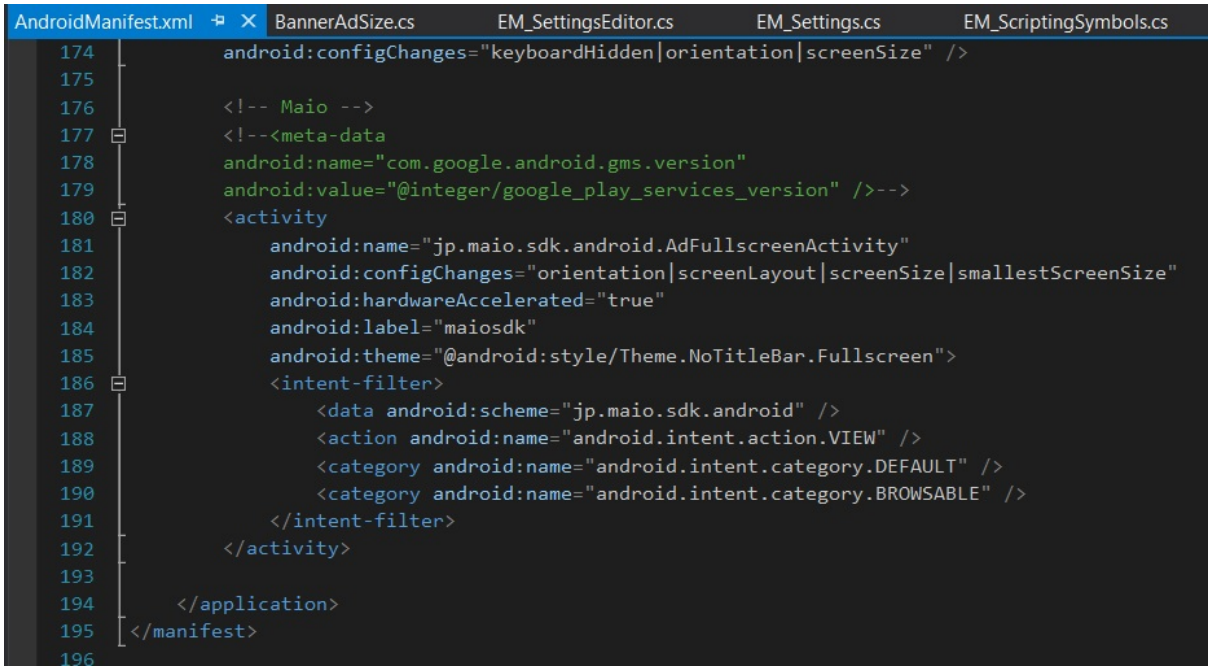
As of version 4.2.1 of the ironSource's adapter for AdMob, you may also need to import the Google Mobile Ads plugin into your project to make it work properly.

Building Notes

Android

The ironSource's instructions for Android can be found here.

If you encounter the "*No resource found that matches the given name (at 'value' with value '@integer/google_play_services_version')"* error, comment out the following lines (177, 178, 179) in the ironSource's AndroidManifest that can be found at folder *Assets/Plugin/Android/IronSource/AndroidManifest.xml*.



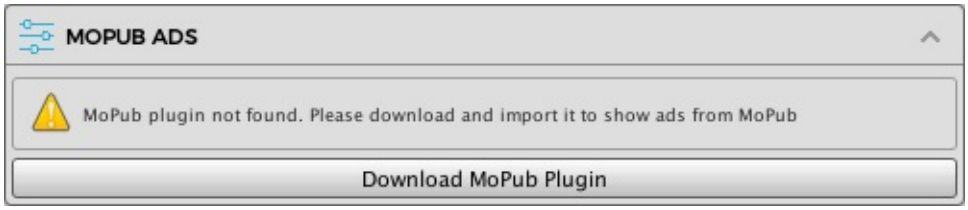
iOS

The ironsource's intructions for iOS can be found here.

Advertising: Settings | Setup MoPub

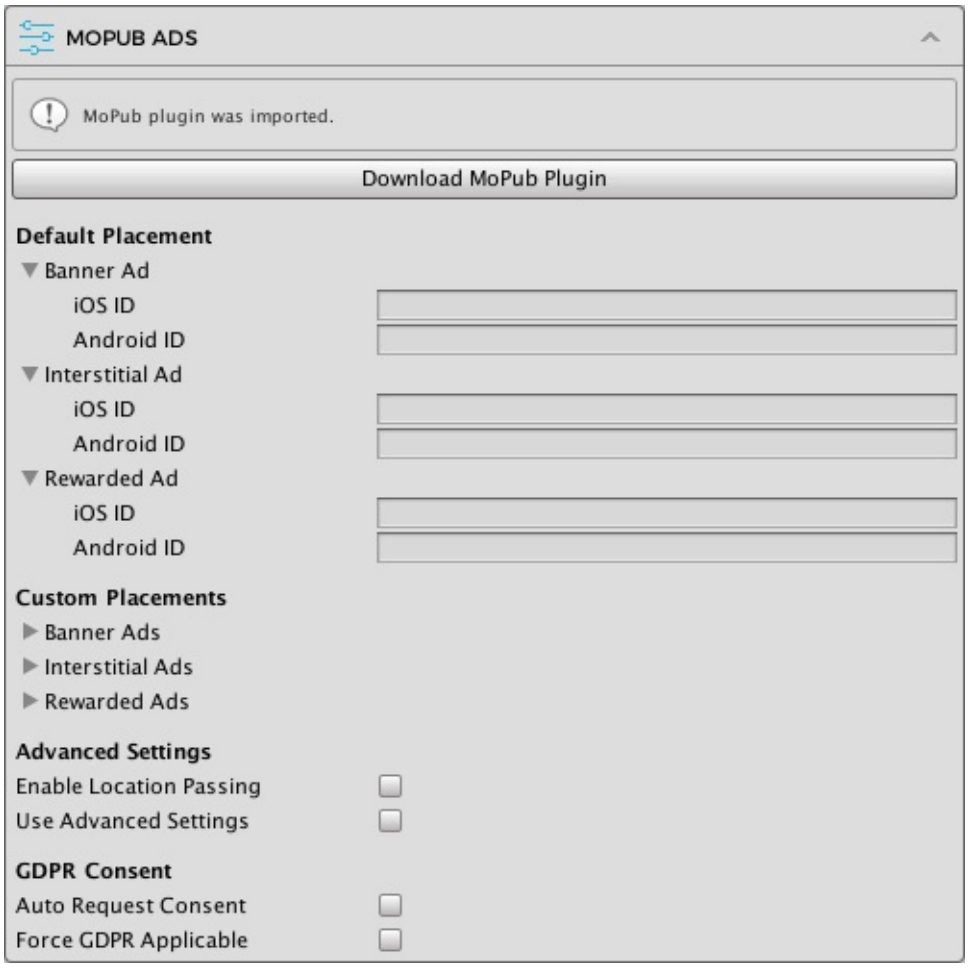
Importing MoPub Plugin

To show ads from MoPub you need to import the MoPub plugin for Unity. In the **MOPUB ADS** section, click the *Download MoPub Plugin* button to open the download page.



Configuring MoPub

After importing the MoPub plugin, the **MOPUB ADS** section will be updated as below.



Default Placement

Here you can enter the ad IDs to be used with the default placement for each platform. These are the only ad IDs required if you are not using any custom placements in your app. Note that you only need to provide IDs for the ad types you want to use, e.g. if you only use MoPub banner ads you can leave the interstitial and rewarded ad IDs empty.

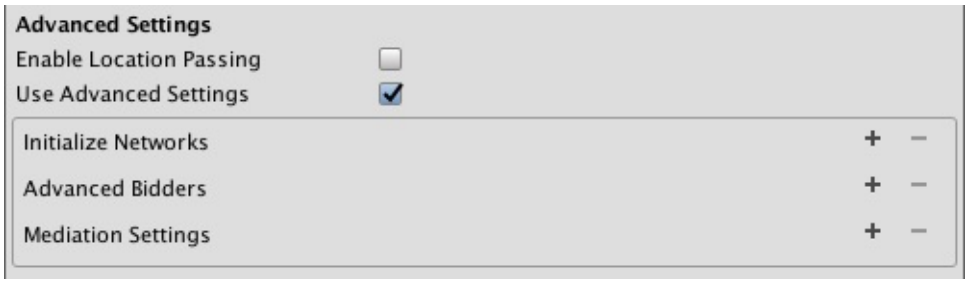
> If you're not familiar with MoPub, please follow the instructions here to setup ad units for your app.

Custom Placements

Here you can optionally enter the ad IDs associated with non-default ad placements to be used in your app. You can have an arbitrary number of custom placements and can use built-in placements or create new placements for your needs.

Advanced Settings

- *Enable Location Passing*: check this if you want to enable location support for banners & interstitials.
- *Use Advanced Setting*: check this to enable MoPub's advanced settings, including initialization with custom configurations, which can read more about here.



GDPR Consent

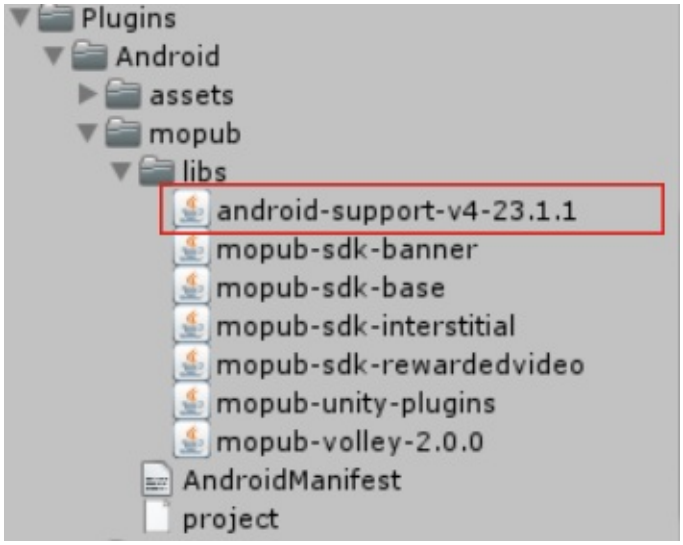
> These settings are only useful if you want to use the GDPR support features provided by MoPub (and thus, specific to MoPub only). If you're using the consent management system and consent dialog provided by Easy Mobile to manage consent for the whole app you should ignore these settings.

- *Auto Request Consent*: check this box if you want the Mopub's GDPR consent dialog to show automatically during initialization.
- *Force GDPR Applicable*: enable this to show the MoPub's GPDR consent dialog in all regions (useful for debugging in development).

Building Notes

Android

MoPub plugin manually includes android-support-v4 jar, which is also required by EasyMobile (and many other plugins) and will be fetched automatically by the Google Play Services Resolver, therefore you need to remove the duplicated file before starting building to avoid errors due to duplication. Navigate to Plugins/Android/mopub/libs and delete the files there.



iOS

MoPub plugin requires iOS 8.0+ and Xcode 9.0+.

Mediation Notes

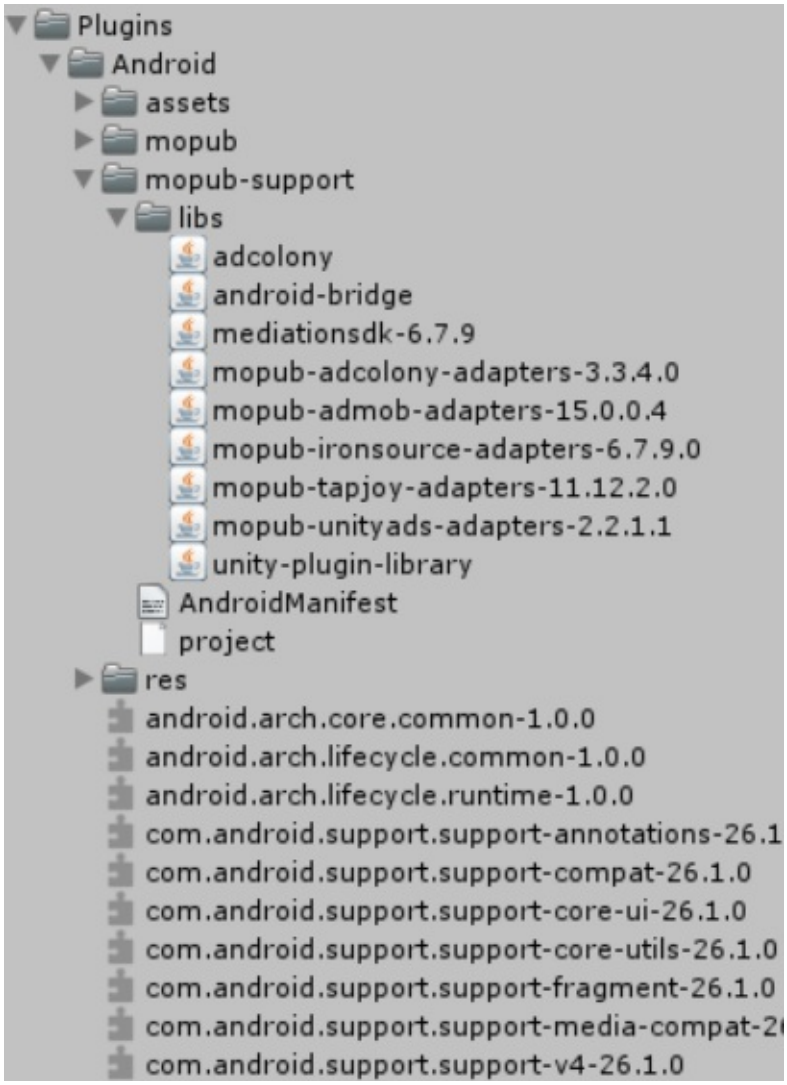
> New mediation networks need to be configured and enabled correctly in the dashboard before you can show them in your app. Follow this instruction to create and setup new networks.

In order to integrate a 3rd-party network, you will need to import that network Unity plugin (you can also use its native plugin, but it can take some extra work, especially on iOS, so it's not recommended) and the MoPub adapter (which can be download here).

Android

Place the adapter .jar files in Plugins/Android/mopub-support/libs folder (directly, not in any subfolder).

If the 3rd-party network plugin uses .jar files, place them in Plugins/Android/mopub-support/libs. If they're .aar files, place them in Plugins/Android. For example:



> There may be conflicts between the AndroidManifest of these networks with the one of Mopub, causing build errors. In such case you should update the AndroidManifest of those networks to solve the conflicts while maintaining the one of Mopub.

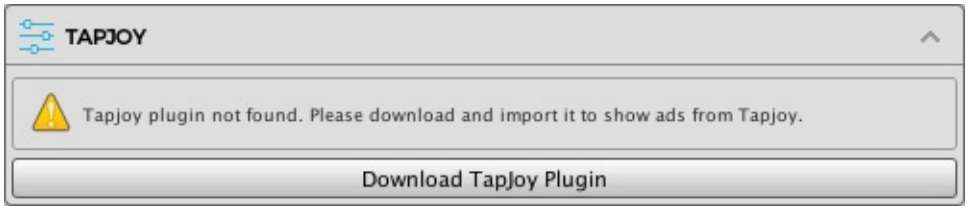
iOS

Just keep all the imported files where they are.

Advertising: Settings | Setup Tapjoy

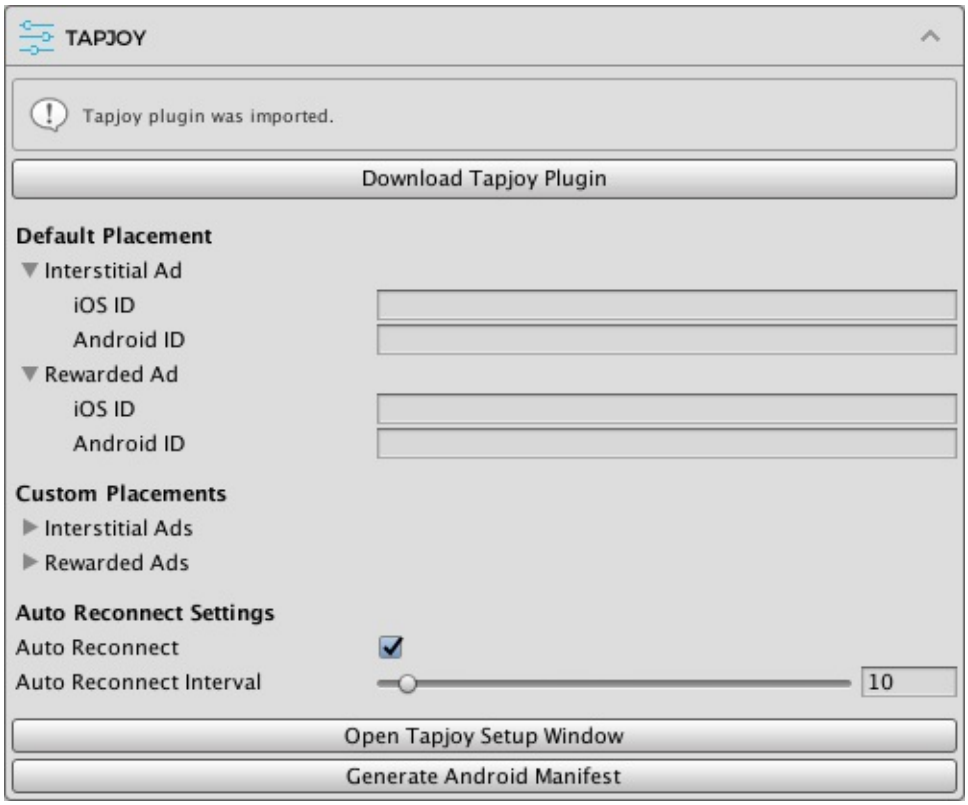
Importing Tapjoy Plugin

To show ads from Tapjoy you need to import the Tapjoy Unity Plugin. In the **TAPJOY** section, click the *Download TapJoy Plugin* button to to open the download page. Download the plugin and import it to your project.



Configuring Tapjoy

After importing the Tapjoy plugin, the **TAPJOY** section will be updated as below.



> Before configuring Tapjoy placements in Easy Mobile settings, you have to create them on the Tapjoy dashboard. If you're not familiar with the process, follow the instruction here. Basically, you'll need to:
>
> - Go to TapJoy dashboard and create a new app.
> - Setup at least one virtual currency.
> - Create some contents to show in your app.
> - Create some placements and assign content for them.

Default Placement

Here you can configure the default placement for each platform. Note that in the ID fields you would enter the *name* of the placements created on the Tapjoy dashboard whose content is relevant to the corresponding ad type (interstitial, rewarded). You only need to provide placement names for the ad types you want to use, e.g. if you only use Tapjoy rewarded ads you can leave the interstitial ad fields empty.

Custom Placements

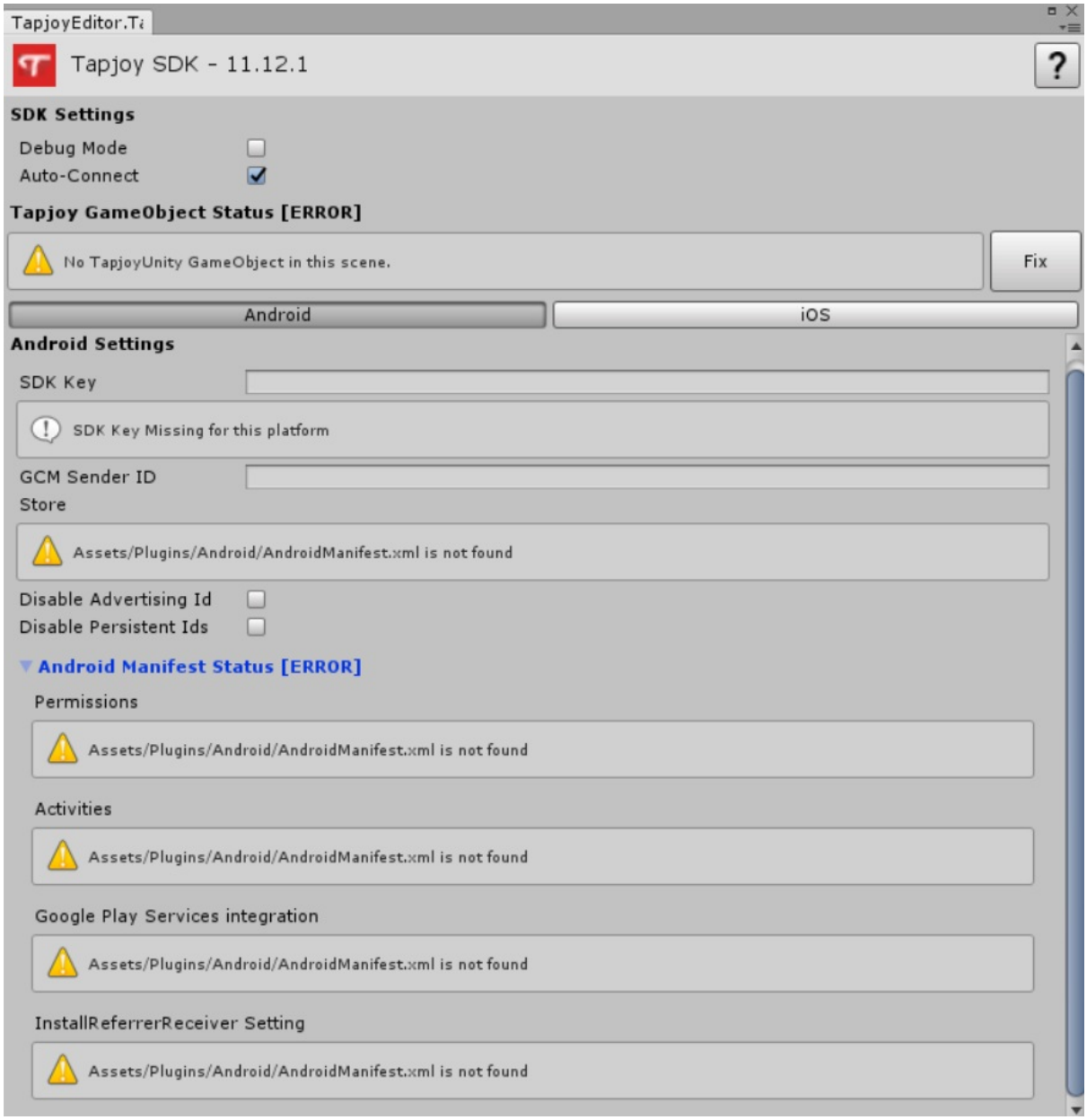
Here you can optionally enter the Tapjoy placement names associated with non-default placements to be used in your app. You can have an arbitrary number of custom placements and can use built-in placements or create new placements for your needs.

Auto Reconnect Settings

- *Auto Reconnect*: should we reconnect to the server automatically until connected. If you uncheck it, players might never be able to connect to the Tapjoy server if they opened your app when their device is offline.
- *Auto Reconnect Interval*: auto reconnect coroutine refresh rate.

Tapjoy Setup Interface

Click the *Open TapJoy Setup Window* button to open the Tapjoy's setup window (you can also navigate to the menu Window > Tapjoy and open it). A window will be opened as below.



- Tapjoy requires a TapjoyUnity GameObject with some components attached to it. The _Fix _button right next to the "No TapjoyUnity GameObject in this scene" warning label can create that GameObject for you, make sure to add it in the scene that appears **first** in your game.
- Enter the SDK Key for each platform.
- Make sure the *Auto-Connect* field is checked.

> More details about these settings can be found here.

Building Notes

Android

- Tapjoy needs Google Play Services library to work, but starting from version 11.10.1, the Tapjoy plugin for Unity no longer includes it in the package, so make sure to add it into your game (you can read about this here).
- You need to make sure your game has a valid **AndroidManifest.xml** placed under Assets/Plugins/Android (read more about it here).

- If there is no AndroidManifest.xml in that folder, you will see some *"not found"* warning labels like the image above. Simply click the *Generate Android Manifest* in the **TAPJOY** section, a valid file will be created automatically.
- If you already have a AndroidManifest in the folder, but some elements are missing, you can at them via Tapjoy's setup window. Click all the fix button there (see the image below) and Tapjoy will do the job automatically.

iOS

After exporting your project to XCode, add the following into your Info.plist file (read more about it here).

```
<key>NSAppTransportSecurity</key>
<dict>
  <!--To support localhost -->
  <key>NSAllowsLocalNetworking</key>
  <true/>
  <!--To continue to work for iOS 9 -->
  <key>NSAllowsArbitraryLoads</key>
  <true/>
</dict>
```

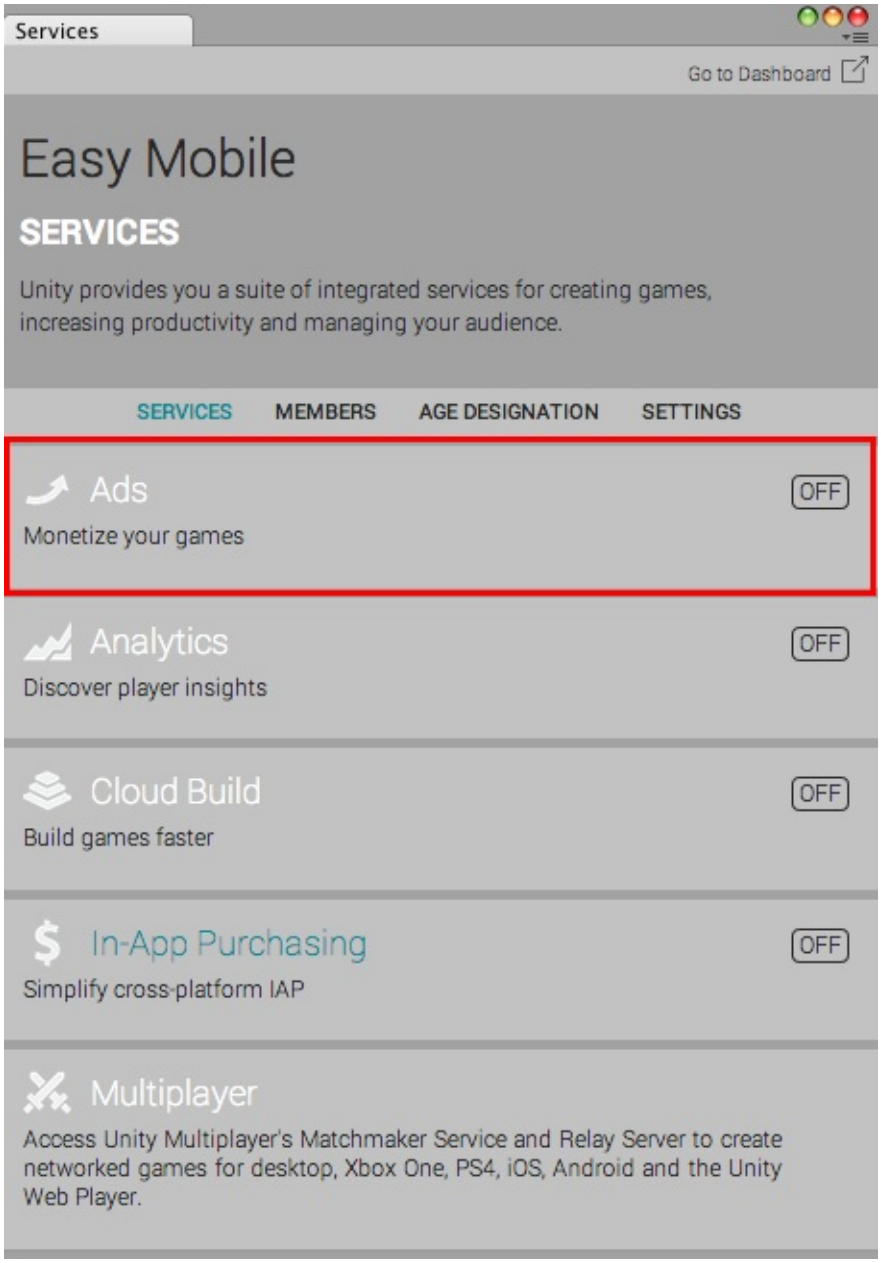
Advertising: Settings | Setup Unity Ads

Enabling Unity Ads Service

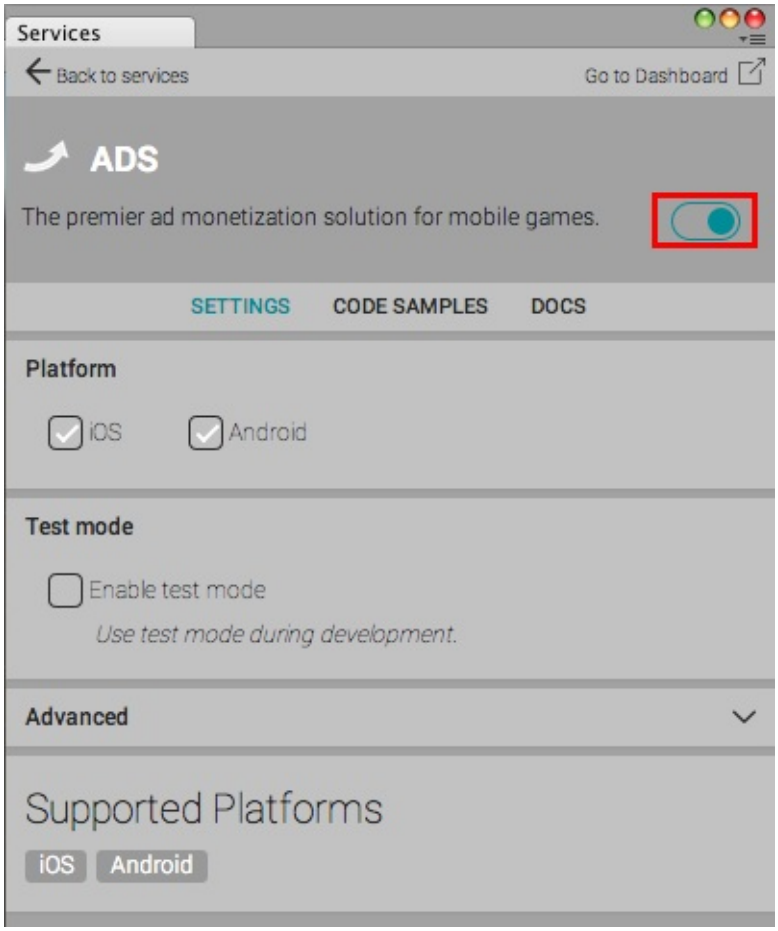
> To use Unity Ads service, you must first set up your project for Unity Services.

To show ads from Unity Ads you need to enable the corresponding service. Easy Mobile will automatically check for the service's availability and warn you to enable it if needed. Below is the **UNITY ADS** section when Unity Ads is not enabled.

To enable Unity Ads switch the platform to iOS or Android, then go to *Window* > *Services* and select the Ads tab.

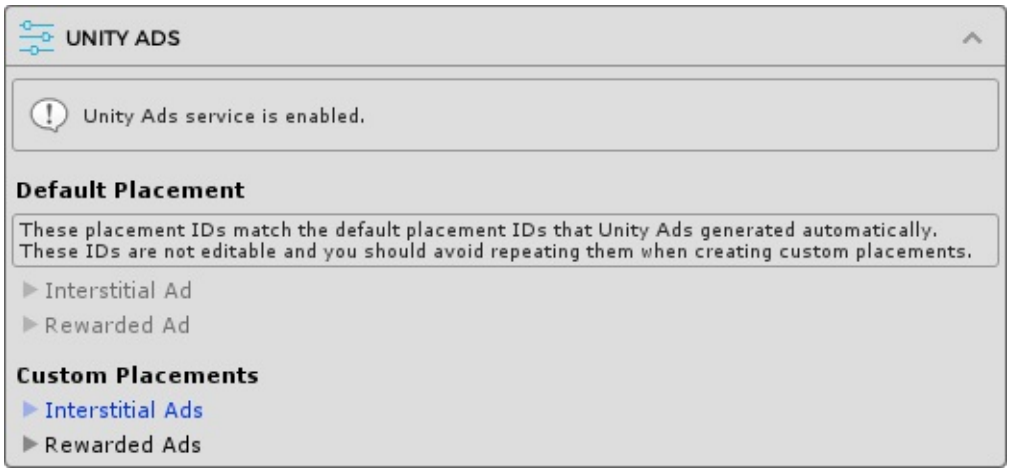


In the opened configuration window, click the toggle at the right-hand side to enable Unity Ads service. You may need to answer a few questions about your game.



Configuring Unity Ads

The **UNITY ADS** section will be updated after Unity Ads has been enabled.



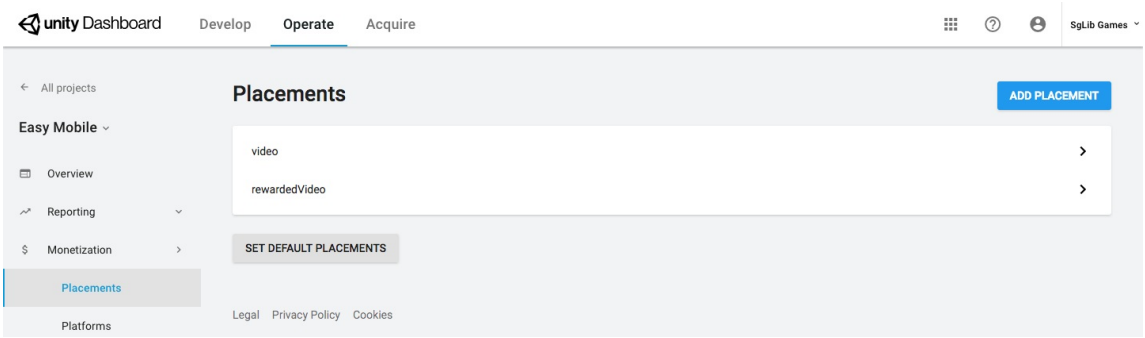
Default Placement

The ad IDs associated with the default placement are prefilled for you automatically. These IDs match the default placement IDs generated by the Unity Ads service when it is enabled and therefore *cannot be changed*.

Custom Placements

Here you can optionally enter the ad IDs associated with non-default ad placements to be used in your app. You can have an arbitrary number of custom placements and can use built-in placements or create new placements for your needs. Note that you must not reuse the default placement IDs with these custom placements.

> Before entering the ad IDs for custom placements in the **Custom Placements** section, you must create these placements in the *Unity Dashboard*. In the dashboard select your project, select the *Operate* tab, then in the left navigation menu select *Monetization* > *Placements* and click the *ADD PLACEMENT* button at the top right corner.



Testing Notes

It is advisable to enable the test mode of Unity Ads during development. This will ensure there's always an ad returned whenever requested. To enable test mode simple check the *Enable test mode* option in the Ads tab in the Services window.

> Remember to disable this test mode when creating your release build.

Advertising: Scripting

This section provides a guide to work with the Advertising module scripting API.

> You can access the Advertising module API via the Advertising class under the EasyMobile namespace.

Working with Consent

Module Consent

The following snippet shows how you grant, revoke or read the module-level consent of the Advertising module.

```
// Grants the module-level consent for the Advertising module.
Advertising.GrantDataPrivacyConsent();

// Revokes the module-level consent of the Advertising module.
Advertising.RevokeDataPrivacyConsent();

// Reads the current module-level consent of the Advertising module.
ConsentStatus moduleConsent = Advertising.DataPrivacyConsent;
```

Vendor Consent

The following snippet shows how you grant, revoke or read the vendor-level consent of an individual network in the Advertising module.

> In this example we use AdMob for demonstration purpose, the same code can be used for any other network.

```
// Grants the vendor-level consent for AdMob.
Advertising.GrantDataPrivacyConsent(AdNetwork.AdMob);

// Revokes the vendor-level consent of AdMob.
Advertising.RevokeDataPrivacyConsent(AdNetwork.AdMob);

// Reads the current vendor-level consent of AdMob.
ConsentStatus admobConsent = Advertising.GetDataPrivacyConsent(AdNetwork.AdMob);
```

Working with Banner Ads

To show a banner ad you need to specify its position using the *BannerAdPosition* enum. The banner will be displayed once it is loaded.

```
// Show banner ad
Advertising.ShowBannerAd(BannerAdPosition.Bottom);
```

To hide the current banner ad (it can be shown again later):

```
// Hide banner ad
Advertising.HideBannerAd();
```

To destroy the current banner ad (a new one will be created on the next banner ad showing):

```
// Destroy banner ad
Advertising.DestroyBannerAd();
```

Working with Interstitial Ads

The method to show an interstial ad requires it to be already loaded. Therefore you should check for the ad's availability before showing it.

```
// Check if interstitial ad is ready
bool isReady = Advertising.IsInterstitialAdReady();

// Show it if it's ready
if (isReady)
{
    Advertising.ShowInterstitialAd();
}
```

An *InterstitialAdCompleted* event will be fired whenever an interstitial ad is closed. You can listen to this event to take appropriate actions, e.g. resume the game.

```
// Subscribe to the event
void OnEnable()
{
    Advertising.InterstitialAdCompleted += InterstitialAdCompletedHandler;
}

// The event handler
void InterstitialAdCompletedHandler(InterstitialAdNetwork network, AdLocation location)
{
    Debug.Log("Interstitial ad has been closed.");
}

// Unsubscribe
void OnDisable()
{
    Advertising.InterstitialAdCompleted -= InterstitialAdCompletedHandler;
}
```

Working with Rewarded Ads

The method to show a rewarded ad requires it to be already loaded. Therefore you should check for the ad's availability before showing it.

```
// Check if rewarded ad is ready
bool isReady = Advertising.IsRewardedAdReady();

// Show it if it's ready
if (isReady)
{
    Advertising.ShowRewardedAd();
}
```

A *RewardedAdCompleted* event will be fired whenever a rewarded ad has completed. You should listen to this event to reward the user for watching the ad. Otherwise, a *RewardedAdSkipped* event will be fired if the ad is skipped before finishing (and the user therefore is not entitled to the reward).

```
// Subscribe to rewarded ad events
void OnEnable()
{
    Advertising.RewardedAdCompleted += RewardedAdCompletedHandler;
    Advertising.RewardedAdSkipped += RewardedAdSkippedHandler;
}
```

```
// Unsubscribe events
void OnDisable()
{
    Advertising.RewardedAdCompleted -= RewardedAdCompletedHandler;
    Advertising.RewardedAdSkipped -= RewardedAdSkippedHandler;
}

// Event handler called when a rewarded ad has completed
void RewardedAdCompletedHandler(RewardedAdNetwork network, AdLocation location)
{
    Debug.Log("Rewarded ad has completed. The user should be rewarded now.");
}

// Event handler called when a rewarded ad has been skipped
void RewardedAdSkippedHandler(RewardedAdNetwork network, AdLocation location)
{
    Debug.Log("Rewarded ad was skipped. The user should NOT be rewarded.");
}
```

Removing Ads

Removing Ads without Updating Consent

In some cases you need to remove/stop showing ads in your game, e.g. when the user purchases the "Remove Ads" product. To remove ads:

```
// Remove ads permanently
Advertising.RemoveAds();
```

The *RemoveAds* method will destroy the banner ad if one is being shown, and prevent future ads from being loaded and shown ***except rewarded ads***, since they are unobtrusive and only shown at the user discretion.

> Note that the RemoveAds method uses Unity's PlayerPrefs to store the ad removal status with no encryption/scrambling.

An *AdsRemoved* event will be fired after ads have been removed. You can listen to this event and take appropriate actions, e.g update the UI.

```
// Subscribe to the event
void OnEnable()
{
    Advertising.AdsRemoved += AdsRemovedHandler;
}

// The event handler
void AdsRemovedHandler()
{
    Debug.Log("Ads were removed.");

    // Unsubscribe
    Advertising.AdsRemoved -= AdsRemovedHandler;
}
```

You can also check at any time if ads were removed or not.

```
// Determine if ads were removed
bool isRemoved = Advertising.IsAdRemoved();
```

Removing Ads and Revoke Consents

Because rewarded ads are still available after removing ads (for good reason!), it may be desirable in some cases to also revoke all consent granted to the Advertising module as well as the individual network. For example, you may offer "Remove Ads" as an in-app purchase that the user can buy to remove intrusive ads (i.e. banner and interstitial ads) *and stop the app from collecting their personal data for advertising purpose.* In such case, you can call the *RemoveAds* and pass *true* to its *revokeConsents* parameter. This parameter is optional and default to false.

```
// Remove ads permanently and also revoke the consent
// of the Advertising module and all ad networks so that
// they stop collecting user data.
Advertising.RemoveAds(true);    // revokeConsents passed as true
```

Re-enabling Ads after Removal

Finally, you can also revoke the ad removing status and allow ads to be shown again.

```
// Revoke ad removing status and allow showing ads again
Advertising.ResetRemoveAds();
```

Manual Ad Loading

With the Automatic Ad Loading feature, you normally don't need to worry about loading ads. However, if you want to disable this feature and control the loading process yourself, you can do so with manual ad loading in script.

> It is advisable to load an ad as far in advance of showing it as possible to allow ample time for the ad to be loaded.

To load an interstitial ad:

```
// Load the default interstitial ad.
Advertising.LoadInterstitialAd();
```

To load a rewarded ad:

```
// Load the default rewarded ad.
Advertising.LoadRewardedAd();
```

Working with Non-Default Ads

Beside the default ads, you can also show ads from non-default networks at non-default placements, thus implementing a more sophisticated ad strategy in your app. Note that each method to load or show ad always has a variant that allows you to specify the target ad network and placement explicitly. For example there're 2 variants of the *LoadInterstitialAd* method. One takes no argument and loads the default interstitial ad. The other loads an interstitial ad from a specified network at an arbitrary placement.

> If you're using the "Load All Defined Placements" auto ad-loading mode, both default and non-default ads are loaded automatically so you never need to worry about loading ads.

If you have AdMob as the default interstitial ad network, you can load and show a non-default interstitial ad like below.

```
// This method shows an interstitial ad from the default network (i.e. AdMob in this example)
// at the default placement. Default ads are loaded automatically unless Automatic Ad Loading is disabled.
if (Advertising.IsInterstitialAdReady())
    Advertising.ShowInterstitialAd();
```

```
// If you're using the "Load All Defined Placements" auto ad-loading mode, non-default ads are loaded automatic
ally too.
// Here we show how you can manually load a non-default interstitial ad, for illustration purpose.
// In practice, you don't need to do this unless the auto ad-loading is disabled or set to "Load Default Ads" m
ode.
// In this example, we'll load and show an interstitial ad from AdMob at the custom placement Startup.
// (Note that an interstitial ad at the default placement but belongs to a non-default network
// would also be considered a non-default ad).
// You can, for example, associate this placement with a house ad unit to show cross-promotion ads
// for other apps in your portfolio, thus having a free cross-promotion system!
Advertising.LoadInterstitialAd(InterstitialAdNetwork.AdMob, AdPlacement.Startup);


// Checks if the AdMob interstitial ad at placement Startup is ready and shows it.
if (Advertising.IsInterstitialAdReady(InterstitialAdNetwork.AdMob, AdPlacement.Startup))
    Advertising.ShowInterstitialAd(InterstitialAdNetwork.AdMob, AdPlacement.Startup);


// Likewise, you can check if a non-default rewarded ad is ready and show it like below
// (assume that IronSource is not set as the default network for rewarded ads).
if (Advertising.IsRewardedAdReady(RewardedAdNetwork.IronSource, AdPlacement.HomeScreen))
    Advertising.ShowRewardedAd(RewardedAdNetwork.IronSource, AdPlacement.HomeScreen);
```

Ad Network Clients

You can access the underlaying clients for the supported ad networks using the corresponding properties of the *Advertising* class.

```
// AdColony client.
AdColonyClientImpl adcolonyClient = Advertising.AdColonyClient;

// AdMob client.
AdMobClientImpl admobClient = Advertising.AdMobClient;

// Chartboost client.
ChartboostClientImpl chartboostClient = Advertising.ChartboostClient;

// Facebook Audience Network client.
AudienceNetworkClientImpl fbanClient = Advertising.AudienceNetworkClient;

// Heyzap client.
HeyzapClientImpl heyzapClient = Advertising.HeyzapClient;

// ironSource client.
IronSourceClientImpl ironSrcClient = Advertising.IronSourceClient;

// MoPub client.
MoPubClientImpl mopubClient = Advertising.MoPubClient;

// Tapjoy client.
TapjoyClientImpl tapjoyClient = Advertising.TapjoyClient;

// Unity Ads client.
UnityAdsClientImpl unityAdsClient = Advertising.UnityAdsClient;
```

From these clients you can access network-specific events, e.g the *OnBannerAdClosed* event provided by the AdMob. Events like this are specific to a certain ad network and normally not available in other networks which is why they are not exposed in the *Advertising* class (the *Advertising* class only exposes a common subset of events that are provided by all ad networks to ensure a consistent behavior across all networks). Note that these network-specific events are only available if the corresponding plugin of that network has been imported to your project. You can use the scripting symbols in the below table to wrap codes that access these event to make sure they only get compiled when the plugin is available. These symbols are defined automatically by Easy Mobile when the corresponding ad plugin is imported, except UNITY_ADS which is defined by Unity when the service is enabled.

Ad Network	Symbol
AdColony	EM_ADCOLONY
AdMob (Google Mobile Ads)	EM_ADMOB
Chartboost	EM_CHARTBOOST
Facebook Audience Network	EM_FBAN
Heyzap (Fyber)	EM_HEYZAP
ironSource	EM_IRONSOURCE
MoPub	EM_MOPUB
Tapjoy	EM_TAPJOY
Unity Ads	UNITY_ADS (defined by Unity)

Advertising: PlayMaker Actions

The PlayMaker actions of the Advertising module are group in the category *Easy Mobile - Advertising* in the PlayMaker's Action Browser.

> Please refer to the AdvertisingDemo_PlayMaker scene in folder *Assets/EasyMobile/Demo/PlayMakerDemo/Modules* for an example on how these actions can be used.

Game Services: Introduction

The Game Services module helps you quickly implement leaderboards and achievements for your game. It works with the Game Center network on iOS and Google Play Games services on Android. Here're some highlights of this module:

- **Leverages official plugins**
  - This module is built on top of Unity's GameCenterPlatform on iOS and Google Play Games plugin on Android
  - GameCenterPlatform is one part of the UnityEngine itself while the other is the official Google Play Games plugin for Unity, so reliability and compatibility can be expected
- **Easy management of leaderboards and achievements**
  - Easy Mobile's custom editor features a friendly interface that help you easily add, edit or remove leaderboards and achievements

Game Services: Settings

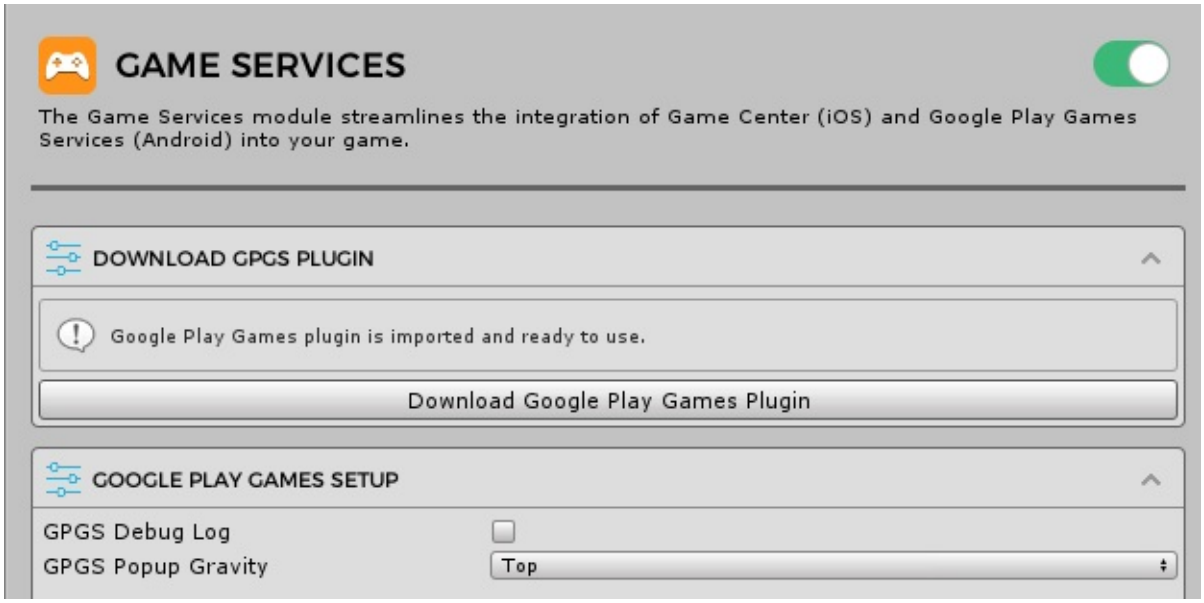
To use the Game Services module you must first enable it. Go to *Window* > *Easy Mobile* > *Settings*, select the Game Services tab, then click the right-hand side toggle to enable and start configuring the module.

Android-Specific Setup

Importing Google Play Games plugin for Unity

As stated earlier, this module is built on top of Google Play Games Plugin on Android. Therefore you need to import it to use the module on this platform. Easy Mobile will automatically detect the availability of the plugin and prompt you to import it if needed. Below is the module settings interface *after switching to Android platform* if the Google Play Games plugin hasn't been imported.

Click the *Download Google Play Games Plugin* button to open the download page, then download the package and import it to your project. Once the import completes the module interface will be updated and ready for you to start with the configuration.



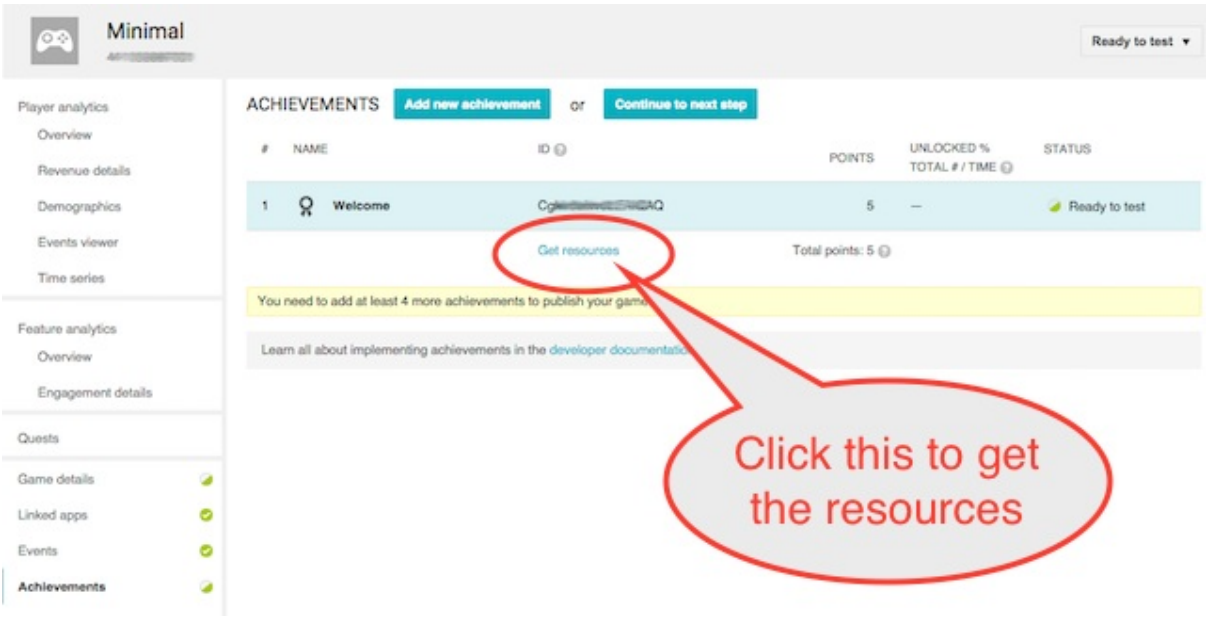
> Since we're not using Google Play Games plugin on iOS, the NO_GPGS symbol will be defined for iOS platform automatically after the plugin is imported in order to disable it.

Setup Google Play Games

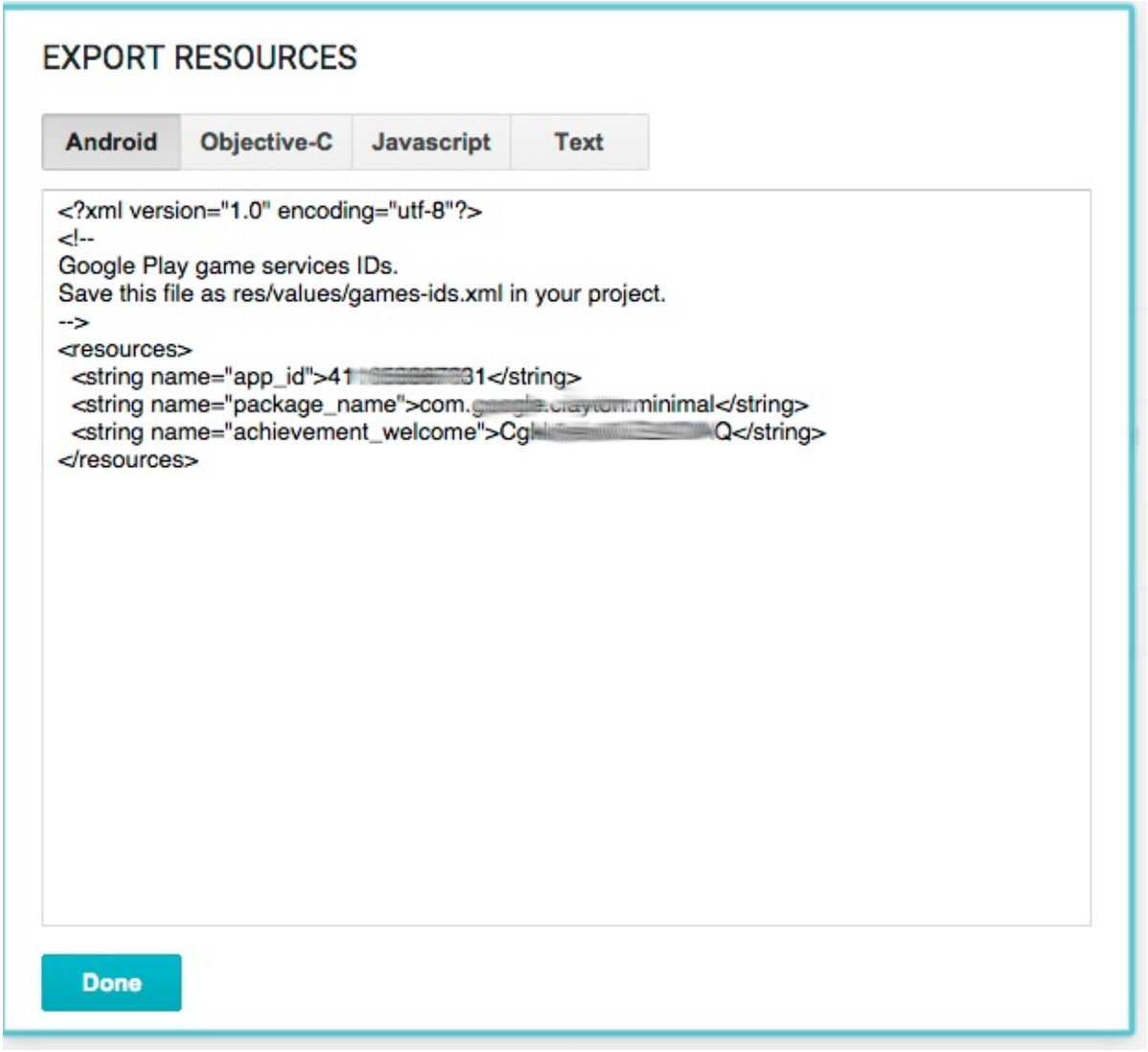
To setup Google Play Games plugin, you need to obtain the game resources from the Google Play Developer Console.

> The game resources are available after you configured your game on the Google Play Developer Console. If you're not familiar with the process, please follow the instructions on creating a client ID, as well as leaderboards and achievements.

To get the game resources, login to your Google Play Developer Console, select *Game services* tabs then select your game. Next go to the *Achievements* tab and click on the *Get Resources* label at the bottom of the list.

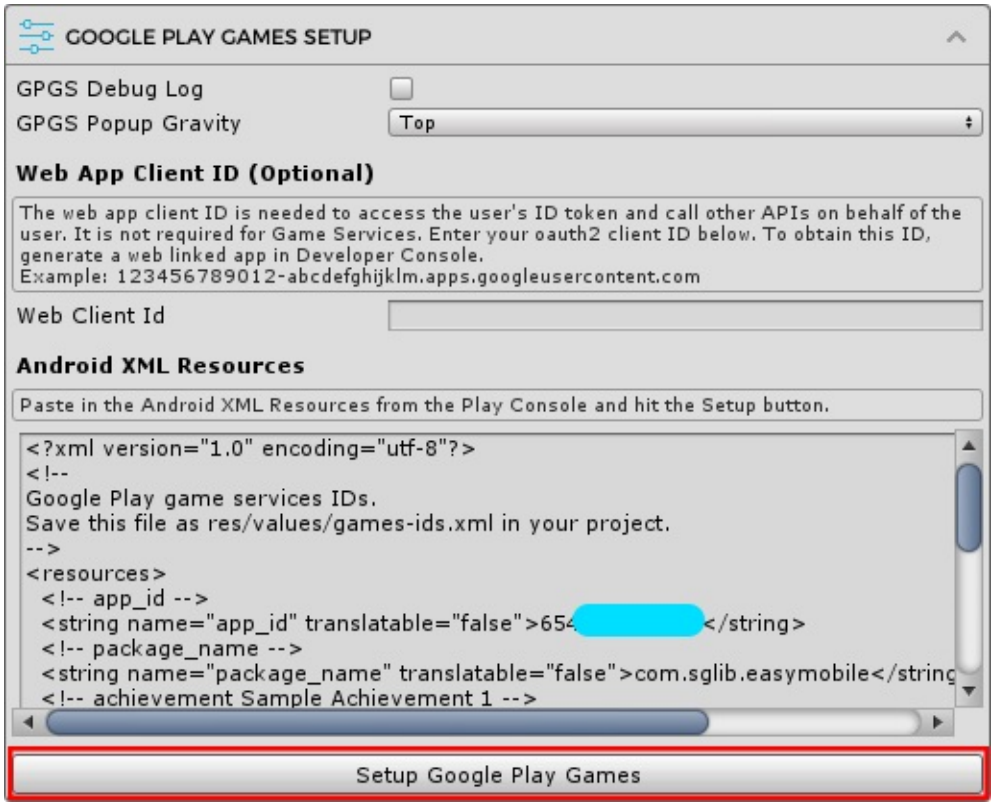


Copy all the xml content from the *Android* tab.



Go back to Unity, in the **GOOGLE PLAY GAMES SETUP** section, paste the obtained xml resouces into the **Android XML Resources** area, then click *Setup Google Play Games*.

> You can optionally provide a Web Client ID before setting up Google Play Games if needed.



After the setup has completed, a new file named EM_GPGSIds will be created at *Assets/EasyMobile/Generated*. This file contains the constants of the IDs of all the leaderboards and achievements in your Android game.

Within the **GOOGLE PLAY GAMES SETUP** section there're other settings including:

- *GPGS Debug Log*: check this to enable Google Play Games debug log.
- *GPGS Popup Gravity*: use this to control the position of Google Play Games popups (e.g. achievement popup).

Auto Initialization

Auto initialization is a feature of the Game Services module that initializes the service automatically when the module starts. Initialization is required before any other actions can be done, e.g. reporting scores.

During the initialization, the system will try to authenticate the user by presenting a login popup.

- On iOS, this popup will show up when the app gets focus (brought to foreground) for the first 3 times. If the user refuses to login all these 3 times, the OS will ignore subsequent authentication calls and stop presenting the login popup (to avoid disturbing the user). Otherwise, if the user has logged in successfully, future authentication will take place silently with no login popup presented.
- On Android, we employ a similar approach but you can configure the maximum number of authentication requests before ignoring subsequent ones.

You can configure the auto initialization feature within the **AUTO-INIT CONFIG** section.

- *Auto Init*: uncheck this option to disable the auto initialization feature, you can start the initialization manually from script (see the **Scripting** section)
- *Auto Init Delay*: how long after the module start that the initialization should take place
- *[Android] Max Login Requests*: maximum number of authentication requests allowed on Android, before ignoring subsequent ones (in case the user refuses to login)

> "Module start" refers to the moment the *Start* method of the module's associated MonoBehavior (attached to the EasyMobile prefab) runs.

Leaderboards & Achievements

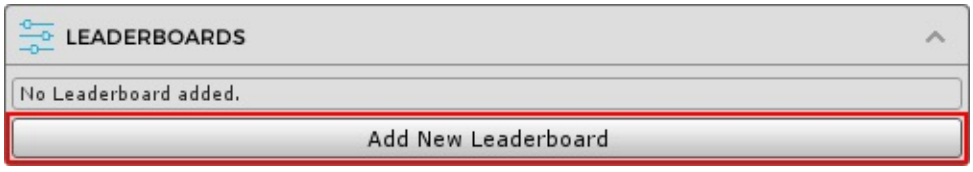
This section provides a guide to manage leaderboards and managements for your game.

> **Before You Begin**
>
> It is assumed that you already configured your game for the targeted gaming networks, i.e. Game Center and Google Play Games. If you're not familiar with the process, here're some useful links:
>
> - Configure for Google Play Games (Android)
> - Creating a Client ID for you game
> - Adding leaderboards
> - Adding achievements
> - Configure for Game Center (iOS)
> - Adding leaderboards and achievements in iTunes Connect

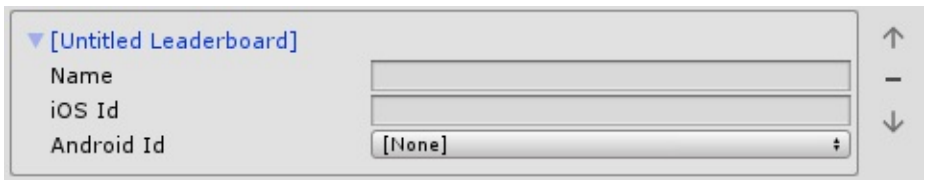
In the **LEADERBOARDS** and **ACHIEVEMENTS** you can add, edit or remove leaderboards and achievements.

Adding a New Leaderboard or Achievement

To add a new leaderboard click the *Add New Leaderboard* button (or *Add New Achievement* button in case of an achievement).



A new empty leaderboard (or achievement) will be added.



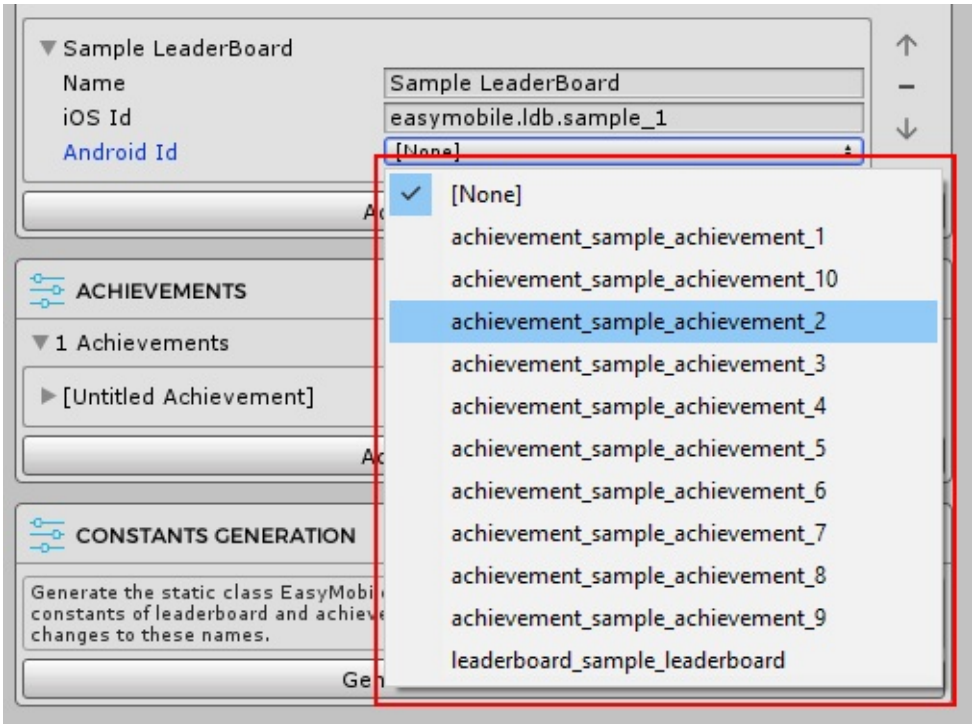
Fill in the required information of the leaderboard (or achievement):

- *Name*: the name of this leaderboard (or achievement), this name can be used when reporting scores to this

leaderboard (or unlocking this achievement)

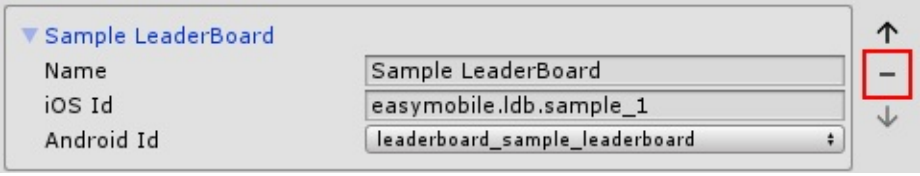
- *iOS Id*: the ID of this leaderboard (or achievement) as declared in iTunes Connect
- *Android Id*: the ID of this leaderboard (or achievement) as declared in Google Play Developer Console

> Google Play Games' leaderboards and achievements have generated IDs which can be difficult to memorize and cumbersome to copy-and-paste, especially if there are many of them. Thankfully, when you setup Google Play Games, the constants of these IDs are generated automatically (remember that EM_GPGSIds file?), allowing Easy Mobile to show a nice dropdown of all defined leaderboard and achievement IDs for you to choose from.



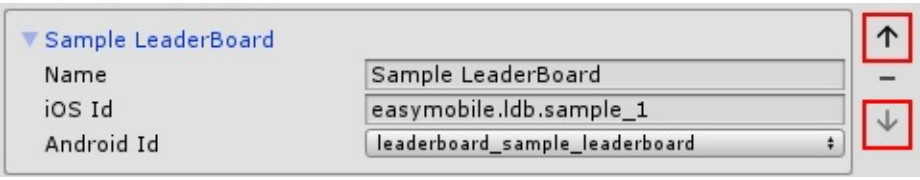
Removing a Leaderboard or Achievement

To remove a leaderboard (or achievement), simply click the [-] button at the right hand side.



Arranging Leaderboards or Achievements

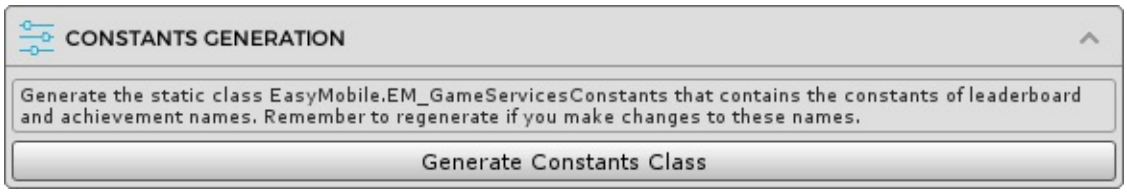
You can use the two arrow-up and arrow-down buttons to move a leaderboard (or achievement) upward or downward within its array.



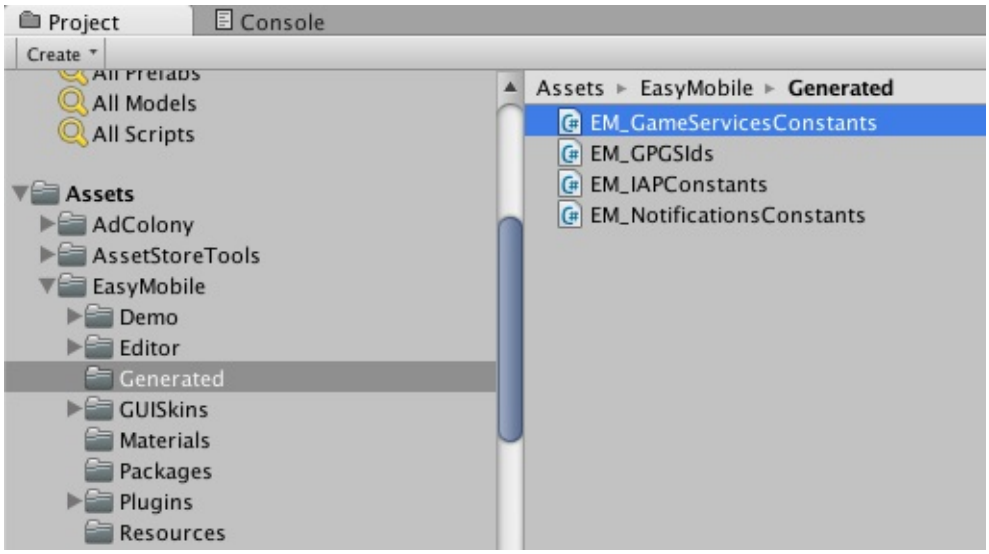
Constants Generation

Constants generation is a feature of the Game Services module. It reads the names of all the added leaderboards and achievements and generates a static class named EM_GameServicesConstants that contains the constants of these names. Later, you can use these constants when reporting scores to a leaderboard or unlocking an achievement in script instead of typing the names directly, thus help prevent runtime errors due to typos and the likes.

To generate the constants class (you should do this after adding all required leaderboards and achievements), click the *Generate Constants Class* button within the **CONSTANTS CLASS GENERATION** section.



When the process completes, a file named EM_GameServicesConstants will be created at *Assets/EasyMobile/Generated*.



Game Services: Scripting

This section provides a guide to work with the Game Services module scripting API.

> You can access the Game Services module API via the GameServices class under the EasyMobile namespace.

Initialization

Initialization is required before any other action, e.g. reporting scores, can be done. It should only be done once when the app is loaded. If you have enabled the Auto initialization feature, you don't need to initialize in script (see **Auto Initialization** section). Otherwise, if you choose to disable that feature, you can start the initialization in a couple of ways.

- Managed initialization: this method respects the *Max Login Requests* value on Android (see **Auto Initialization** section), which means it will ignore all subsequent calls once the user has dismissed the login popup for a number of time determined by *Max Login Requests*
- Unmanaged initialization: this method simply initializes the module, on Android it shows the login popup every time as long as the user hasn't been authenticated

> On iOS, the system automatically limits the maximum number of login requests to 3 no matter which method is used.

To use the managed initialization method:

```
// Managed init respects the Max Login Requests value
GameServices.ManagedInit();
```

To use the unmanaged initialization method:

```
// Unmanaged init
GameServices.Init();
```

Note that the initialization should be done early and only once, e.g. you can put it in the *Start* method of a MonoBehaviour, preferably a singleton one so that it won't run again when the scene reloads.

```
// Initialization in the Start method of a MonoBehaviour script
void Start()
{
    // Managed init respects the Max Login Requests value
    GameServices.ManagedInit();

    // Do other stuff...
}
```

A *UserLoginSucceeded* event will be fired when the initialization completes and the user logins successfully. Otherwise, a *UserLoginFailed* event will be fired instead. You can optionally subscribe to these events and take appropriate actions depended on the user login status.

```
// Subscribe to events in the OnEnable method of a MonoBehavior script
void OnEnable()
{
    GameServices.UserLoginSucceeded += OnUserLoginSucceeded;
    GameServices.UserLoginFailed += OnUserLoginFailed;
}
```

```
// Unsubscribe
void OnDisable()
{
    GameServices.UserLoginSucceeded -= OnUserLoginSucceeded;
    GameServices.UserLoginFailed -= OnUserLoginFailed;
}

// Event handlers
void OnUserLoginSucceeded()
{
    Debug.Log("User logged in successfully.");
}

void OnUserLoginFailed()
{
    Debug.Log("User login failed.");
}
```

You can also check if the module has been initialized at any point using the *IsInitialized* method.

```
// Check if initialization has completed (the user has been authenticated)
bool isInitialized = GameServices.IsInitialized();
```

Leaderboards

This section focuses on working with leaderboards.

Show Leaderboard UI

To show the default leaderboard UI (the system view of leaderboards):

```
// Show leaderboard UI
GameServices.ShowLeaderboardUI();
```

You should check if the initialization has finished (the user has been authenticated) before showing the leaderboard UI, and take appropriate actions if the user is not logged in, e.g. show an alert or start another initialization process.

```
// Check for initialization before showing leaderboard UI
if (GameServices.IsInitialized())
{
    GameServices.ShowLeaderboardUI();
}
else
{
    #if UNITY_ANDROID
    GameServices.Init();    // start a new initialization process
    #elif UNITY_IOS
    Debug.Log("Cannot show leaderboard UI: The user is not logged in to Game Center.");
    #endif
}
```

To show the UI of a specific leaderboard, simply pass the name of the leaderboard into the *ShowLeaderboardUI* method. You can also optionally specify the time scope:

```
// Show a specific leaderboard UI
GameServices.ShowLeaderboardUI("YOUR_LEADERBOARD_NAME");

// Show a specific leaderboard UI in the Week time scope
GameServices.ShowLeaderboardUI("YOUR_LEADERBOARD_NAME", TimeScope.Week);
```

Report Scores

To report scores to a leaderboard you need to specify the name of that leaderboard.

> It is strongly recommended that you use the constants of leaderboard names in the generated EM_GameServicesConstants class (see **Game Services Constants Generation** section) instead of typing the names directly in order to prevent runtime errors due to typos and the likes.

```
// Report a score of 100
// EM_GameServicesConstants.Sample_Leaderboard is the generated name constant
// of a leaderboard named "Sample Leaderboard"
GameServices.ReportScore(100, EM_GameServicesConstants.Sample_Leaderboard);
```

Load Local User's Score

You can load the score of the local user (the authenticated user) on a leaderboard, to do so you need to specify the name of the leaderboard to load score from and a callback to be called when the score is loaded.

```
// Put this on top of the file to use IScore
UnityEngine.SocialPlatforms;



// Load the local user's score from the specified leaderboard
// EM_GameServicesConstants.Sample_Leaderboard is the generated name constant
// of a leaderboard named "Sample Leaderboard"
GameServices.LoadLocalUserScore(EM_GameServicesConstants.Sample_Leaderboard, OnLocalUserScoreLoaded);



// Score loaded callback
void OnLocalUserScoreLoaded(string leaderboardName, IScore score)
{
    if (score != null)
    {
        Debug.Log("Your score is: " + score.value);
    }
    else
    {
        Debug.Log("You don't have any score reported to leaderboard " + leaderboardName);
    }
}
```

Load Scores

You can load a set of scores from a leaderboard with which you can specify the start position to load score, the number of scores to load, as well as the time scope and user scope.

```
// Put this on top of the file to use IScore
UnityEngine.SocialPlatforms;



// Load a set of 20 scores starting from rank 10 in Today time scope and Global user scope
// EM_GameServicesConstants.Sample_Leaderboard is the generated name constant
// of a leaderboard named "Sample Leaderboard"
GameServices.LoadScores(
    EM_GameServicesConstants.Sample_Leaderboard,
    10,
    20,
    TimeScope.Today,
    UserScope.Global,
    OnScoresLoaded
);
```

```
// Scores loaded callback
void OnScoresLoaded(string leaderboardName, IScore[] scores)
{
    if (scores != null && scores.Length > 0)
    {
        Debug.Log("Loaded " + scores.Length + " from leadeboard " + leaderboardName);
        foreach (IScore score in scores)
        {
            Debug.Log("Score: " + score.value + "; rank: " + score.rank);
        }
    }
    else
    {
        Debug.Log("No score loaded.");
    }
}
```

You can also load the default set of scores, which contains 25 scores around the local user's score in the *AllTime* time scope and *Global* user scope.

```
// Put this on top of the file to use IScore
UnityEngine.SocialPlatforms;



// Load the default set of scores
// EM_GameServicesConstants.Sample_Leaderboard is the generated name constant
// of a leaderboard named "Sample Leaderboard"
GameServices.LoadScores(EM_GameServicesConstants.Sample_Leaderboard, OnScoresLoaded);



// Scores loaded callback
void OnScoresLoaded(string leaderboardName, IScore[] scores)
{
    if (scores != null && scores.Length > 0)
    {
        Debug.Log("Loaded " + scores.Length + " from leadeboard " + leaderboardName);
        foreach (IScore score in scores)
        {
            Debug.Log("Score: " + score.value + "; rank: " + score.rank);
        }
    }
    else
    {
        Debug.Log("No score loaded.");
    }
}
```

Get All Leaderboards

You can obtain an array of all leaderboards created in the module settings interface:

```
// Get the array of all leaderboards created in the Game Service module settings
// Leaderboard is the class representing a leaderboard as declared in the module settings
// The GameServices property of EM_Settings class holds the settings of this module
Leaderboard[] leaderboards = EM_Settings.GameServices.Leaderboards;

// Print all leaderboard names
foreach (Leaderboard ldb in leaderboards)
{
    Debug.Log("Leaderboard name: " + ldb.Name);
}
```

Achievements

This section focuses on working with achievements.

Show Achievement UI

To show the achievements UI (the system view of achievements):

```
// Show achievements UI
GameServices.ShowAchievementsUI();
```

You should check if the initialization has finished (the user has been authenticated) before showing the achievements UI, and take appropriate actions if the user is not logged in, e.g. show an alert or start another initialization process.

```
// Check for initialization before showing achievements UI
if (GameServices.IsInitialized())
{
    GameServices.ShowAchievementsUI();
}
else
{
    #if UNITY_ANDROID
    GameServices.Init();    // start a new initialization process
    #elif UNITY_IOS
    Debug.Log("Cannot show achievements UI: The user is not logged in to Game Center.");
    #endif
}
```

Reveal an Achievement

To reveal a hidden achievement, simply specify its name.

> As in the case of leaderboards, it is strongly recommended that you use the constants of achievement names in the generated EM_GameServicesConstants class instead of typing the names directly.

```
// Reveal a hidden achievement
// EM_GameServicesConstants.Sample_Achievement is the generated name constant
// of an achievement named "Sample Achievement"
GameServices.RevealAchievement(EM_GameServicesConstants.Sample_Achievement);
```

Unlock an Achievement

To unlock an achievement:

```
// Unlock an achievement
// EM_GameServicesConstants.Sample_Achievement is the generated name constant
// of an achievement named "Sample Achievement"
GameServices.UnlockAchievement(EM_GameServicesConstants.Sample_Achievement);
```

Report Incremental Achievement's Progress

To report the progress of an incremental achievement:

```
// Report a rogress of 50% for an incremental achievement
// EM_GameServicesConstants.Sample_Incremental_Achievement is the generated name constant
// of an incremental achievement named "Sample Incremental Achievement"
GameServices.ReportAchievementProgress(EM_GameServicesConstants.Sample_Incremental_Achievement, 50.0f);
```

Get All Achievements

You can obtain an array of all achievements created in the module settings interface:

```
// Get the array of all achievements created in the Game Service module settings
// Achievement is the class representing an achievement as declared in the module settings
// The GameService property of EM_Settings class holds the settings of this module
Achievement[] achievements = EM_Settings.GameServices.Achievements;

// Print all achievement names
foreach (Achievement acm in achievements)
{
    Debug.Log("Achievement name: " + acm.Name);
}
```

User Profiles

You can load the profiles of friends of the local (authenticated) user. When the loading completes the provided callback will be invoked.

```
// Put this on top of the file to use IUserProfile
UnityEngine.SocialPlatforms;


// Load the local user's friend list
GameServices.LoadFriends(OnFriendsLoaded);


// Friends loaded callback
void OnFriendsLoaded(IUserProfile[] friends)
{
    if (friends.Length > 0)
    {
        foreach (IUserProfile user in friends)
        {
            Debug.Log("Friend's name: " + user.userName + "; ID: " + user.id);
        }
     }
     else
     {
         Debug.Log("Couldn't find any friend.");
     }
}
```

You can also load user profiles by providing their IDs.

```
// Put this on top of the file to use IUserProfile
UnityEngine.SocialPlatforms;


// Load the profiles of the users with provided IDs
// idArray is the (string) array of the IDs of the users to load profiles
GameServices.LoadUsers(idArray, OnUsersLoaded);


// Users loaded callback
void OnUsersLoaded(IUserProfile[] users)
{
    if (users.Length > 0)
    {
        foreach (IUserProfile user in users)
        {
            Debug.Log("User's name: " + user.userName + "; ID: " + user.id);
        }
     }
     else
```

```
        {
            Debug.Log("Couldn't find any user with the specified IDs.");
        }
    }
```

Sign Out

To sign the user out, simply call the *SignOut* method. Note that this method is only effective on Android.

```
// Sign the user out on Android
GameServices.SignOut();
```

Game Services | Saved Games: Introduction

Saving game data is among the most desirable features of video games in general, and mobile games in particular. Nowadays, it's not uncommon for a user to own more than one mobile device, be it phone or tablet. Being able to start a game on one device, and then continue playing on another device without losing any progress brings a seamless - if not natural - user experience.

The Saved Games feature of Easy Mobile makes it possible - and easy - to save a player's game data to the cloud and synchronize it across multiple devices. Saving user data to the cloud also means that their game progression is preserved and can be restored in cases such as reinstallation or device failure.

> On iOS, the game data is saved to iCloud via the Game Center (GameKit) API. On Android, it is saved to Google Drive via the Google Play Game Services API (GPGS).

Understanding Saved Games

A saved game consists of two parts:

- An unstructured binary blob - this can represent whatever data you deem relevant to your game, and your game is responsible for generating and intepreting it.
- Structured metadata - additional properties associated with the binary data and provide information about this data.

The table below describes common saved game properties.

Property	Description
Name	A developer-supplied short name of the saved game
ModificationDate	A timestamp corresponding to the last modification of the saved game
DeviceName	[iOS only] The name of the device that committed the saved game data
Description	[GPGS only][Optional] A developer-supplied description of the saved game
CoverImageURL	[GPGS only] [Optional] The URL of the PNG cover image of the saved game
TotalTimePlayed	[GPGS only][Optional] A developer-supplied value (in milisesconds) representing the played time of the saved game
IsOpen	Whether the saved game is "Open". A saved game can only be read or written to if it is open.

It's up to you to decide how and when users can save a game. Depending on your game design, you might want to allow only a single saved game, or you might want to allow the player to create multiple saved games with different names (so they can, for example, go back to various checkpoints and try different actions).

The Underlying Cloud Services

As mentioned earlier, saved games are stored on iCloud (iOS/Game Center) and Google Drive (Android/GPGS). Therefore, it's mandatory that the user has an iCloud or Google account to use the feature on the corresponding platform.

> On iOS, the saved games are tied to the user's iCloud account, not the Game Center account.
> On Android, the Google Drive associated with the user's Google account that was authenticated with GPGS is used.

Limitations

iOS (iCloud/Game Center)	Android (Google Drive/GPGS)
No hard limit on the number of saved games	No hard limit on the number of saved games
The size of a saved game data is limited to the amount of available space in the user's iCloud account)	GPGS currently enforce size limits on binary data and cover image sizes of 3 MB and 800 KB respectively.

> You should always strive to minimize the amount of data being saved. This prevents the user from running out of space and decreases the amount of time required to fetch or save a game file. Also note that the game saving operation may fail if there's not enough room in the iCloud or Google Drive account of the user.

Offline Support

Your game can still read and write to a saved game when the player's device is offline, but will not be able to sync with the cloud services until network connectivity is established. Once reconnected, the synchronization will be done automatically and asynchronously.

Conflict Resolution

When a user plays your game on multiple devices and uses the saved games feature, it's not uncommon to have multiple saved games with the same name and from different devices, thus creating conflicts. These conflicts typically occur when an instance of your game is unable to reach the cloud service while attempting to sync the save game data, or when it updates the saved game data on the cloud without loading the latest data first. In general, the best way to avoid data conflicts is to always load the latest data from the cloud service when your game starts up or resumes, and save data to the service with reasonable frequency. However, it is not always possible to avoid data conflicts. Your application should make every effort to handle conflicts to preserve users' data as well as maintain a good user experience. Fortunately, the Saved Games API can help you resolve these conflicts automatically using several default resolution strategies. It also provides relevant methods to help you implement your own resolution strategy to better suit your needs.

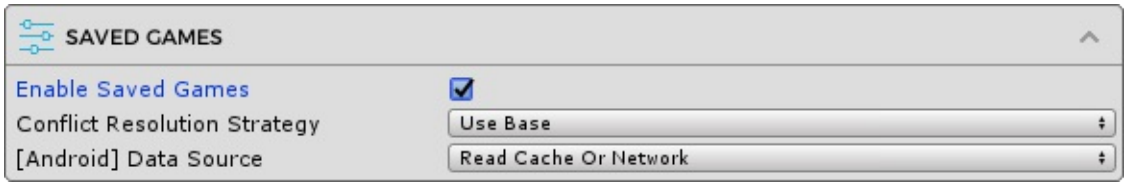
In this GameOn! - Saved Games In-Depth (Part 2) YouTube video by Google Developers you 'll find in-depth explanation on conflicts between saved games, how they happen, how to resolve them as well as other important concepts. The video is dedicated to the Saved Games feature of Google Play Games Services, but the concepts are also applicable to Game Center. A must watch.

Useful Links

1. Saving A Game - Game Center Programming Guide
2. Saved Games - Google Play Game Services
3. GameOn! - Saved Games In-Depth (Part 2) YouTube video

Game Services | Saved Games: Settings

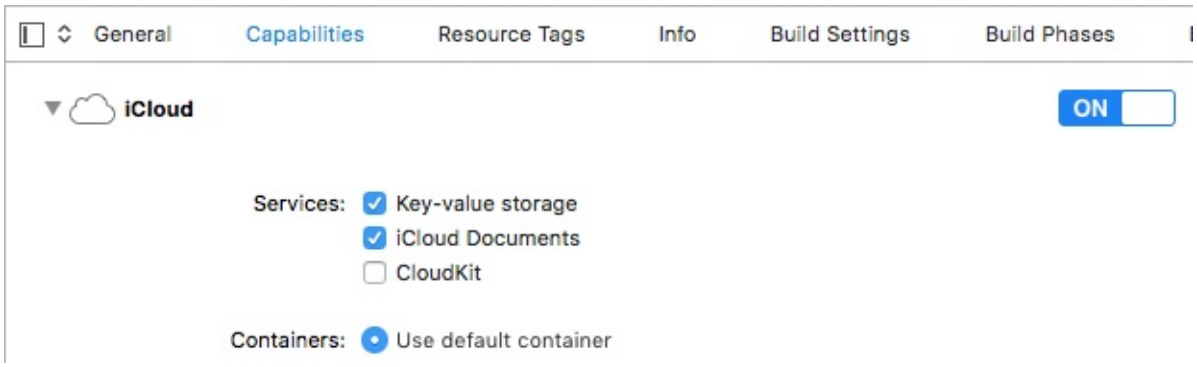
The Saved Games feature can be configured in the **SAVED GAMES CONFIG** section in the Game Service module settings.



- *Enable Saved Games*: you must enable the Saved Games feature before using it
- *Conflict Resolution Strategy*: the default strategy used by the automatic conflict resolution feature
- *[Android] Data Source*: where the game data can be fetched from, only applicable on Android/Google Play Game Services platform

iOS Setup

To use the Saved Games service on iOS, you must enable the iCloud capability for your app in the Xcode project. Make sure the *iCloud Documents* service is selected.



Also, for the feature to function on their iOS devices, the users must signed into their iCloud account and have the iCloud Drive service enabled in the Settings app.

Android Setup

Again, on Android we employ the Saved Games feature provided by the Google Play Game Services. Therefore, you need to enable this feature for your app in the Google Play Console. Select your app, then select the **Game Services** tab and enable the feature in the **Game details** tab.

> Note that you need to wait at least 24 hours after enabling the Saved Games service for it to be available. Attempting to authenticate during this time may cause the app to crash.

Game Services | Saved Games: Scripting

This section provides a guide to work with the Saved Games scripting API of the Game Services module.

> You can access the Saved Games API via the SavedGames property of the GameServices class under the EasyMobile namespace.

Working with saved games involves the following operations:

Operation	Description
Open	A saved game must be opened before it can be used for read or write operation. If you attempt to open a non-existing saved game, a new one will be created and will be opened automatically. You must resolve any conflicts associated with a saved game when opening it. You can have the conflicts resolved automatically using one of the default strategies, or implement your own strategy to resolve them manually.
Write	Update the data associated with a saved game, the saved game must be open before writing, and it will be closed automatically after the operation has finished.
Read	Retrieve the data associated with a saved game, the saved game must be open.
Delete	Delete a saved game from the cloud service

Opening Saved Game

You can open a saved game using either the *OpenWithAutomaticConflictResolution* or the *OpenWithManualConflictResolution* method. Both methods open a saved game with the specified name, or create a new one if none exists. The saved game returned in their callbacks will be open which means it can be used for read or write operation. The difference between the two is whether saved game conflicts, if any, will be resolved automatically or manually.

> If the current platform is Google Play Game Services, these methods use the data source specified in the module settings.

Open With Automatic Conflict Resolution

As its name suggests, when opening a saved game using this method, any outstanding conflicts will be resolved automatically using the resolution strategy specified in the module settings.

```
// Put this on top of the script
using EasyMobile;


// To store the opened saved game.
private SavedGame mySavedGame;

// Open a saved game with automatic conflict resolution
void OpenSavedGame()
{
        // Open a saved game named "My_Saved_Game" and resolve conflicts automatically if any.
        GameServices.SavedGames.OpenWithAutomaticConflictResolution("My_Saved_Game", OpenSavedGameCallback);
}

// Open saved game callback
void OpenSavedGameCallback(SavedGame savedGame, string error)
{
        if (string.IsNullOrEmpty(error))
        {
```

```
                Debug.Log("Saved game opened successfully!");
                mySavedGame = savedGame;        // keep a reference for later operations
        }
        else
        {
                Debug.Log("Open saved game failed with error: " + error);
        }
}
```

Open With Manual Conflict Resolution

If the saved game being opened has outstanding conflicts, they will be resolved manually using the specified conflict resolution function. This function must be implemented by you and it is where you provide your custom conflict resolution strategy, in case none of the default strategies suits your needs. The function will be invoked automatically when a conflict is encountered while opening a saved game and can be invoked multiple times if the saved game has more than one outstanding conflict. Therefore it must be designed to handle multiple invocations.

The conflict resolution function receives the Base and Remote versions of the conflicting saved game (please check out this GameOn! - Saved Games In-Depth (Part 2) YouTube video by Google Developers for an excellent explanation on the concepts of "base" and "remote"). These passed saved games are all open. If *OpenWithManualConflictResolution* was invoked with *prefetchDataOnConflict* set to true, the binary data associated with these saved games will loaded and passed to the conflict resolution function too. Use the return value of this function to determine whether the base or the remote will be chosen as the canonical version of the saved game.

The callback will be invoked when all conflicts (if any) have been resolved and the operation finishes.

```
// Put this on top of the script
using EasyMobile;


// To store the opened saved game.
private SavedGame mySavedGame;

// Open a saved game with manual conflict resolution
void OpenSavedGame()
{
        // Open a saved game named "My_Saved_Game" and resolve any outstanding conflicts manually using
        // the specified resolution function.
        GameServices.SavedGames.OpenWithManualConflictResolution(
                "My_Saved_Game",
                true,        // prefetchDataOnConflict
                MyConflictResolutionFunction,
                OpenSavedGameCallback
        );
}

// The conflict resolution function.
// baseGame and remoteGame are all open.
// If OpenWithManualConflictResolution was invoked with prefetchDataOnConflict set to true,
// baseData and remoteData will contain the binary data associated with baseGame and remoteGame respective.
// They will be null otherwise.
// In this function you can perform required calculation, comparison between two versions, etc. to decide
// which one will be the canonical version of the saved game. Use the return value to indicate your decision.
SavedGameConflictResolutionStrategy MyConflictResolutionFunction(SavedGame baseGame, byte[] baseData,
                                                                 SavedGame remoteGame,byte[] remoteData)
{
        {
                // Perform whatever required calculation, comparison, etc. on the two versions
                // and their associated data to help you decide which version should be chosen.
                ...

                // After determining the canonical version, use the return value to indicate your choice
                return SavedGameConflictResolutionStrategy.UseBase;        // use the base version

                // If you want to select the remote version instead, just change it to
                // return SavedGameConflictResolutionStrategy.UseRemote;
        }
```

```
}

// Open saved game callback
void OpenSavedGameCallback(SavedGame savedGame, string error)
{
        if (string.IsNullOrEmpty(error))
        {
                Debug.Log("Saved game opened successfully!");
                mySavedGame = savedGame;        // keep a reference for later operations
        }
        else
        {
                Debug.Log("Open saved game failed with error: " + error);
        }
}
```

> In case you want to merge the data from different versions, simply specify either the base or the remote as the chosen version. Once all conflicts are resolved and the saved has been opened successfully, perform a write operation using the merge data.

Writing Saved Game Data

To commit new data to a saved game, use the *WriteSavedGameData* method. As mention earlier, the saved game must be open before writing or the operation will fail. When this method completes successfully, the data is durably persisted to disk and will eventually be uploaded the the cloud (in practice, this process happens very quickly unless the device doesn't have a network connection). After the operation finishes, the saved game will be closed automatically. This is to force it to be opened once again (thus resolving any outstanding conflicts) before another commit can be made.

```
// Put this on top of the script
using EasyMobile;



// Updates the given binary data to the specified saved game
void WriteSavedGame(SavedGame savedGame, byte[] data)
{
    if (savedGame.IsOpen)
    {
        // The saved game is open and ready for writing
        GameServices.SavedGames.WriteSavedGameData(
            savedGame,
            data,
            (SavedGame updatedSavedGame, string error) =>
            {
                if (string.IsNullOrEmpty(error))
                {
                    Debug.Log("Saved game data has been written successfully!");
                }
                else
                {
                    Debug.Log("Writing saved game data failed with error: " + error);
                }
        );
    }
    else
    {
        // The saved game is not open. You can optionally open it here and repeat the process.
        Debug.Log("You must open the saved game before writing to it.");
    }
}
```

Beside the binary data, you can also update the metadata (properties) of a saved game. Just use the overloading version of *WriteSavedGameData* that accepts a *SavedGameInfoUpdate* struct.

> Some saved game properties are only available on a certain platform, please review the **Game Service > Module Configuration > Saved Game** section for detailed information.

```
// Put this on top of the script
using EasyMobile;



// Updates the binary data AND the properties of a saved game
void WriteSavedGame(SavedGame savedGame, byte[] data)
{
    if (savedGame.IsOpen)
    {
        // The saved game is open and ready for writing
        // Prepare the updated metadata of the saved game
        SavedGameInfoUpdate.Builder builder = new SavedGameInfoUpdate.Builder();
        builder.WithUpdatedDescription("New_Description");
        builder.WithUpdatedPlayedTime(TimeSpan.FromMinutes(30));    // update the played time to 30 minutes
        SavedGameInfoUpdate infoUpdate = builder.Build();

        GameServices.SavedGames.WriteSavedGameData(
            savedGame,
            data,
            infoUpdate,    // update saved game properties
            (SavedGame updatedSavedGame, string error) =>
            {
                if (string.IsNullOrEmpty(error))
                {
                    Debug.Log("Saved game data has been written successfully!");
                }
                else
                {
                    Debug.Log("Writing saved game data failed with error: " + error);
                }
        );
    }
    else
    {
        // The saved game is not open. You can optionally open it here and repeat the process.
        Debug.Log("You must open the saved game before writing to it.");
    }
}
```

Reading Saved Game Data

To read a saved game data, use the ReadSavedGameData method. The saved game must be open before reading. The callback will be invoked when the operation finishes and will receive the retrieved data as a byte array, which can be empty if the saved game has no data committed previously.

```
// Put this on top of the script
using EasyMobile;



// Retrieves the binary data associated with the specified saved game
void ReadSavedGame(SavedGame savedGame)
{
    if (savedGame.IsOpen)
    {
        // The saved game is open and ready for reading
        GameServices.SavedGames.ReadSavedGameData(
            savedGame,
            (SavedGame game, byte[] data, string error) =>
            {
                if (string.IsNullOrEmpty(error))
                {
                    Debug.Log("Saved game data has been retrieved successfully!");
                    // Here you can process the data as you wish.
```

```
                    if (data.Length > 0)
                    {
                        // Data processing
                        ...
                    }
                    else
                    {
                        Debug.Log("The saved game has no data!");
                    }
                }
                else
                {
                    Debug.Log("Reading saved game data failed with error: " + error);
                }
            );
        }
        else
        {
            // The saved game is not open. You can optionally open it here and repeat the process.
            Debug.Log("You must open the saved game before reading its data.");
        }
    }
```

Deleting Saved Game

To delete a saved game, simply call the DeleteSavedGame method.

```
// Put this on top of the script
using EasyMobile;



// Deletes a saved game
void DeleteSavedGame(SavedGame savedGame)
{
    GameServices.SavedGames.DeleteSavedGame(savedGame);
}
```

Fetching All Saved Games

When implement the saved games feature in your game, chances are you will want to show the user a list of existing saved games for them to choose from. In such case, you can use the *FetchAllSavedGames* method to retrieve all know saved games. A callback will be invoked when the method completes, receiving an array of saved games which can be empty if no saved game was created before. Note that all the returned saved games are NOT open.

> If the current platform is Google Play Game Services, this method retrieves saved games from the data source specified in the module settings.

```
// Put this on top of the script
using EasyMobile;



// Fetches all know saved games.
void FetchSavedGames()
{
    GameServices.SavedGames.FetchAllSavedGames(
        (SavedGame[] games, string error) =>
        {
            if (string.IsNullOrEmpty(error))
            {
                Debug.Log("Fetched saved games successfully! Got " + games.Length + " saved games.");

                // Here you can show a UI to display these saved games to the user...
```

```
            }
            else
            {
                Debug.Log("Fetching saved games failed with error " + error);
            }
        }
    );
}
```

[Android] Built-in Saved Game UI

On Android, the Saved Games feature of Google Play Game Services offers a built-in UI from which the user can open, select or delete a saved games. You can show this UI by calling the *ShowSelectSavedGameUI* method. A callback will be invoked when the UI is closed, receiving the selected saved game if any.

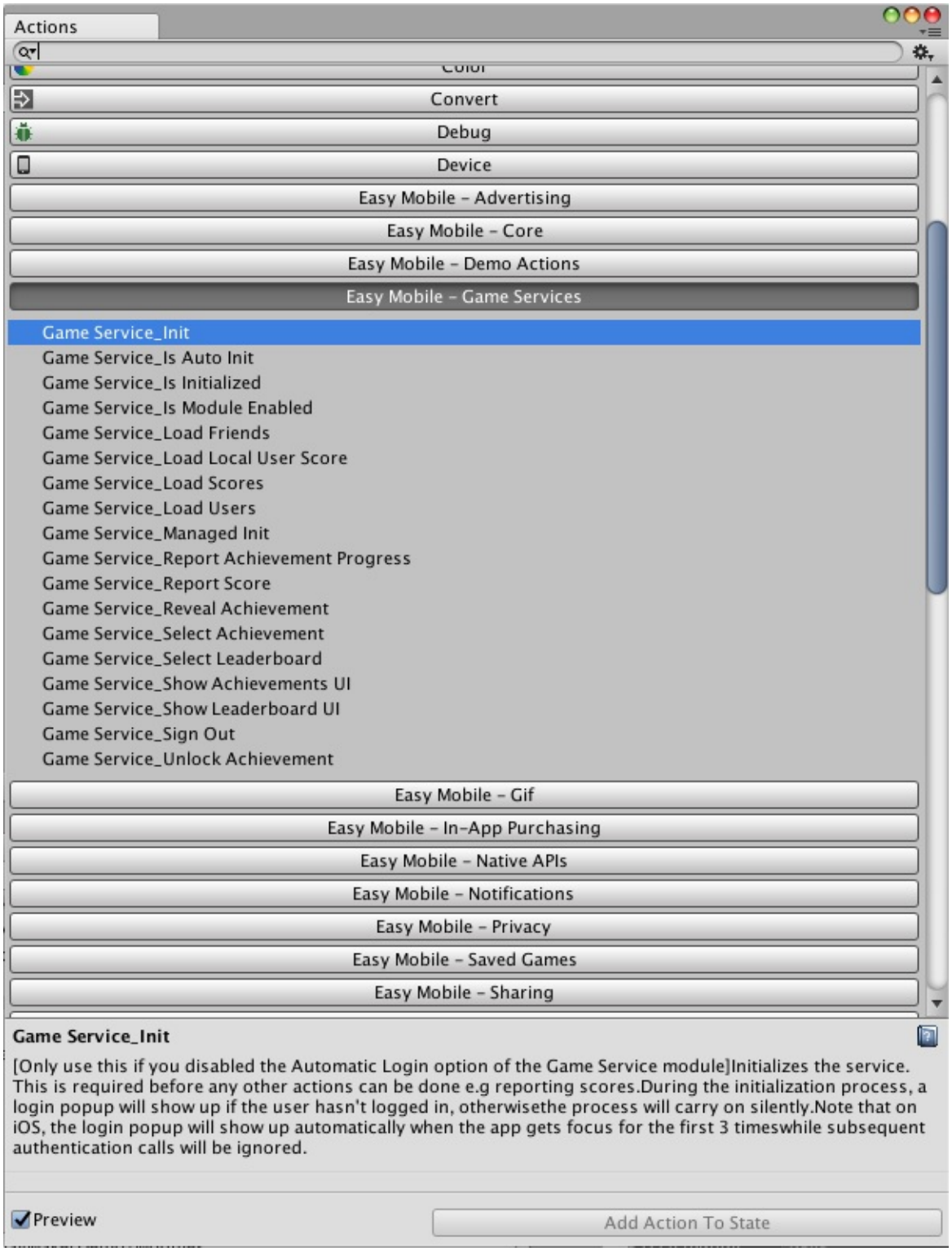
> This method is a no-op on iOS/Game Center platform.

```
// Put this on top of the script
using EasyMobile;



// Shows the GPGS built-in saved game UI.
void ShowGPGSSavedGameUI()
{
    GameServices.SavedGames.ShowSelectSavedGameUI(
        "Select Saved Game",    // UI title
        5,        // maximum number of displayed saved games
        true,    // allow creating saved games
        true,    // allow deleting saved games
        (SavedGame game, string error) =>
        {
            if (string.IsNullOrEmpty(error))
            {
                Debug.Log("You selected saved game: " + game.Name);
            }
            else
            {
                Debug.Log(error);
            }
        }
    );
}
```

Game Services: PlayMaker Actions

The PlayMaker actions of the Game Services module are group in the category *Easy Mobile - Game Services* in the PlayMaker's Action Browser.

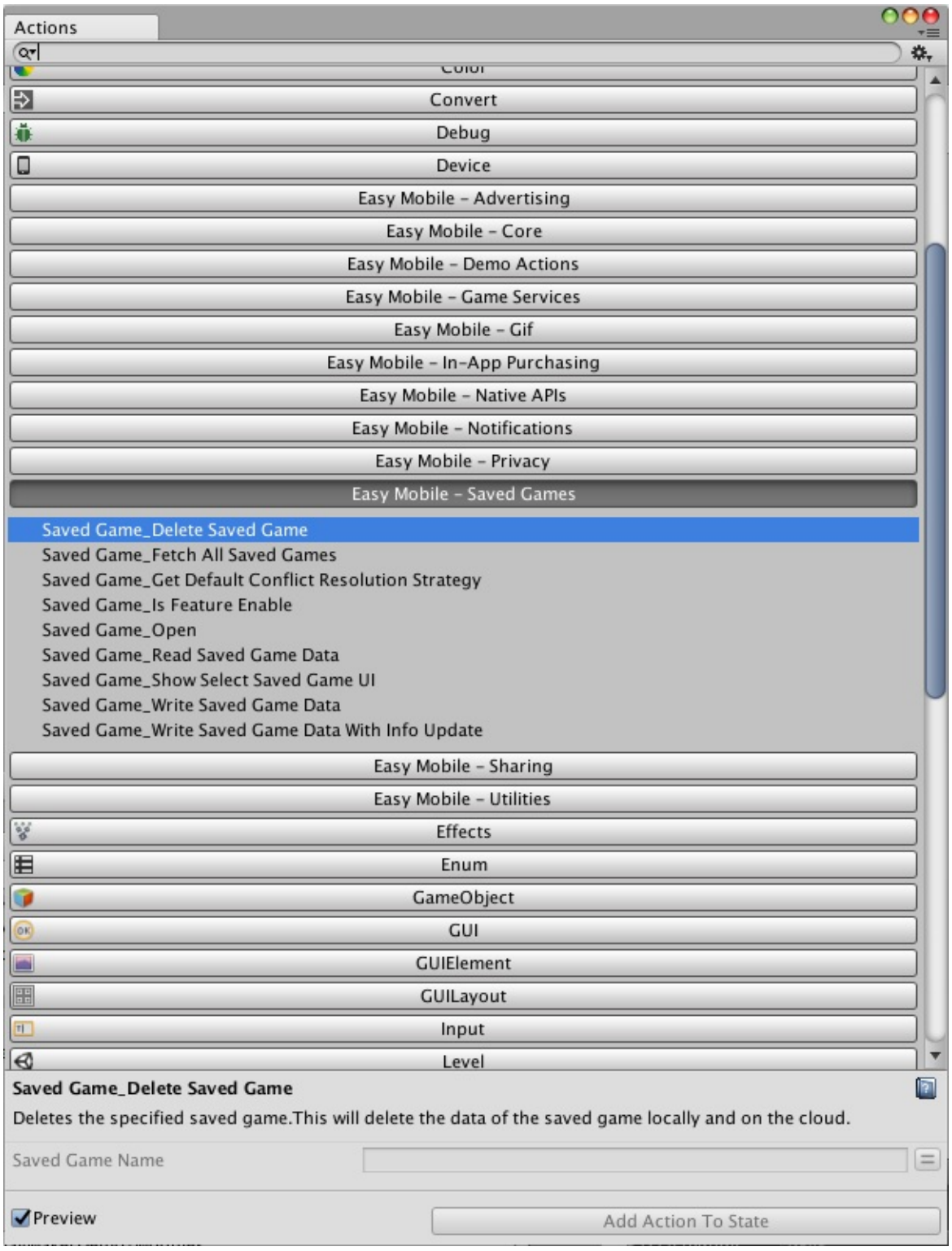
> Please refer to the GameServicesDemo_PlayMaker scene in folder *Assets/EasyMobile/Demo/PlayMakerDemo/Modules* for an example on how these actions can be used.



Saved Games

The PlayMaker actions of the Saved Games feature are group in the category *Easy Mobile - Saved Games* in the PlayMaker's Action Browser.

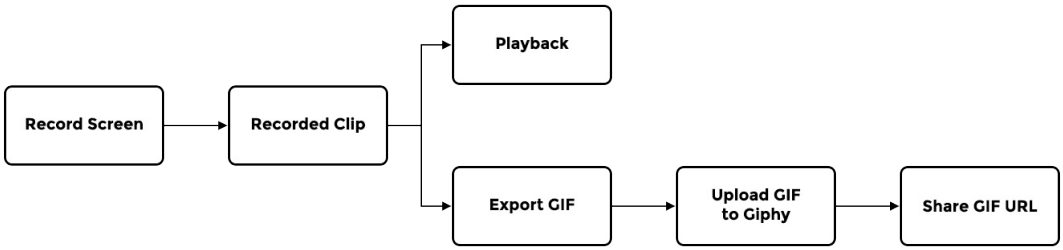
> Please refer to the GameServicesDemo_SavedGames_PlayMaker scene in folder *Assets/EasyMobile/Demo/PlayMakerDemo/Modules* for an example on how these actions can be used.



GIF: Introduction

> *GIF module is available on Easy Mobile Pro only.*

The GIF module provides you convenient tools to record screen activities into a short clip, play the recorded clip and export it into a GIF image. You can then upload the GIF file to hosting sites like Giphy and finally share its URL to social networks. In short, this module helps you easily add the GIF sharing feature to your game, which allows the user to share animated GIF images of the gameplay, instead of still screenshots, to social networks including Facebook and Twitter. The following picture illustrates a typical workflow of such feature.



Here're some highlights of this module:

- **High performance, mobile-friendly GIF generator**
  - Low overhead screen/camera recorder
  - GIF generation is done in native code (iOS and Android) on a separate thread to allow fast exporting while minimizing impact to the main thread. Export callbacks are still called from main thread though, so you can safely access Unity API in the callback handlers
- **Flexible, fully controllable process**
  - You have full control on the sizes, length, frame rate, loop mode and quality of the exported GIF
  - You can also set the priority of the exporting thread to best suit your needs
- **High quality GIF**
  - Exported GIF employs GIF89a format and uses 256-color local palettes (one palette per frame)
  - Frame image data is LWZ compressed
- **Works in Unity editor**
  - GIF exporting also works in the editor, mostly for testing purpose. On mobiles, the exporting is done in native code, while in editor it is done in managed code using an adapted version of the Moments plugin (see Acknowledgement)
- **Easy GIF sharing**
  - This module also provides Giphy API for uploading GIF images to Giphy, so that they can be shared and played on major social networks including Facebook and Twitter using the Giphy hosted URLs

Acknowledgement

The recorder used in this module is adapted from the recorder of the Moments plugin by Chman (Thomas Hourdel). Also, in Unity editor, GIF generation is done using an adapted version of this plugin.

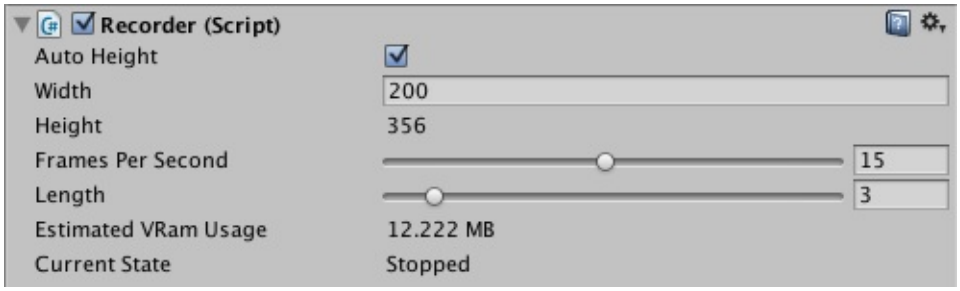
GIF: Setup

This section explains the various components, objects and concepts involved in clip recording, clip playing and GIF exporting. It also provides a guide on creating and configuring relevant objects and components.

Recorder Component

The Recorder component records the content rendered by a camera and returns the recorded clip. To start recording, simply add a Recorder component to the camera that renders the content you're interested in recording (normally this will be the Main Camera). To add the component to a camera, select that camera in the Hierarchy, then click *Add Component* > *Easy Mobile* > *Recorder*.

Once the Recorder component is added to the camera, you can start configuring it in the inspector to determine how the recorded clip (and as a result, the exported GIF) will be like.



- *Auto Height*: whether the clip height should be computed automatically from the specified width and the camera's aspect ratio, which is useful to make sure the exported GIF has a correct aspect ratio
- *Width*: the width of the recorded clip in pixels
- *Height*: the height of the recorded clip in pixels
- *Frames Per Second*: the frame rate of the clip
- *Length*: the clip length in seconds; the recorder automatically discards old content to preserve this length, e.g. if you set this value to 3 seconds, only last 3 seconds of the recording will be stored in the resulted clip, the rest will

be discarded
- *Estimated VRam Usage*: the estimated memory used for recording, calculated based on the above settings
- *Current State*: the current status of the recorder, which is either *Stopped* or *Recording*

Now that the recorder is configured, you can start and stop its recording activity from script (see the **Scripting** section). Once the recording is stopped, the recorded clip will be returned for playback of GIF exporting.

> **Recording the UI**
>
> To record the UI (Canvas content), you need to set the Canvas Render Mode to World Space or Screen Space - Camera, and set the Render Camera to the one containing the Recorder component in the latter case.
>
> **Recording multiple composited cameras**
>
> If your scene contains multiple cameras being composited (using Camera.depth and Clear flags), you can add the Recorder component to the top-most camera, so it captures whatever content being composited and shown by that camera.

AnimatedClip Class

Recorded clips are represented by the *AnimatedClip* class, which has following properties:

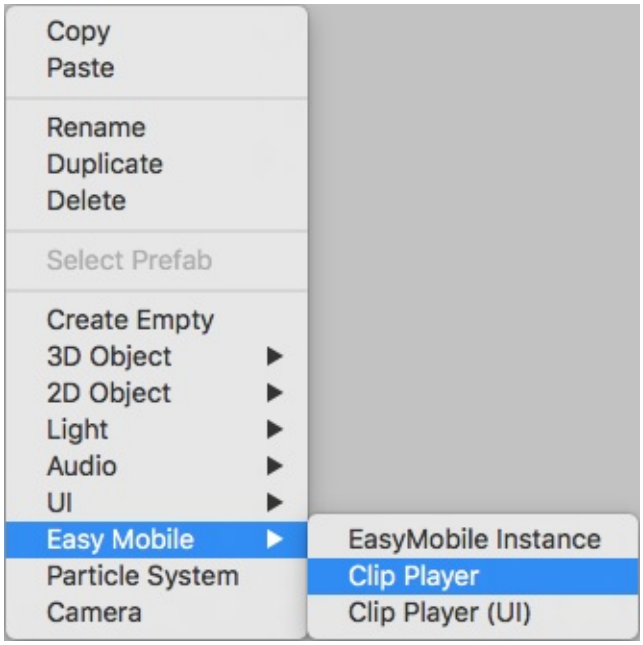
- *Width*: the width of the clip in pixels
- *Height*: the height of the clip in pixels
- *Frame Per Second*: the frame rate of the clip
- *Length*: the length of the clip in seconds
- *Frames*: an array of frames, each frame is a *Render Texture* object

Playback

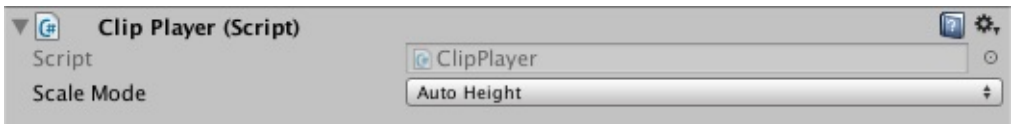
Easy Mobile provides two built-in objects dedicated for playing recorded clips: the Clip Player and Clip Player UI objects. You can create them from the context menu (as you would with other Unity built-in objects), configure them in the inspector, and start or stop their playing activity from script (see the **Scripting** section).

Clip Player

The Clip Player is a non-UI object, which is basically a Quad object equipped with a ClipPlayer component. It is meant to be used inside the game world. To create a Clip Player object, right-click in the Hierarchy window to open the context menu, then select *Easy Mobile* > *Clip Player*.



Each Clip Player object contains a ClipPlayer component.

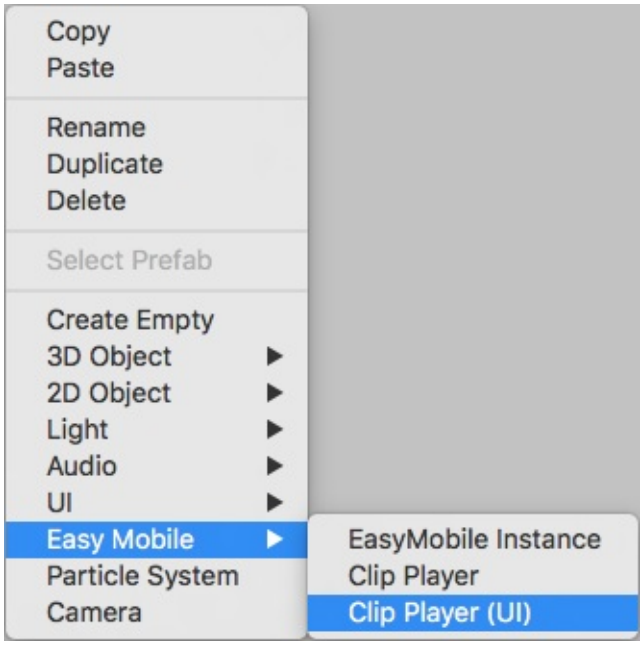


The only parameter of this component is the *Scale Mode*, which can take one of 3 values:

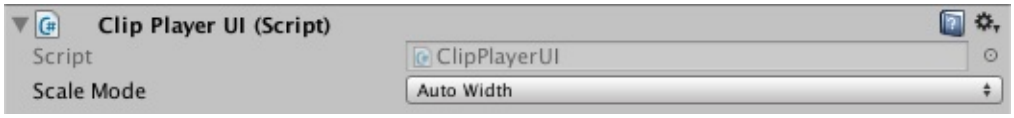
- *None*: don't adjust the object sizes
- *Auto Height*: keeps the current height of the object (the Y component of its localScale), and adjust the width (the X component of its localScale) to match the aspect ratio of the clip being played
- *Auto Width*: keeps the current width of the object (the X component of its localScale), and adjust the height (the Y component of its localScale) to match the aspect ratio of the clip being played

Clip Player UI

The Clip Player UI, as it name implies, is a UI object living inside a Canvas. It is the object to use when you want to play a clip inside the UI. It is basically a Raw Image object equipped with a ClipPlayerUI component. To create a Clip Player UI object, right-click in the Hierarchy window to open the context menu, then select *Easy Mobile* > *Clip Player (UI)*.



Each Clip Player UI object contains a ClipPlayerUI component.



The only parameter of this component is the *Scale Mode*, which can take one of 3 values:

- *None*: don't adjust the object sizes
- *Auto Height*: keeps the current height of the object (the Height value in its Rect Transform), and adjust the width (the Width value in its Rect Transform) to match the aspect ratio of the clip being played
- *Auto Width*: keeps the current width of the object (the Width value in its Rect Transform), and adjust the height (the Height value in its Rect Transform) to match the aspect ratio of the clip being played

Custom Clip Player

Beside the two built-in clip players provided by Easy Mobile, you can construct your own player to serve your specific needs. To make it consistent with other players, and compatible with Easy Mobile API, this player should contain a script implementing the *IClipPlayer* interface, which is responsible for applying the frames (*RenderTextures*) of the clip, at the required frame rate, to whatever texture-displaying component it is equipped with.

GIF: Scripting

This section provides a guide to work with the GIF module scripting API. At this stage, it's assumed that you have setup a recorder for the camera you want to record, and created an appropriate clip player to play the recorded clip. If you're not familiar with these concepts, please review the Setup section.

> You can access the GIF module API via the Gif class under the EasyMobile namespace. As for Giphy API, use the Giphy class.

Recording

To start recording on the created recorder, use the *StartRecording* method. You can do this as soon as the game starts; the recorder only stores a few last seconds (specified by the *Length* parameter in the Recorder inspector) of the recording, and automatically discards the rest.

```
// Put this on top of the script
using EasyMobile;



// Drag the camera with the Recorder component to this field in the inspector
public Recorder recorder;



// You can start recording as soon as your game starts
// (suppose you have a method named StartGame, which is called when the game starts)
void StartGame()
{
    // Start recording!
    Gif.StartRecording(recorder);

    // Do other stuff...
}
```

To stop recording, simply call the *StopRecording* method, passing the relevant recorder. The method returns an *AnimatedClip* object, which can be played or exported into a GIF image afterward. To continue the previous example:

```
// Put this on top of the script
using EasyMobile;



// Drag the camera with the Recorder component to this field in the inspector
public Recorder recorder;

// The recorded clip
AnimatedClip myClip;

// You can start recording as soon as your game starts
// (suppose you have a method named StartGame, which is called when the game starts)
void StartGame()
{
    // Start recording!
    Gif.StartRecording(recorder);

    // Do other stuff...
}



// A suitable time to stop recording may be when the game ends (the player dies)
```

```
// (suppose you have a method named GameOver, called when the game ends)
void GameOver()
{
    // Stop recording
    myClip = Gif.StopRecording(recorder);

    // Do other stuff...
}
```

Playback

To play a recorded clip using a pre-created clip player, use the *PlayClip* method. This method receives as argument an *IClipPlayer* interface, which is implemented by both *ClipPlayer* and *ClipPlayerUI* classes, therefore it works with both Clip Player and Clip Player UI object. The second argument is an *AnimatedClip* object. Other arguments include an optional delay time before the playing starts, and the looping mode. You can pause, resume and stop the player using the *PausePlayer*, *ResumePlayer* and *StopPlayer* methods, respectively.

To continue the previous example:

```
// Put this on top of the script
using EasyMobile;



// Drag the camera with the Recorder component to this field in the inspector
public Recorder recorder;

// Suppose you've created a ClipPlayerUI object (ClipPlayer will also work)
// Drag the pre-created clip player to this field in the inspector
public ClipPlayerUI clipPlayer;

// The recorded clip
AnimatedClip myClip;

// You can start recording as soon as your game starts
// (suppose you have a method named StartGame, which is called when the game starts)
void StartGame()
{
    // Start recording!
    Gif.StartRecording(recorder);

    // Do other stuff...
}



// A suitable time to stop recording may be when the game ends (the player dies)
// (suppose you have a method named GameOver, called when the game ends)
void GameOver()
{
    // Stop recording
    myClip = Gif.StopRecording(recorder);

    // Play the recorded clip!
    PlayMyClip();
}



// This method plays the recorded clip on the created player,
// with no delay before playing, and loop indefinitely.
void PlayMyClip()
{
    Gif.PlayClip(clipPlayer, myClip);
}

// This method plays the recorded clip on the created player,
// with a delay of 1 seconds before playing, and loop indefinitely,
```

```
// (you can set loop = false to play the clip only once)
void PlayMyClipWithDelay()
{
    Gif.PlayClip(clipPlayer, myClip, 1f, true);
}

// This method pauses the player.
void PausePlayer()
{
    Gif.PausePlayer(clipPlayer);
}

// This method un-pauses the player.
void UnPausePlayer()
{
    Gif.ResumePlayer(clipPlayer);
}

// This method stops the player.
void StopPlayer()
{
    Gif.StopPlayer(clipPlayer);
}
```

Exporting GIF

To export the recorded clip into a GIF image, use the *ExportGif* method. In the editor, the exported GIF file will be stored right under the *Assets* folder; on mobile devices, the storage location is Application.persistentDataPath. You can specify the filename and the quality of the GIF image as well as the priority of the exporting thread. The quality setting accepts values from 1 to 100 (inputs will be clamped to this range). Bigger values will result in better looking GIFs, but will take slightly longer processing time; 80 is generally a good value in terms of time-quality balance. This method has two callbacks: one is called repeatedly during the process and receives the progress value (0 to 1), the other is called when the export completes and receives the file path of the generated image. Though the GIF generation process is done in a separate thread, these callbacks are guaranteed to be called from the main thread, so you can safely access all Unity API from within them.

In the rare case that you want to control the looping mode of the exported GIF (the default is loop indefinitely), use the variant of *ExportGif* that has a *loop* parameter (note that some GIF players may ignore this setting):

- *loop* < *0*: disable looping (play once)
- *loop* = *0*: loop indefinitely
- *loop* > *0*: loop a number of times

In the following example, we'll export a GIF image from the recorded clip returned after the recording has stopped.

```
// Put this on top of the script
using EasyMobile;



// Drag the camera with the Recorder component to this field in the inspector
public Recorder recorder;

// The recorded clip
AnimatedClip myClip;

// You can start recording as soon as your game starts
// (suppose you have a method named StartGame, which is called when the game starts)
void StartGame()
{
    // Start recording!
    Gif.StartRecording(recorder);

    // Do other stuff...
```

```
}



// A suitable time to stop recording can be when the game ends (the player dies)
// (suppose you have a method named GameOver, called when the game ends)
void GameOver()
{
    // Stop recording
    myClip = Gif.StopRecording(recorder);

    // Export GIF image from the resulted clip
    ExportMyGif();
}



// This method exports a GIF image from the recorded clip.
void ExportMyGif()
{
    // Parameter setup
    string filename = "myGif";    // filename, no need the ".gif" extension
    int loop = 0;                 // -1: no loop, 0: loop indefinitely, >0: loop a set number of times
    int quality = 80;             // 80 is a good value in terms of time-quality balance
    System.Threading.ThreadPriority tPriority = System.Threading.ThreadPriority.Normal; // exporting thread pri
ority

    Gif.ExportGif(myClip,
                  filename,
                  loop,
                  quality,
                  tPriority,
                  OnGifExportProgress,
                  OnGifExportCompleted);
}


// This callback is called repeatedly during the GIF exporting process.
// It receives a reference to original clip and a progress value ranging from 0 to 1.
void OnGifExportProgress(AnimatedClip clip, float progress)
{
    Debug.Log(string.Format("Export progress: {0:P0}", progress));
}


// This callback is called once the GIF exporting has completed.
// It receives a reference to the original clip and the filepath of the generated image.
void OnGifExportCompleted(AnimatedClip clip, string path)
{
    Debug.Log("A GIF image has been created at " + path);
}
```

Disposing of AnimatedClip

Internally, each *AnimatedClip* object consists of an array of *RenderTexture*, a "native engine object" type, which is not garbage collected as normal managed types. That means these render textures won't be "destroyed" automatically when their containing clip is garbage collected (the clip object does get collected, but the render textures it references don't, thus creating memory leaks). To take care of this issue, we have the *AnimatedClip* implement the *IDisposable* interface and provide the *Dispose* method to release the render textures, as Unity advised. It's strongly recommended that you call this *Dispose* method, preferably as soon as you're done with using a clip (e.g. after playing or exporting GIF), to make sure the render textures are properly released and not cause memory issues.

We'll extend the *OnGifExportCompleted* callback handler of the previous example to dispose the recorded clip as soon as we've generated a GIF image from it.

```
// This callback is called once the GIF exporting has completed.
// It receives a reference to the original clip and the filepath of the generated image.
```

```
void OnGifExportCompleted(AnimatedClip clip, string path)
{
    Debug.Log("A GIF image has been created at " + path);

    // We've done using the clip, dispose it to save memory
    if (clip == myClip)
    {
        myClip.Dispose();
        myClip = null;
    }
}
```

> Since version 2.1.0, we've updated *AnimatedClip* such that it automatically releases the RenderTexture object once it is collected, thus avoiding memory leaks even if you forget to call *Dispose*. However, it can take a long time before a clip gets collected by the garbage collector, so it's still a good practice to dispose a clip as soon as you're done using it to avoid wasting memory.

Sharing GIF

Now that a GIF image has been created, you may want to share it (because it's not fun otherwise, is it?). A common approach is to first upload the image to Giphy, a popular GIF hosting site, and then share the returned URL to other social networks like Facebook and Twitter, using Easy Mobile's Native Sharing feature (see the *Native Sharing* > *Scripting* section, in particular the *ShareURL* method).

> According to Giphy API documentation, hosted Giphy URLs are supported and play on every major social network.

Upload to Giphy

To upload a GIF image to Giphy, use the *Upload* method of the *Giphy* class. You can upload a local image on your device, or an image hosted online, provided that you have its URL. Before doing so, you'll need to prepare the upload content by creating a *GiphyUploadParams* struct. In this struct you'll specify either the file path of the local image, or the URL of the online image to upload. Note that if both parameters are provided, the local file path will be used over the URL. Within this struct you can also specify other optional parameters such as image tags, the source of the image (e.g. your website), or mark the image as private (only visible by you on Giphy). The *Upload* method has three callbacks: the first one is called repeatedly during the upload process, receiving a progress value (0 to 1); the second one is called once the upload has completed, receiving the URL of the uploaded image; and the last one will be called if the upload has failed, receiving the error message. All callbacks are called from the main thread.

> **Giphy Beta and Production Key**
>
> The *Upload* method has two variants: one using Giphy's public beta key, and the other using your own channel username and production API key. The public beta key is meant to be used in development only. According to Giphy Upload API documentation, it is "subject to rate limit constraints", and they "do not encourage live production deployments to use the public key". If you have created a Giphy channel and want to upload GIF images directly to that channel, you'll need to request an Upload Production Key, then provide that key and your channel username to the *Upload* method.

We'll extend the above example, and modify the *OnGifExportCompleted* callback handler to upload the GIF image to Giphy once it is created. We'll demonstrate two cases: upload using the public beta key and upload using your own production key.

```
// This callback is called once the GIF exporting has completed.
// It receives a reference to the original clip and the filepath of the generated image.
void OnGifExportCompleted(AnimatedClip clip, string path)
{
    Debug.Log("A GIF image has been created at " + path);
```

```
    // We've done using the clip, dispose it to save memory
    if (clip == myClip)
    {
        myClip.Dispose();
        myClip = null;
    }

    // The GIF image has been created, now we'll upload it to Giphy
    // First prepare the upload content
    var content = new GiphyUploadParams();
    content.localImagePath = path;    // the file path of the generated GIF image
    content.tags = "easy mobile, sglib games, unity";    // optional image tags, comma-delimited
    content.sourcePostUrl = "YOUR_WEBSITE_ADDRESS";    // optional image source, e.g. your website
    content.isHidden = false;    // optional hidden flag, set to true to mark the image as private

    // Upload the image to Giphy using the public beta key
    UploadToGiphyWithBetaKey(content);
}

// This method uploads a GIF image to Giphy using the public beta key,
// no need to specify any username or API key here.
void UploadToGiphyWithBetaKey(GiphyUploadParams content)
{
    Giphy.Upload(content, OnGiphyUploadProgress, OnGiphyUploadCompleted, OnGiphyUploadFailed);
}

// This method uploads a GIF image to your own Giphy channel,
// using your channel username and production key.
void UploadToGiphyWithProductionKey(GiphyUploadParams content)
{
    Giphy.Upload("YOUR_CHANNEL_USERNAME", "YOUR_PRODUCTION_KEY",
                 content,
                 OnGiphyUploadProgress,
                 OnGiphyUploadCompleted,
                 OnGiphyUploadFailed);
}

// This callback is called repeatedly during the uploading process.
// It receives a progress value ranging from 0 to 1.
void OnGiphyUploadProgress(float progress)
{
    Debug.Log(string.Format("Upload progress: {0:P0}", progress));
}

// This callback is called once the uploading has completed.
// It receives the URL of the uploaded image.
void OnGiphyUploadCompleted(string url)
{
    Debug.Log("The GIF image has been uploaded successfully to Giphy at " + url);
}

// This callback is called if the upload has failed.
// It receives the error message.
void OnGiphyUploadFailed(string error)
{
    Debug.Log("Uploading to Giphy has failed with error: " + error);
}
```

Display the Giphy Attribution Marks

To request a Production Key, Giphy require you to display the "Powered by Giphy" attribution marks whenever their API is utilized in your app, and provide screenshots of your attribution placement when submitting for the key. To take care of this, we provide the static *IsUsingAPI* boolean property inside the *Giphy* class. This property will be true as long as Giphy API is in use, to let you know when to show their attribution marks. You can display the attribution logo using an Image or a Sprite object, then poll this property inside the Update() function, and activate or deactivate the object accordingly.

> You can download Giphy's official attribution marks here.

```
// Drag the object displaying the attribution marks to this field in the inspector
public GameObject attribution;



void Update()
{
    attribution.SetActive(Giphy.IsUsingAPI);
}
```

Share Giphy URLs

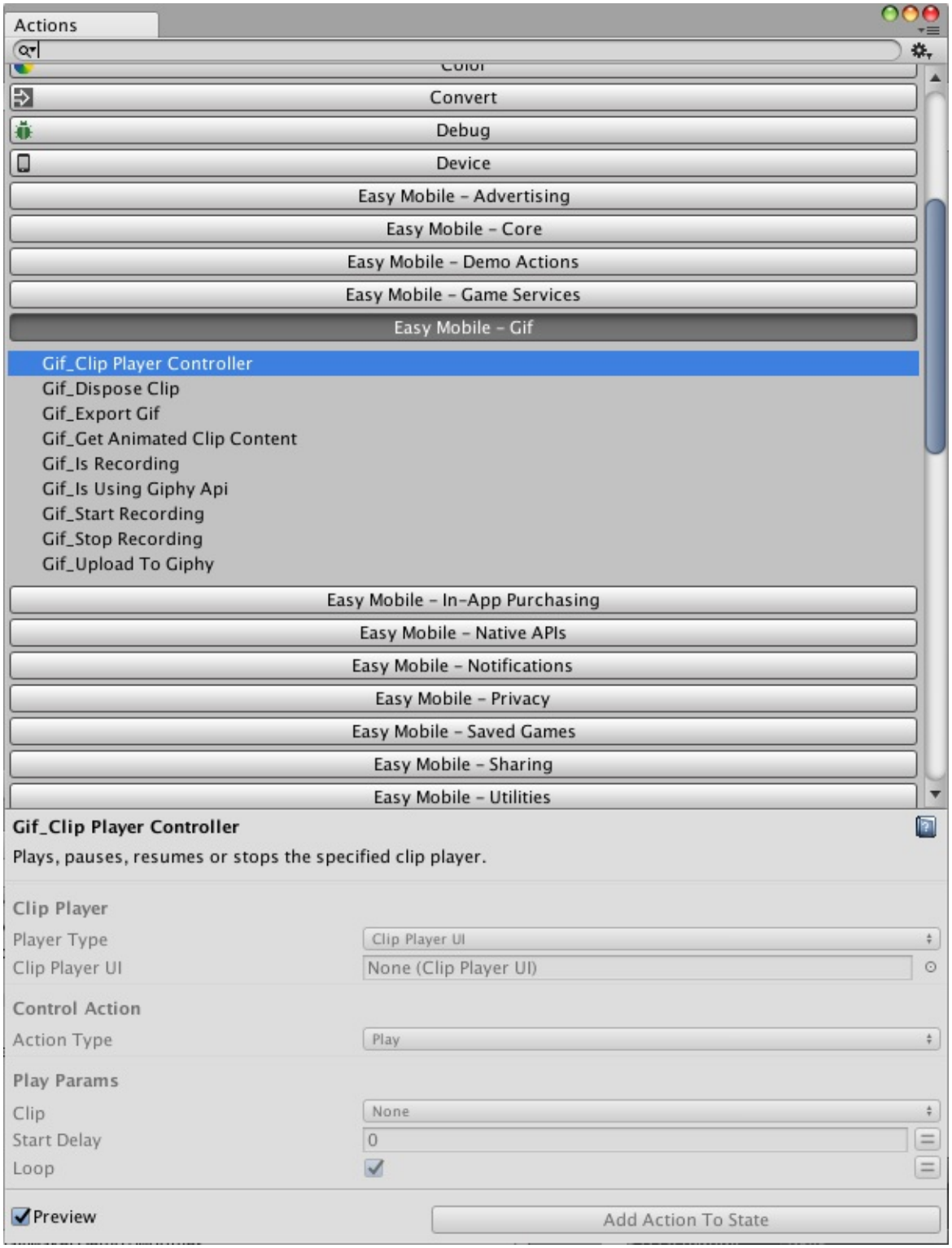
After uploading your GIF image to Giphy and obtain its URL, you can share this URL using the *ShareURL* method of the *MobileNativeShare* class. In the example below, we'll modify the *OnGiphyUploadCompleted* callback handler of the previous example to store the returned URL into a global variable, which can be used for later sharing.

```
// Global variable to hold the Giphy URL of the uploaded GIF
string giphyURL;



// This callback is called once the uploading has completed.
// It receives the URL of the uploaded image.
void OnGiphyUploadCompleted(string url)
{
    Debug.Log("The GIF image has been uploaded successfully to Giphy at " + url);

    // Store the URL into our global variable
    giphyURL = url;
}



// This method shares the URL using the native sharing utility on iOS and Android
public void ShareGiphyURL()
{
    if (!string.IsNullOrEmpty(giphyURL))
    {
        MobileNativeShare.ShareURL(giphyURL);
    }
}
```

GIF: PlayMaker Actions

The PlayMaker actions of the GIF module are group in the category *Easy Mobile - Gif* in the PlayMaker's Action Browser.

> Please refer to the GifDemo_PlayMaker scene in folder *Assets/EasyMobile/Demo/PlayMakerDemo/Modules* for an example on how these actions can be used.



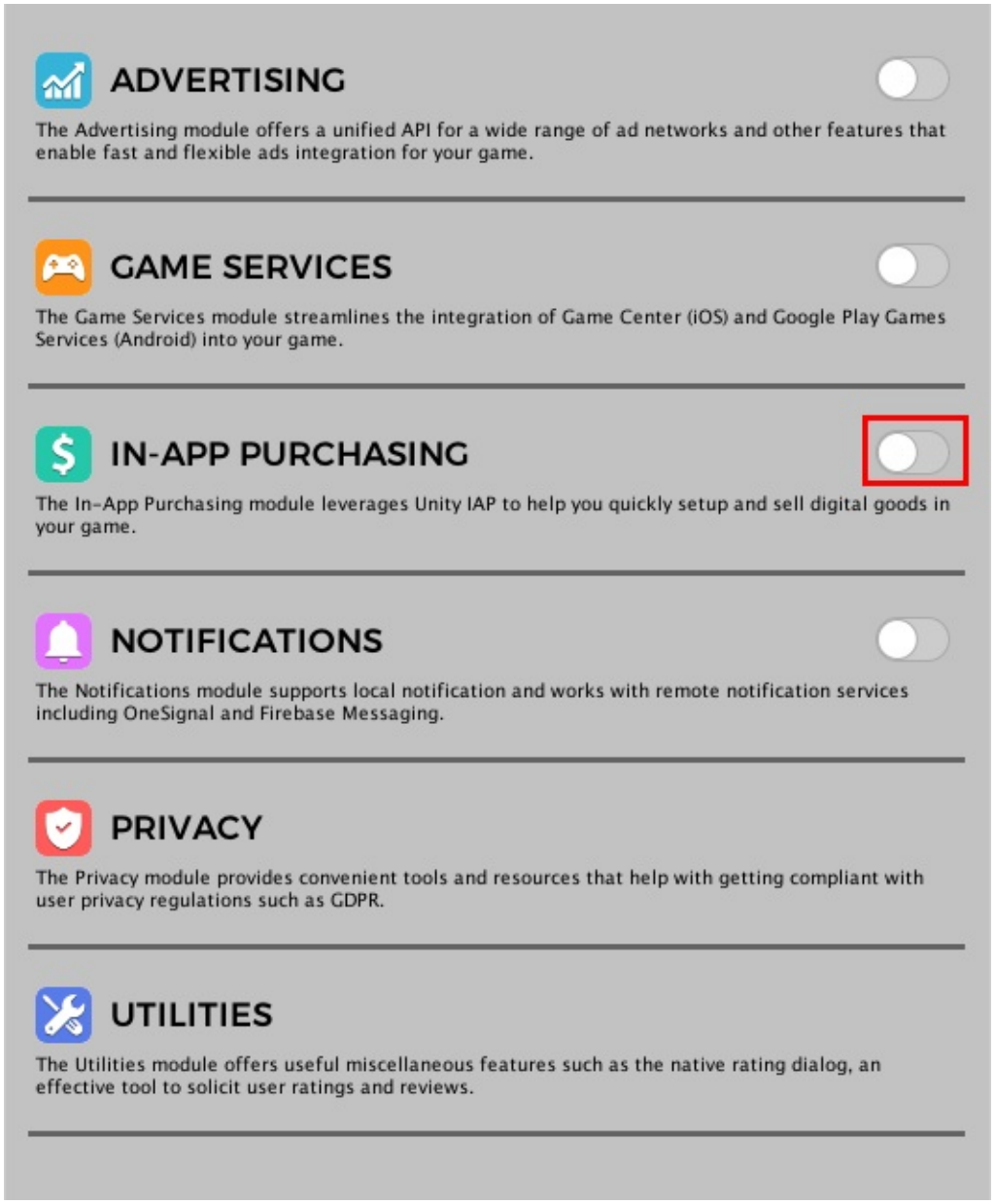
In-App Purchasing: Introduction

The In-App Purchasing module helps you quickly setup and sell digital products in your game. Here're some highlights of this modules:

- **Leverages Unity In-App Purchasing service**
  - This module is built on top of Unity IAP service, a powerful service that supports most app stores including iOS App Store, Google Play, Amazon Apps, Samsung GALAXY Apps and Tizen Store
  - Unity IAP is tightly integrated with the Unity engine, so compatibility and reliability can be expected
- **Easy management of product catalog**
  - Easy Mobile's custom editor features a friendly interface that helps you easily add, edit or remove products
- **Receipt validation**
  - Local receipt validation that offers extra security

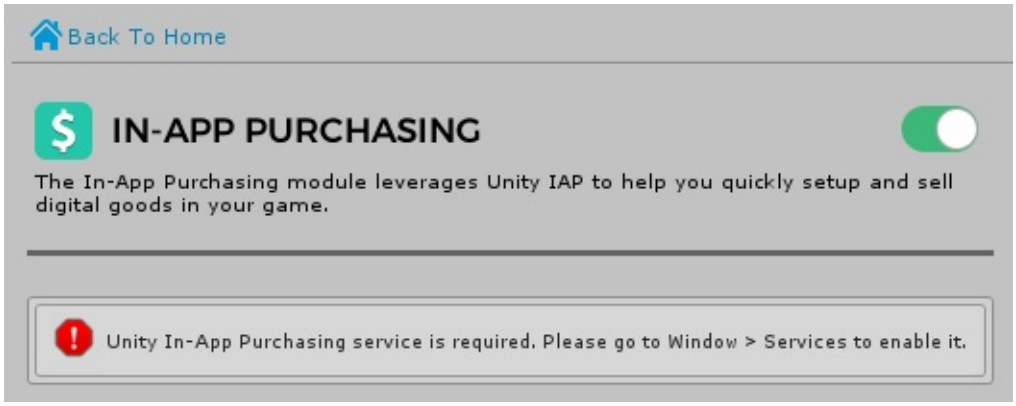
In-App Purchasing: Settings

To use the In-App Purchasing module you must first enable it. Go to *Window* > *Easy Mobile* > *Settings*, select the In-App Purchasing tab, then click the right-hand side toggle to enable and start configuring the module.



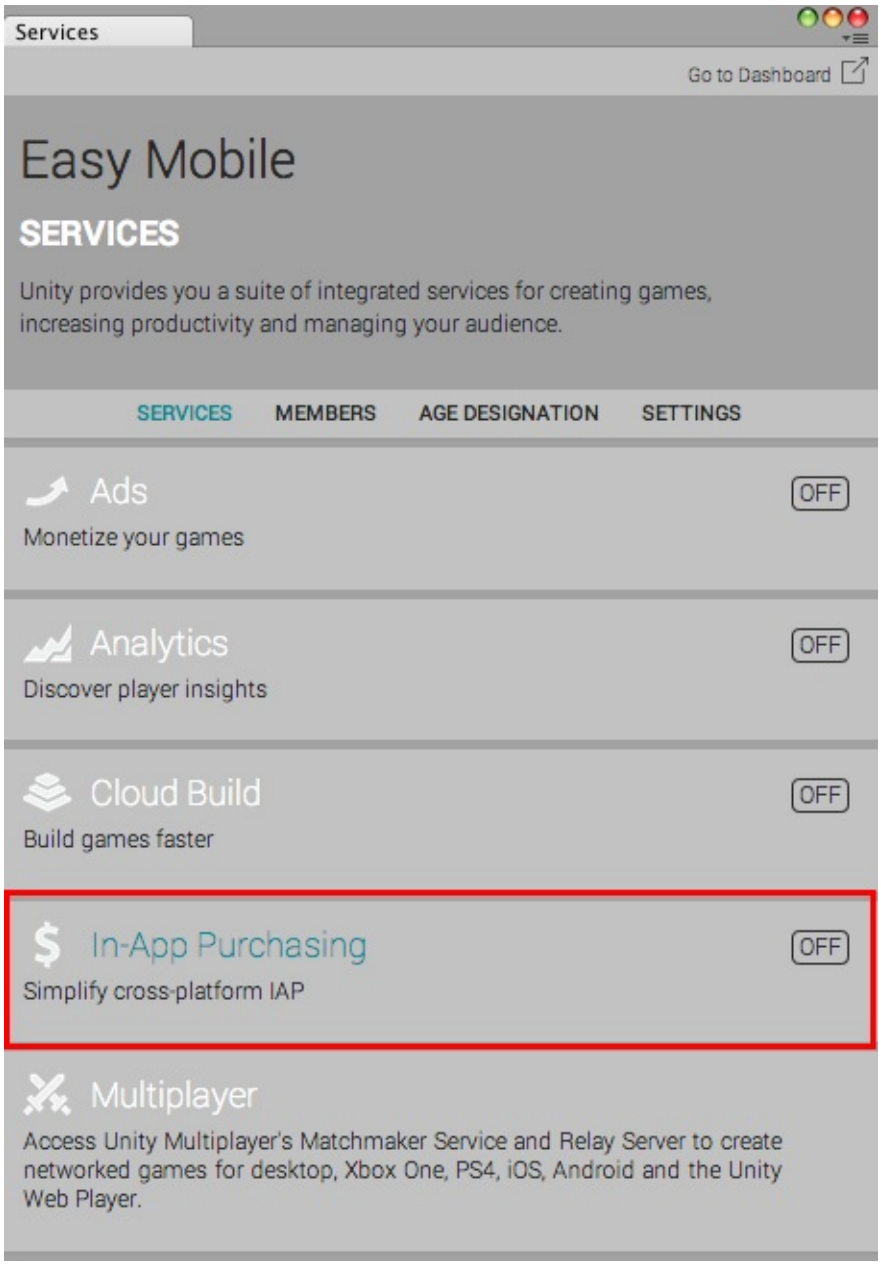
Enabling Unity IAP

The In-App Purchasing module requires Unity IAP service to be enabled. It will automatically check for the service's availability and prompt you to enable it if needed. Below is the module settings interface when Unity IAP is disabled.

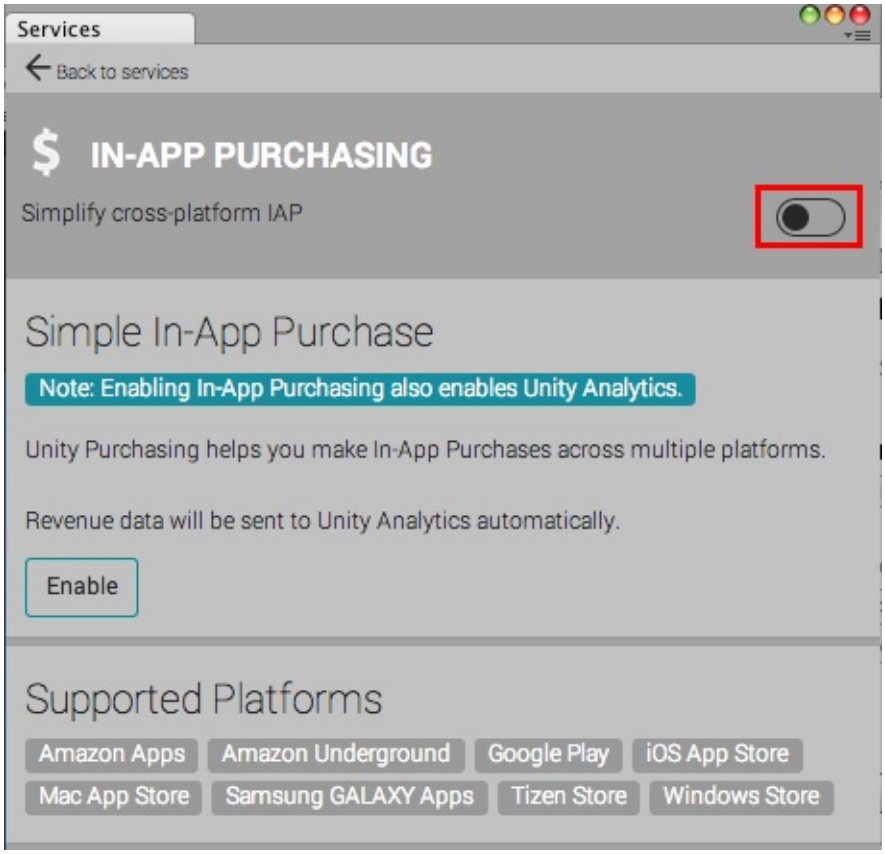


> To use Unity In-App Purchasing service, you must first set up your project for Unity Services.

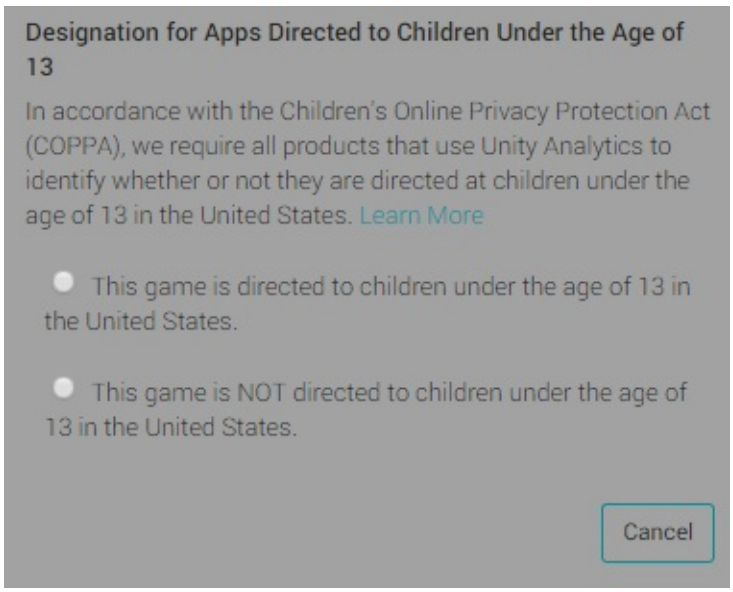
To enable Unity IAP service go to *Window* > *Services* and select the In-App Purchasing tab.



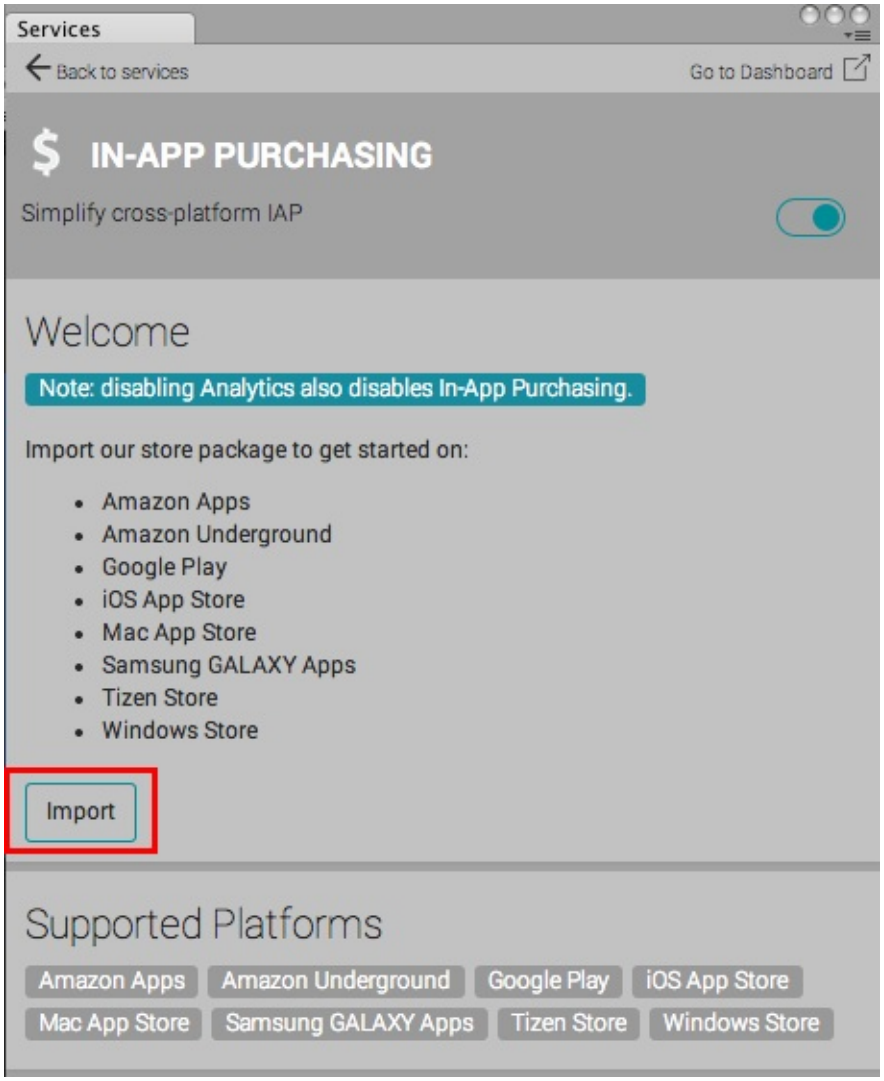
In the opened configuration window, click the toggle at the right-hand side or the *Enable* button to enable Unity IAP service.



A dialog window will appear asking a few questions about your game in order to ensure COPPA compliance.

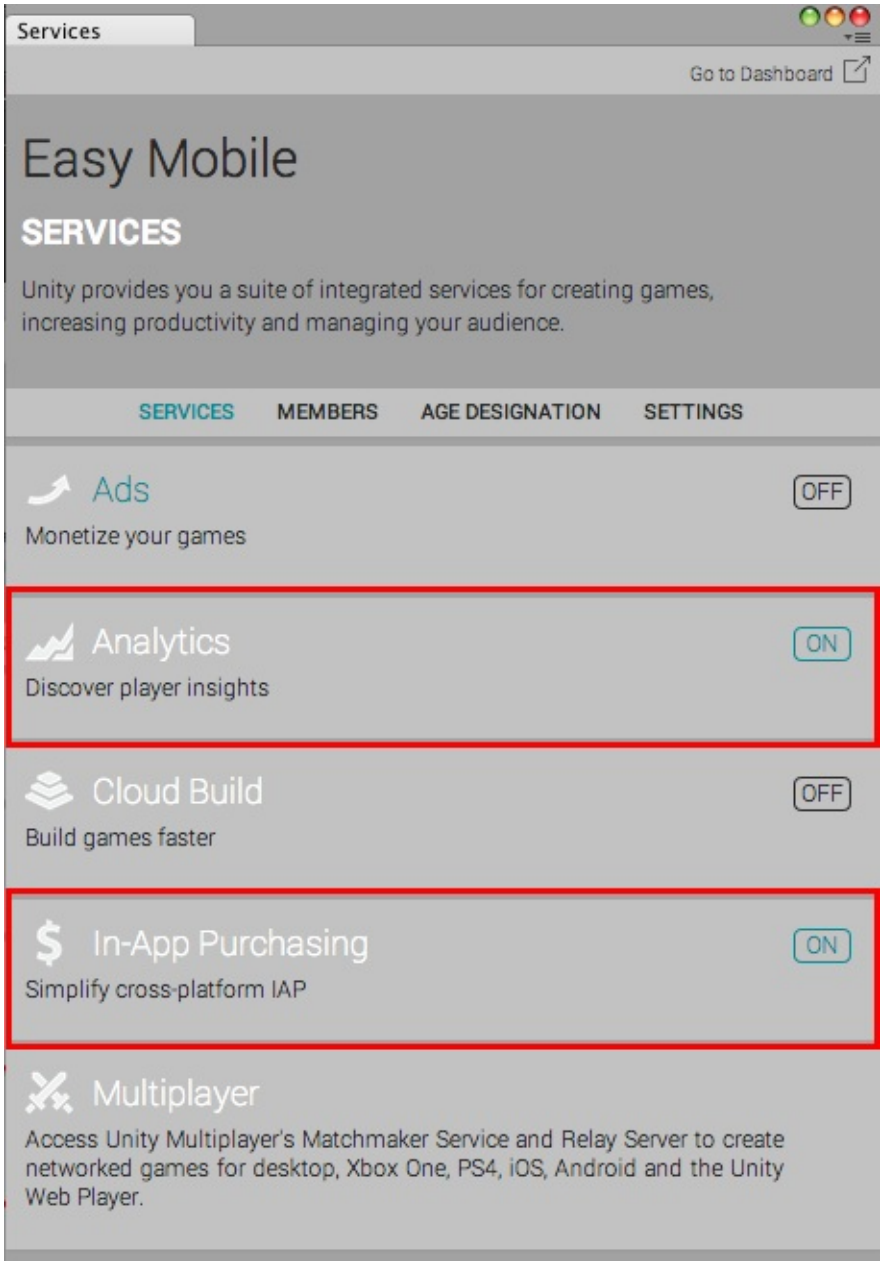


Next click the *Import* button to import the Unity IAP package to your project.



> After importing, there should be a UnityPurchasing folder added under Assets/Plugins folder.

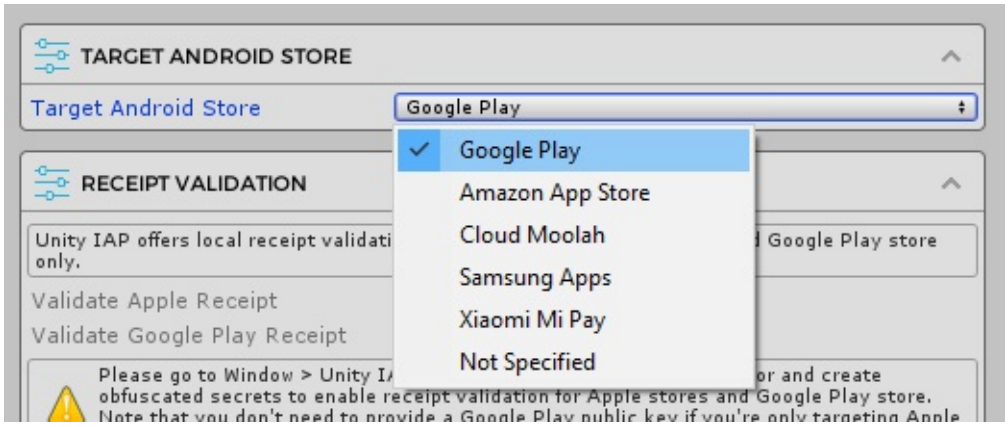
Enabling Unity IAP service will automatically enable the Unity Analytics service (if it's not enabled before), this is a requirement to use Unity IAP. Go back to the Services panel and make sure that both In-App Purchasing and Analytics services are now enabled.



After enabling Unity IAP service, the settings interface of the In-App Purchasing module will be updated and ready for you to start configuring.

Target Android Store

If you're building for Android platform, you need to specify your target store. In the **[ANDROID] TARGET STORE** section select your target store from the dropdown.



Apple Ask-To-Buy

Since iOS 8.0, Apple introduces a new parental control feature called Ask To Buy. Basically, Ask To Buy purchases will defer for parental approval. When this occurs, the In-App Purchasing module will notify your app by raising the *PurchaseDeferred* event, which you can subscribe to perform necessary actions, e.g. updating your UI to reflect the deferred state of the purchases.

In the **APPLE ASK-TO-BUY** section you can check the *Simulate Ask-To-Buy* option to enable the simulation of this feature in the sandbox environment, which is useful for testing during development.

> This setting has no effect on non-Apple platforms.

Apple Promotional Purchases

Apple allows you to promote in-app purchases through your app's product page. Unlike conventional in-app purchases, Apple promotional purchases initiate directly from the App Store on iOS and tvOS. The App Store then launches your app to complete the transaction, or prompts the user to download the app if it isn't installed.

You can instruct the In-App Purchasing module to intercept these promotional purchases by checking the *Intercept Promotional Purchases* option in the **APPLE PROMOTIONAL PURCHASES** section. Once a promotional purchase is intercepted, the *PromotionalPurchaseIntercepted* event will be fired. You can subscribe to this event and in its handler perform necessary actions such as presenting parental gates, sending analytics events, etc. before sending the purchase back to Apple by calling the *ContinueApplePromotionalPurchases* method, which will initiate any queued-up payments. If you do not enable this option, the promotional purchases will go through immediately with the *PurchaseCompleted* or *PurchaseFailed* event being fired according to the purchase result, and the *PromotionalPurchaseIntercepted* will never be raised.

> - The *Intercept Promotional Purchases* setting has no effect on non-Apple platforms.
> - It is vital to call the *ContinueApplePromotionalPurchases* method in the handler of the *PromotionalPurchaseIntercepted* event for the purchases to be processed properly after being intercepted.

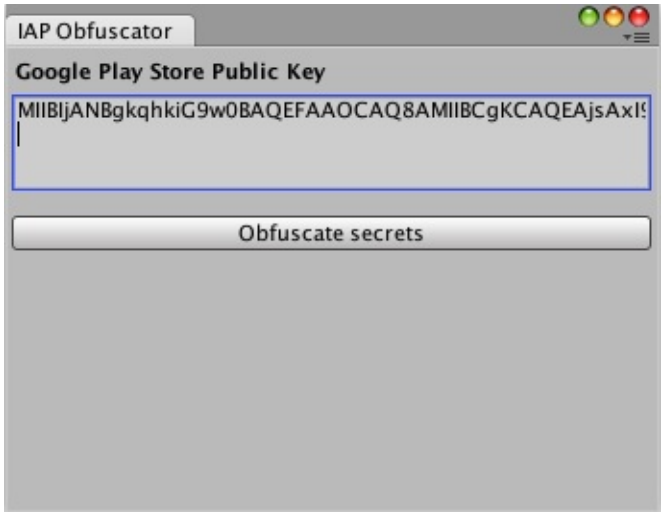
Receipt Validation

The receipt validation feature provides extra security and helps prevent prevent fraudulent users from accessing content they have not purchased. This feature employs Unity IAP's local receipt validation, which means the validation takes place on the target device, without the need to connect to a remote server.

> Receipt validation is available for Apple stores and Google Play store. Please find more information about Unity IAP's receipt validation here.

Obfuscating Encryption Keys

To enable receipt validation, you must first create obfuscated encryption keys. The purpose of this obfuscating process is to prevent a fraudulent user from accessing the actual keys, which are used for the validation process. To obfuscate your encryption keys, go to *Window* > *Unity IAP* > *Receipt Validation Obfuscator*.



In the opened **IAP Obfuscator** window, paste in your Google Play public key and hit the *Obfuscate secrets* button. According to Unity documentation, this will obfuscate both Apple's root certificate (bundle with Unity IAP) and the provided Google Play public key and create two C# files AppleTangle and GooglePlayTangle at *Assets/Plugins/UnityPurchasing/generated*. These files are required for the receipt validation process.

> To obtain the Google Play public key for your app, login to your Google Play Developer Console, select your app, then navigate to the **Services & APIs** section and find your key under the section labeled **YOUR LICENSE KEY FOR THIS APPLICATION**.
>
> Note that you don't need to provide a Google Play public key if you're only targeting Apple stores.

Enabling Receipt Validation

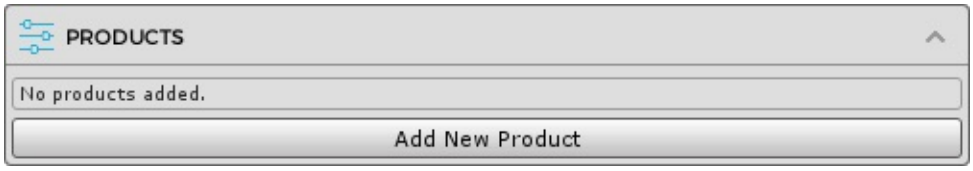
After creating the obfuscated encryption keys, you can now enable receipt validation for your game. Open the In-App Purchasing module settings, then in the **RECEIPT VALIDATION** section check the corresponding options for your targeted stores.

Product Management

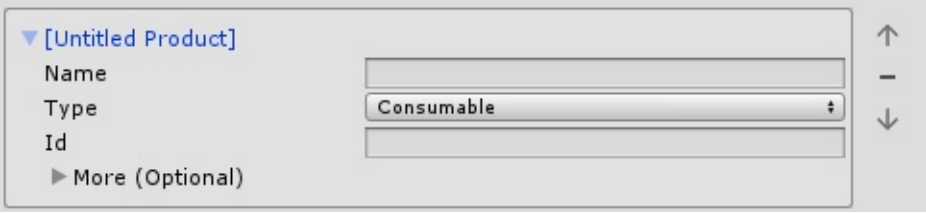
In the **PRODUCTS** section you can easily add, edit or remove your IAP products.

Adding a New Product

To add a new product, click the *Add New Product* button.



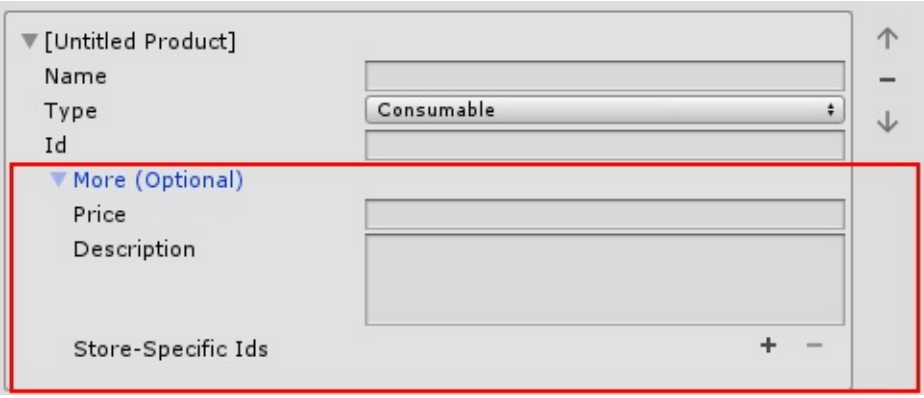
A new empty product will be added.



Fill in the required information for your new product:

- *Name*: the product name, can be used when making purchases
- *Type*: the product type, can be Consumable, Non-Consumable or Subscription
- *Id*: the unified product identifier, you should use this ID when declaring the product on your targeted stores; otherwise, if you need to have a different ID for this product on a certain store, add it to the *Store-Specific Ids* array (see below)

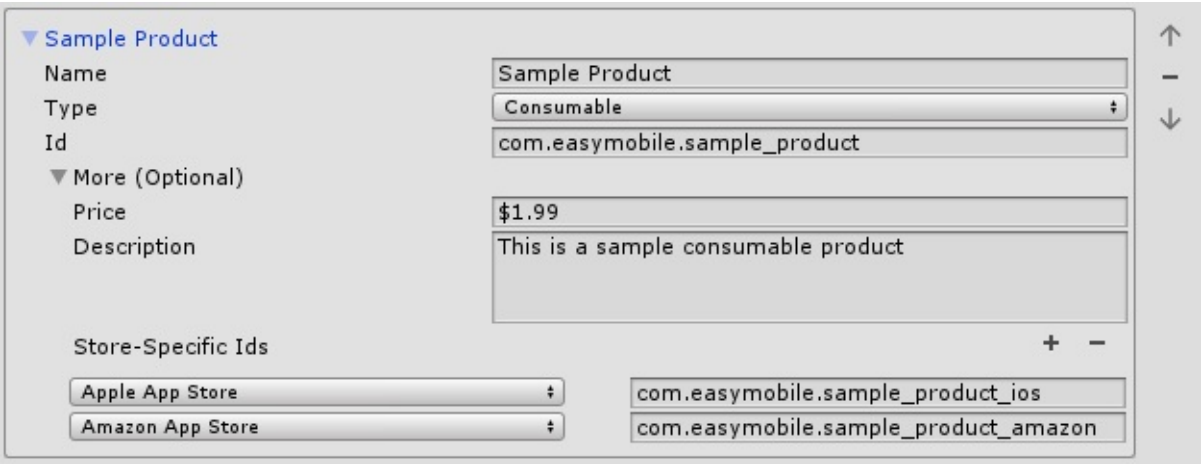
Click *More* if you need to enter store-specific IDs or fill in optional information for your product.



- *Price*: the product price string for displaying purpose
- *Description*: the product description for displaying purpose
- *Store-Specific Ids*: if you need to use a different product ID (than the unified ID provided above) on a certain store, you can add it here

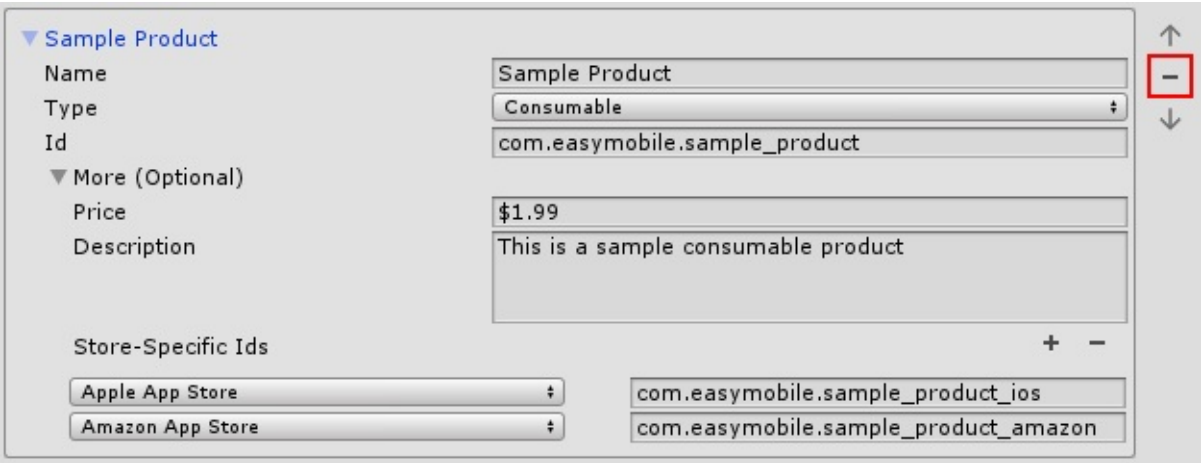
> **Adding Store-Specific ID**
> To add a new ID to the *Store-Specific Ids* array, increase the array size by adjusting the number in the right-hand side box. A new record will be added where you can select the targeted store and enter the corresponding product ID for that store.

Below is a sample product with all the information entered including the two store-specific IDs.



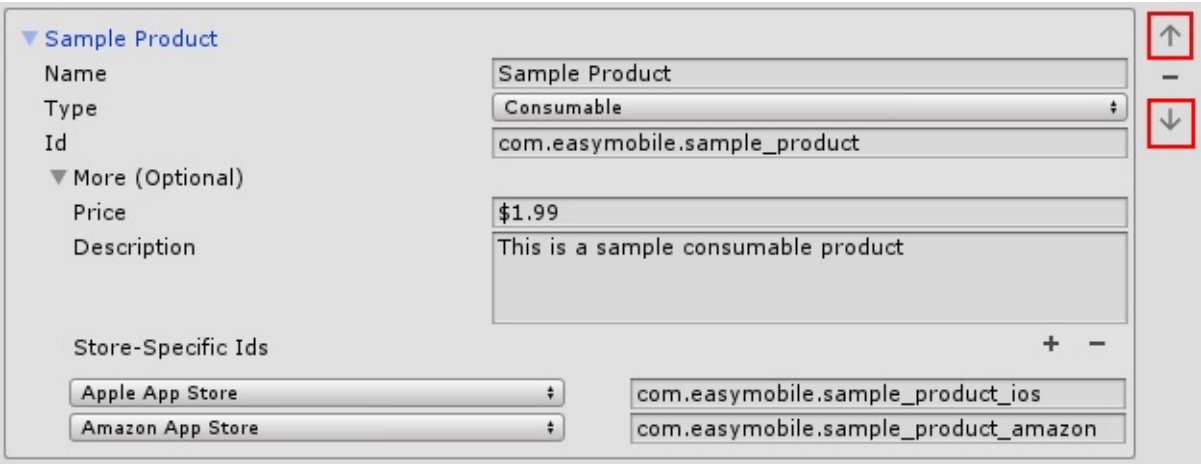
Removing a Product

To remove a product, simply click the [-] button at the right hand side.



Arranging Product List

You can use the two arrow-up and arrow-down buttons to move a product upward or downward within the product list.



Setup Products for Targeted Stores

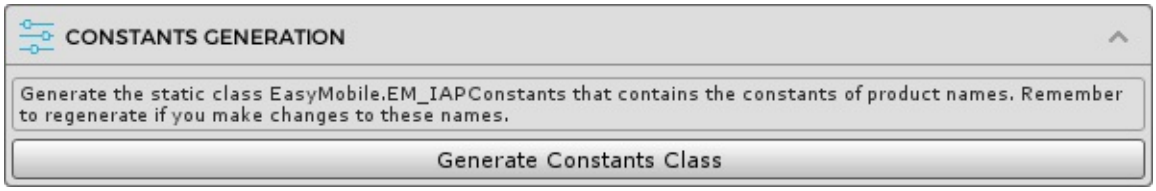
Beside creating the product list in Unity, you also need to declare similar products for your targeted stores, e.g. if you're targeting iOS App Store you need to create the products in iTunes Connect. If you're not familiar with the process, you can follow Unity's instructions on configuring IAP for various stores, which also include useful information about IAP testing.

> On Google Play store, both consumable and non-consumable products are defined as Managed product. If a product is set to Consumable type in Unity, the module will automatically handle the consumption of the product once it is bought and make it available for purchase again.

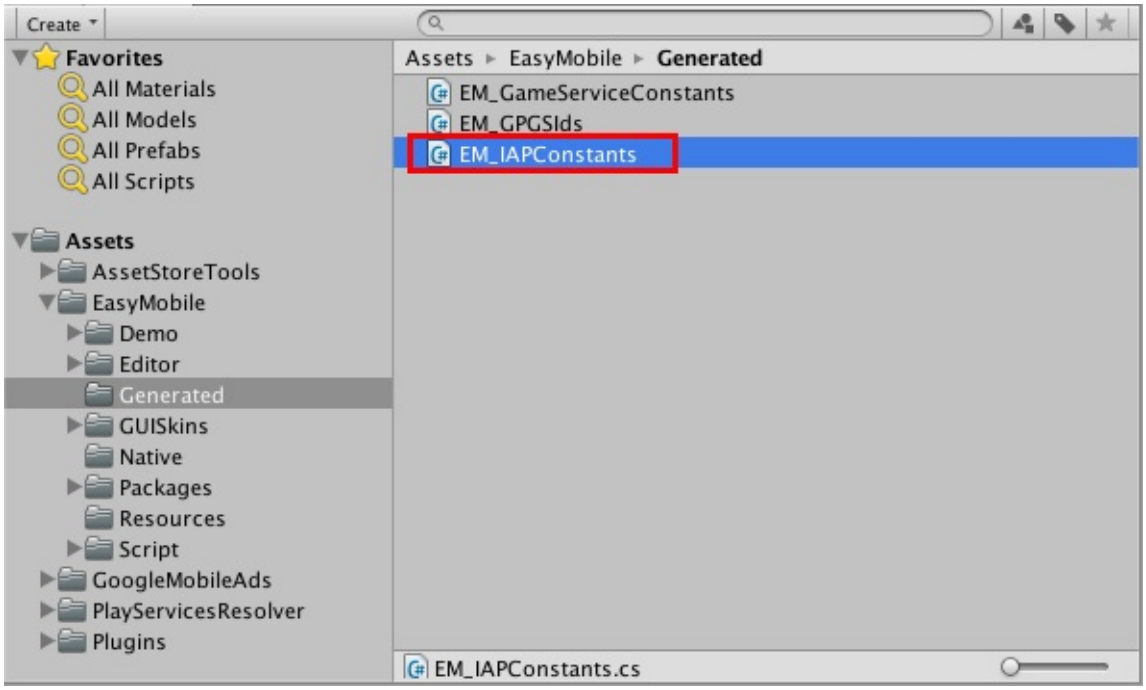
Constants Generation

Constants generation is a feature of the In-App Purchasing module. It reads all the product names and generates a static class named EM_IAPConstants that contains the constants of these names. Later, you can use these constants when making purchases in script instead of typing the product names directly, thus help prevent runtime errors due to typos and the likes.

To generate the constants class (you should do this after finishing with product editing), click the *Generate Constants Class* button within the **CONSTANTS CLASS GENERATION** section.



When the process completes, a file named EM_IAPConstants will be created at *Assets/EasyMobile/Generated*.



In-App Purchasing: Scripting

This section provides a guide to work with the In-App Purchasing module scripting API.

> You can access the In-App Purchasing module API via the InAppPurchasing class under the EasyMobile namespace.

Initialization

The module will automatically initialize Unity IAP at start without you having to do anything. All further API calls can only be made after the initialization has finished. You can check if Unity IAP has been initialized:

```
// Check if Unity IAP has been initialized
bool isInitialized = InAppPurchasing.IsInitialized();
```

Obtaining Product List

You can obtain the array of all products created in the module settings interface:

```
// Get the array of all products created in the In-App Purchasing module settings
// IAPProduct is the class representing a product as declared in the module settings
IAPProduct[] products = InAppPurchasing.GetAllIAPProducts();

// Print all product names
foreach (IAPProduct prod in products)
{
    Debug.Log("Product name: " + prod.Name);
}
```

Making Purchases

You can purchase a product using its name.

> It is strongly recommended that you use the constants of product names in the generated EM_IAPConstants class (see **IAP Constants Generation** section) instead of typing the names directly in order to prevent runtime errors due to typos and the likes.

```
// Purchase a product using its name
// EM_IAPConstants.Sample_Product is the generated name constant of a product named "Sample Product"
InAppPurchasing.Purchase(EM_IAPConstants.Sample_Product);
```

A *PurchaseCompleted* event will be fired if the purchase is successful, otherwise, a *PurchaseFailed* event will be fired instead. You can listen to these events and take appropriate actions, e.g. grant the user digital goods if the purchase has succeeded.

```
// Subscribe to IAP purchase events
void OnEnable()
{
    InAppPurchasing.PurchaseCompleted += PurchaseCompletedHandler;
    InAppPurchasing.PurchaseFailed += PurchaseFailedHandler;
}

// Unsubscribe when the game object is disabled
```

```
void OnDisable()
{
    InAppPurchasing.PurchaseCompleted -= PurchaseCompletedHandler;
    InAppPurchasing.PurchaseFailed -= PurchaseFailedHandler;
}

// Purchase the sample product
public void PurchaseSampleProduct()
{
    // EM_IAPConstants.Sample_Product is the generated name constant of a product named "Sample Product"
    InAppPurchasing.Purchase(EM_IAPConstants.Sample_Product);
}

// Successful purchase handler
void PurchaseCompletedHandler(IAPProduct product)
{
    // Compare product name to the generated name constants to determine which product was bought
    switch (product.Name)
    {
        case EM_IAPConstants.Sample_Product:
            Debug.Log("Sample_Product was purchased. The user should be granted it now.");
            break;
        case EM_IAPConstants.Another_Sample_Product:
            Debug.Log("Another_Sample_Product was purchased. The user should be granted it now.");
            break;
        // More products here...
    }
}

// Failed purchase handler
void PurchaseFailedHandler(IAPProduct product)
{
    Debug.Log("The purchase of product " + product.Name + " has failed.");
}
```

Checking Ownership

You can check if a product is owned by specifying its name. A product is considered "owned" if its receipt exists and passes the receipt validation (if enabled).

```
// Check if the product is owned by the user
// EM_IAPConstants.Sample_Product is the generated name constant of a product named "Sample Product"
bool isOwned = InAppPurchasing.IsProductOwned(EM_IAPConstants.Sample_Product);
```

> Consumable products' receipts are not persisted between app restarts, therefore this method only returns true for those products in the session they're purchased.
>
> In the case of subscription products, this method simply checks if a product has been bought (subscribed) before and has a receipt. It doesn't check if the subscription is expired or not.

Restoring Purchases

Non-consumable and subscription products are restorable. App stores maintain a permanent record of each user's non-consumable and subscription products, so that he or she can be granted these products again when reinstalling your game.

Apple normally requires a *Restore Purchases* button to exist in your game, so that the users can explicitly initiate the purchase restoration process. On other platforms, e.g. Google Play, the restoration is done automatically during the first initialization after reinstallation.

> During the restoration process, a *PurchaseCompleted* event will be fired for each owned product, as if the user has just purchased them again. Therefore you can reuse the same handler to grant the user their products as normal purchases.

On iOS, you can initiate a purchase restoration as below.

```
// Restore purchases. This method only has effect on iOS.
InAppPurchasing.RestorePurchases();
```

A *RestoreCompleted* event will be fired if the restoration is successful, otherwise, a *RestoreFailed* event will be fired instead. Note that these events only mean the success or failure of the restoration itself, while the *PurchaseCompleted* event will be fired for each restored product, as noted earlier. You can listen to these events and take appropriate actions, e.g. inform the user the restoration result.

> The *RestoreCompleted* and *RestoreFailed* events are only raised on iOS.

```
// Subscribe to IAP restore events, these events are fired on iOS only.
void OnEnable()
{
    InAppPurchasing.RestoreCompleted += RestoreCompletedHandler;
    InAppPurchasing.RestoreFailed += RestoreFailedHandler;
}

// Successful restoration handler
void RestoreCompletedHandler()
{
    Debug.Log("All purchases have been restored successfully.");
}

// Failed restoration handler
void RestoreFailedHandler()
{
    Debug.Log("The purchase restoration has failed.");
}

// Unsubscribe
void OnDisable()
{
    InAppPurchasing.RestoreCompleted -= RestoreCompletedHandler;
    InAppPurchasing.RestoreFailed -= RestoreFailedHandler;
}
```

Apple Ask-To-Buy

On iOS 8.0 or newer, Ask To Buy purchases will defer for parental approval. You can subscribe to the *PurchaseDeferred* event to acknowledge when this occurs, and perform relevant actions such as updating UI to reflect the deferred state of the purchase. When the purchase is approved or rejected, the normal *PurchaseCompleted* or *PurchaseFailed* events will be fired.

```
// Subscribe to Ask To Buy purchases deferred event, this event is fired on iOS only.
void OnEnable()
{
    InAppPurchasing.PurchaseDeferred += PurchaseDeferredHandler;
}

// Unsubscribe.
void OnDisable()
{
    InAppPurchasing.PurchaseDeferred -= PurchaseDeferredHandler;
}

// This handler is invoked once an Ask To Buy purchase is deferred for parental approval.
void PurchaseDeferredHandler(IAPProduct product)
```

```
{
    Debug.Log("Purchase of product " + product.Name + " has been deferred.");

    // Perform necessary actions, e.g. updating UI to inform user that
    // the purchase has been deferred...
}
```

> You can simulate Ask To Buy feature in the sandbox app store for testing during development, see Settings/Apple Ask-To-Buy.

Apple Promotional Purchases

If you have enabled the *Intercept Promotional Purchases* option in the In-App Purchasing module settings, the *PromotionalPurchaseIntercepted* event will be fired every time a promotional purchase is intercepted. In the handler of this event you can perform relevant actions such as presenting parental gates, sending analytics events, etc. After that you must call the *ContinueApplePromotionalPurchases* method to continue the normal processing of the purchase. This will initiate any queued-up payments. Once the transaction is done, the normal *PurchaseCompleted* or *PurchaseFailed* event will be fired according to the purchase result.

> If the *Intercept Promotional Purchases* option is disabled, the *PromotionalPurchaseIntercepted* event will never occur.

```
// Subscribe to promotional purchase intercepted event, this event is fired on iOS only.
void OnEnable()
{
    InAppPurchasing.PromotionalPurchaseIntercepted += PromotionalPurchaseInterceptedHandler;
}

// Unsubscribe.
void OnDisable()
{
    InAppPurchasing.PromotionalPurchaseIntercepted -= PromotionalPurchaseInterceptedHandler;
}

// This handler is invoked once a promotional purchase is intercepted.
void PromotionalPurchaseInterceptedHandler(IAPProduct product)
{
    Debug.Log("Promotional purchase of product " + product.Name + " has been intercepted.");

    // Here you can perform necessary actions, e.g. presenting parental gates,
    // sending analytics events, etc.


    // Finally, you must call the ContinueApplePromotionalPurchases method
    // to continue the normal processing of the purchase!
    InAppPurchasing.ContinueApplePromotionalPurchases();
}
```

You can also use the *SetAppleStorePromotionVisibility* and *SetAppleStorePromotionOrder* methods to respectively set the visibility of a promotional product and the order of visible promotional products on the Apple app store of the current device.

In-App Purchasing: Advanced Scripting

This section describes the methods to accomplish tasks beyond the basic ones such as making or restoring purchases. These tasks include retrieving product localized data, reading product receipts, refreshing receipts, etc.

> Most of the methods described in this section are only available once Easy Mobile's IAP module and Unity IAP service are enabled, which is indicated by the definition of the symbol **EM_UIAP**. Therefore, you should always wrap the use of these methods inside a check for the existing of this symbol.
>
> Also, the types exposed in these methods are only available when the Unity IAP package is imported, and you should include the UnityEngine.Purchasing and UnityEngine.Purchasing.Security namespaces at the top of your script for these types to be recognized.

Getting Unity IAP's Product Object

The in-app products are represented in Unity IAP by the Product class, which is different from Easy Mobile's IAPProduct class, whose main purpose is for settings and displaying. This Product class is the entry point to access product-related data including its metadata and receipt, which is populated automatically by Unity IAP. To obtain the Product object of an in-app product, call the *GetProduct* method with the product name.

```
#if EM_UIAP
using UnityEngine.Purchasing;
#endif



// Obtain the Product object of the sample product and print its data
public void GetSampleProduct()
{
    #if EM_UIAP
    // EM_IAPConstants.Sample_Product is the generated name constant of a product named "Sample Product"
    Product sampleProduct = InAppPurchasing.GetProduct(EM_IAPConstants.Sample_Product);

    if (sampleProduct != null)
    {
        Debug.Log("Available To Purchase: " + sampleProduct.availableToPurchase.ToString());
        if (sampleProduct.hasReceipt)
        {
            Debug.Log("Receipt: " + sampleProduct.receipt);
        }
    }
    #endif
}
```

Getting Product Localized Data

You can get a product's metadata retrieved from targeted app stores, e.g. localized title, description and price. This information is particularly useful when building a storefront in your game for displaying the in-app products. To get the localized data of a product, call the *GetProductLocalizedData* and specify the product name. The following example iterates through the product list and retrieve the localized data of each item.

```
#if EM_UIAP
using UnityEngine.Purchasing;
#endif
```

```
// Iterate through the product list and get the localized data retrieved from the targeted app store.
// Note the check for the EM_UIAP symbol.
void PrintProductsMetadata()
{
    #if EM_UIAP
    // Get all products created in the In-App Purchasing module settings
    IAPProduct[] products = EM_Settings.InAppPurchasing.Products;

    foreach (IAPProduct prod in products)
    {
        // Get product localized data.
        ProductMetadata data = InAppPurchasing.GetProductLocalizedData(prod.Name);

        if (data != null)
        {
            Debug.Log("Localized title: " + data.localizedTitle);
            Debug.Log("Localized description: " + data.localizedDescription);
            Debug.Log("Localized price string: " + data.localizedPriceString);
        }
    }
    #endif
}
```

Getting Subscription Info

You can get a subscription product's information such as expire date using the *GetSubscriptionInfo* method. Internally, this method uses Unity IAP's SubscriptionManager class to retrieve the subscription data. The following example iterates through the product list and prints the information of each subscription product.

```
#if EM_UIAP
using UnityEngine.Purchasing;
#endif


// Iterates through all available products and prints the data of subscriptions.
public void PrintSubscriptionInfo()
{
    #if EM_UIAP
    // Get all products created in the In-App Purchasing module settings.
    IAPProduct[] products = EM_Settings.InAppPurchasing.Products;

    foreach (IAPProduct p in products)
    {
        // If this is a subscription product.
        if (p.Type == IAPProductType.Subscription)
        {
            // Get the subscription information of the current product,
            // note that this method takes the product name as input.
            SubscriptionInfo info = InAppPurchasing.GetSubscriptionInfo(p.Name);

            if (info == null)
            {
                Debug.Log("The subscription information of this product could not be retrieved.");
                continue;
            }

            // Prints subscription info.
            Debug.Log("Product ID: " + info.getProductId());
            Debug.Log("Purchase Date: " + info.getPurchaseDate());
            Debug.Log("Is Subscribed: " + info.isSubscribed());
            Debug.Log("Is Expired: " + info.isExpired());
            Debug.Log("Is Cancelled: " + info.isCancelled());
            Debug.Log("Is FreeTrial: " + info.isFreeTrial());
            Debug.Log("Is Auto Renewing: " + info.isAutoRenewing());
            Debug.Log("Remaining Time: " + info.getRemainingTime().ToString());
            Debug.Log("Is Introductory Price Period: " + info.isIntroductoryPricePeriod());
            Debug.Log("Introductory Price Period: " + info.getIntroductoryPricePeriod().ToString());
            Debug.Log("Introductory Price Period Cycles: " + info.getIntroductoryPricePeriodCycles());
            Debug.Log("Introductory Price: " + info.getIntroductoryPrice());
```

```
            Debug.Log("Expire Date: " + info.getExpireDate());
        }
    }
    #endif
}
```

Working with Receipts

This sections describes methods to work with receipts. Currently, Unity IAP only supports parsing receipts from Apple stores and Google Play store.

> Note that for the receipt reading methods to work, you need to enable receipt validation feature (see the **Receipt Validation** section).

Apple App Receipt

On iOS, you can get the parsed Apple App Receipt for your app using the *GetAppleAppReceipt* method.

```
#if EM_UIAP
using UnityEngine.Purchasing;
using UnityEngine.Purchasing.Security;
#endif



// Read the App Receipt on iOS. Receipt validation is required.
void ReadAppleAppReceipt()
{
    #if EM_UIAP
    if (Application.platform == RuntimePlatform.IPhonePlayer)
    {
        AppleReceipt appReceipt = InAppPurchasing.GetAppleAppReceipt();

        // Print the receipt content.
        if (appReceipt != null)
        {
            Debug.Log("App Version: " + appReceipt.appVersion);
            Debug.Log("Bundle ID: " + appReceipt.bundleID);
            Debug.Log("Number of purchased products: " + appReceipt.inAppPurchaseReceipts.Length);
        }
    }
    #endif
}
```

Apple InAppPurchase Receipt

On iOS, you can get the parsed Apple InAppPurchase receipt for a particular product, using the *GetAppleIAPReceipt* method with the name of the product.

```
#if EM_UIAP
using UnityEngine.Purchasing;
using UnityEngine.Purchasing.Security;
#endif



// Read the InAppPurchase receipt of the sample product on iOS.
// Receipt validation is required.
void ReadAppleInAppPurchaseReceipt()
{
    #if EM_UIAP
    if (Application.platform == RuntimePlatform.IPhonePlayer)
    {
        // EM_IAPConstants.Sample_Product is the generated name constant of a product named "Sample Product".
```

```
        AppleInAppPurchaseReceipt receipt = InAppPurchasing.GetAppleIAPReceipt(EM_IAPConstants.Sample_Product);

        // Print the receipt content.
        if (receipt != null)
        {
            Debug.Log("Product ID: " + receipt.productID);
            Debug.Log("Original Purchase Date: " + receipt.originalPurchaseDate.ToShortDateString());
            Debug.Log("Original Transaction ID: " + receipt.originalTransactionIdentifier);
            Debug.Log("Purchase Date: " + receipt.purchaseDate.ToShortDateString());
            Debug.Log("Transaction ID: " + receipt.transactionID);
            Debug.Log("Quantity: " + receipt.quantity);
            Debug.Log("Cancellation Date: " + receipt.cancellationDate.ToShortDateString());
            Debug.Log("Subscription Expiration Date: " + receipt.subscriptionExpirationDate.ToShortDateString())
;
        }
    }
    #endif
}
```

Google Play Receipt

On Android, you can get the parse GooglePlay receipt for a particular product, using the *GetGooglePlayReceipt* method with the name of the product.

```
#if EM_UIAP
using UnityEngine.Purchasing;
using UnityEngine.Purchasing.Security;
#endif



// Read the GooglePlay receipt of the sample product on Android.
// Receipt validation is required.
void ReadGooglePlayReceipt()
{
    #if EM_UIAP
    if (Application.platform == RuntimePlatform.Android)
    {
        // EM_IAPConstants.Sample_Product is the generated name constant of a product named "Sample Product".
        GooglePlayReceipt receipt = InAppPurchasing.GetGooglePlayReceipt(EM_IAPConstants.Sample_Product);

        if (receipt != null)
        {
            Debug.Log("Package Name: " + receipt.packageName);
            Debug.Log("Product ID: " + receipt.productID);
            Debug.Log("Purchase Date: " + receipt.purchaseDate.ToShortDateString());
            Debug.Log("Purchase State: " + receipt.purchaseState.ToString());
            Debug.Log("Transaction ID: " + receipt.transactionID);
            Debug.Log("Purchase Token: " + receipt.purchaseToken);
        }
    }
    #endif
}
```

Refreshing Apple App Receipt

Apple provides a mechanism to fetch a new App Receipt from their servers, typically used when no receipt is currently cached in local storage SKReceiptRefreshRequest. You can refresh the App Receipt on iOS using the *RefreshAppleAppReceipt* method. Note that this will prompt the user for their password.

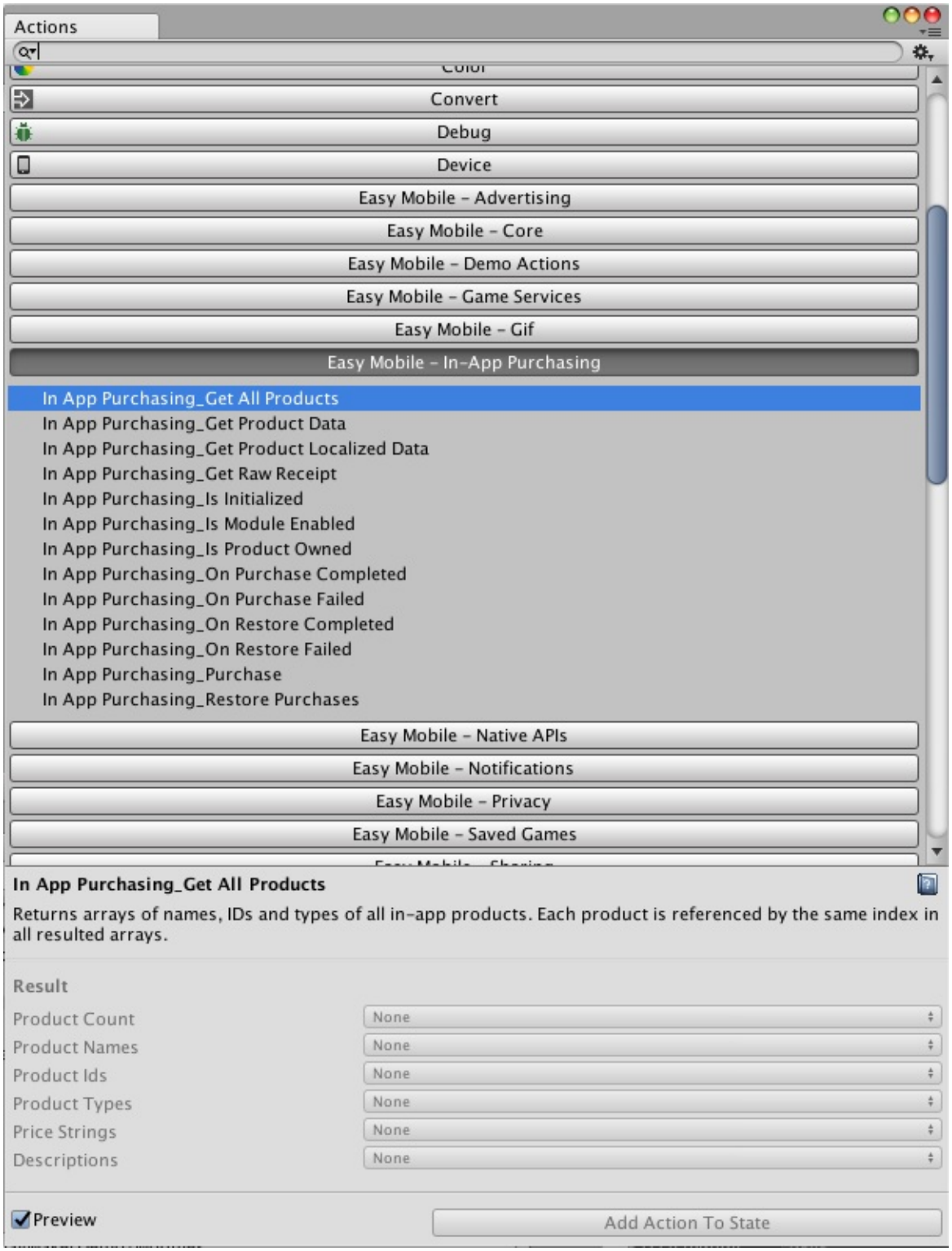
```
// Fetch a new Apple App Receipt on iOS. This will prompt the user for their password.
void RefreshAppleAppReceipt()
{
    if (Application.platform == RuntimePlatform.IPhonePlayer)
    {
        InAppPurchasing.RefreshAppleAppReceipt(SuccessCallback, ErrorCallback);
```

```
    }
}

void SuccessCallback(string receipt)
{
    Debug.Log("App Receipt refreshed successfully. New receipt: " + receipt);
}

void ErrorCallback()
{
    Debug.Log("App Receipt refreshing failed.");
}
```

In-App Purchasing: PlayMaker Actions

The PlayMaker actions of the In-App Purchasing module are group in the category *Easy Mobile - In-App Purchasing* in the PlayMaker's Action Browser.

> Please refer to the InAppPurchasingDemo_PlayMaker scene in folder *Assets/EasyMobile/Demo/PlayMakerDemo/Modules* for an example on how these actions can be used.



Native APIs: Introduction

The Native APIs module allows access to mobile native functionalities. The first feature available is native UI. More functionalities will be added soon.

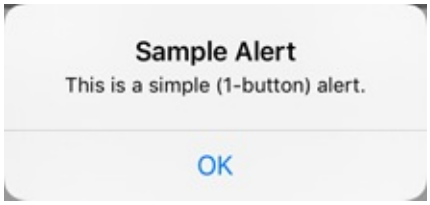
Native APIs | Native UI: Introduction

The Native UI module allows you to access native mobile UI elements such as alerts and dialogs. This module requires no configuration and all tasks can be done from script.

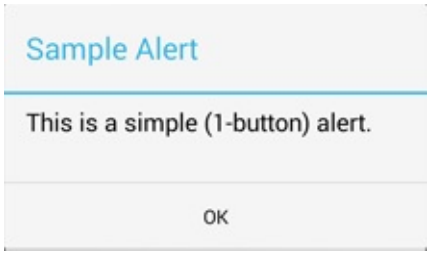
Alerts

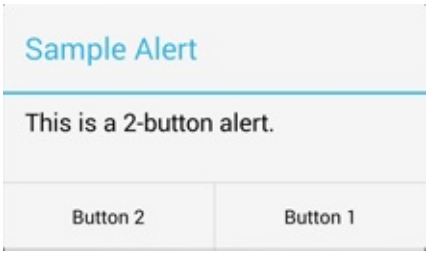
Alerts are useful in providing the users contextual information, asking for confirmation or prompting them to make a selection out of several options. An alert can have one, two or three buttons with it.

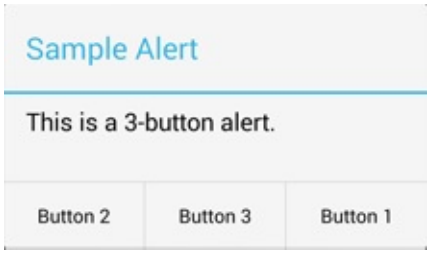
Below are the three types of alert on iOS.



And below are the three types of alert on Android.







Toasts

Toasts are short messages displayed at the bottom of the screen. They automatically disappear after a timeout. Toasts are available on Android only. Below is a sample toast message.


HOME
NATIVE UI
isFirstButtonClicked: FALSE
isSecondButtonClicked: FALSE
isThirdButtonClicked: TRUE
ALERT
2-BUTTON ALERT
3-BUTTON ALERT
A toast is a view containing a quick little message for the user. It's only available on Android.
ANDROID TOAST

This is a sample Android toast

Native APIs | Native UI: Scripting

This section provides a guide to work with Native UI scripting API.

> You can access the Native UI module API via the NativeUI class under the EasyMobile namespace.

Alerts

Alerts are available on both iOS and Android platform and can have up to three buttons.

Simple (one-button) alerts are useful in giving the user contextual information. To show a simple alert with the default OK button, you only need to provide a title and a message for the alert:

```
// Show a simple alert with OK button
NativeUI.AlertPopup alert = NativeUI.Alert("Sample Alert", "This is a sample alert with an OK button.");
```

You can also show a one-button alert with a custom button label.

```
// Show an alert with a button labeled as "Got it"
NativeUI.AlertPopup alert = NativeUI.Alert(
    "Sample Alert",
    "This is a sample alert with a custom button.",
    "Got it"
    );
```

Two-button alerts can be useful when needing to ask for user confirmation. To show a two-button alert, you need to specify the labels of these two buttons.

```
// Show a two-button alert with the buttons labeled as "Button 1" & "Button 2"
NativeUI.AlertPopup alert = NativeUI.ShowTwoButtonAlert(
    "Sample Alert",
    "This is a two-button alert.",
    "Button 1",
    "Button 2"
    );
```

Three-button alerts can be used to present the user with several options, a typical usage of it is to implement the Rate Us popup. To show a three-button alert, you need to specify the labels of the three buttons.

```
// Show a three-button alert with the buttons labeled as "Button 1", "Button 2" & "Button 3"
NativeUI.AlertPopup alert = NativeUI.ShowThreeButtonAlert(
    "Sample Alert",
    "This is a three-button alert.",
    "Button 1",
    "Button 2",
    "Button 3"
    );
```

Whenever an alert is shown, a *NativeUI.AlertPopup* object is returned, when the alert is closed, this object will fire an *OnComplete* event and then destroy itself. The argument of this event is the index of the clicked button. You should listen to this event and take appropriate action depending on the button selected.

```
// Show a three button alert and handle its OnComplete event
NativeUI.AlertPopup alert = NativeUI.ShowThreeButtonAlert(
    "Sample Alert",
    "This is a three-button alert.",
```

```
    "Button 1",
    "Button 2",
    "Button 3"
    );

// Subscribe to the event
if (alert != null)
{
    alert.OnComplete += OnAlertCompleteHandler;
}

// The event handler
void OnAlertCompleteHandler(int buttonIndex)
{
    switch (buttonIndex)
    {
        case 0:
            // Button 1 was clicked
            break;
        case 1:
            // Button 2 was clicked
            break;
        case 2:
            // Button 3 was clicked
            break;
        default:
            break;
    }
}
```

Only one alert popup can be shown at a time. Any call to show an alert while another one is being displayed will be ignored. You can check if an alert is being shown using the *IsShowingAlert* method.

```
// Check if an alert is being displayed.
bool isShowingAlert = NativeUI.IsShowingAlert();
```

Toasts

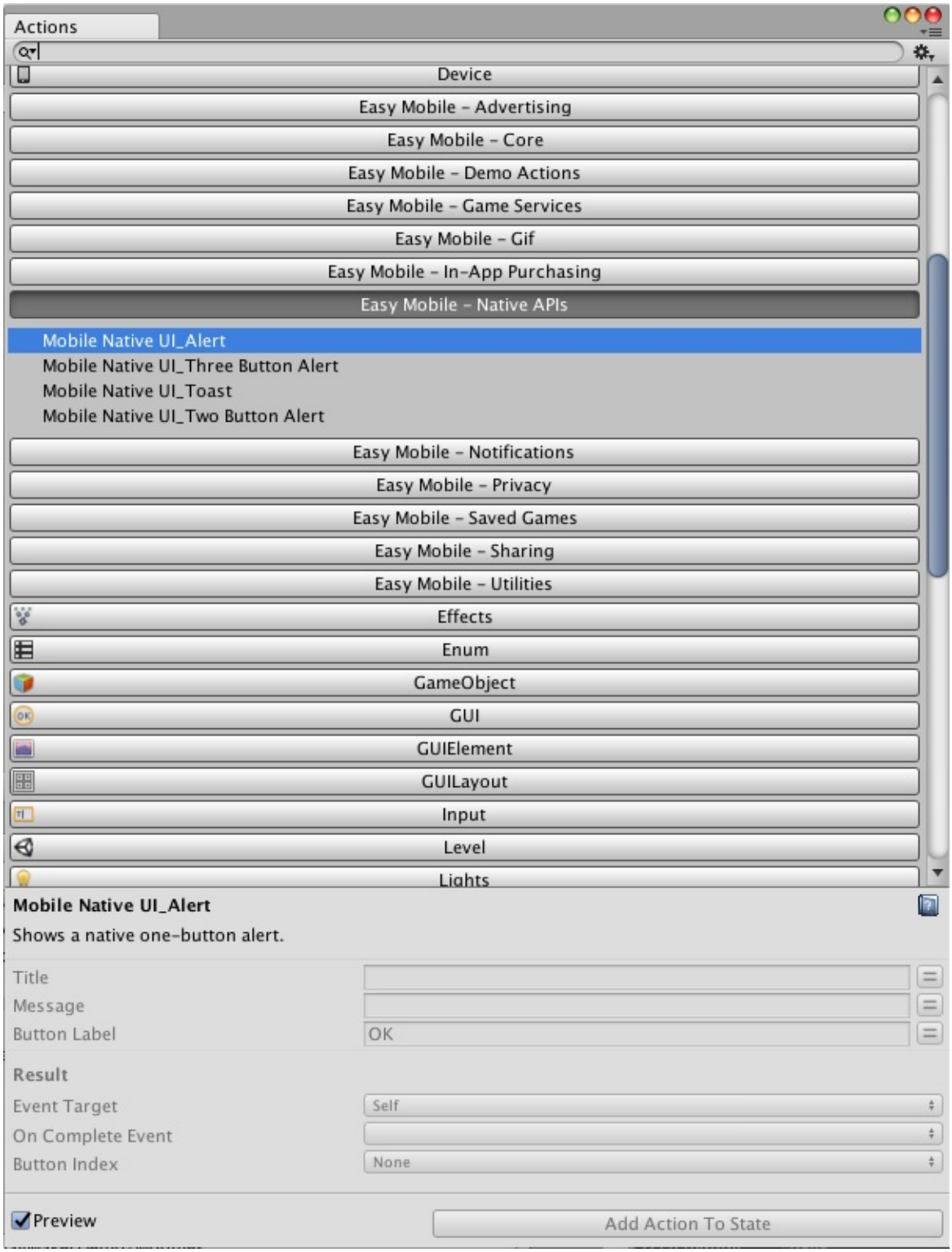
Toast is a short message displayed at the bottom of the screen and automatically disappears after a timeout. Toasts are available only on Android platform. To show a toast message:

```
// Show a sample toast message
NativeUI.ShowToast("This is a sample Android toast");
```

Native APIs: PlayMaker Actions

The PlayMaker actions of the Native APIs module are group in the category *Easy Mobile - Native APIs* in the PlayMaker's Action Browser.

> Please refer to the NativeUIDemo_PlayMaker scene in folder *Assets/EasyMobile/Demo/PlayMakerDemo/Modules* for an example on how Native UI actions can be used.



Notifications: Introduction

The Notifications module helps you quickly implement notifications feature in your app. Here're some highlights of this module:

- **Remote (push) notification**
  - The module is currently compatible with OneSignal and Firebase Cloud Messaging (FCM). Both are free and popular cross-platform push notification delivery services.
- **Local notification**
  - One-time and repeat notifications.
  - Fully-customizable notifications: sounds, icons, badge, light, vibration, lock-screen visibility, etc. (feature availability varies between iOS and Android)
  - Supports notification custom action buttons.
- **Notification Category**
  - Unifies Android notification category (channel) and iOS notification category, makes it easy to customize and organize notifications in your app.
  - Fully supports notification channels introduced and required since Android O, while maintaining backward-compatibility with older Android versions, where notification channels don't exist.
  - Fully supports notification channel groups introduced since Android O.
- **Friendly editor**
  - Makes it easy to setup remote notification service, manage categories, adding resources, etc.

> Easy Mobile's notification API works on iOS 10 or newer (more than 95% of iOS devices) and Android API 14 or newer (more than 99% of Android devices).

Understanding Notifications

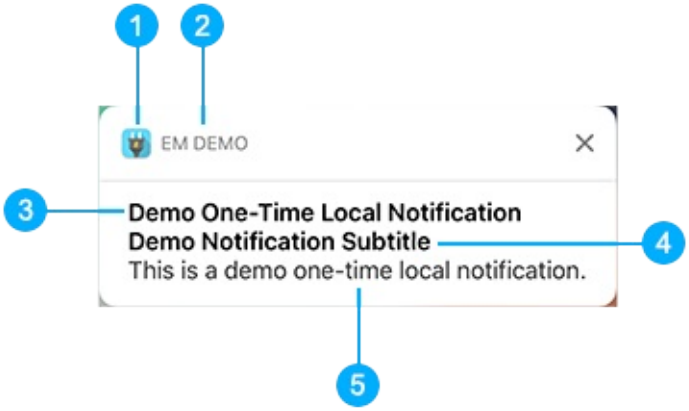
A notification is basically a message that is displayed by the system outside of your app's UI to provide the user with some timely information about your app. The user can open a notification to bring your app to foreground and take further actions if they wish.

> If your app is in foreground at the moment the notification is delivered, then the notification won't be posted (in other words, it will be silenced). Instead, its data will be sent to the app directly in form of an event.

Notifications appear to users in different locations and formats, plus their appearance varies slightly between iOS and Android, and among different versions of these platforms. Depending on the current state of your device, notifications can appear in the status bar (Android), the notification center (iOS), or the lock-screen. Below are the typical anatomies of iOS and Android notifications.

iOS Notification Anatomy

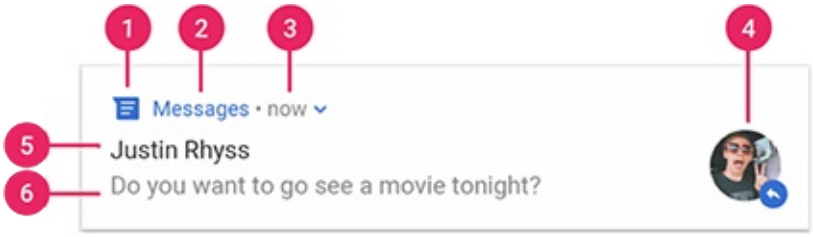
Here's a typical iOS notification with basic details.



1. *Small icon*: provided by the system
2. *App name*: provided by the system
3. T*itle*: the notification title provided by you
4. S*ubtitle*: the optional notification subtitle provided by you
5. B*ody*: the main notification message provided by you

Android Notification Anatomy

Here's a typical Android notification with basic details.



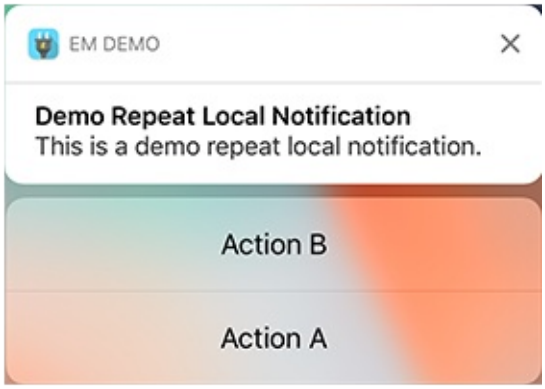
1. *Small icon*: required, provided by you (if no valid icon specified, Easy Mobile automatically uses the fallback icon, which is a bell)
2. *App name*: provided by the system
3. *Time stamp*: provided by the system
4. *Large icon*: optional, provided by you (this is usually used only for contact photos; do not use it for your app icon)
5. *Title*: the notification title provided by you
6. *Body*: the main notification message provided by you

> You can learn more about iOS notifications here and Android notifications here.

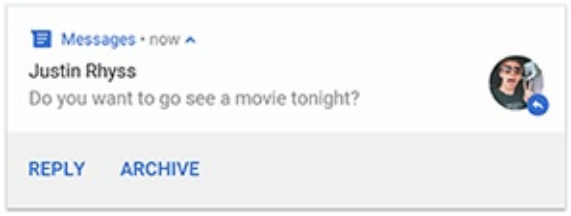
Notification Custom Actions

Beside basic content, you can optionally attach additional action buttons to the notification for specific tasks. Using unique action IDs, your app can acknowledge the action selected by the user and response accordingly, e.g. open a particular UI for the user to perform further interaction.

A notification on iOS can have up to 4 action buttons. However, the number of actions actually displayed depends on how and where the notification is displayed. For example, banners display no more than two actions. Here's an example of an iOS notification with 2 action buttons, Action A and Action B.



A notification on Android can have up to 3 action buttons. Here's a typical Android notification with 2 action buttons, REPLY and ARCHIVE.



> On Android, Easy Mobile supports notification actions on API 23 or newer.

Local Vs. Remote Notifications

Local notifications are notifications scheduled by your app locally. Your app configures the notification content, specifies a trigger condition, and passes these details to the system, which then handles the delivery of the notification when the trigger condition is met.

> **One-time vs. Repeat local notifications**
>
> A one time notification is delivered only once. A repeat notification is delivered once every time the repeat interval passed. Both notification types survive device reboots. If the delivery time is past at the moment the device has finished reboot, the notification will be delivered immediately.

Remote notifications are sent (pushed) to user devices from a remote server (be it your own server or a 3rd party server like OneSignal's) via the Apple Push Notification service (APNs) on iOS, or the Google Cloud Messaging service (GCM) on Android.

Notification Categories

Easy Mobile's cross-platform notification category unifies Android notification channel/category and iOS notification category, providing a simple and convenient way to customize and organize notifications in your app. You can use category to control multiple aspects of notifications including importance, light, vibration, sound and action buttons (configuration varies between platforms, e.g. importance and light are Android-only). All notifications posted to the same category share the same customization. Every time you schedule a notification, simply set the category it belongs to and all the settings of that category will be applied to the notification automatically. Using categories is therefore a more intuitive and efficient way to customize and organize notifications. You can have multiple categories in your app, each controls a different type of notifications. For example, you can have a category dedicated for game event notifications, and another category for user (chat) message notifications.

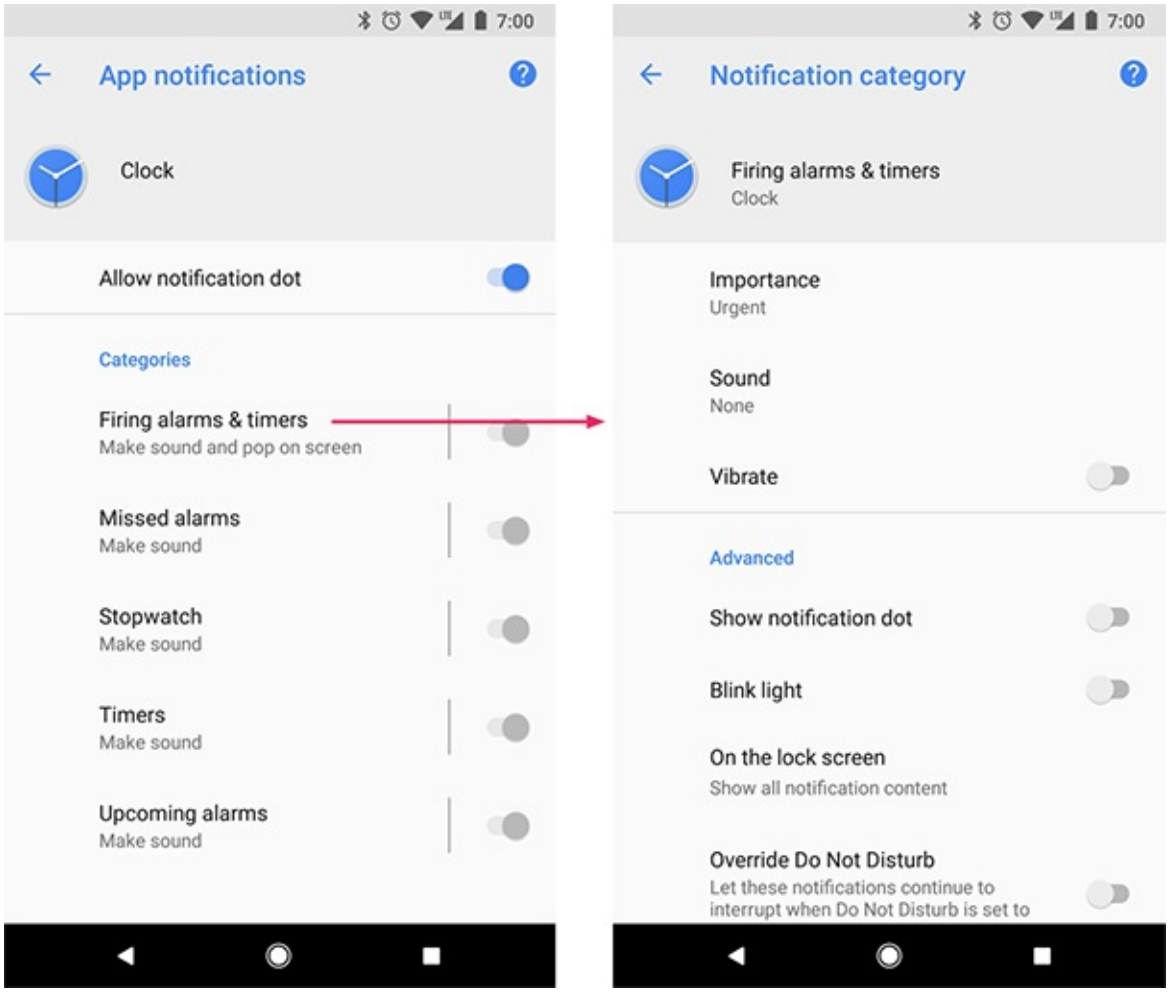
Notification Categories on iOS

On iOS, cross-platform categories automatically translate to native notification categories, taking only the information on custom actions because iOS notification categories are mostly responsible for configuring custom actions. About the other settings of the cross-platform category, those are applicable on iOS (e.g. sound) will automatically be applied when a notification of this category is scheduled.

> iOS notification categories are not visible to users. You can learn more about them here.

Notification Categories on Android

On Android 8.0 Oreo (API level 26), a new feature is introduced which is notification channels/categories. Starting in this version, all notifications must be assigned to a channel or it will not appear. Notification channels offer the ability to group notifications, and allow users to control the visual and auditory options of each channel from the Android system settings.



> The Android user interface refers to channels as "categories".

On Android 8.0 and newer, Easy Mobile's cross-platform notification categories automatically translate to native notification channels. Most settings are applicable to Android channels, except the custom actions (Android notification channels are not responsible for custom actions). When a local notification is scheduled, it will be assigned to the native notification channel, and the custom actions (if any) found in its cross-platform category are automatically added when constructing the notification.

On Android 7.1 (API level 25) and lower, there're no notification channels. When a local notification is scheduled, individual settings will be read from its cross-platform category and applied automatically when constructing the notification.

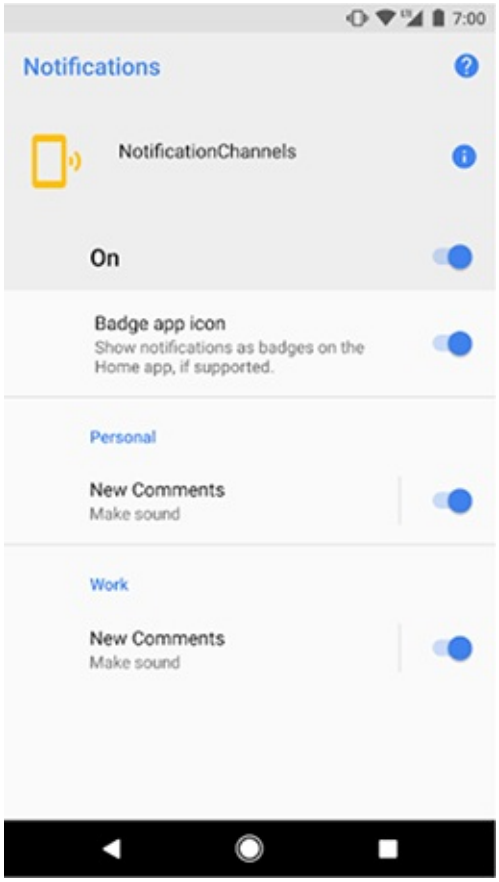
> You can learn more about Android notification channels/categories here.

Default Notification Category

For notifications to work consistently across iOS and Android platform, your app requires at least one notification category. Easy Mobile has a built-in default category. If you're scheduling a local notification and not specifying any category, then the default one will be used. This default category can be customized from the module settings, but it cannot be removed.

Notification Category Groups

You can organize your notification categories into category groups. On Android category groups directly translate to native channel groups. Below is an example of how notification channels are organized into groups in the system settings on Android.



> Notification category groups don't have any effect on iOS.

GDPR Compliance

> We recommend you to read the Privacy chapter first to gain a comprehensive understanding of the tools and resources offered by Easy Mobile to help your app get compliant with GDPR, including the consent dialog and the consent management system.

Remote notification is one of those services affected by the GDPR, because most providers (including OneSignal and Firebase) need to collect a device's unique identifier of one kind or another to deliver messages to that device. Therefore it may be advisable to get the user consent (explicit opt-in) before initializing the service, especially in EEA

region. Easy Mobile provides a native, multi-purpose dialog which you can use to collect user consent for the remote notification service as well as other relevant services in your app.

> Allowing the user to provide and manage consent for all services via a single interface (dialog) is advisable in terms of user experience, because the user may find it irritating being presented multiple dialogs asking consent for various things.

Once the push notification consent is collected, you can communicate it to the Notifications module using the *GrantDataPrivacyConsent* or *RevokeDataPrivacyConsent* methods (see Scripting/Working with Consent). The consent will then be automatically forwarded to the selected remote notification service during its initialization. Therefore it is important to collect the consent before initializing the remote notification service. Practically, this means collecting consent before initializing Easy Mobile runtime, which will automatically initialize the remote notification service (provided that the Auto Initialization feature of the Notifications module is enabled). The next sections detail how consent is applied to each remote notification service when it is used in your app.

> Currently there's only module-level consent available to the Notifications module. Because consent is only applicable to remote notification service, and only one service can be selected at a time, providing vendor-level consent would be irrelevant. Instead, the module-level consent (if any) will be used for whichever service being selected.

OneSignal

If OneSignal is selected as the remote notification provider in your app, the Notifications module will perform appropriate actions according to the specified consent before initializing OneSignal service. The table below summarizes the actions taken in each case.

Consent Status	Actions Taken
Granted	Calling *OneSignal.SetRequiresUserPrivacyConsent(true);* and *OneSignal.UserDidProvideConsent(true);* to inform OneSignal that the consent was granted and then initialize it.
Revoked	Calling *OneSignal.SetRequiresUserPrivacyConsent(true);* and *OneSignal.UserDidProvideConsent(false);* to inform OneSignal that the consent was not granted/revoked.
Unknown	Do nothing regarding consent. The initialization of OneSignal is carried out as normal (the "pre-GDPR" behavior).

Reference: https://documentation.onesignal.com/docs/unity-sdk#section--setrequiresuserprivacyconsent-

Firebase Cloud Messaging

Once the FCM SDK is imported into your project, it will automatically perform initialization in runtime, during which it generates an Instance ID, which is used as a registration token within FCM. When an Instance ID is generated the library will upload the identifier and configuration data to Firebase. Therefore if you want to get an explicit opt-in before using FCM, you must disable its auto initialization. Please follow the instructions in the Prevent auto initialization chapter in the FCM documentation for that purpose.

If the consent communicated to the Notifications module is Granted, it will automatically re-enable FCM during the initialization process by calling the following command:

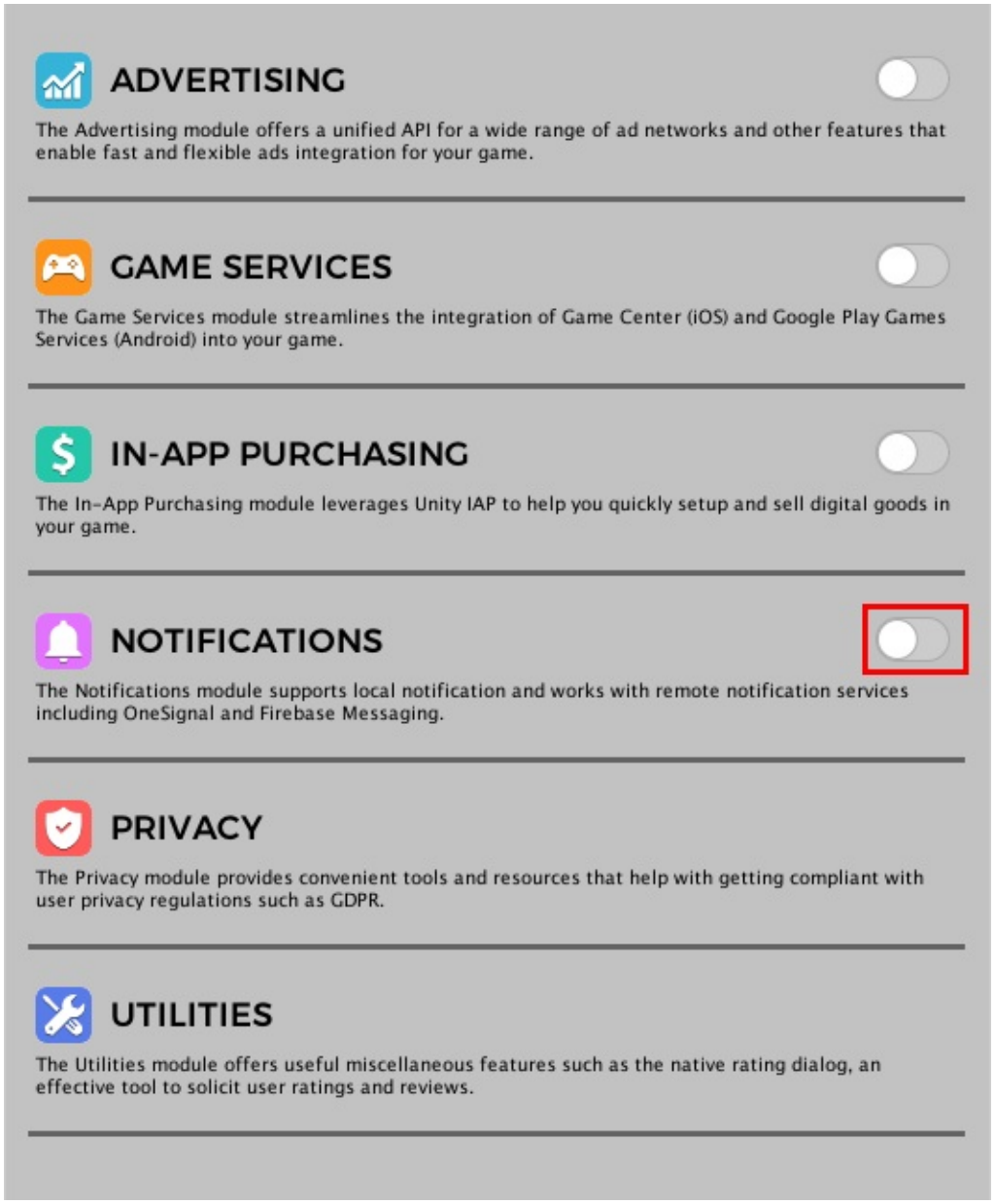
```
Firebase.Messaging.FirebaseMessaging.TokenRegistrationOnInitEnabled = true;
```

> **Summary**

> - If you want to get an explicit opt-in before using FCM, you must disable its auto initialization. If the consent provided to the Notifications module is Granted, FCM will be re-enabled automatically during initialization.
> - If you disable the FCM auto initialization and the consent provided to the Notifications module is Revoked or Unknown, the FCM service won't be re-enabled and it won't function.
> - If you don't want to get an explicit opt-in before using FCM, you can ignore all described actions. The FCM service will be initialized automatically as normal.

Notifications: Settings

To use the Notifications module you must first enable it. Go to *Window* > *Easy Mobile* > *Settings*, select the Notifications tab, then click the right-hand side toggle to enable and start configuring the module.



Auto Initialization

Auto initialization is a feature of the Notifications module that initializes the service automatically when the module starts. You can configure this feature in the **AUTO-INIT CONFIG** section.

> On iOS, a popup will appear during the first initialization following the app installation to ask for the user's permission to enable notifications for your game.

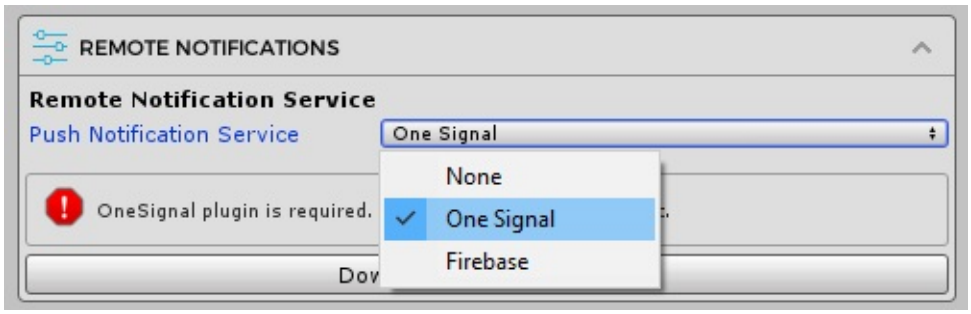
- *Auto Init*: uncheck this option to disable the auto initialization feature, you can start the initialization manually from script (see Scripting)
- *Auto Init Delay*: how long after the module start that the initialization should take place

> "Module start" refers to the moment the *Start* method of the module's associated MonoBehavior (attached to the EasyMobile prefab) runs. If you add the EasyMobile prefab instance to the first scene of your game then this moment is mostly identical to the launch time of the app.

Remote Notification Setup

Enable Remote Notifications

To enable remote/push notifications for your app, select a valid service provider from the *Push Notification Service* dropdown in **REMOTE NOTIFICATION SETUP** section. Currently OneSignal and Firebase Cloud Messaging services are supported.



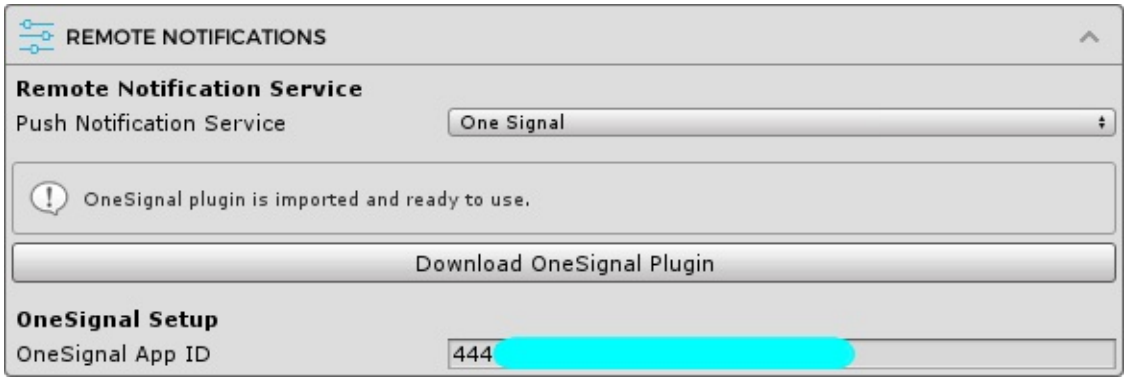
Setup OneSignal

> **Before You Begin**
>
> Before setting up OneSignal in Unity, you must first generate appropriate credentials for your targeted platforms. If you're not familiar with the process, please follow the guides listed here. You should also follow the instructions included in that document on performing necessary setup when building for each platform.

Importing OneSignal Plugin

Using OneSignal service requires the OneSignal plugin for Unity. Easy Mobile will automatically check for the availability of the plugin and prompt you to import it if needed. You can click the *Download OneSignal Plugin* button to open the download page for the plugin. Once it is imported into Unity the settings interface will be updated and ready for you to start configuring.

In fact all you need to do is enter your *OneSignal App Id*.



Customizing Notification Sounds and Icons

Here're the guides on customization OneSignal notification sounds and Android notification icons:

- Customize OneSignal Android Notification Icons
- Customize OneSignal Notification Sounds

Setup Firebase Cloud Messaging

> **Before You Begin**
>
> Before setting up Firebase Cloud Messaging in Unity, you will need to create a project in Firebase console. If you're not familiar with the process, please follow this guide.
>
> Note that for iOS you need to associate your project with an APNs certificate:
>
> - Inside your project in the Firebase console open Project Settings then select the Cloud Messaging tab
> - Select the Upload Certificate button to upload your development or production certificate, or both. For each certificate, select the .p12 file and provide password if any. Make sure the bundle ID for this certificate matches the bundle ID of your app
> - Save
>
> After setting up the project, download the *GoogleService-Info.plist* file (iOS) or *Google-Services.json* file (Android) from the console and drag it into your Unity project (you can place these files anywhere under the *Assets* folder of your project).

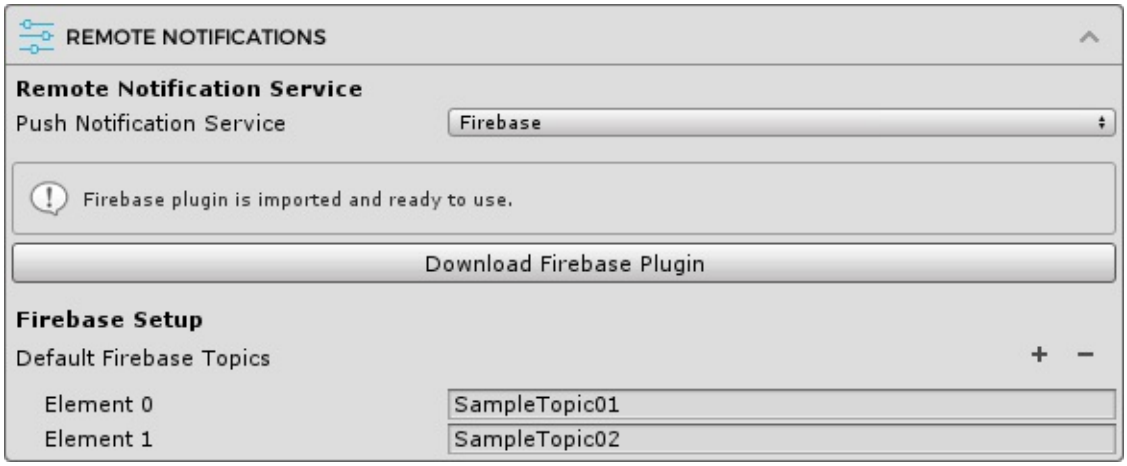
Importing Firebase Messaging plugin

Using Firebase service requires the Firebase plugin for Unity. Easy Mobile will automatically check for the availability of the plugin and prompt you to import it if needed. You can click the *Download Firebase Plugin* button to download the Firebase Unity SDK. After the downloading finishes, unzip the downloaded file and import the *FirebaseMessaging* package into your project. Once it is imported into Unity the settings interface will be updated and ready for you to start configuring.

Registering Default Notification Topics

Firebase Cloud Messaging has a concept of topic messaging, which allows you to send a message to multiple devices that have opted into a particular topic. If you want to have some default topics that every instance of your app subscribes to when it's installed, you can register them in the settings interface. At the *Firebase Topics* field click the "+" button and enter all the default topics that you want.

> You can learn more about Firebase topic messaging here.



Sending Notifications to Your App

Firebase provides 3 ways to send notifications to your app:

1. Using the notification composer in the Firebase Console - learn more here
2. Using the Admin SDK
3. Using HTTP and XMPP Protocols

Customizing Notification Sounds and Icons

Currently the Firebase notification composer doesn't allow customizing the notification sound and icon (you can only toggle sound on or off). On iOS, the app icon will be used as the notification icon. On Android, you can setup a default notification icon for Firebase by adding the following lines between the *<application>* element of the AndroidManifest.xml file at the *Assets/notifications/Plugins/Android* folder.

```
<meta-data
        android:name="com.google.firebase.messaging.default_notification_icon"
        android:resource="@drawable/firebase_notification_icon" />
```

Then you need to create notification icons at appropriate resolutions (you can use the Android Asset Studio), name them *firebase_notification_icon*, place them into a *res* folder and import the folder into your Unity project using the *Import Res Folder* button under the **ANDROID NOTIFICATION RESOURCES** section of the settings interface (see chapter **Adding Notification Resources**). You can also use another name for the icons as long as the name in the AndroidManifest.xml and the name of the icon files are same.

> If you're using other methods to send notifications to your app, you can specify the notification icons and sounds in the notification content. Just make sure that the specified icons and sounds are available in your app.

iOS Build Notes

After exporting to Xcode, perform the following steps:

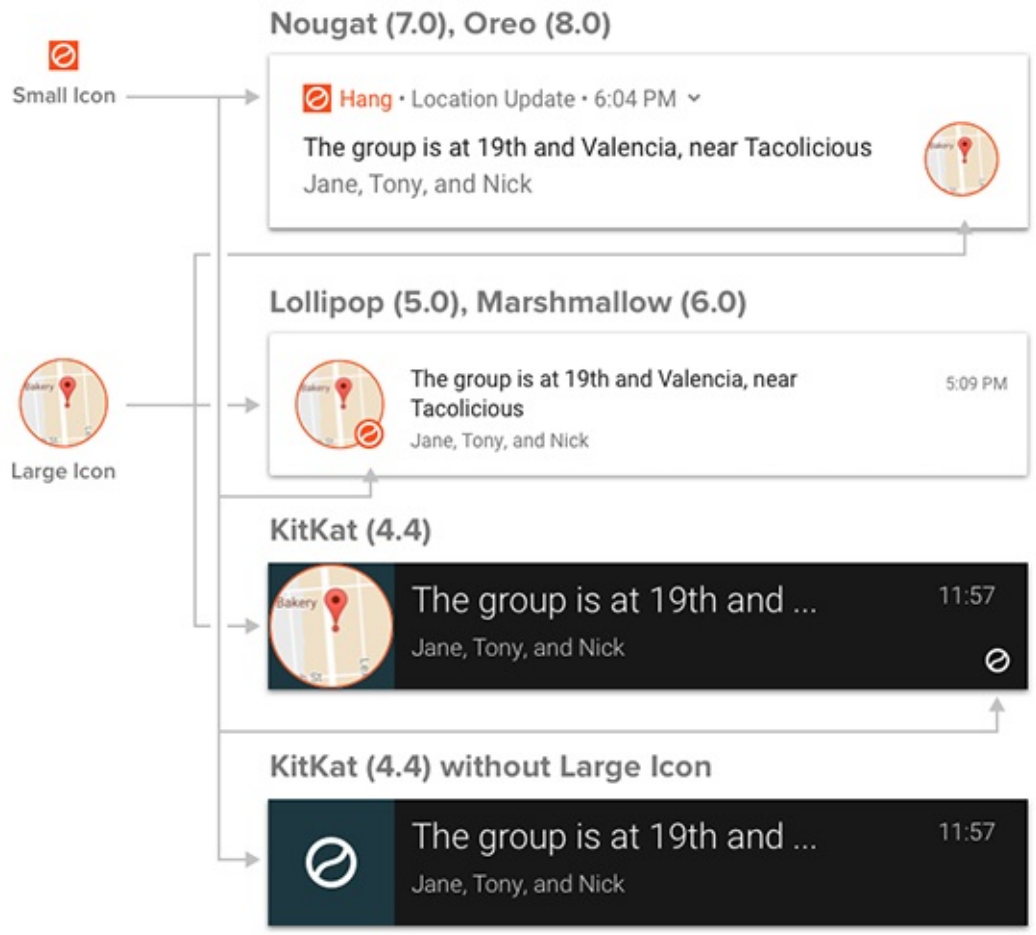
- In the *General* tab, add the UserNotifications framework into your project if it hasn't been added yet.
- In the *Capabilities* tab, turn on *Push Notifications* and *Background Modes*, then check the *Remote Notifications* box under *Background Modes*.

Adding Notification Resources

Android Notification Resources

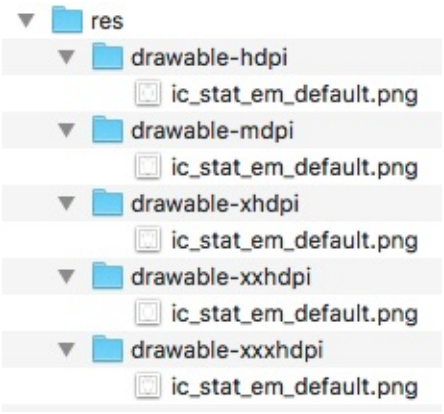
Android notification resources include notification icons and custom notification sounds.

- **Custom notification sounds:** if you don't want to use the system's default notification sound, you can provide custom sounds to be played when your notifications are delivered.
- **Small notification icons:** small icons or status bar icons are required and will be used to represent notifications from your app in the status bar. Starting with Android 5, the system forces small notification icons to be all white when your app targets Android API 21+. If you don't make a correct icon, it will most likely be displayed as the fallback icon (bell) or solid white icon in the status bar.
- **Large notification icons:** large notifications icons are optional and will show up at different positions on the notification depending on the Android version (see below screenshot). If you do not specify a large icon, the small icon will be used.

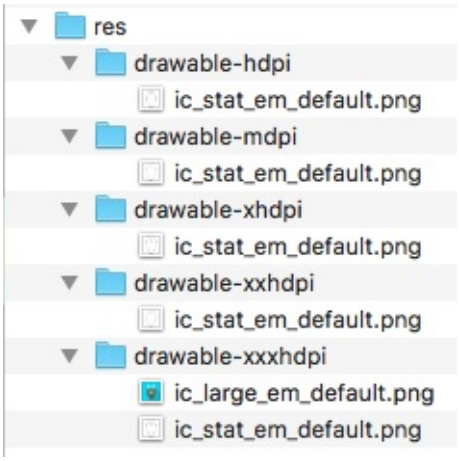


Preparing Android Notification Icons

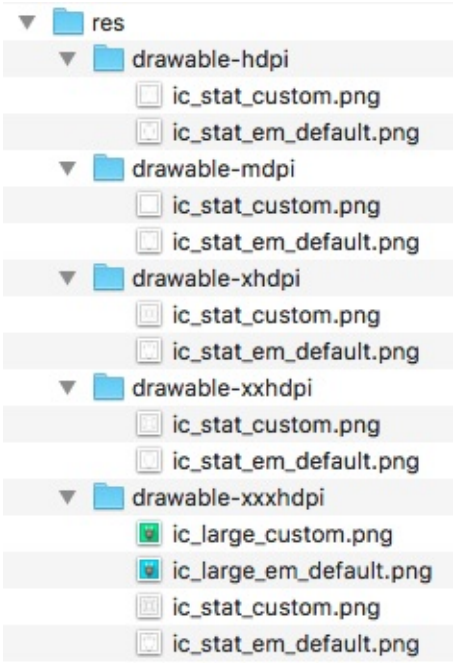
It's advisable to use the Android Asset Studio to quickly and easily generate small icons with the correct settings. Note that the default small icons must be named **ic_stat_em_default** so that Easy Mobile can recognize it. Below is an example of a *res* folder containing the default small icons generated by the Android Asset Studio. Later we will import this folder to Unity so that the icons can be used in your project.



If you want to have a default large icon, create a 256x256 icon, name it **ic_large_em_default** and place it in the *drawable-xxxhdpi* folder of the same *res* folder generated previously.

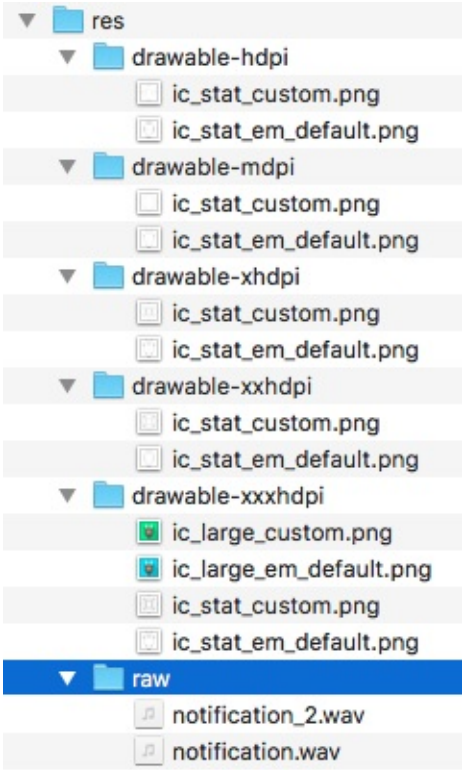


> **Non-Default Notification Icons**
>
> Beside the default icons, you can add more custom icons which can be used when scheduling different types of notifications. Just use the Android Asset Studio to generate small icons, and create large icons at the correct sizes (256x256). Then merge them into the same *res* folder that contains the default icons, making sure the icons go into their correct subfolders. When scheduling a notification, you can select these custom icons using their names.



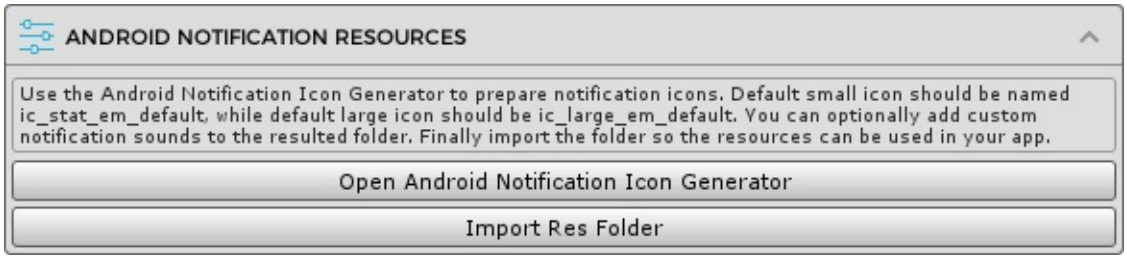
Preparing Android Notification Sounds

To add custom notification sounds, create a folder named *raw* inside the *res* folder that contains the default notification icons and place the sound files there. Later these custom sounds can be specified in your project with their names. Below is an example *res* folder that contains default icons, custom icons and custom sounds for Android notifications.



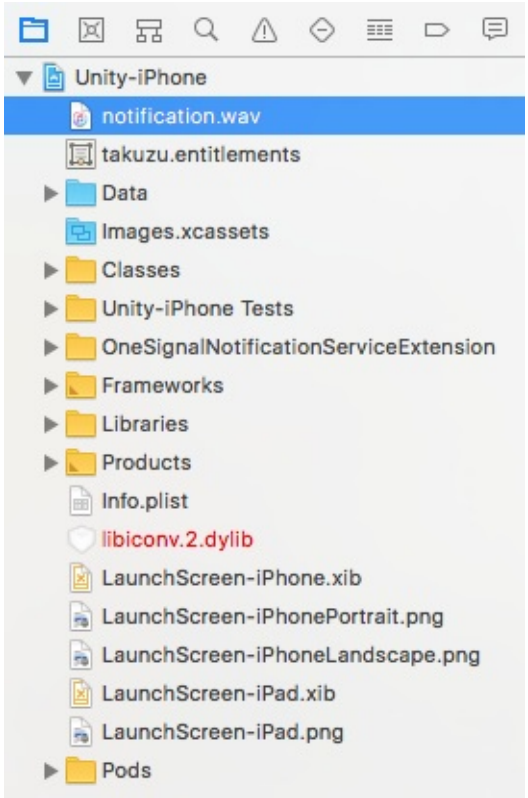
Importing Android Notification Resources

After constructing the *res* folder with all the required icons and sounds, the next step is import the folder into your Unity project so the resources can be used in your app. In the **ANDROID NOTIFICATION RESOURCES** section, click the *Import Res Folder* button, then select the generated *res* folder to import. That's it!



iOS Notification Resources

On iOS, notification icons are provided by the system, but you can still have custom notification sounds. To add a custom sound to your iOS app, simply place the sound file anywhere in your Unity project, build your project for iOS platform, and then drag the sound file to the Xcode project root (remember to select *Copy items if needed* on the Xcode import dialog).



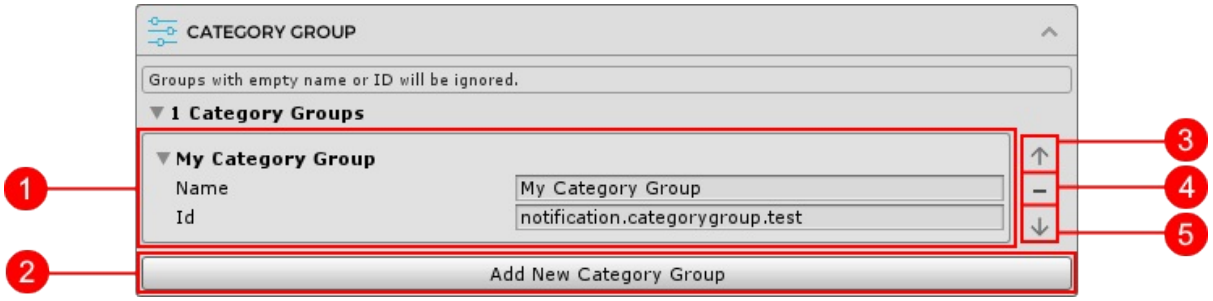
Notes on Notification Sounds

- It's recommended to have notification sounds in .wav format so that they can be used on both Android and iOS.
- Make sure sound names are consistent across iOS and Android.
- Notification sounds must be less than 30 seconds to make sure they can be played on both Android and iOS.

Category Management

Notification Category Groups

In the **CATEGORY GROUPS** section you can add, edit or remove groups for the notification categories in your app.



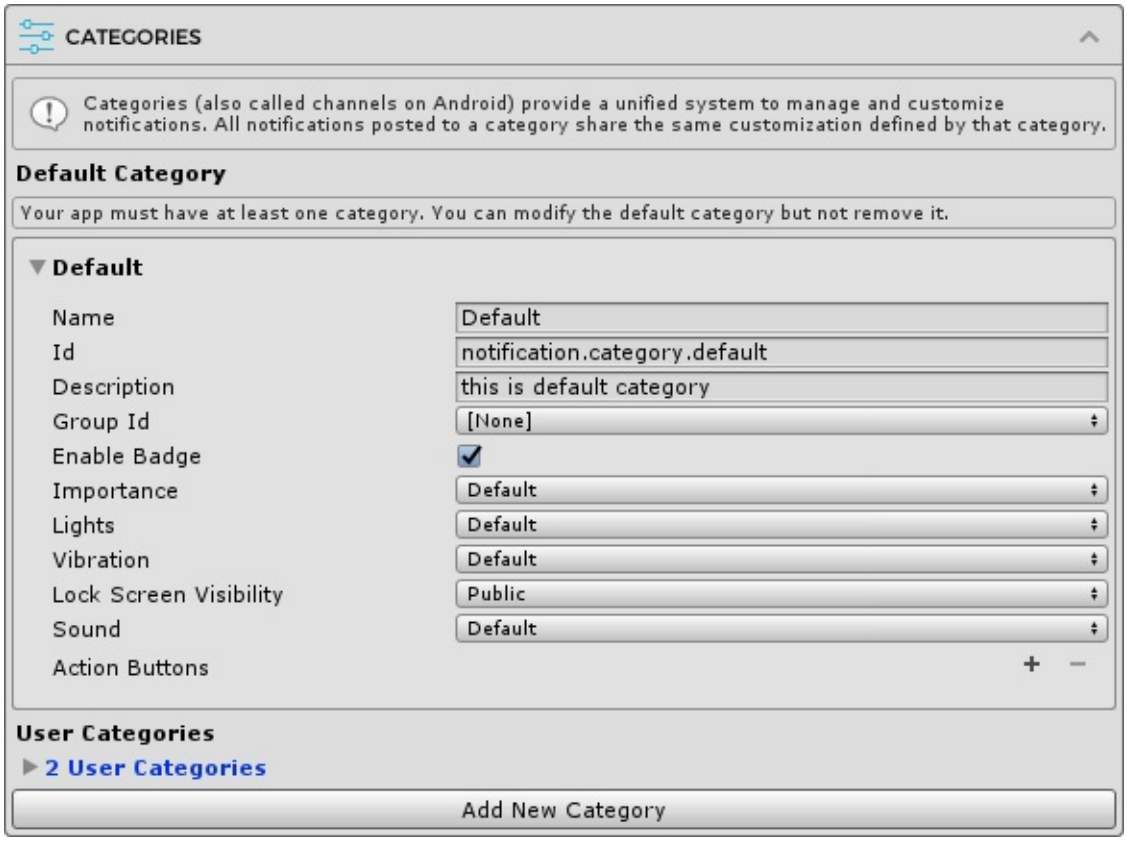
1. *Group content*: where you fill in group information
2. *Add button*: use this button to add a new group
3. *Move up button*: move the group up, for arrangement purpose
4. *Delete button*: use this button to remove the current group
5. *Move down button*: move the group down, for arrangement purpose

A category group content includes following fields:

- *Name*: the group name, must not be empty
- *Id*: the group ID, must not be empty; a category specifies the group it belongs to using this ID

Notification Categories

You can manage the notification categories in your app within the **CATEGORIES** section.



A category content includes following fields:

- *Name*: category name, only visible on Android devices, this is required

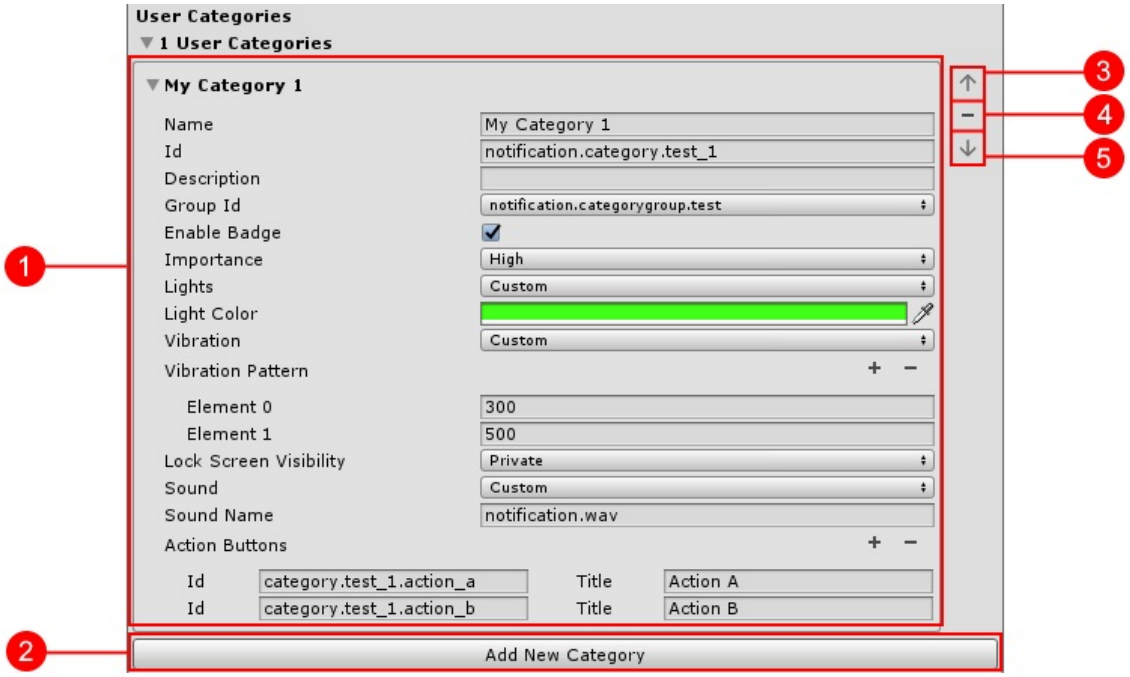
- *Id*: category ID, this is required; a notification specifies the category it belongs to using this ID
- *Description*: optional category description, only visible on Android devices
- *Group Id*: the identifier of the group this category belongs to
- *Enable Badge*: [Android only] whether the notifications of this category can appear as badges in a Launcher application
- *Importance*: [Android only] how much interruptive - visually and audibly - the notifications of this category should be
- *Lights*: [Android only] determines how the notification light should be displayed, on devices that support this feature
- *Light Color*: [Android only] the color of the notification light, only configurable when *Lights* is set to *Custom*
- *Vibration*: [Android only] determines how the device should vibrate when a notification arrives
- *Vibration Pattern*: [Android] the vibration pattern of the device, only configurable when *Vibration* is set to *Custom*
- *Lock-screen Visibility*: [Android only] determines how notifications should be displayed on lock-screen
- *Sound*: determine whether the default sound, or a custom sound, or no sound at all should be played when a notification arrives
- *Sound Name*: the filename (with extension) of the custom notification sound, only configurable if Sound is set to Custom
- *Action buttons*: the custom action buttons to be attached to the notifications of this category, each action button requires the following fields:
  - *Id*: action ID, used to distinguish between actions
  - *Title*: action title, used to display the action button on the notification

Default Category

Your app must have at least one notification category, and Easy Mobile provides a built-in default category. You can customize it in the **Default Category** sub-section of the **CATEGORIES** section. You can't remove the default category.

User Categories

Beside the default category, you can add as many more categories as you wish. We call these user categories, and they can be managed in the **User Categories** sub-section of the **CATEGORIES** section.

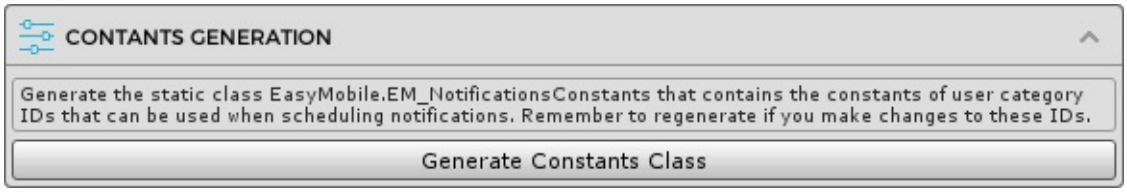


1. *Category content*: where you fill in category information
2. *Add button*: use this button to add a new user category
3. *Move up button*: move the category up, for arrangement purpose
4. *Delete button*: use this button to remove the current user category
5. *Move down button*: move the category down, for arrangement purpose

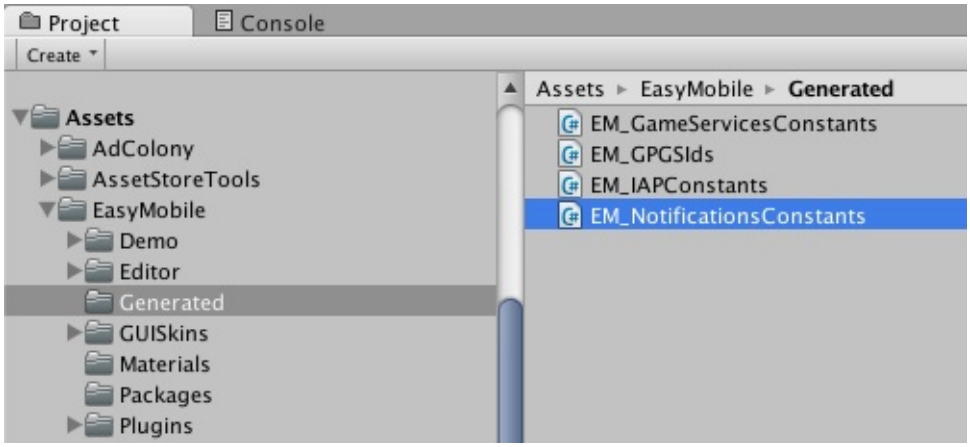
Constants Generation

Constants generation is a feature of the Notifications module. It reads the IDs of all user categories added and generates a static class named EM_NotificationsConstants that contains the constants of these IDs. Later, you can use these constants when scheduling a local notification in script instead of typing the IDs directly, thus help prevent runtime errors due to typos and the likes.

To generate the constants class (you should do this after adding all required user categories), click the *Generate Constants Class* button within the **CONSTANTS CLASS GENERATION** section.



When the process completes, a file named EM_NotificationsConstants will be created at *Assets/notifications/EasyMobile/Generated*.



Notifications: Scripting

This section provides a guide to work with the Notifications module scripting API.

> You can access the Notifications module API via the Notifications class under the EasyMobile namespace.

Working with Consent

The following snippet shows how you grant, revoke and read the module-level consent of the Notifications module.

```
// Grants the module-level consent for the Notifications module.
Notifications.GrantDataPrivacyConsent();

// Revokes the module-level consent of the Notifications module.
Notifications.RevokeDataPrivacyConsent();

// Reads the current module-level consent of the Notifications module.
ConsentStatus moduleConsent = Notifications.DataPrivacyConsent;
```

Initialization

Before notifications can be used, the module needs to be initialized. This initialization should only be done once when the app is loaded, and before any other calls to the API are made. If you have enabled the Auto initialization feature (see **Module Configuration** section), you don't need to start the initialization from script. Otherwise, if you choose to disable that feature, you can initialize the service using the *Init* method.

```
// Initialize push notification service
Notifications.Init();
```

Note that the initialization should be done early and only once, e.g. you can put it in the *Start* method of a MonoBehaviour, preferably a singleton so that it won't run again when the scene reloads.

```
// Initialization in the Start method of a MonoBehaviour script
void Start()
{
    // Initialize push notification service
    Notifications.Init();

    // Do other stuff...
}
```

You can check if the module has been initialized at any point using the *IsInitialized* method.

```
// Check if initialization has completed.
bool isInitialized = Notifications.IsInitialized();
```

Working with Local Notifications

Constructing a Notification Content

Before scheduling a notification, you must first prepare its content. To do this simply create a new *NotificationContent* object and fill it with appropriate information.

```
// Construct the content of a new notification for scheduling.
NotificationContent PrepareNotificationContent()
{
    NotificationContent content = new NotificationContent();

    // Provide the notification title.
    content.title = "Demo Notification";

    // You can optionally provide the notification subtitle, which is visible on iOS only.
    content.subtitle = "Demo Subtitle";

    // Provide the notification message.
    content.body = "This is a demo notification.";

    // You can optionally attach custom user information to the notification
    // in form of a key-value dictionary.
    content.userInfo = new Dictionary<string, object>();
    content.userInfo.Add("string", "OK");
    content.userInfo.Add("number", 3);
    content.userInfo.Add("bool", true);

    // You can optionally assign this notification to a category using the category ID.
    // If you don't specify any category, the default one will be used.
    // Note that it's recommended to use the category ID constants from the EM_NotificationsConstants class
    // if it has been generated before. In this example, UserCategory_notification_category_test is the
    // generated constant of the category ID "notification.category.test".
    content.categoryId = EM_NotificationsConstants.UserCategory_notification_category_test;

    // If you want to use default small icon and large icon (on Android),
    // don't set the smallIcon and largeIcon fields of the content.
    // If you want to use custom icons instead, simply specify their names here (without file extensions).
    content.smallIcon = "YOUR_CUSTOM_SMALL_ICON";
    content.largeIcon = "YOUR_CUSTOM_LARGE_ICON";

    return content;
}
```

Scheduling Local Notifications

To schedule a local notification, prepare the notification content and feed it to the *ScheduleLocalNotification* method.

One-Time Local Notifications

You can schedule a notification to be delivered at a specific time (in the future). If the specified time is in the past, the notification will be delivered immediately.

```
using System;    // to use DateTime



// Schedule a notification to be delivered by 08:08AM, 08 August 2018.
void ScheduleLocalNotification()
{
    // Prepare the notification content (see the above section).
    NotificationContent content = PrepareNotificationContent();

    // Set the delivery time.
    DateTime triggerDate = new DateTime(2018, 08, 08, 08, 08, 08);

    // Schedule the notification.
    Notifications.ScheduleLocalNotification(triggerDate, content);
}
```

Instead of a trigger date, you can also schedule a one-time notification to be delivered after some delay time.

```
using System;
```

```
// Schedule a notification to be delivered after 08 hours, 08 minutes and 08 seconds.
void ScheduleLocalNotification()
{
    // Prepare the notification content (see the above section).
    NotificationContent content = PrepareNotificationContent();

    // Set the delay time as a TimeSpan.
    TimeSpan delay = new TimeSpan(08, 08, 08);

    // Schedule the notification.
    Notifications.ScheduleLocalNotification(delay, content);
}
```

Repeat Local Notifications

If you want the notification to repeat automatically, schedule it to be delivered after some delay time, and then specify a fixed repeat interval. Repeat interval can be one of the following values:

- *None*: no repeat
- *EveryMinute*: notification repeats once every minute
- *EveryHour*: notification repeats once every hour
- *EveryDay*: notification repeats once every day
- *EveryWeek*: notification repeats once every week

```
using System;



// Schedule a notification to be delivered after 08 hours, 08 minutes and 08 seconds,
// then repeat once every day.
void ScheduleRepeatLocalNotification()
{
    // Prepare the notification content (see the above section).
    NotificationContent content = PrepareNotificationContent();

    // Set the delay time as a TimeSpan.
    TimeSpan delay = new TimeSpan(08, 08, 08);

    // Schedule the notification.
    Notifications.ScheduleLocalNotification(delay, content, NotificationRepeat.EveryDay);
}
```

Managing Pending Local Notifications

Get Pending Local Notifications

To get all pending (scheduled but not yet delivered) local notifications, use the *GetPendingLocalNotifications* method. The pending notifications will be returned as an array of *NotificationRequest* objects via a callback. Each pending notification request is identified by its request ID.

```
// Gets all pending local notification requests.
void GetPendingLocalNotifications()
{
    Notifications.GetPendingLocalNotifications(GetPendingLocalNotificationsCallback);
}

// Callback.
void GetPendingLocalNotificationsCallback(NotificationRequest[] pendingRequests)
{
    foreach (var request in pendingRequests)
    {
        NotificationContent content = request.content;        // notification content
```

```
        Debug.Log("Notification request ID: " + request.id);    // notification request ID
        Debug.Log("Notification title: " + content.title);
        Debug.Log("Notification body: " + content.body);
    }
}
```

Cancel a Pending Local Notification

To cancel a particular pending local notification, call the *CancelPendingLocalNotification* method with the request ID of the notification to be canceled. Canceled notifications will no longer be delivered.

```
// Cancels a pending local notification with known request ID.
Notifications.CancelPendingLocalNotification("REQUEST_ID_TO_CANCEL");
```

Cancel All Pending Local Notifications

To cancel all pending local notifications, simply call the *CancelAllPendingLocalNotifications* method.

```
// Cancels all pending local notifications.
Notifications.CancelAllPendingLocalNotifications();
```

Working with Remote Notifications

Remote notifications are sent from a remote server and are typically scheduled from a website, e.g. OneSignal's dashboard. Within your app, you only need to write code to handle when a remote notification is delivered and opened.

Handling Opened Notifications

Whenever a local or remote notification is opened - either by the user tapping on it (default open action) or selecting an action button - your app will be brought to foreground and then a *LocalNotificationOpened _or a* RemoteNotificationOpened *event will be raised. You can subscribe to these events and take appropriate actions according to your app design. It's recommended to subscribe to the events as soon as your app is loaded, e.g. in the _OnEnable* method of a MonoBehaviour script in your first scene. Otherwise you risk missing them.

> If your app is in foreground when a notification is delivered - be it local or remote - then the notification is not posted (not displayed) and the *LocalNotificationOpened* or *RemoteNotificationOpened* event will be raised immediately as if the notification was opened with the default action.

```
// Subscribes to notification events.
void OnEnable()
{
    Notifications.LocalNotificationOpened += OnLocalNotificationOpened;
    Notifications.RemoteNotificationOpened += OnRemoteNotificationOpened;
}

// Unsubscribes notification events.
void OnDisable()
{
    Notifications.LocalNotificationOpened -= OnLocalNotificationOpened;
    Notifications.RemoteNotificationOpened -= OnRemoteNotificationOpened;
}

// This handler will be called when a local notification is opened.
void OnLocalNotificationOpened(LocalNotification delivered)
{
```

```
    // The actionId will be empty if the notification was opened with the default action.
    // Otherwise it contains the ID of the selected action button.
    if (!string.IsNullOrEmpty(delivered.actionId))
    {
        Debug.Log("Action ID: " + delivered.actionId);
    }

    // Whether the notification is delivered when the app is in foreground.
    Debug.Log("Is app in foreground: " + delivered.isAppInForeground.ToString());

    // Gets the notification content.
    NotificationContent content = delivered.content;

    // Take further actions if needed...
}

// This handler will be called when a remote notification is opened.
void OnRemoteNotificationOpened(RemoteNotification delivered)
{
    // The actionId will be empty if the notification was opened with the default action.
    // Otherwise it contains the ID of the selected action button.
    if (!string.IsNullOrEmpty(delivered.actionId))
    {
        Debug.Log("Action ID: " + delivered.actionId);
    }

    // Whether the notification is delivered when the app is in foreground.
    Debug.Log("Is app in foreground: " + delivered.isAppInForeground.ToString());

    // Gets the notification content.
    NotificationContent content = delivered.content;

    // If OneSignal service is in use you can access the original OneSignal payload like below.
    // If OneSignal is not in use this will be null.
    OneSignalNotificationPayload osPayload = delivered.oneSignalPayload;

    // If Firebase Messaging service is in use you can access the original Firebase
    // payload like below. If Firebase is not in use this will be null.
    FirebaseMessage fcmPayload = delivered.firebasePayload;

    // Take further actions if needed...
}
```

> **Handling Firebase Notifications on Android**
>
> On Android, if a Firebase notification arrives when your app is not running or is in background, only the "data payload" will be sent to the app, while the "notification component" won't because it is intended to be displayed to the user only. Effectively, the NotificationContent associated with the RemoteNotification object returned by the RemoteNotificationOpened event will contain empty fields (title, subtile, body, etc.) with the exception of the 'userInfo' field which represents the data payload. If the app is in foreground then both the data payload and the notification component will be forwarded to the app.
>
> You can learn more about this here and here.

Removing Delivered Notifications

Normally delivered notifications are cleared automatically when they're opened. You can manually clear all previously shown notifications of your app from the notification center or status bar with the *ClearAllDeliveredNotifications* method.

```
// Clear all delivered notifications (local and remote).
Notifications.ClearAllDeliveredNotifications();
```

[iOS] Setting Application Icon Badge Number

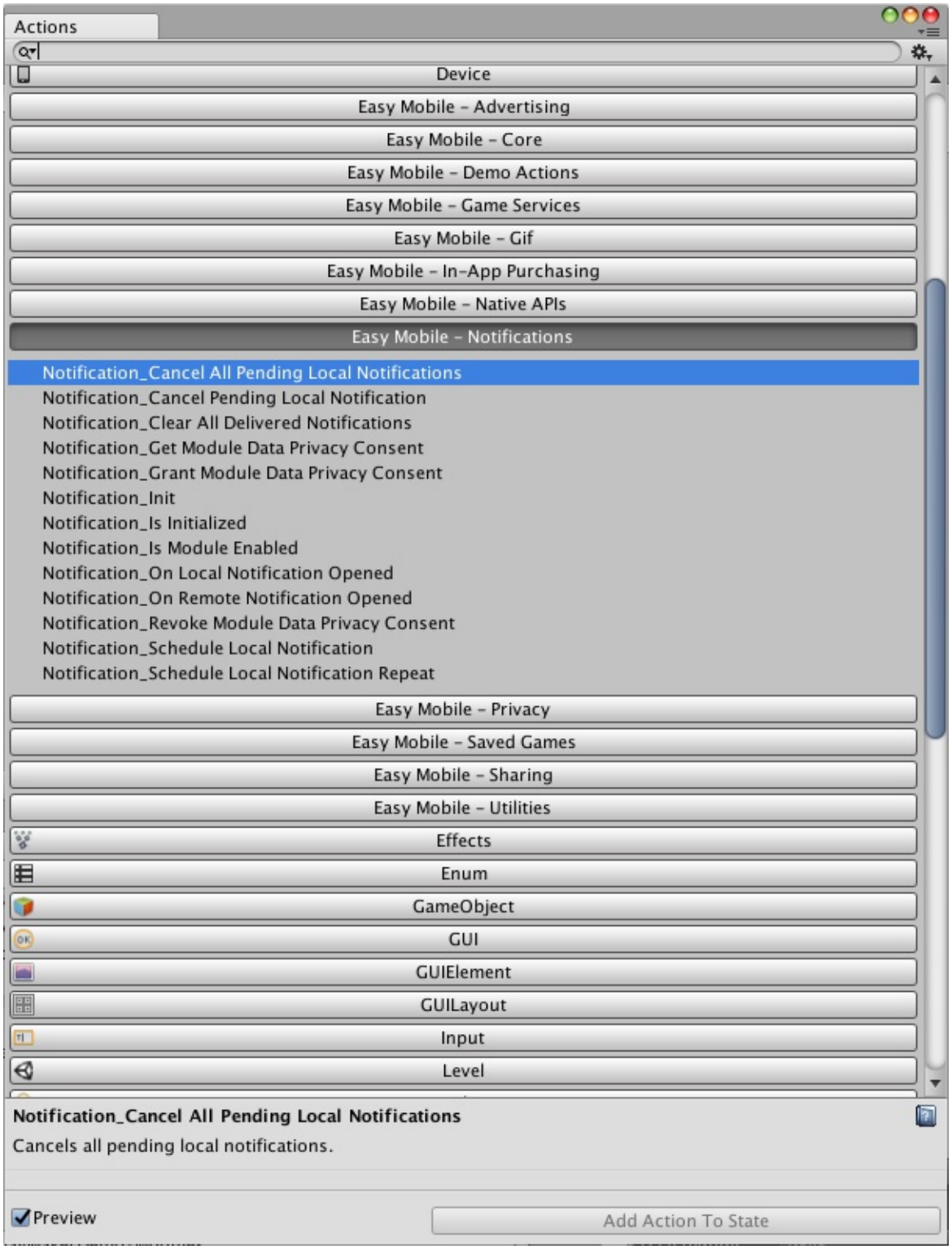
iOS allows us to get and set the application icon badge number directly. Easy Mobile provides the following methods so you can manage the badge number when working with notifications on iOS. Note that they don't have any effect on Android.

```
// Get iOS application icon badge number.
int badgeNumber = Notifications.GetAppIconBadgeNumber();

// Set iOS application icon badge number.
Notifications.SetAppIconBadgeNumber(badgeNumber + 1);    // increase the badge number by 1
```

Notifications: PlayMaker Actions

The PlayMaker actions of the Notifications module are group in the category *Easy Mobile - Notifications* in the PlayMaker's Action Browser.

> Please refer to the NotificationsDemo_PlayMaker scene in folder *Assets/EasyMobile/Demo/PlayMakerDemo/Modules* for an example on how these actions can be used.



Privacy: Introduction

The Privacy module provides you convenient tools and resources to streamline the process of getting compliant with user privacy directive and regulations such as the General Data Privacy Regulation (GDPR). Here're some highlights of this modules:

- **Comprehensive, flexible consent management system**
  - Multi-level, prioritized consent system for flexible management of various consent-requirable services in your app
  - Consent is automatically stored to be persistent across app sessions
  - Consent is automatically applied/communicated to relevant services in runtime
- **Multi-purpose built-in native consent dialog**
  - Easy to use dialog that can be used as the common interface for collecting user consent for all relevant services (to avoid annoying the user with multiple popups asking consent for various things)
  - The dialog content and layout can be flexibly adapted to your consent management needs
  - The dialog can be created from script or by using the built-in graphical composer
  - Native look makes sure the dialog can visually fit into apps of any graphic style
- **Location detection tool**
  - Built-in API to check whether the current device is in the European Economic Area (EEA) region, which is regulated by the GDPR

Consent Management System

Consent States

A consent can take one of 3 values: Unknown, Granted or Revoke.

State	Description
Unknown	The default state of consent, which means the user neither accepts nor denies the requested use of their data.
Granted	The user allows permission to the requested use of their data
Revoked	The user denies (or revokes a previously granted) permission to the requested use of their data

> A consent is considered "defined" if it has a state other than Unknown.

Consent Levels and Priorities

The primary design target of the consent system is flexibility, that it allows setting a global consent or granular consent for individual modules or services. Different consent-levels have different priorities such that the higher prioritized consent overrides the lower one. In other words, more specific consent is prioritized over less-specific one. The table below summarizes the types of consent provided by Easy Mobile.

Type	Level	Priority	Description
Vendor/Provider Consent	Lowest (most specific)	Highest	Consent given to a specific vendor of a certain service, e.g. AdMob's consent
Module Consent	Medium	Medium	Consent given to a certain module, e.g. Advertising module's consent

Global Consent	Highest (least specific)	Lowest	The common consent given to the whole app

Whenever a service seeks for an applicable consent, it searches from the lowest level to the highest level and get the defined consent with highest priority. The following pseudo-code illustrates the process:

```
if there’s a defined vendor-consent
    Use vendor-consent
else if there’s a defined module-consent
    Use module-consent
else if there’s a define global-consent
    Use global-consent
else
    Fallback to service's default-behaviour (the “pre-GDPR” behaviour)
```

ConsentManager Class

Each consent type is managed by a corresponding consent manager. Specifically,

- The vendor consent is managed by the client of the corresponding vendor, e.g. AdMob's consent is managed by the *Advertising.AdMobClient* object
- The module consent is managed by the corresponding consent manager of that module, e.g. the Advertising module's consent is managed by the *AdvertisingConsentManager*.
- The global consent is managed by the *GlobalConsentManager*.

Each of these manager extends the *ConsentManager* class, which is an abstract class that implements the *IConsentRequirable* interface. It provides an API for following tasks:

- Querying the current state of the managed consent
- Granting or revoking consent
- Observing consent changes using event

> The ConsentManager stores the specified consent state in PlayerPrefs so that it persists across app launches.
>
> You can create classes extending the ConsentManager class to manage consent for other services in your app that are not governed by Easy Mobile.

Consent Communication

Specified consent (Granted or Revoked) will be automatically communicated (applied) to corresponding services during their initialization, which most of the time is done automatically during the initialization of Easy Mobile (see Using Easy Mobile > Initializing), unless you turned off the auto initialization feature of these services in the Settings interface. If the consent is changed after the corresponding service has been initialized, that change will be applied in the next initialization. In such case you should inform the user accordingly.

> Details on how consent is applied for each individual service can be found in the chapter on GDPR of the corresponding module. Currently, two modules being affected are the Advertising and Notifications module.

Consent Dialog

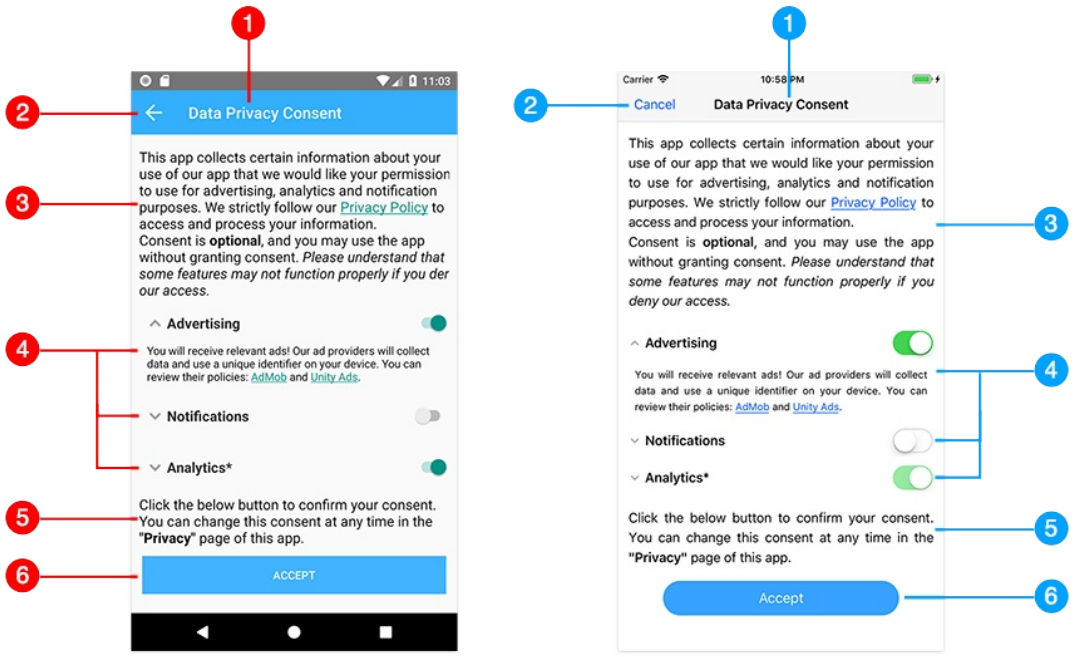
Collecting user consent may sound trivial, but if not done right could negatively affect the user experience. A typical app could incorporate multiple consent-requirable services such as ads, analytics and remote notifications. Each service may provide its own interface to collect user consent, e.g. AdMob provides the Consent SDK and the MoPub

SDK offers a built-in dialog for the same purpose. However, it may not be ideal to use these vendor-specific interfaces because it means the user would be presented with multiple dialogs asking consent for different things, which would not be a great experience, to say the least.

To solve this problem, Easy Mobile provides a multi-purpose consent dialog that can serve as a common interface for collecting user consent for all relevant services in your app. The dialog is built from native UI elements to give it a mobile-native look that easily fits into your app regardless of its graphic style.

Consent Dialog Anatomy

The following image depicts how the consent dialog looks on Android and iOS (left to right).



A consent dialog consists of following parts:

- *Title bar:* on top of the dialog is a title bar containing:
  - *Title (1):* the title for your dialog
  - *Dismiss button (2):* optional button that allows the user to cancel the dialog without updating the consent, when this button is clicked the dialog closes and raises its *Dismissed* event.
- *Main content:* the body of the dialog is comprised of the following element:
  - *Texts (3)(5):* you can use common HTML tags such as <b>, <i> to style the text, as well as using the <a> tag to include hyperlinks, which is useful for providing links to privacy policies.
  - *Toggles (4):* you can optionally include toggles in your dialog asking the user to provide granular consent for different services. Toggles can be turned off by default, making them a good means for users to provide explicit consent (they have to turn them on explicitly). Each toggle consists of a title, a switch and a description text, which is collapsible. Similar to the body text, you can use common HTML tags such as <b>, <i> and <a> in this description text. Also, you can, for example, make a toggle on by default and make it unclickable for a service that is vital for the operation of your app to indicate the user that consent for such service is required for continued use of the app.
  - *Buttons (6):* whenever a button in the dialog is clicked, the dialog closes and raises its *Completed* event with the ID of the clicked button and the current values of the toggles if any.

> - You can construct consent dialogs directly in script or by using the built-in consent dialog composer (see Settings).
> - You can flexibly interleave text, toggles and buttons in the body of your consent dialog to achieve the desired layout.

> - You can have zero or more toggles, but the dialog should always have at least one button otherwise the *Completed* event will never fire and the dialog results (clicked button ID, toggle values) won't be returned.

Consent Dialog Localization

One of the requirement of the GDPR is the the request for consent must be intelligible, so it may be desirable to localize the consent dialog into multiple languages. Easy Mobile's consent dialog was designed with that in mind. All elements of the consent dialog can be accessed and altered at runtime, making it easy to localize them. A typical approach would be using known string-patterns as placeholders for the texts in the dialog (e.g title, body texts, toggle description, button label) and replace them with appropriate localized texts in runtime before showing the dialog.

> You may want to use a dedicated localization asset, which can be easily found on the Unity Asset Store, for the purpose of managing and selecting correct translation for the placeholder texts.

EEA Region Checking

Since the GDPR applies specifically to the EEA region, it may be desirable to have different behaviors for your app between EEA and non-EEA regions, especially on privacy-sensitive tasks. To serve that purpose, Easy Mobile provides a built-in validator to detect whether the current device is in EEA region or not. The tool employs several different methods for the test which are summarized in the table below.

Method	Description
Google Service	Validating using the service provided by Google at https://adservice.google.com/getconfig/pubvendors, requires an internet conection
Telephony	Validating using the country code obtained from the device's mobile carrier information (SIM card information), no connection required but won't work without a SIM card
Timezone	Validating using the device's timezone setting. No connection required; note that this setting can be changed by the user
Locale	Validating using the device's locale setting. No connection required; note that this setting can be changed by the user

The EEA region validator returns one of the following results: *InEEA*, *NotInEEA* or *Unknown*. It allows you to use all available validating methods or a subset of them in a predefined order of priorities. If a higher prioritized method fails, the validator will use the next method until an explicit result (either an *InEEA* or *NotInEEA* value) is determined. In the rare case that all methods fail, the result will be returned as *Unknown*. For example scripts on using the EEA validator see the Scripting chapter.

> The codes of those countries currently included in the EEA region are stored in the *EEACountries* enum.

```
public enum EEACountries
{
    None = 0,
    AT, BE, BG, HR, CY, CZ, DK, EE, FI, FR, DE, GR, HU, IE, IT, LV, LT, LU, MT, NL, PL, PT, RO, SK, SI, ES
, SE, GB, // 28 member states.
    GF, PF, TF, // French territories French Guiana, Polynesia, Southern Territories.
    EL, UK,  // Alternative EU names for GR and GB.
    IS, LI, NO, // Not EU but in EAA.
    CH, // Not in EU or EAA but in single market.
    AL, BA, MK, XK, ME, RS, TR // Candidate countries.
}
```

A Proposed Workflow for Working with Consent

It's up to you to decide how your app should react to the GDPR, including whether user consent should be collected for those services in the app that may be subject to the regulation. The following proposed workflow can be optionally employed in the case you decide to collect user consent and configure relevant service with that consent.

> SgLib Games offer tools and information as a resource to assist you in getting compliant with the GDPR, but we don't offer legal advice. We recommend you contact your legal counsel to find out how GDPR affects you.

When building a consent aware app using Easy Mobile, following considerations can be made:

- Since the consent is applied during service initialization, it's advisable to collect user consent and forward it to relevant modules or services *before* initializing the Easy Mobile runtime.
- A consent dialog should be presented the first time the app launches to collect user consent for all relevant services; this dialog *should not be dismissible*, as we want the user to specify explicit consent once before the normal operation runs.
- The consent dialog should be accessible again (e.g. via a menu button) so that the user can change their consent; this dialog should reflect the current consent states for each individual services and *should be dismissible*, so the user can leave it without updating their consent in case the dialog was opened unintentionally. If the user does change their consent, they should be informed that the changes will take place during the next app launch (the next initialization).
- It may be advisable to only collect and apply consent for devices in the EEA region, while having a "pre-GDPR" behavior in other region (not setting any consent, as what's normally done before the GDPR came out).
- It may be desirable to use a localization tool to localize the consent dialog to languages in the EEA region.

Since it's common to have multiple consent-requirable services in your app, some of those may be governed by Easy Mobile while others may be not, we recommend having a custom "app consent" class that represents the collection of all consent given to each individual service in the app. This class should have several functionalities:

- Represents the collection of consent given to various services, specifically advertising, analytics and notifications as in the example script
- Forwards the collected consent to corresponding services using appropriate API provided by Easy Mobile (and other relevant 3rd party SDKs if any)
- Allows saving and retrieving the collected consent to and from PlayerPrefs in form of a JSON string (thus it is attributed as *Serializable*) so that we can restore the consent values and reflect them on the consent dialog when it is shown again

For the purpose of explaining this proposed workflow, we reuse the *DemoAppConsent* class found in our Privacy demo scene as an example of the aforementioned class.

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

namespace EasyMobile.Demo
{
    /// <summary>
    /// This class represents a collection of individual consents for
    /// all 3rd-party services that require consent in our app.
    /// This class can be used to manage consent for any
    /// 3rd-party service, not just those managed by Easy Mobile.
    /// Note that this class is serializable so we can serialize it
    /// to a JSON string and store in PlayerPrefs.
    /// </summary>
    [Serializable]
    public class DemoAppConsent
    {
        public const string DemoStorageKey = "EM_Demo_AppConsent";

        #region 3rd-party Services Consent

        // The consent for the whole Advertising module.
```

```
        // (we could have had different consents for individual ad networks, but for
        // the sake of simplicity in this demo, we'll ask the user a single consent
        // for the whole module and use it for all ad networks).
        public ConsentStatus advertisingConsent = ConsentStatus.Unknown;

        // The consent for the whole Notifications module.
        // Note that data consent is only applicable to push notifications,
        // local notifications don't require any consent.
        public ConsentStatus notificationConsent = ConsentStatus.Unknown;

        // Since this demo app also has In-App Purchase, which forces the use of
        // Unity Analytics, we could have had to ask a consent for that too. However,
        // according to Unity it's sufficient to provide the user with an URL
        // so they can opt-out on Unity website. So we will include that URL in our
        // consent dialog and not need to ask and store any explicit consent locally.

        // Here you can add consent variables for other 3rd party services if needed,
        // including those not managed by Easy Mobile...

        #endregion

        /// <summary>
        /// To JSON string.
        /// </summary>
        /// <returns>A <see cref="System.String"/> that represents the current <see cref="EasyMobile.Demo.AppCo
nsent"/>.</returns>
        public override string ToString()
        {
            return JsonUtility.ToJson(this);
        }

        /// <summary>
        /// Converts this object to JSON and stores in PlayerPrefs with the provided key.
        /// </summary>
        /// <param name="key">Key.</param>
        public void Save(string key)
        {
            PlayerPrefs.SetString(key, ToString());
        }

        /// <summary>
        /// Forwards the consent to relevant modules of EM.
        /// </summary>
        /// <param name="consent">Consent.</param>
        /// <remarks>
        /// In a real-world app, you'd want to write similar method
        /// to forward the obtained consent not only to relevant EM modules
        /// and services, but also to other relevant 3rd-party SDKs in your app.
        public static void ApplyDemoAppConsent(DemoAppConsent consent)
        {
            // Forward the consent to the Advertising module.
            if (consent.advertisingConsent == ConsentStatus.Granted)
                Advertising.GrantDataPrivacyConsent();
            else if (consent.advertisingConsent == ConsentStatus.Revoked)
                Advertising.RevokeDataPrivacyConsent();

            // Forward the consent to the Notifications module.
            if (consent.notificationConsent == ConsentStatus.Granted)
                Notifications.GrantDataPrivacyConsent();
            else if (consent.notificationConsent == ConsentStatus.Revoked)
                Notifications.RevokeDataPrivacyConsent();

            // Here you can forward the consent to other relevant 3rd-party services if needed...

        }

        /// <summary>
        /// Saves the give app consent to PlayerPrefs as JSON using the demo storage key.
        /// </summary>
        /// <param name="consent">Consent.</param>
        public static void SaveDemoAppConsent(DemoAppConsent consent)
        {
            if (consent != null)
                consent.Save(DemoStorageKey);
```

```
        }

        /// <summary>
        /// Loads the demo app consent from PlayerPrefs, returns null if nothing stored previously.
        /// </summary>
        /// <returns>The demo app consent.</returns>
        public static DemoAppConsent LoadDemoAppConsent()
        {
            string json = PlayerPrefs.GetString(DemoStorageKey, null);

            if (!string.IsNullOrEmpty(json))
                return JsonUtility.FromJson<DemoAppConsent>(json);
            else
                return null;
        }
    }
}
```

The following pseudo-code illustrates the process that should be done at every app launch to collect and forward consent (if needed) and then initialize Easy Mobile in a consent-aware app.

```
if NOT in EEA region                            // not affected by GDPR
{
    Initialize Easy Mobile                      // no consent needed, go ahead with initialization
}
else                                            // is in EEA region, affected by GDPR
{
    if NOT first app launch                     // consent should be given already in the 1st launch
    {
        Initialize Easy Mobile                  // go ahead with initialization
    }
    else                                        // this is first app launch, should ask for consent
    {
        Get the default consent dialog          // or create a new one
        Localize the consent dialog             // if needed, preferably using a localization tool
        Show non-dismissible consent dialog     // user can not skip it
        Construct new "app consent" object      // consent collected, construct the object off it
        Forward consent to relevant services    // using the "app consent" object
        Save consent values                     // using the "app consent" object
        Initialize Easy Mobile                  // everything is set, can now start working!
    }
}
```

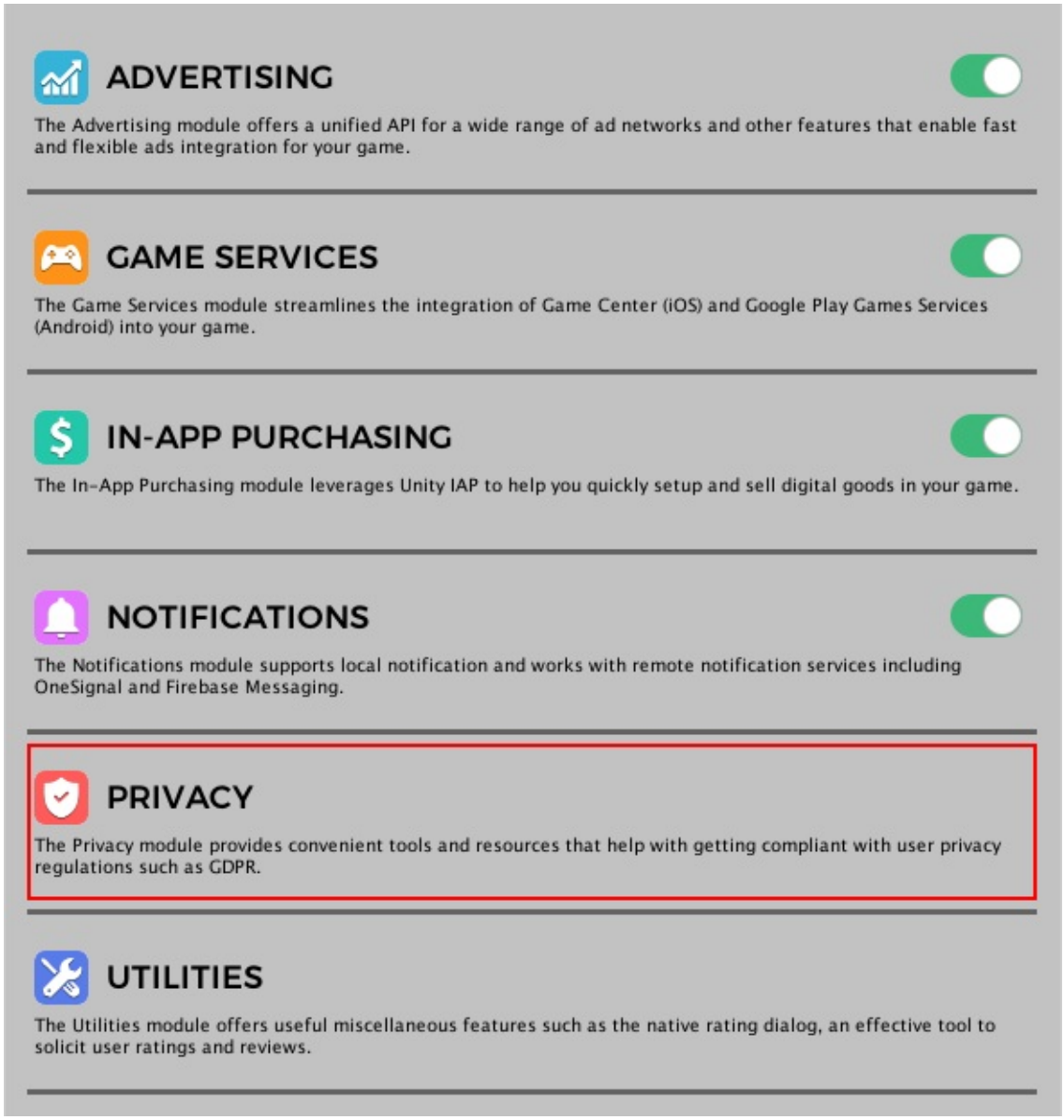
As mentioned previously, apart from collecting consent at the first app launch we also need to provide a way for the user to access the consent dialog again to update their consent if needed. The simplest way is having a menu button that can open the consent dialog at the user discretion. This dialog should reflect the current consent status and is dismissible. The following pseudo-code illustrate the process to handle such consent dialog opening button.

```
if button clicked
{
    Get the default consent dialog              // or create a new one
    Localize the consent dialog                 // if needed, preferably using a localization tool

    if having a stored consent                  // check using the "app consent" class
    {
        Retrieve the stored consent             // using the "app consent" class
        Update dialog with stored consent       // reflect the current consent status
    }

    Show dismissible consent dialog             // user can skip it if opened unintentionally

    if consent is updated                       // dialog is close & consent has changed
    {
        Construct new "app consent" object      // consent collected, construct the object off it
        Forward consent to relevant services    // using the "app consent" object
        Save consent values                     // using the "app consent" object
        Show alert                              // inform user changes will be applied since next app launch
    }
}
```

> Please see folder *Assets/EasyMobile/Demo/Scripts/PrivacyDemo* for example scripts on this subject.

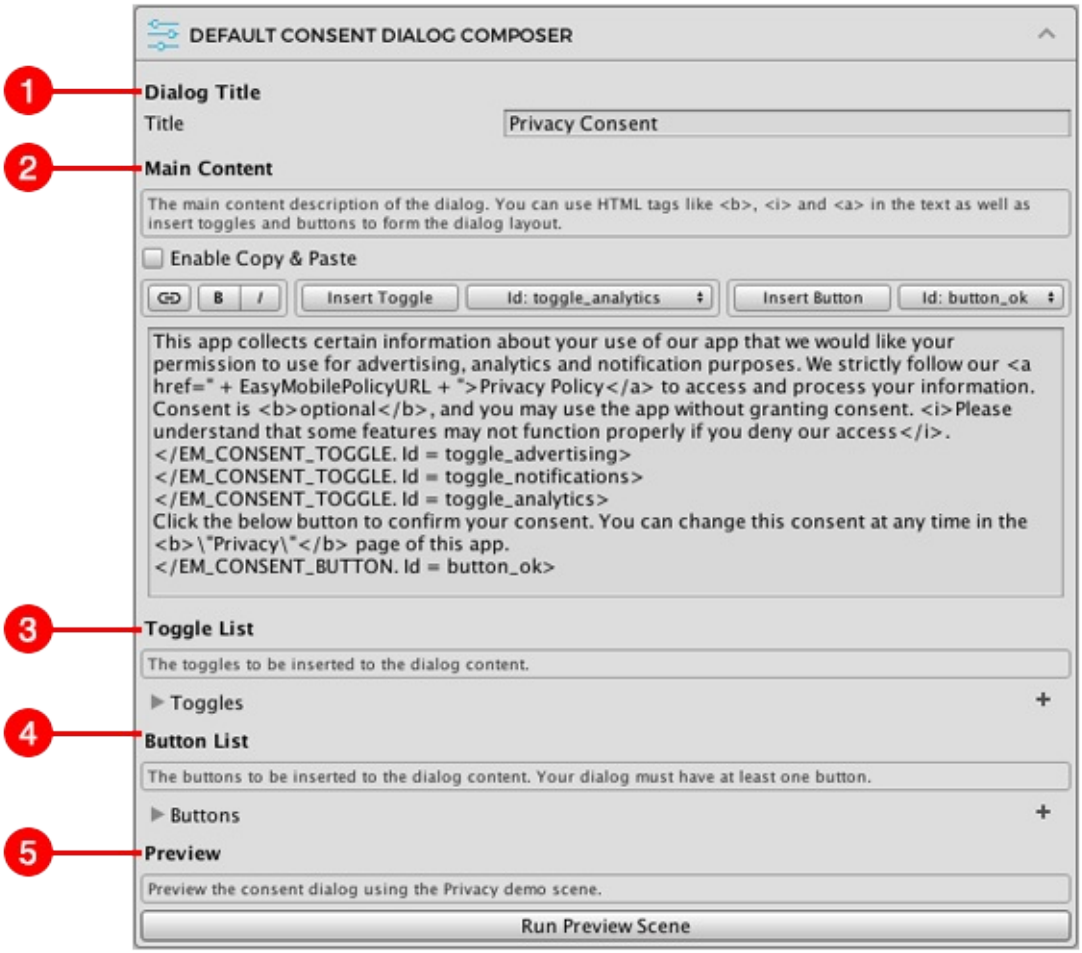
Privacy: Settings

The settings interface of the Privacy module can be accessed from the Privacy tab at menu *Window* > *Easy Mobile* > *Settings*.



Default Consent Dialog Composer

There are two ways to construct an Easy Mobile consent dialog: building it in script or using the built-in graphical consent dialog composer. The latter can be found in the Privacy module settings interface. The dialog composed with this composer is the default consent dialog and can be obtained from script using the *Privacy.GetDefaultConsentDialog()* method (see Working with Default Consent Dialog).



The composer consists of following sections:

1. *Title*: for editing the dialog's title
2. *Main Content*: this is where you compose the main content of the dialog by inserting texts, toggles or buttons.
3. *Toogle List*: this is where you manage the toggles in your dialog, adding or removing toggles, setting their parameters such as ID, title, description, etc.
4. *Button List*: this is where you manage the buttons in your dialog, adding or removing buttons, setting their parameters such as ID, label, colors, etc.
5. *Preview*: you can use the button in this section to open the preview scene for the composed dialog.

Composing Main Content

You can edit the body of your consent dialog in the **Main Content** section of the composer.

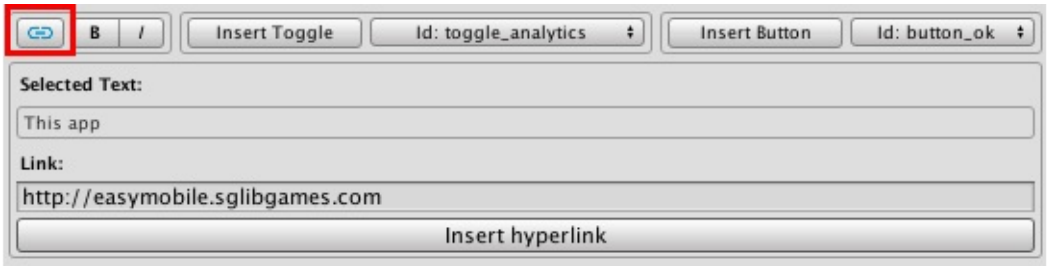
1. *Enable Copy & Paste*: in some versions of Unity we can't have both text selection function (for styling or inserting hyperlinks) and copy-paste function at the same time. In such case you can check this option when you want to do copy-paste and uncheck it to use the toolbar buttons (which require text selection).
2. *Toolbar*: containing buttons for text styling (bold, italic), inserting hyperlinks, inserting toggles and buttons.
3. *Text Area*: for inputing the text that describes the content of your consent dialog.

Text Styling

Common HTML tags including <b> (bold) and <i> (italic) can be used to style the text. In the composer you can select a chunk of text and use the B or *I* button on the toolbar to make it bold or italic respectively, the appropriate tag will be added to the text in the below text area automatically.

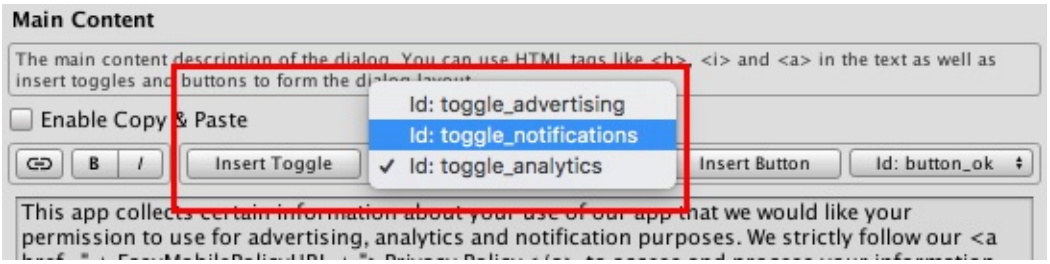
Inserting Hyperlinks

It's important to provide links to appropriate privacy policy in the consent dialog, which can be done by inserting hyperlinks. To insert a hyperlink, simply select the appropriate text in the text area and click the hyperlink button on the toolbar and enter the appropriate URL.



Inserting Toggles

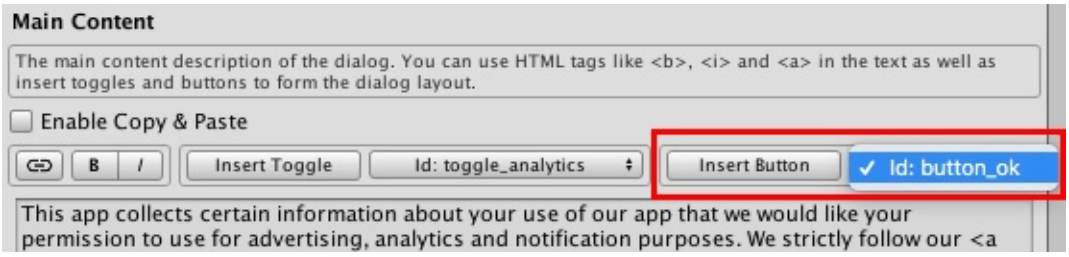
Before inserting toggles to the consent dialog, you must create them in the **Toggle List** section first (see Managing Toggles). Then select the toggle to insert using its ID in the toggle dropdown on the toolbar and click the *Insert Toggle* button. A special toggle tag (e.g. </EM_CONSENT_TOGGLE. Id = toggle_advertising>) will be inserted at the current cursor position in the text area. In runtime, this will be replaced by an actual toggle constructed with the parameters corresponding to the toggle ID in the toggle list.



Inserting Buttons

Before inserting buttons to the consent dialog, you must create them in the **Button List** section first (see Managing Buttons). Then select the button to insert using its ID in the button dropdown on the toolbar and click the *Insert Button* button. A special button tag (e.g. </EM_CONSENT_BUTTON. Id = button_ok>) will be inserted at the current cursor position in the text area. In runtime, this will be replaced by an actual button constructed with the parameters corresponding to the button ID in the button list.

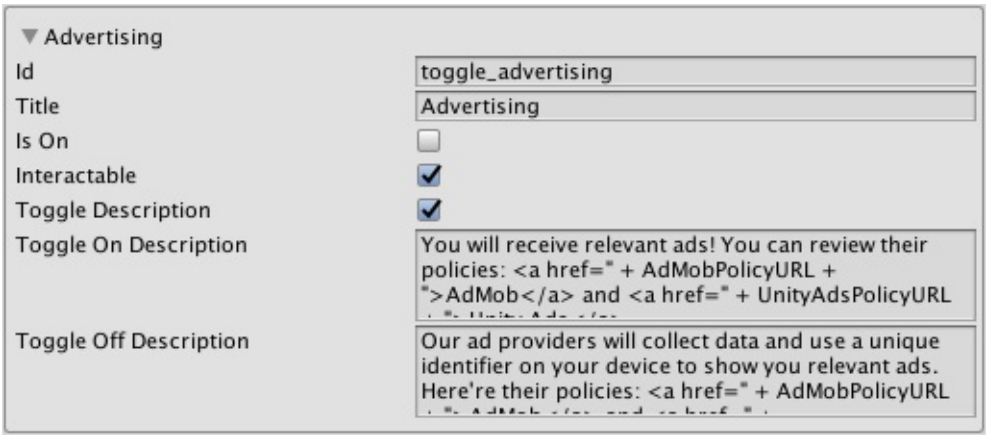
> Your consent dialog must have at least one button so that it can be close with the *Completed* event.



Managing Toggles

Toggles in your consent dialog can be designed and managed in the **Toggle List** section. To create a new toggle click the plus button at the top right corner. To remove a toggle click the minus button to the right of that toggle.

The following shows the parameters of a toggle.

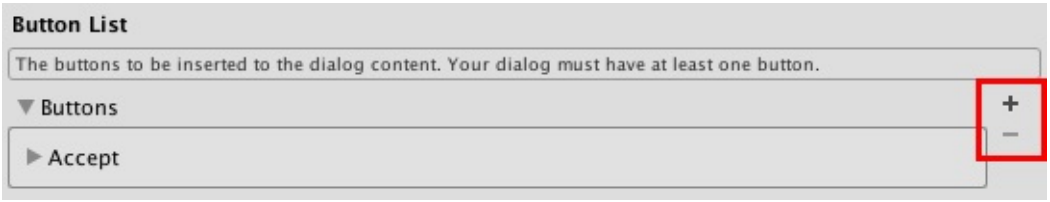


- *Id*: the ID of the toggle.
- *Title*: the title of the toggle.
- *Is On*: whether the toggle should be turned on by default.
- *Interactable*: whether the toggle is clickable, this value can be changed from script in runtime.
- *Toggle Description*: whether the description should be updated according to the on off state of the toggle, if this value is false, the *Toggle On Description* will always show.
- *Toggle On Description*: the description to be shown when the toggle is on, or when Toggle Description is false.
- *Toggle Off Description*: the description to be shown when the toggle is off and *Toggle Description* is true.

> Similar to the main body text, you can use common HTML tags such as <b>, <i> and <a> in the *Toggle On Description* and *Toggle Off Description*.

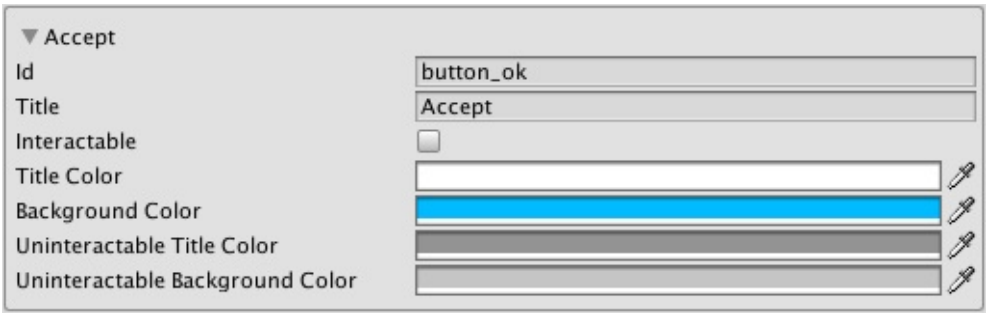
Managing Buttons

Buttons in your consent dialog can be designed and managed in the **Button List** section. To create a new button click the plus button at the top right corner. To remove a button click the minus button to the right of that button.



> Again, you consent dialog must have at least one button so the last one won't be removable.

The following shows the parameters of a button.

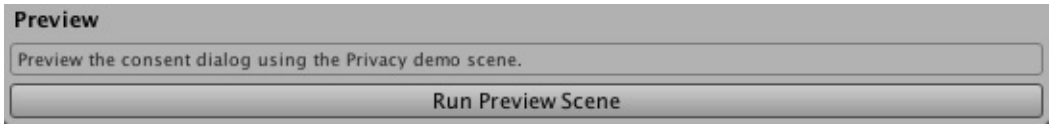


- *Id*: the ID of the button.
- *Title*: the text on the button
- *Interactable*: whether the button is clickable, this value can be changed from script in runtime.
- *Title Color*: the color of the button text in normal state (clickable).
- *Background Color*: the color of the button body in normal state (clickable).

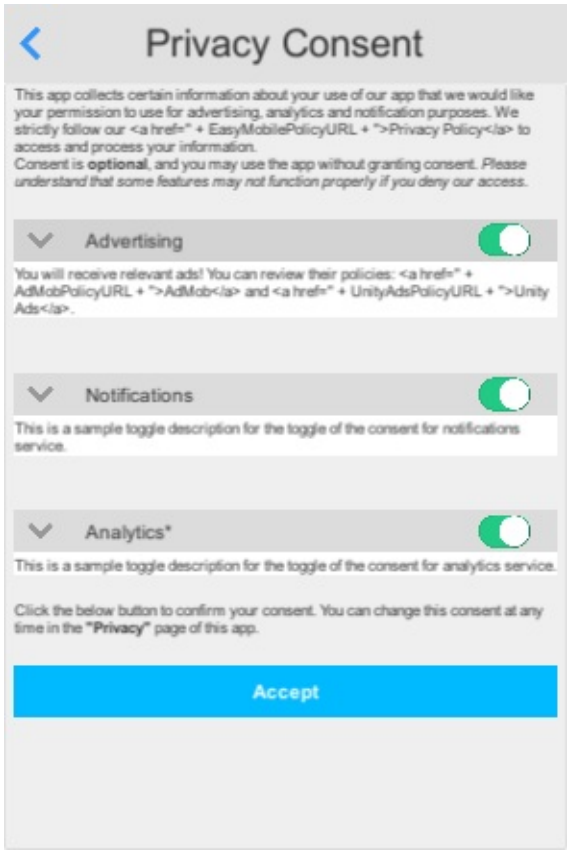
- *Uninteractable Title Color*: the color of the button text when it is not clickable.
- *Uninteractable Background Color*: the color of the button body when it is not clickable.

Previewing Consent Dialog

You can preview the consent dialog while composing it without having to export a build to a mobile device. In the **Preview** section click the *Run Preview Scene* button to run the *Privacy Demo* scene.



In the opened scene click the *DEFAULT CONSENT DIALOG* button to open the default consent dialog which has just been composed. The consent dialog in the editor looks and functions similarly to its counterparts on iOS or Android and is useful for quick verifying of your dialog design.



Privacy: Scripting

This section provides a guide to work with the Privacy module scripting API.

EEA Region Checking

Easy Mobile provides a built-in validator to detect whether the current device is in the European Economic Area (EEA) region, where the GDPR governs. This validator performs the region check using the methods defined in the *EEARegionValidationMethods* enum (see Introduction chapter for a detailed explanation of these methods): GoogleService, Telephony, Timezone or Locale. It will return one of the following results: InEEA, NotInEEA or Unknown.

The validator can use any subset of the available methods, in any order. To instruct it to use a certain collection of methods in a specific order, simply prepare a *List* object containing the methods-to-use in the desired order. The validator will perform validating using the provided methods in their order in the containing list. If a method fails to return an explicit result (either InEEA or NotInEEA), the next one in the list will be used until a clear result is found. If all methods fail, the validator will return *Unknown* as the result.

Validating Using Default Method List

The default EEA region validating method list is defined as the *DefaultMethods* variable of the *EEARegionValidator* class. It employs all the available methods in the following order:

1. GoogleService
2. Telephony
3. Timezone
4. Locale

To perform EEA region validation using the default method list, call the *IsInEEARegion* method of the *Privacy* class. This method is actually a wrapper of the *ValidateEEARegionStatus* method of the *EEARegionValidator* class so you can also use that method directly.

```
using UnityEngine;
using EasyMobile;

public static class ExampleEEARegionDetector
{
    // Checks if we're in EEA region.
    public static void Check()
    {
        // You can use the method of the Privacy class.
        Privacy.IsInEEARegion(CheckEEARegionCallback);

        // You can also do this.
        // EEARegionValidator.ValidateEEARegionStatus(CheckEEARegionCallback);
    }

    // Callback to be invoked when the validation completes.
    static void CheckEEARegionCallback(EEARegionStatus result)
    {
        if (result == EEARegionStatus.InEEA)
            Debug.Log("We're in EEA region!");
        else if (result == EEARegionStatus.NotInEEA)
            Debug.Log("We're not in EEA region!");
        else
            Debug.Log("Result is Unknown: couldn't determine if we're in EEA region or not.");
    }
}
```

Validating Using Custom Method List

You can instruct the validator to use a custom list of desired methods in the desired order. To do so use the *ValidateEEARegionStatus* method of the *EEARegionValidator* class but provide it your custom method list.

```
using UnityEngine;
using EasyMobile;
using System.Collections.Generic;

public static class ExampleEEARegionDetector
{
    // Checks if we're in EEA region using custom method list.
    public static void CheckWithCustomMethodList()
    {
        // First create a custom list of methods in the preferred order.
        var methodList = new List<EEARegionValidationMethods>()
            {
                EEARegionValidationMethods.Telephony,
                EEARegionValidationMethods.GoogleService,
                EEARegionValidationMethods.Locale,
                EEARegionValidationMethods.Timezone
            };

        // Validates using the custom method list.
        EEARegionValidator.ValidateEEARegionStatus(CheckEEARegionCallback, methodList);
    }

    // Callback to be invoked when the validation completes.
    static void CheckEEARegionCallback(EEARegionStatus result)
    {
        if (result == EEARegionStatus.InEEA)
            Debug.Log("We're in EEA region!");
        else if (result == EEARegionStatus.NotInEEA)
            Debug.Log("We're not in EEA region!");
        else
            Debug.Log("Result is Unknown: couldn't determine if we're in EEA region or not.");
    }
}
```

Working with Consent Dialog

Getting the Default Consent Dialog

You can get the default consent dialog that was composed using the built-in composer (see Consent Dialog Composer) using the *GetDefaultConsentDialog* method of the *Privacy* class.

```
ConsentDialog dialog = Privacy.GetDefaultConsentDialog();
```

Localizing the Consent Dialog

You can localize the main content of the consent dialog by grabbing its *Content* property and replace the placeholder texts with the appropriate localized texts.

```
// Replace placeholder texts in main content with localized texts.
// You may need to repeat this multiple times to replace all placeholders.
ddialog Content = dialog.Content.Replace("PLACEHOLDER_TEXT", "LOCALIZED_TEXT");
```

To localize toggles in the dialog you can iterate through the *Toggles* property and localize the title and the description texts of each toggle.

```
// Iterate through all toggles in the dialog and localize them.
foreach (ConsentDialog.Toggle toggle in dialog.Toggles)
{
    // Localize the toggle title.
    toggle.Title = toggle.Title.Replace("PLACEHOLDER_TOGGLE_TITLE", "LOCALIZED_TOGGLE_TITLE");

    // Localize the toggle on description.
    toggle.OnDescription = toggle.OnDescription.Replace("PLACEHOLDER_ON_DESCRIPTION", "LOCALIZED_ON_DESCRIPTION"
);

    // Localize the toggle off description if needed.
    toggle.OffDescription = toggle.OffDescription.Replace("PLACEHOLDER_OFF_DESCRIPTION", "LOCALIZED_OFF_DESCRIP
TION");
}
```

To localize buttons in the dialog you can iterate through the *ActionButtons* property and localize each button title.

```
// Iterate through all buttons in the dialog and localize them.
foreach (ConsentDialog.Button button in dialog.ActionButtons)
{
    // Localize the button text.
    button.Title = button.Title.Replace("PLACEHOLDER_BUTTON_TITLE", "LOCALIZED_BUTTON_TITLE");
}
```

> For the sake of simplicity, we use sample strings in the above example snippets. In a practical app you may use a dedicated localization tool, as well as checking toggle or button ID, to pick the correct translation for each placeholder text.

Showing Consent Dialog

To show a consent dialog simply call its *Show* method. This method has an optional 'dismissible' parameter. You can set it to 'true' to allow the user to dismiss the dialog using a cancel button that is added to the dialog automatically. Otherwise the dialog can only be closed by one of the buttons in its content.

```
// Shows the dialog and don't allow the user to dismiss it (must provide explicit consent).
dialog.Show(false);
```

Only one consent dialog can be shown at a time. Attempts to show a consent dialog when another is being shown will be ignored. You can check if there's any consent dialog being shown using the *IsShowingAnyDialog* method.

```
// Checks if any consent dialog is being shown.
bool isShowingAnotherDialog = ConsentDialog.IsShowingAnyDialog();
```

Consent Dialog Events

A consent dialog has following events:

Event	Description
ToggleStateUpdated	This event is raised when the value of a toggle in the (being shown) consent dialog has been updated. You may subscribe to this event if you want to update the dialog according to the values of its toggles.
Dismissed	This event is raised when the dialog is dismissed (closed by the cancel button)
Completed	This event is raised when the dialog is completed (close by one of the action buttons). The handler of this event will be called with the ID of the selected button and the values of the toggles in the dialog at the time it is closed.

Updating Toggles in Runtime

You can update the toggle value of a being-shown consent dialog using the *SetToggleIsOn* method on the dialog.

```
dialog.SetToggleIsOn("TOGGLE_ID", true); // turn the toggle on
```

You can also change the interactability of a toggle while the dialog is being shown using the *SetToggleInteractable* method.

```
dialog.SetToggleInteractable("TOGGLE_ID", false);   // make the toggle un-clickable
```

Updating Buttons in Runtime

In some cases, you may want to update the interactability of a button on a being-shown consent dialog, e.g. only make the button clickable if a certain toggle is on. For that purpose you can use the *SetButtonInteractable* method.

```
dialog.SetButtonInteractable("BUTTON_ID", true);    // make the button clickable
```

A Complete Example

The following script gives an example on how to grab the default consent dialog, localize and show it, as well as handle its events.

```
using UnityEngine;
using EasyMobile;
using System.Collections.Generic;

public static class ExampleConsentDialogDisplayer
{
    static bool hasSubscribedEvents = false;

    // Grabs the default consent dialog, localizes and then shows it.
    public static void ShowDefaultConsentDialog()
    {
        // Grab the default consent dialog that was built with the composer.
        ConsentDialog dialog = Privacy.GetDefaultConsentDialog();

        // Replace placeholder texts in main content with localized texts.
        // You may need to repeat this multiple times to replace all placeholders.
        dialog.Content = dialog.Content.Replace("PLACEHOLDER_TEXT", "LOCALIZED_TEXT");

        // Iterate through all toggles in the dialog and localize them.
        foreach (ConsentDialog.Toggle toggle in dialog.Toggles)
        {
            // Localize the toggle title.
            toggle.Title = toggle.Title.Replace("PLACEHOLDER_TOGGLE_TITLE", "LOCALIZED_TOGGLE_TITLE");

            // Localize the toggle on description.
            toggle.OnDescription = toggle.OnDescription.Replace("PLACEHOLDER_ON_DESCRIPTION", "LOCALIZED_ON_DES
CRIPTION");

            // Localize the toggle off description if needed.
            toggle.OffDescription = toggle.OffDescription.Replace("PLACEHOLDER_OFF_DESCRIPTION", "LOCALIZED_OFF
_DESCRIPTION");

            // Here you can also set the toggle value according to the
            // stored consent (if any) to reflect the current consent status.
            // toggle.IsOn = TRUE_OR_FALSE;
        }

        // Iterate through all buttons in the dialog and localize them.
        foreach (ConsentDialog.Button button in dialog.ActionButtons)
        {
            // Localize the button text.
```

```
            button.Title = button.Title.Replace("PLACEHOLDER_BUTTON_TITLE", "LOCALIZED_BUTTON_TITLE");
        }

        // Show the default consent dialog. If you want to allow the user to dismiss the dialog
        // without updating their consent, pass 'true' to the 'dismissible' argument. Otherwise
        // the dialog can only be closed with one of the action buttons.
        if (!ConsentDialog.IsShowingAnyDialog())
        {
            // Subscribe to the default consent dialog events.
            // Only do this once.
            if (!hasSubscribedEvents)
            {
                dialog.ToggleStateUpdated += DefaultDialog_ToggleStateUpdated;
                dialog.Dismissed += DefaultDialog_Dismissed;
                dialog.Completed += DefaultDialog_Completed;
                hasSubscribedEvents = true;
            }

            // Now shows the dialog and don't allow the user to dismiss it (must provide explicit consent).
            dialog.Show(false);
        }
        else
        {
            Debug.Log("Another consent dialog is being shown!");
        }
    }

    // Event handler to be invoked when the value of a toggle in the consent dialog is updated.
    static void DefaultDialog_ToggleStateUpdated(ConsentDialog dialog, string toggleId, bool isOn)
    {
        Debug.Log("Toggle with ID " + toggleId + " now has value " + isOn);

        // If there's a service mandatory to the operation of your app,
        // you may disable all buttons until the toggle associated with that service
        // is turn on, so that the user can only close the dialog once they grant consent to that service.
        if (toggleId.Equals("MANDATORY_TOGGLE"))
            dialog.SetButtonInteractable("SOME_BUTTON_ID", true);   // make the button clickable
    }


    // Event handler to be invoked when the consent dialog is dismissed.
    static void DefaultDialog_Dismissed(ConsentDialog dialog)
    {
        Debug.Log("The consent dialog has been dismissed!");
    }

    // Event handler to be invoked when the consent dialog completed.
    static void DefaultDialog_Completed(ConsentDialog dialog, ConsentDialog.CompletedResults results)
    {
        // The 'results' argument contains the ID of clicked button.
        Debug.Log("The consent dialog has completed with button ID " + results.buttonId);

        // The 'results' argument also returns the values of the toggles in the dialog.
        foreach (KeyValuePair<string, bool> kvp in results.toggleValues)
        {
            Debug.Log("Toggle with ID " + kvp.Key + " has value " + kvp.Value);

            // Here you can perform relevant actions, e.g. update the consent
            // for individual services according to the toggle value...
        }
    }
}
```

Creating Consent Dialog Programmatically

Apart from building consent dialog using the graphical composer, you can construct a dialog completely from script, using methods such as AppendText, AppendToggle and AppendButton to add texts, toggles and buttons to the dialog content, respectively. The following example shows how a consent dialog can be constructed programmatically.

> For the sake of simplicity we use sample hard-coded strings in the example script. In a practical app you may use a localization tool to pick the correct translation for each text. Also you can load the consent provided by the user previously and set the toggle values accordingly to reflect the current consent status.

```
using UnityEngine;
using EasyMobile;
using System.Collections.Generic;

public static class ExampleConsentDialogDisplayer
{
    // Constructs a consent dialog from script.
    public static ConsentDialog ConstructConsentDialog()
    {
        // First create a new consent dialog.
        ConsentDialog dialog = new ConsentDialog();

        // Set the title.
        dialog.Title = "SOME_CONSENT_DIALOG_TITLE";

        // Add the first paragraph.
        dialog.AppendText("FIRST_PARAGRAPH_TEXT");

        // Build and append the toggle for advertising service consent.
        ConsentDialog.Toggle adsToggle = new ConsentDialog.Toggle("ADS_TOGGLE_ID");
        adsToggle.Title = "ADS_TOGGLE_TITLE";
        adsToggle.OnDescription = "ADS_TOGGLE_ON_DESCRIPTION";
        adsToggle.OffDescription = "ADS_TOGGLE_OFF_DESCRIPTION";
        adsToggle.ShouldToggleDescription = true;   // make the description change with the toggle state.
        adsToggle.IsOn = false;     // make the toggle off by default

        // Append the toggle after the 1st paragraph.
        dialog.AppendToggle(adsToggle);

        // Build and append the toggle for notifications service consent.
        ConsentDialog.Toggle notifsToggle = new ConsentDialog.Toggle("NOTIFS_TOGGLE_ID");
        notifsToggle.Title = "NOTIFS_TOGGLE_TITLE";
        notifsToggle.OnDescription = "NOTIFS_TOGGLE_ON_DESCRIPTION";
        notifsToggle.ShouldToggleDescription = false;   // use same description for both on & off states.
        notifsToggle.IsOn = false; // make the toggle off by default

        // Append the toggle below the previous toggle.
        dialog.AppendToggle(notifsToggle);

        // Build and append the toggle for analytics service consent.
        ConsentDialog.Toggle uaToggle = new ConsentDialog.Toggle("ANALYTICS_TOGGLE_ID");
        uaToggle.Title = "ANALYTICS_TOGGLE_TITLE";
        uaToggle.OnDescription = "ANALYTICS_TOGGLE_ON_DESCRIPTION";
        uaToggle.ShouldToggleDescription = false;   // the description won't change when the toggle switches be
tween on & off states.
        uaToggle.IsInteractable = false; // not interactable
        uaToggle.IsOn = true;   // assuming analytics is vital to our app, make its toggle on by default

        // Append the toggle below the previous toggle.
        dialog.AppendToggle(uaToggle);

        // Append the second paragraph.
        dialog.AppendText("SECOND_PARAGRAPH_TEXT");

        // Build and append the accept button.
        // A consent dialog should always have at least one button!
        ConsentDialog.Button okButton = new ConsentDialog.Button("OK_BUTTON_ID");
        okButton.Title = "OK_BUTTON_TITLE";
        okButton.TitleColor = Color.white;
        okButton.BodyColor = new Color(66 / 255f, 179 / 255f, 1);

        // Append the button to the bottom of the dialog.
        dialog.AppendButton(okButton);

        return dialog;
    }
}
```

Managing Global Consent

You can grant or revoke the global consent of your app using the *GrantGlobalDataPrivacyConsent* and *RevokeGlobalDataPrivacyConsent* methods of the Privacy class, respectively.

```
// Grants global consent.
Privacy.GrantGlobalDataPrivacyConsent();

// Revokes global consent.
Privacy.RevokeGlobalDataPrivacyConsent();
```

You can get the current state of the global consent of your app using the *GlobalDataPrivacyConsent* property of the Privacy class.

```
// Gets current global consent.
ConsentStatus globalConsent = Privacy.GlobalDataPrivacyConsent;
```

These APIs of the Privacy class are actually wrapper of those of the *GlobalConsentManager*, which is the object that actually manages the global consent. So the above snippets are equivalent to the following script.

```
// Grants global consent.
GlobalConsentManager.Instance.GrantDataPrivacyConsent();

// Revokes global consent.
GlobalConsentManager.Instance.RevokeDataPrivacyConsent();

// Gets current global consent.
ConsentStatus globalConsent = GlobalConsentManager.Instance.DataPrivacyConsent;
```

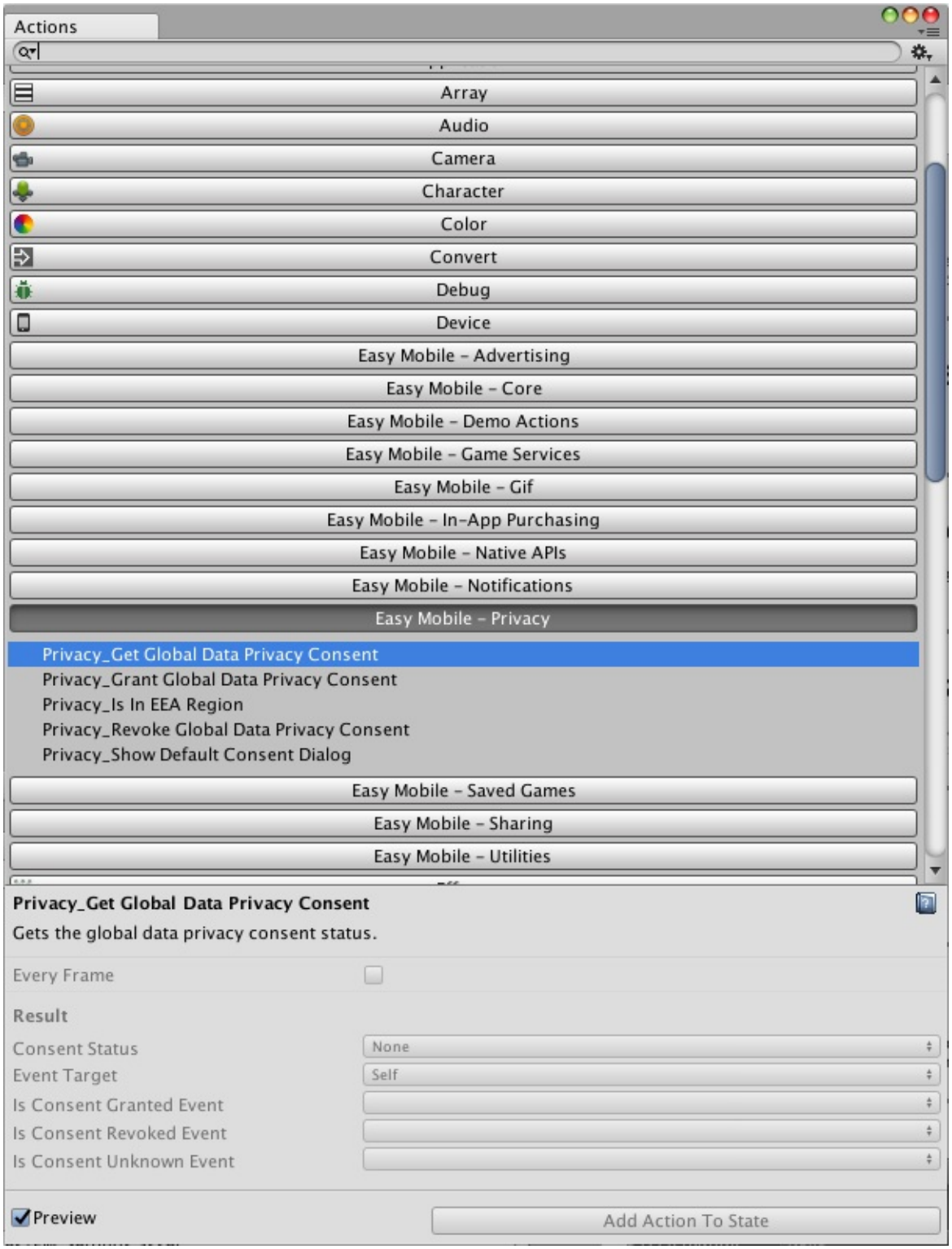
You can also acknowledge when the global consent is updated by subscribing to the *DataPrivacyConsentUpdated* event.

```
using UnityEngine;
using EasyMobile;

public class ExampleConsentClass : MonoBehaviour
{
    void OnEnable()
    {
        // Subscribe.
        GlobalConsentManager.Instance.DataPrivacyConsentUpdated += OnGlobalDataPrivacyConsentUpdated;
    }

    void OnDisable()
    {
        // Unsubscribe.
        GlobalConsentManager.Instance.DataPrivacyConsentUpdated -= OnGlobalDataPrivacyConsentUpdated;
    }

    void OnGlobalDataPrivacyConsentUpdated(ConsentStatus consent)
    {
        if (consent == ConsentStatus.Granted)
            Debug.Log("Global consent has been granted!");
        else if (consent == ConsentStatus.Revoked)
            Debug.Log("Global consent has been revoked!");
        else
            Debug.Log("Global consent is unknown.");
    }
}
```

Privacy: PlayMaker Actions

The PlayMaker actions of the Privacy module are group in the category *Easy Mobile - Privacy* in the PlayMaker's Action Browser.

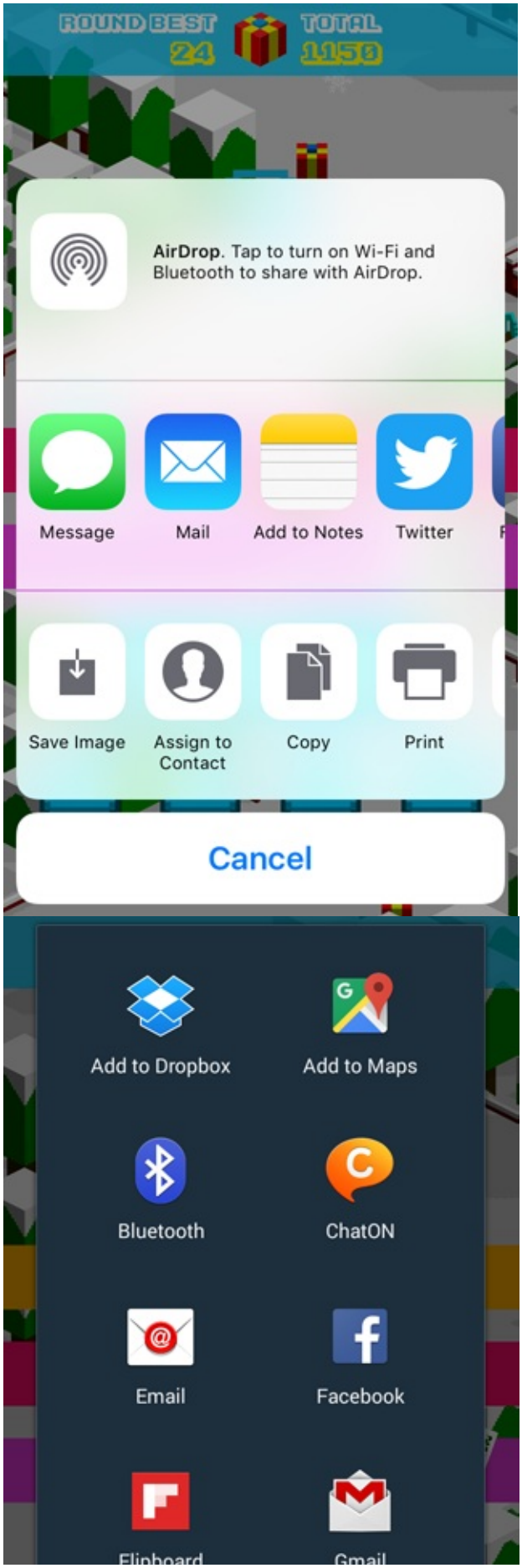
> Please refer to the PrivacyDemo_PlayMaker scene in folder *Assets/EasyMobile/Demo/PlayMakerDemo/Modules* for an example on how these actions can be used.

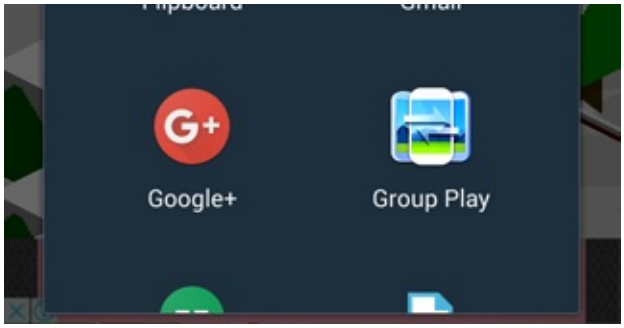


Sharing: Introduction

The Sharing module helps you easily share texts and images to social networks including Facebook, Twitter and Google+ using the native sharing functionality. In addition, it also provides convenient methods to capture the screenshots to be shared.

Below are the sharing interfaces on iOS and Android, respectively.


ROUND BEST
24
TOTAL
1158
AirDrop. Tap to turn on Wi-Fi and Bluetooth to share with AirDrop.
Message
Mail
Add to Notes
Twitter
Save Image
Assign to Contact
Copy
Print
Cancel
Add to Dropbox
Add to Maps
Bluetooth
ChatON
Email
Facebook
Flipboard
Gmail

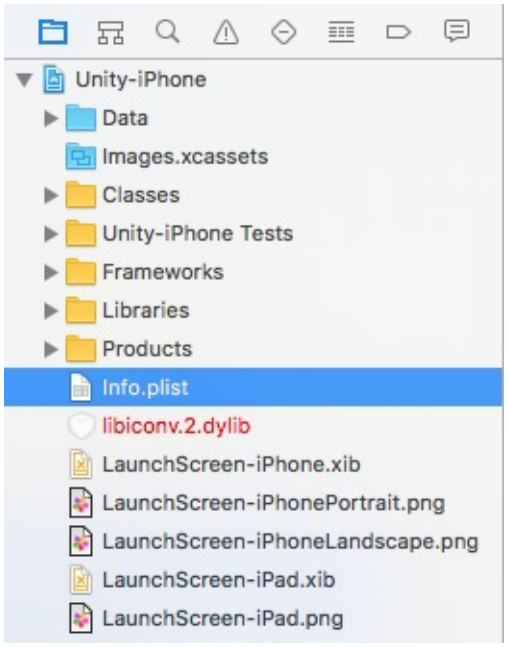


[iOS] Request Photo Library Access Permission

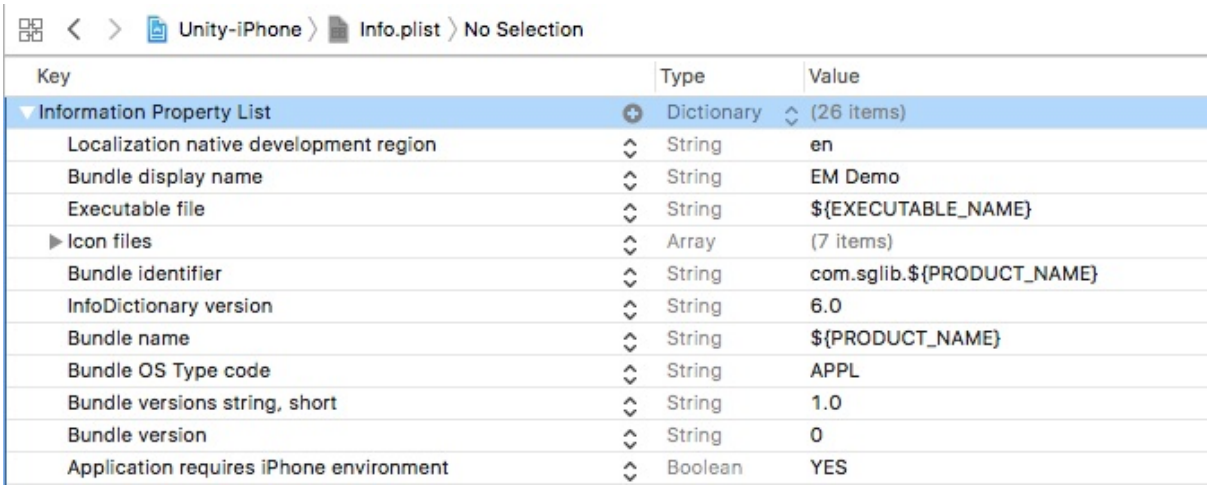
Since iOS 10, in order to use the "Save Image" feature of the sharing utility, the app needs to ask for user permission before it can access the photo library. Failure to do so will cause the app to crash as soon as the user selects the option. To request the photo library access permission, you need to add the **Privacy - Photo Library Usage Description** and **Privacy - Photo Library Additions Usage Description** properties to the Info.plist of your Xcode project.

> As of this writing, out tests show that on iOS 10, the **Privacy - Photo Library Usage Description** property is required. While iOS 11 asks for the **Privacy - Photo Library Additions Usage Description** property. Therefore it's recommended to add both properties if your target platforms including iOS 10 and above.

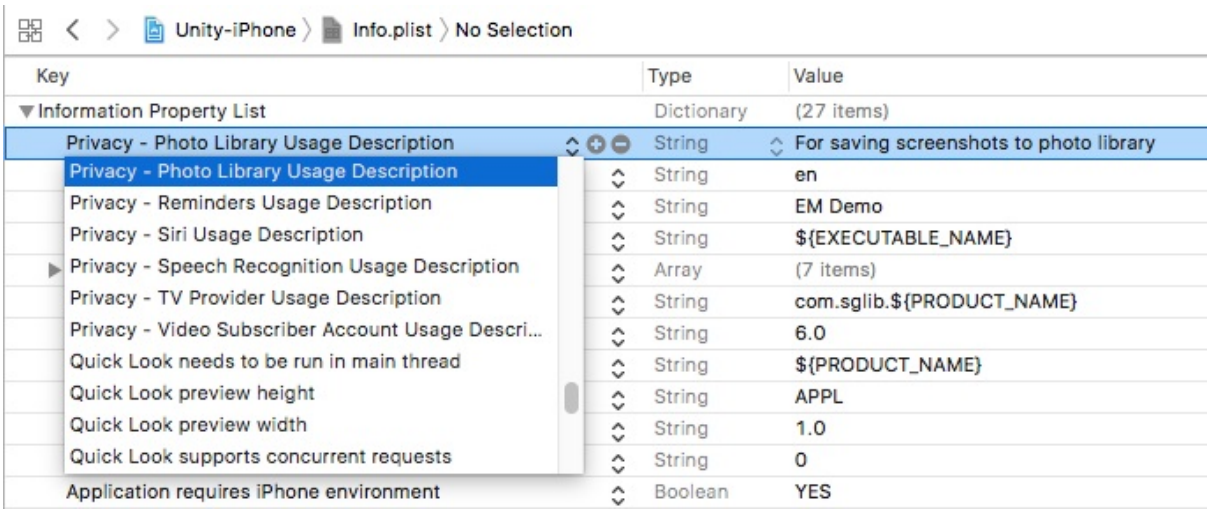
In your generated Xcode project open the Info.plist file.



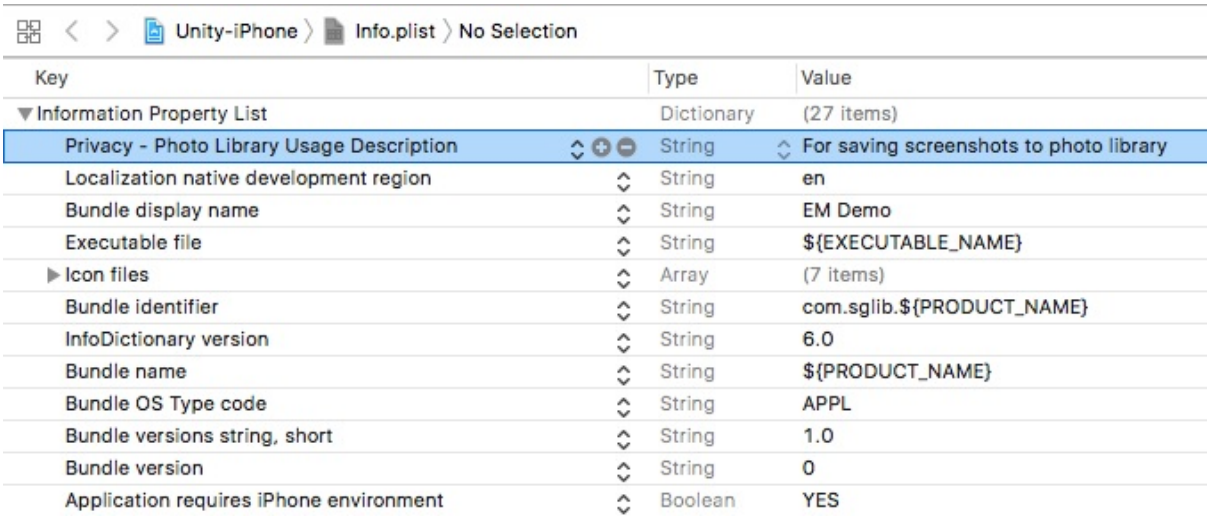
Click the **+** button on the right of **Information Property List** to add a new key.



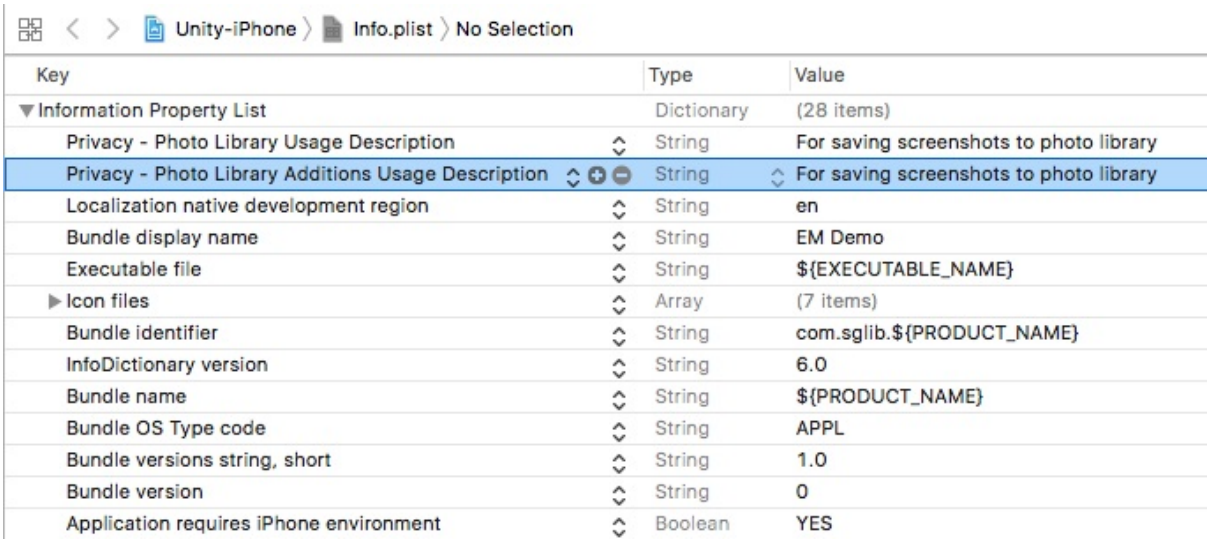
Scroll down to find the **Privacy - Photo Library Usage Description** key.



Enter a value for the key, this message will be displayed as the app requests access permission when the user selects the "Save Image" option.

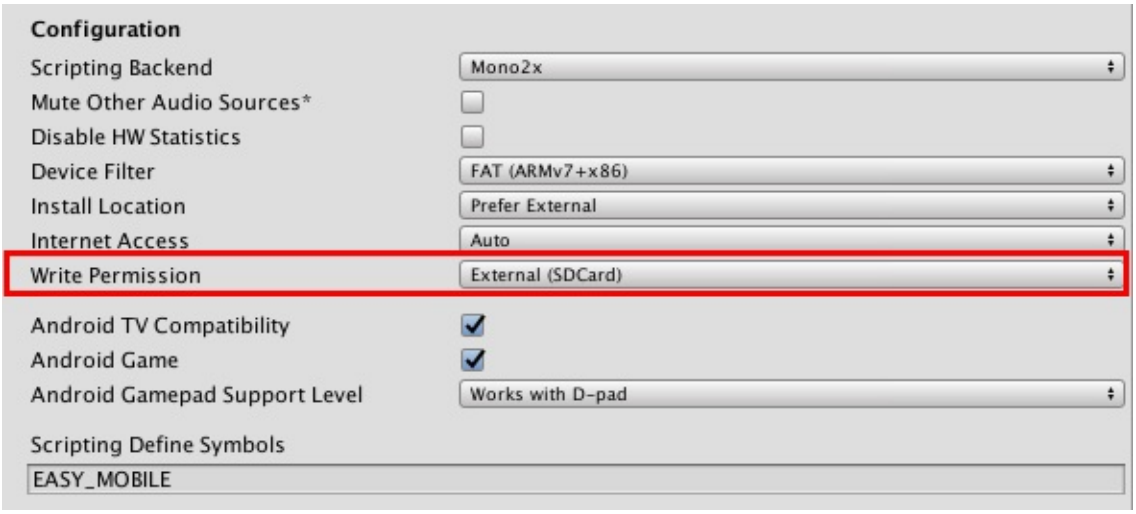


Repeat the process to add the **Privacy - Photo Library Additions Usage Description** property.



[Android] Enable External Write Permission

For this module to function on Android, it is necessary to enable the permission to write to external storage. To do so, go to *Edit* > *Project Settings* > *Player*, select *Android settings* tab, then locate the **Configuration** section and set the *Write Permission* to *External (SDCard)*.



Sharing: Scripting

This section provides a guide to work with Sharing module scripting API.

> You can access the Sharing module API via the Sharing class under the EasyMobile namespace.

Screenshot Capturing

To capture the device's screenshot, you have a few options.

Screenshot as PNG Image

To capture and save a screenshot of the whole device screen, simply specify the file name to be saved. This screenshot will be saved as a PNG image in the directory pointed by Application.persistentDataPath. Note that this method, as well as other screenshot capturing methods, needs to be called at the end of a frame (when the rendering has done) for it to produce a proper image. Therefore you should call it within a coroutine after *WaitForEndOfFrame()*.

```
// Coroutine that captures and saves a screenshot
IEnumerator SaveScreenshot()
{
    // Wait until the end of frame
    yield return new WaitForEndOfFrame();

    // The SaveScreenshot() method returns the path of the saved image
    // The provided file name will be added a ".png" extension automatically
    string path = Sharing.SaveScreenshot("screenshot");
}
```

You can also captures and saves just a portion of the screen:

```
// Coroutine that captures and saves a portion of the screen
IEnumerator SaveScreenshot()
{
    // Wait until the end of frame
    yield return new WaitForEndOfFrame();

    // Capture the portion of the screen starting at (50, 50),
    // has a width of 200 and a height of 400 pixels.
    string path = Sharing.SaveScreenshot(50, 50, 200, 400, "screenshot");
}
```

Screenshot as Texture2D

In some cases you may want to capture a screenshot and obtain a Texture2D object of it instead of saving to disk, e.g. to create a sprite from the texture and display it in-game.

```
// Coroutine that captures a screenshot and generates a Texture2D object of it
IEnumerator CaptureScreenshot()
{
    // Wait until the end of frame
    yield return new WaitForEndOfFrame();

    // Create a Texture2D object of the screenshot using the CaptureScreenshot() method
    Texture2D texture = Sharing.CaptureScreenshot();
}
```

Similar to the case above, you can also capture only a portion of the screen.

```
// Coroutine that captures a portion of the screenshot and generates a Texture2D object of it
IEnumerator CaptureScreenshot()
{
    // Wait until the end of frame
    yield return new WaitForEndOfFrame();

    // Create a Texture2D object of the screenshot using the CaptureScreenshot() method
    // The captured portion starts at (50, 50) and has a width of 200, a height of 400 pixels.
    Texture2D texture = Sharing.CaptureScreenshot(50, 50, 200, 400);
}
```

> Note that screenshot capturing should be done at the end of the frame.

Sharing

To share an image you also have a few options. You can also attach a message to be shared with the image.

> Due to Facebook policy, pre-filled messages will be ignored when sharing to this network, i.e. sharing messages must be written by the user.

Share a Saved Image

You can share a saved image by specifying its path.

```
// Share a saved image
// Suppose we have a "screenshot.png" image stored in the persistentDataPath,
// we'll construct its path first
string path = System.IO.Path.Combine(Application.persistentDataPath, "screenshot.png");

// Share the image with the path, a sample message and an empty subject
Sharing.ShareImage(path, "This is a sample message");
```

Share a Texture2D

You can also share a Texture2D object obtained some point before the sharing time. Internally, this method will also create a PNG image from the Texture2D, save it to the persistentDataPath, and finally share that image.

```
// Share a Texture2D
// sampleTexture is a Texture2D object captured some time before
// This method saves the texture as a PNG image named "screenshot.png" in persistentDataPath,
// then shares it with a sample message and an empty subject
Sharing.ShareTexture2D(sampleTexture, "screenshot", "This is a sample message");
```

Share a Text

You can share a text-only message using the *ShareText* method. Note that Facebook doesn't allow pre-filled sharing messages, so the text will be discarded when sharing to this particular network.

```
// Share a text
Sharing.ShareText("Hello from Easy Mobile!");
```

Share a URL

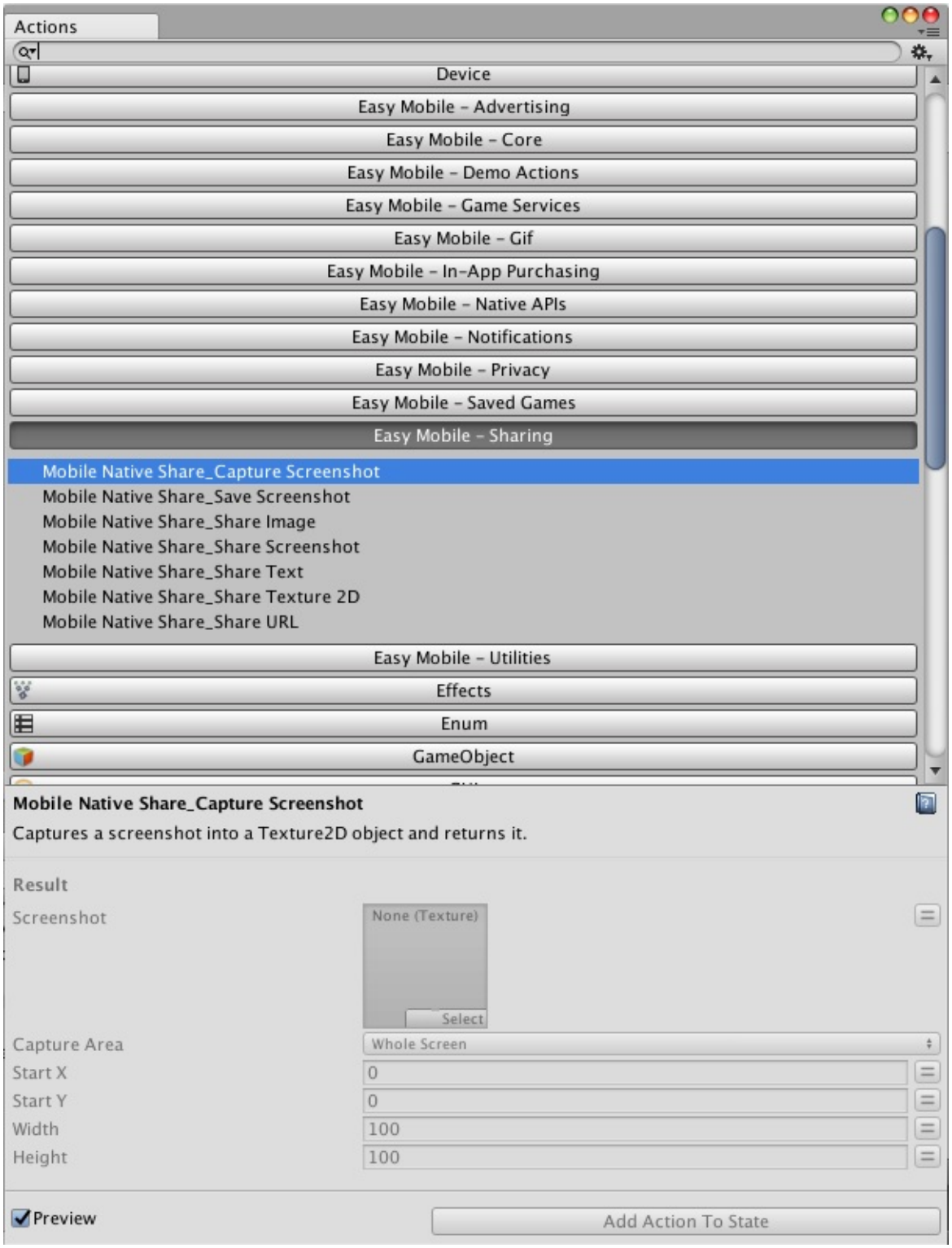
To share a URL, use the *ShareURL* method. On networks like Facebook or Twitter, a summary of the page will be shown if the shared URL points to a website. URLs are also useful to share GIF images hosted on sites like Giphy (see the *GIF* > *Scripting* section).

```
// Share a URL
Sharing.ShareURL("www.sglibgames.com");
```

Sharing: PlayMaker Actions

The PlayMaker actions of the Sharing module are group in the category *Easy Mobile - Sharing* in the PlayMaker's Action Browser.

> Please refer to the SharingDemo_PlayMaker scene in folder *Assets/EasyMobile/Demo/PlayMakerDemo/Modules* for an example on how these actions can be used.



Utilities: Introduction

The Utilities module is a place to hold useful miscellaneous features. The first feature added to this module is Store Review.

Utilities | Store Review: Introduction

Ratings and reviews can have a crucial impact on the performance of an app on app stores. Therefore it's a common practice to ask users for ratings when appropriate. The Store Review feature gives you an efficient way to do that using a native and highly customizable rating dialog.

This rating dialog has different appearances and behaviors depended on the platform it is being used.

iOS

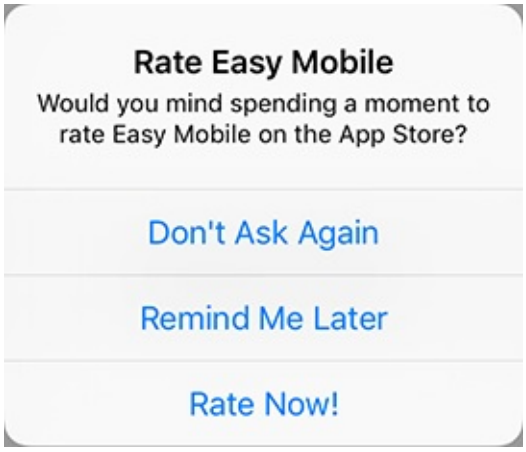
iOS 10.3 and newer

On iOS 10.3 or newer, the system-provided rating dialog is employed. This dialog is built-in to iOS since its 10.3 release, and is the preferred method to solicit user ratings on this platform.

> You can find more information about this built-in rating prompt at https://developer.apple.com/ios/human-interface-guidelines/interaction/ratings-and-reviews/
>
> It's worth noting that the Submit button on this rating popup will be disabled while your app is still in sandbox mode. It will be functioning normally when the app is actually live on App Store.

iOS Before 10.3

On iOS older than 10.3, a typical 3-button alert is used as the rating prompt. This is mainly for backward-compatibility purpose, since the new built-in rating prompt is preferred and will be used on the majority of iOS devices in the near future, given the high adoption rate of new iOS versions.

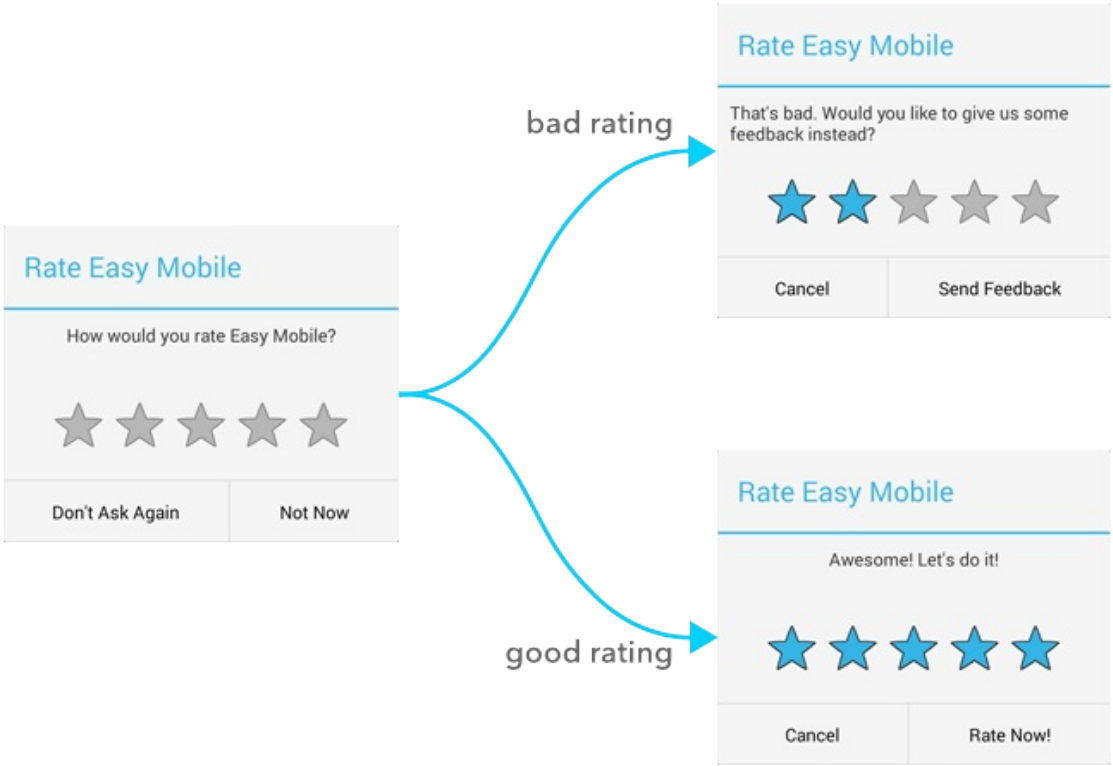


Unlike the built-in dialog, you can customize the title, message and button labels of this alert to suit your needs. The default behavior of this rating prompt is described below.

- *Don't Ask Again*: close and never show this prompt again
- *Remind Me Later*: close this alert
- *Rate Now!*: open the "Write A Review" page of the current app on the App Store, the prompt will never be displayed again

Android

On Android, we built a native, custom alert that employs the RatingBar component to form the rating dialog. The picture below illustrates how this dialog looks and behaves.



The idea is to ask the user how they would rate the app, and the dialog will update itself based on the given rating. You can set a "minimum accepted rating" value, which is the lowest number of stars expected for your app. Any rating lower than this value is considered a bad rating, and vice versa. If the user is giving a good rating, we will take them to the store to do the actual rating and review. Otherwise, we will suggest them to send a feedback to your support email instead. The default behavior of this rating dialog is described below. Again, you can discard this default behavior and implement a custom one if you wish.

- *Don't Ask Again*: close and never show this prompt
- *Not Now/Cancel*: close this prompt
- *Send Feedback*: open email client for the user to send feedback to your email address
- *Rate Now*!: open the product page of the app on the Google Play Store, the prompt will never be displayed again

> On Android or iOS older than 10.3, you can discard the default behavior of the rating dialog and give your own behavior implementation if you wish.

Display Policy

It is up to you to decide when to show the rating prompt in your game to maximize its effectiveness while maintaining the best user experience. Generally, it is advisable to not annoy the user by asking repeatedly or too frequently. For that purpose, the rating request feature provides a few general constraints to help regulate the display of the rating prompt. You are free to configure these values appropriately to suit your needs. These constraints include:

- *Annual Cap*: the maxium number of requests allowed each year
- *Delay After Installation*: the required waiting time (days) since app installation before the first rating request can be made
- *Cooling-Off Period*: the minimum interval (days) required between two consecutive requests

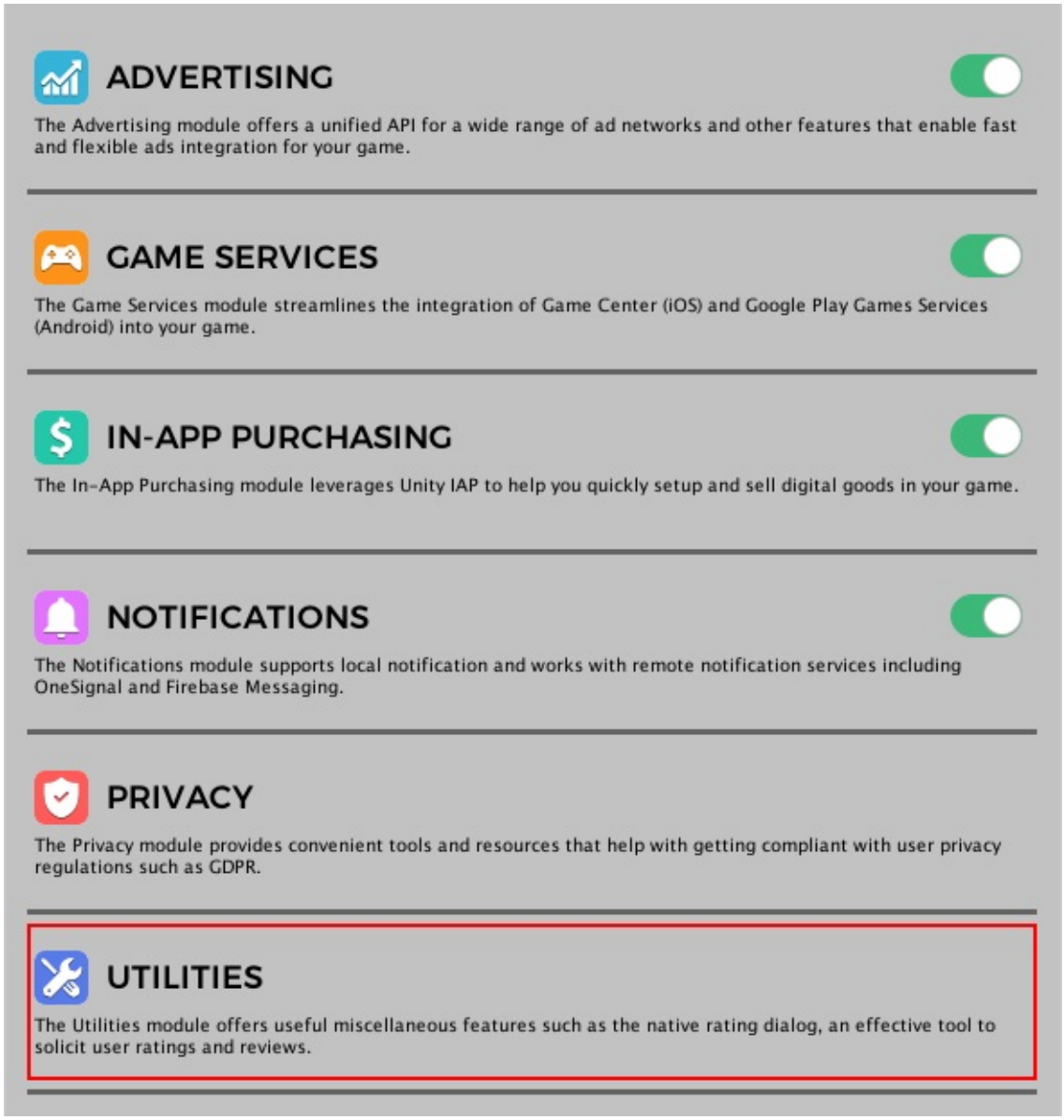
> On iOS 10.3 and newer (where the built-in rating prompt is used), the *Annual Cap* is overwritten by the OS and will always be set to 3.
>
> For the *Delay After Installation* constraint to function properly, it is required that an instance of the EasyMobile prefab is added to the first scene of your game (so that it can record the installation timestamp).

Dialog Content and Localization

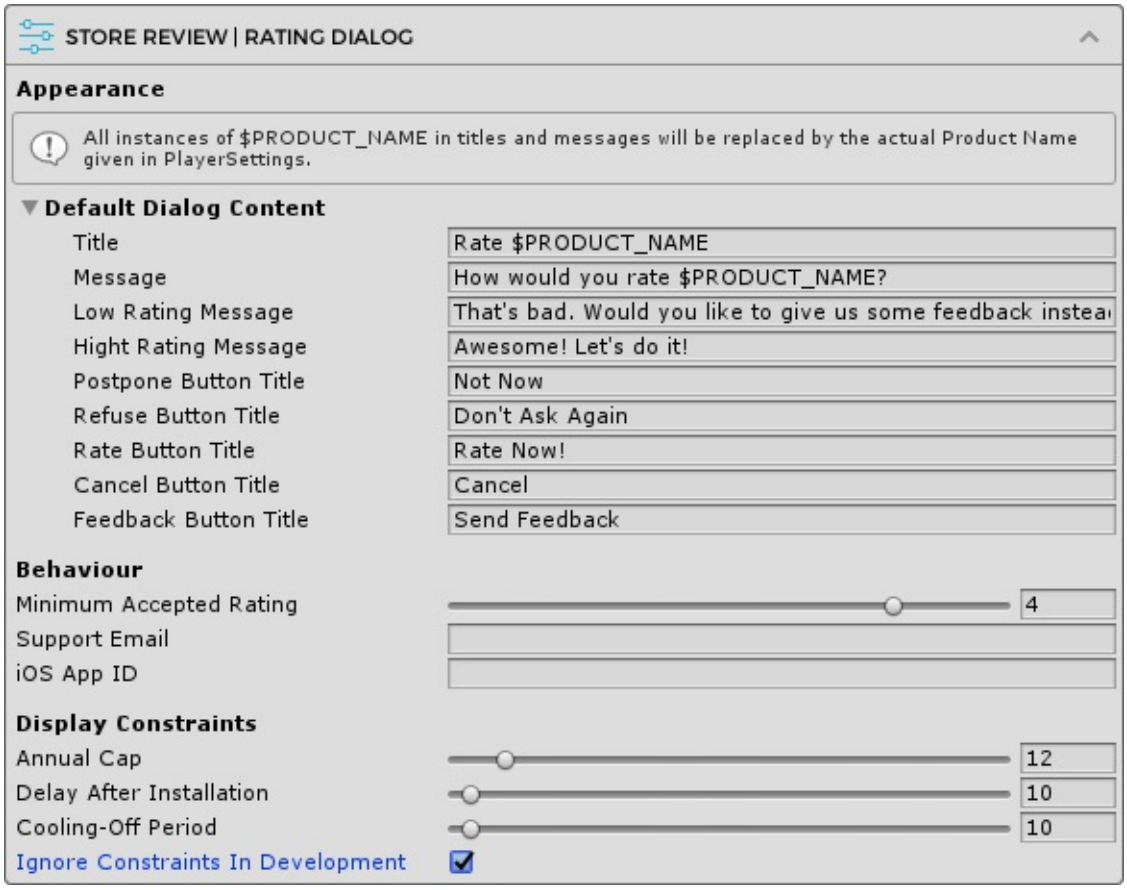
The default content (texts) of the rating dialog can be entered in the settings UI (see the **Configuration** section). This content can be altered in runtime (see the **Scripting** section), so that you can use this feature in conjunction with another localization plugin to fully localize your popup.

Utilities | Store Review: Settings

You can configure the Store Review feature from the Utilities module. Go to *Window* > *Easy Mobile* > *Settings*, then select the Utilities tab to reveal it.



In the **[STORE REVIEW] REQUEST DIALOG CONFIG** section, you can customize the appearance, behavior and display constraints of the rating dialog.



- *Default Dialog Content*: the default texts of the rating dialog used on Android and iOS older than 10.3 (on iOS 10.3 or newer this content is governed by the system)
- *Minimum Accepted Rating*: the lowest number of stars required to be considered as a good rating, you can set it to 0 to disable the feedback feature (accept all ratings); *note that this is only applicable on Android*
- *Support Email*: your email address for receiving feedback
- *iOS App Id*: your app Id on the Apple App Store, this is required to open the review page of the app on iOS older than 10.3
- *Annual Cap*: the maxium number of requests allowed each year
- *Delay After Installation*: the required waiting time (days) since app installation before the first rating request can be made
- *Cooling-Off Period*: the minimum interval (days) required between two consecutive requests
- *Ignore Constraints In Development*: ignore all display constraints so the rating popup can be shown every time in Development builds (unless it was disabled before)

Utilities | Store Review: Scripting

This section provides a guide to work with the Store Review scripting API.

> You can access the Store Review API via the StoreReview class under the EasyMobile namespace.

Making Rating Request

To show the rating dialog using its default content and retain its default behavior, use the *RequestRating* method without any parameter. Note that this method is a no-op if the rating dialog has been disabled, or one of display constraints is not satisfied. You should call this method when it makes sense in the experience flow of your app, to maximize the effectiveness of the request.

> On iOS 10.3 or newer, the actual display of the rating dialog is governed by App Store policy. When your app is still in sandbox/development mode, the dialog is always displayed for testing purpose. However, it won't be shown in an app that you distribute using TestFlight.

```
// Show the rating dialog with default behavior
StoreReview.RequestRating();
```

To check if the rating dialog has been disabled (because the user selected *Don't Ask Again* or already gave a rating):

```
// Check if the rating dialog has been disabled
bool isDisabled = StoreReview.IsRatingRequestDisabled();
```

To get the number of used and remaining requests in the current year:

```
// Get the number of requests used this year
int usedRequests = StoreReview.GetThisYearUsedRequests();

// Get the number of unused requests this year
int unusedRequests = StoreReview.GetThisYearRemainingRequests();
```

To get the timestamp of the last request:

```
// Get the time when the last rating popup is shown
DateTime lastTime = StoreReview.GetLastRequestTimestamp();
```

To check if it's eligible to show the rating dialog (which means it hasn't been disabled and all display constraints are satisfied):

```
// Check if it's eligible to show the rating dialog and then show it
if (StoreReview.CanRequestRating())
{
    StoreReview.RequestRating();
}
```

Making Rating Request with Custom Callback

On Android or iOS older than 10.3, you can discard the default behavior of the rating dialog and provide your own implementation to suit your needs (again, on iOS 10.3 or newer we employ the native rating prompt whose behavior is governed by the system itself). This can be useful in cases when you want to perform additional tasks like recording

the number of users who gave good ratings (maybe for analytics purpose). To do so, simply call the *RequestRating* method passing a callback in which the custom behavior is implemented. This callback takes as input an enum value representing the user action, which you can use to decide whatever action should be taken. Note that you can use the *DisableRatingRequest* method to prevent the rating dialog from being displayed in the future, if the user selects "Don't Ask Again" option. Also note that you can pass a *null* RatingDialogContent object to use the default content, otherwise create a new object as described in the **Localized the Rating Dialog** section above.

> From the analytics point of view, it's worth noting that the rating given in the rating dialog on Android is merely *a suggestion of how the user would rate the app*. There's currently no reliable way to verify if it is the actual rating given on the app stores or not.

```
// Show rating dialog with a callback for custom behavior
// Passing null for the RatingDialogContent parameter to use the default content
StoreReview.RequestRating(null, RatingCallback);

// The rating callback
private void RatingCallback(StoreReview.UserAction action)
{
    switch (action)
    {
        case StoreReview.UserAction.Refuse:
            // Don't ask again. Disable the rating dialog
            // to prevent it from being shown in the future.
            StoreReview.DisableRatingRequest();
            break;
        case StoreReview.UserAction.Postpone:
            // User selects Not Now/Cancel button.
            // The dialog automatically closes.
            break;
        case StoreReview.UserAction.Feedback:
            // Bad rating, user opts to send feedback email.
            break;
        case StoreReview.UserAction.Rate:
            // Good rating, user wants to rate.
            break;
    }
}
```

Localizing Rating Dialog

To localize the content of the rating dialog, simply create a new RatingDialogContent to hold the translated texts (which you may obtain from a standard localization plugin), and pass it to the *RequestRating* method.

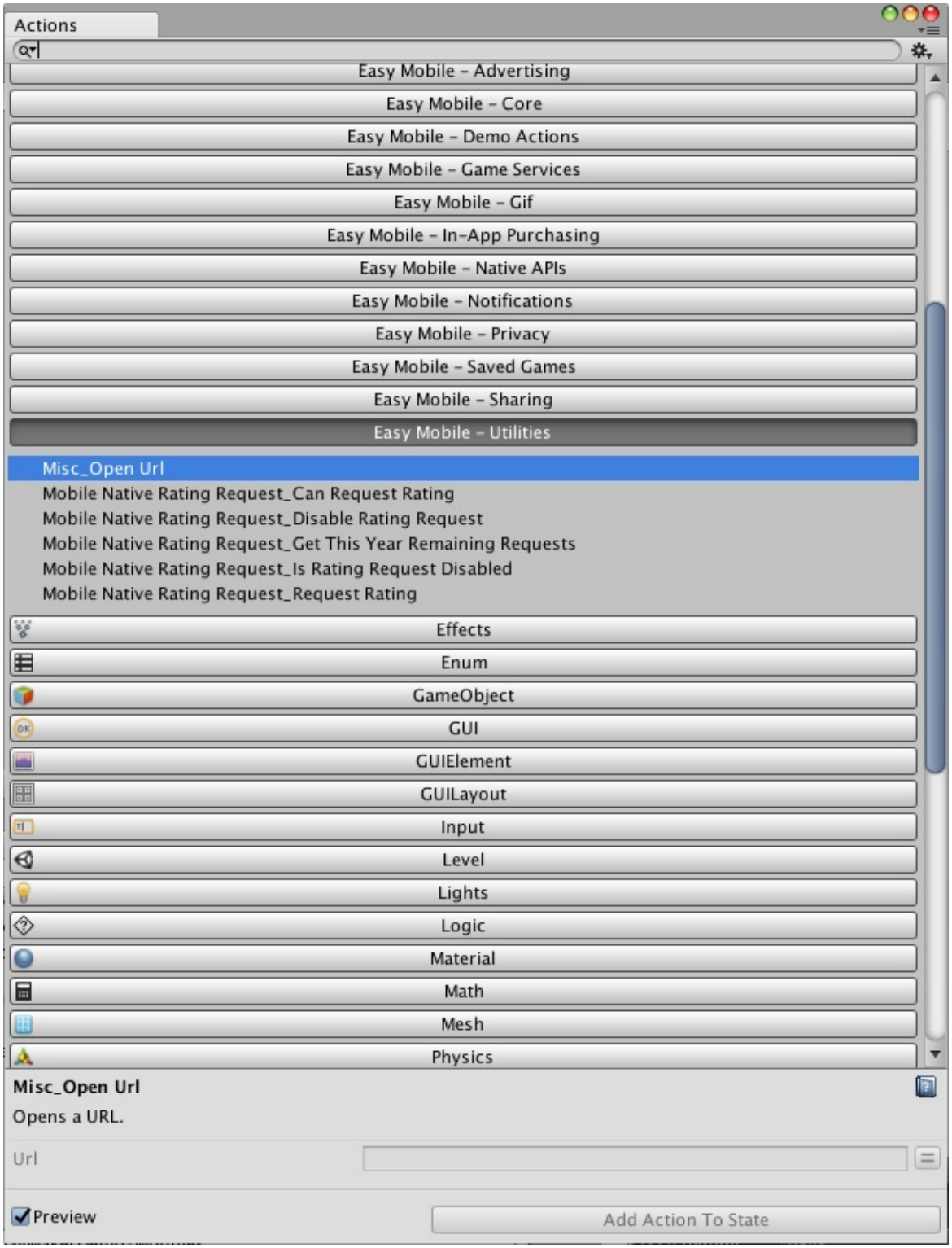
```
// Create a RatingDialogContent object to hold the translated content of the dialog
var localized = new RatingDialogContent(
            YOUR_LOCALIZED_TITLE + RatingDialogContent.PRODUCT_NAME_PLACEHOLDER,
            YOUR_LOCALIZED_MESSAGE + RatingDialogContent.PRODUCT_NAME_PLACEHOLDER + "?",
            YOUR_LOCALIZED_LOW_RATING_MESSAGE,
            YOUR_LOCALIZED_HIGH_RATING_MESSAGE,
            YOUR_LOCALIZED_POSTPONE_BUTTON_LABEL,
            YOUR_LOCALIZED_REFUSE_BUTTON_LABEL,
            YOUR_LOCALIZED_RATE_BUTTON_LABEL,
            YOUR_LOCALIZED_CANCEL_BUTTON_LABEL,
            YOUR_LOCALIZED_FEEDBACK_BUTTON_LABEL
        );

// Show the rating popup with the localized texts
StoreReview.RequestRating(localized);
```

> Any instance of RatingDialogContent.PRODUCT_NAME_PLACEHOLDER (literal value "$PRODUCT_NAME") will be automatically replaced by the actual product name (given in PlayerSettings) by the *RequestRating* method.

Utilities: PlayMaker Actions

The PlayMaker actions of the Utilities module are group in the category *Easy Mobile - Utilities* in the PlayMaker's Action Browser.

> Please refer to the UtilitiesDemo_PlayMaker scene in folder *Assets/EasyMobile/Demo/PlayMakerDemo/Modules* for an example on how these actions can be used.



Release Notes

Version 2.1.0

New Features

- **Advertising module:**
  - Improved Automatic Ad Loading feature, now it can either load only default ads, or load ads at all placements defined in the module settings, which means you can completely forget about manual ad loading now.
- **GIF module:**
  - *AnimatedClip* can now automatically release the associated RenderTexture objects while being garbage collected, which is useful to avoid memory leaks in case you forget to call *Dispose*.

Changes

- **Advertising module:**
  - The *Advertising.IsAutoLoadDefaultAds*, *Advertising.EnableAutoLoadDefaultAds* and *Advertising.SetAutoLoadDefaultAds* are now deprecated and replaced by the *Advertising.AutoAdLoadingMode* property.

Bug Fixes

- **Notifications module:**
  - Fixed a bug related to applying consent during initializing OneSignal.
  - Fixed irrelevant warning message when generating constants without having any user categories defined.

Version 2.0.0

New Features

- **Advertising module:**
  - Added support for Facebook Audience Network.
  - Added support for ironSource.
  - Added support for Mopub Ads.
  - Added support for Tapjoy.
  - Added support for showing ads at multiple ad placements in an app.
  - Added consent support (GDPR compliance).
- **Game Services module:**
  - Added popup gravity setting for Google Play Games platform.
- **In-App Purchasing module:**
  - Added a method to get subscription product information using Unity IAP's SubscriptionManager class.
  - Added support for Apple's Ask-To-Buy feature.
  - Added support for Apple's Promotional Purchases.
  - Added an option to enable Amazon store sandbox testing.
  - Added a method to get Amazon user ID.

- **Notifications module:**
  - Added consent support (GDPR compliance).
  - Android local notifications are now expandable (Android 4.1 or newer only).
- **Privacy module:** brand new module introduced in version 2.0.0 that offers convenient tools and resources to help with getting GDPR-compliant including:
  - A comprehensive, flexible consent management system.
  - A native, multi-purpose, customizable consent dialog that can serve as a common interface for collecting user consent for all services in an app.
  - Easy-to-use graphical composer for editing consent dialog content.
  - A location checking tool, for detecting if the current device is in the European Economic Region (EEA), which is regulated by GDPR.
- **Editor:**
  - Brand new redesigned settings UI that is cleaner, friendlier and prettier :)
  - Brand new redesigned settings UI, once again. (sorry we got too excited, but the new UI looks really cool, you'll like it :D)
- **PlayMaker support:**
  - Added new actions for initializing Easy Mobile runtime (for replacing the EasyMobile prefab, see **Changes**).
  - Added new actions for the Privacy module including granting and revoking consent, displaying consent dialog, etc.
  - Added a PlayMaker demo scene for the Privacy module.
  - Added an action for the RewardedAdSkipped event.
  - Added actions for the LocalNotificationOpened and RemoteNotificationOpened events.

Changes

- **Editor:**
  - The long-time EasyMobile prefab is now officially deprecated. Easy Mobile initialization can now be done from script using the RuntimeManager class (see Initializing).
  - Upgraded to Google Play Services Resolver version 1.2.89.
  - The minimum required version of Unity is now 5.5.5f1.

Bug Fixes

- **Notifications module:**
  - Fixed a bug causing the local repeat notifications in the Notifications demo scene to not function on Android devices due to missing category definition in the module settings.
- **Utilities module:**
  - Fixed a bug causing the rating dialog to be dismissible by tapping outside of the popup and prevent subsequent dialogs from being shown.
- **PlayMaker support:**
  - Fixed the issue that Saved Games actions don't function correctly with global variables.

Version 1.3.0

New Features

- **Notifications module:**
  - Added support for Firebase Cloud Messaging as a remote notification service.
  - Added methods to get and set application icon badge number on iOS.

Changes

- **Editor:**
  - Build managing script now uses IPreProcessBuildWithReport and IPostProcessBuildWithReport interfaces on Unity 2018.1.0 and newer instead of the deprecated IPreProcessBuild and IPostProcessBuild interfaces.

Version 1.2.1

Changes

- **Game Services module:**
  - Improved PlayMaker actions for Saved Games API.
  - Improved PlayMaker demo scene for Saved Games feature.
- **Editor:**
  - Updated to Google Play Services Resolver 1.2.69.

Version 1.2.0p1

Bug Fixes

- **Notifications module:**
  - Fixed a bug that may cause local notifications to not be scheduled properly if the trigger date is not specified in local timezone.

Version 1.2.0

This is a major update in which we're adding brand new features, revamping the whole API as well as fixing some known issues. We also renamed some modules and revised related wording to better present the plugin content. Most importantly, with this update we're restructuring the Easy Mobile product line. Specifically:

- The existing Easy Mobile version will now be Easy Mobile Pro, which is the premium version and contains all the available features of the plugin. Current Easy Mobile users therefore will own the Pro version automatically.
- A new version named Easy Mobile Basic will be introduced at a lower price than the Pro version. It will contain all the core features but without a few advanced ones such as GIF and Saved Games. For details about feature differences between Pro and Basic versions please see the Feature Comparison table.
- The Easy Mobile: GIF Tools version will be deprecated.

New Features

- **Game Services module:**
  - Added the brand new feature Saved Games, which allow easy synchronization of game data to cloud services including iCloud (iOS) and Google Drive (Android).
  - Allows specifying an optional the Web Client ID when setup Google Play Games.
  - New PlayMaker actions for Saved Games feature.
- **Notifications module:**
  - Added support for fully-customizable local notifications.

- Fully compatible with Android 8.0 notification channels and channel groups.
- New PlayMaker actions for local notifications.

- **Native APIs module:**
  - Native UI feature: added a method to check whether an alert is being displayed.

Changes

- **Advertising module:**
  - Removed *IsShowingBannerAd* and *GetActiveBannerAdNetworks* methods because there's currently no reliable way to obtain this information that would work consistently across all the supported networks.
- **Scripting:**
  - Introduced much of code refactoring to enhance stability, maintainability, scalability and readability.
  - Renamed major classes to make the API more intuitive; specifically, each module/feature now has a main class with the same name, where its API can be accessed.
  - Removed the feature that automatically disables debug logs in production builds.
- **Editor:**
  - Upgraded to Google Play Services Resolver version 1.2.64.0.

Bug Fixes

- **GIF module:**
  - Fixed Giphy upload error "401 Unauthorized".

Version 1.1.5p2

New Features

- **Advertising module:**
  - AdManager class now exposes a RewardedAdSkipped event, which is raised when a rewarded ad was closed before finishing.

Bug Fixes

- **Editor:**
  - Replaced the old Chartboost SDK download URL with a new working one.

Version 1.1.5p1

Bug Fixes

- **PlayMaker Actions:**
  - Fixed a minor error in the script for MobileNativeShare_CaptureScreenshot action.

Version 1.1.5

New Features

- **Editor:**
  - Incorporated the Google Play Services Resolver for Unity plugin for Android dependencies management.
  - Added the **Import Play Services Resolver** item to Easy Mobile menu for manual import of this resolver if needed (normally it will be imported automatically upon importing Easy Mobile)

Changes

- **Editor:**
  - Easy Mobile's native code is now statically included in folder Assets/EasyMobile/Plugins folder, rather than being imported automatically from script into Assets/Plugins folder as before. This enhances the plugin's robustness as it prevents build errors due to unintended removal of plugin files in the Assets/Plugins folder.
  - Removed the **Reimport Native Package** item from Easy Mobile menu (as a result of the above change).

Bug Fixes

- **Native Sharing module:**
  - Fixed a bug causing image sharing to fail on Android 7 (Nougat) and above. Image sharing on these platforms now uses FileProvider to comply with the new Android security requirements.

Notes

** Since the plugin structure changes quite a lot in this version, you need to do some cleanup before importing the new plugin. Please see the **Upgrade Guide** section for more details.*

Version 1.1.4b

Changes

- **GIF module:**
  - Optimized memory usage when exporting GIF.

Version 1.1.4a

Bug Fixes

- **In-App Purchasing module:**
  - Updated editor scripts to be compatible with UnityIAP version 1.14.0.

Notes

** If you're upgrading Easy Mobile from an existing project that uses IAP module, you need to upgrade (re-import) UnityIAP package too. Please see the **Upgrade Guide** section for more details.*

Version 1.1.4

New Features

- **Game Service module:**
  - Added a new method to show the UI of a specific leaderboard in an (optional) time scope.
- **In-App Purchasing module:**
  - Added a new method to get all IAP products created in the module settings.
- **Utilities module - Rating Request feature:**
  - Added new display constraints: delay after installation & cooling-off period.
  - Added an option to ignore display constraints while in development mode.
  - Added new methods to get the timestamp of the last request, the number of requests used in the current year, etc.
  - Added the ability to update the dialog content in runtime for localization purposes (see the user guide for details).
- **Editor:**
  - [Android] leaderboard & achievement IDs are now sorted alphabetically in the settings UI.
  - We've now got a little cute About window where you can quickly find out the version of your Easy Mobile :)

Changes

- **Game Service module:**
  - *UserAuthenticated* event is now officially removed.

Version 1.1.3

New Features

- **Introducing brand new PlayMaker actions!**
  - Easy Mobile is now compatible with PlayMaker, starting with nearly 100 custom actions covering all modules!
- **Utilities module:**
  - Added new method GetAnnualRequestLimit to get the annual cap of the rating request popup from script

Changes

- **Game Service module:**
  - Added optional callback to ReportScore, RevealAchievement, UnlockAchievement & ReportAchievementProgress to acknowledge if the operation succeeds or not

Bug Fixes

- **Editor:**
  - Fixed a bug on Unity 5.6+ causing EasyMobile prefab instance to not be detected properly if the containing scene is not active -> a false **"Easy Mobile Instance Not Found"** alert is shown before building

---

Version 1.1.2

Changes

- **In-App Purchasing module:**
  - Updated the receipt validation method to handle cases when the input receipt is null or empty.

Version 1.1.1

New Features

- **In-App Purchasing module:**
  - Added new methods to read receipts from Apple stores and Google Play store
  - Added a new method to refresh Apple App Receipt

Version 1.1.0

This is a major release with many new features and improvements!

New Features

- **Introducing brand new module GIF!**
  - Low overhead screen/camera recorder
  - Built-in players for playback of recorded clips
  - High performance, mobile-friendly GIF image generator
  - Giphy upload API for sharing GIF images to social networks
- **Native Sharing module:**
  - Added *ShareText* and *ShareURL* methods to *MobileNativeShare* class
- **Editor:**
  - Added a new context menu for creating EasyMobile instance and other built-in objects in the Hierarchy window
  - Added new item **Reimport Native Package** to Easy Mobile menu
  - [Unity 5.6+] Added a warning popup which is shown when an iOS or Android build starts while no EasyMobile instance was added to any scene

---

Version 1.0.4

New Features

- **Game Service module:**
  - Added *SignOut* method to *GameServiceManager* class.

Version 1.0.3

This update introduces important improvements and bug fixes.

New Features

- **Advertising module:**
  - AdMob rewarded ad is now supported
  - Added support for new ad network: **AdColony**

Changes

- **Advertising module:**
  - Ad events are now raised from main thread when using AdMob
  - *RewardedAdCompleted* event is now raised after the ad is closed, to ensure a consistent behavior across different ad networks

Bug Fixes

- **Native Sharing module:**
  - Fixed a potential memory leak issue caused by the *SaveScreenshot* method of the *MobileNativeShare* class

---

Version 1.0.2

New Features

- **Introducing whole new module Utilities:**
  - The first feature of this module is **Rating Request**, an effective way to ask for rating using a native and highly customizable "rate my app" popup.
- **Game Service module:**
  - Updated *GameServiceManager* class, introducing new events *UserLoginSucceeded* and *UserLoginFailed*; _UserAuthenticated _event is now obsolete.

Version 1.0.1

Changes

- **Game Service module:**
  - Updated scripts to be compatible with version 0.9.37 of the Google Play Games plugin for Unity.

Version 1.0.0

First release.

Upgrade Guide

This section describes the required actions you may need to take when upgrading to a certain version of Easy Mobile. Please visit this place before upgrading Easy Mobile to avoid unnecessary issues.

Upgrading to version 2.0.0

- Version 2.0.0 is a major update which introduces lots of changes and improvements. To make the product better, we had to make some necessary changes that break backward compatibility. Therefore we recommend a clean upgrade if you are moving from an older version to Easy Mobile Basic 2.0.0 or newer. That means you should remove the *Assets/EasyMobile* folder completely before importing the new version and setup the plugin again (re-apply previous settings in the Settings UI).
- Version 2.0.0 also removes the longtime EasyMobile prefab. Now you no longer have to add it to the first scene of your app. Instead you would call the *RuntimeManager.Init* method (see Using Easy Mobile > Initializing).

Upgrading to version 1.2.0

Version 1.2.0 is a major update in which Easy Mobile has been renamed to Easy Mobile Pro and lots of improvements and modifications were introduced, most notably API changes. If you're upgrading from an older version to 1.2.0, we strongly recommend removing the old version completely before importing the new one to avoid potential issues. Please follow these steps:

- Backup the Assets/EasyMobile/Resources and Assets/EasyMobile/Generated folders and save them somewhere safe.
- Remove the whole Assets/EasyMobile folder.
- Remove the file/folder named com.sglib.easymobile.easy-mobile-1.0.2 in folder Assets/Plugins/Android.
- Import Easy Mobile Pro 1.2.0.
- Copy the backed up Resources and Generated folders back to Assets/EasyMobile folders.
- Go to menu Assets > Play Services Resolver > Android Resolver > Force Resolve.
- If you're using Game Services module on Android, run Setup Google Play Games once again in the settings UI.
- Optionally update your scripts to fix warnings due to old classes being deprecated (they still function normally, we're just introducing new classes with different names to make the API more intuitive).

Upgrading to version 1.1.5 or newer

Since version 1.1.5, Easy Mobile incorporates the Google Play Services Resolver for Unity plugin for Android dependencies management, as well as moves all native code into the Assets/EasyMobile/Plugins folder. If you're upgrading from an older version to version 1.1.5 or newer, please remove the following files before importing the new package to avoid potential issues:

- Assets/Plugins/Android/easy-mobile.aar.
- Assets/Plugins/Android/libs/armeabi-v7a/libeasymobile.so
- Assets/Plugins/Android/libs/x86/libeasymobile.so
- Assets/Plugins/iOS/libEasyMobile.a

Upgrading to version 1.1.4a or newer

Since version 1.14.0, the UnityIAP package has made changes to its API that cause some conflicts with Easy Mobile editor scripts. We addressed this problem in version 1.1.4a. If you're upgrading from an older version to 1.1.4a, and your project uses the In-App Purchasing module, you need to upgrade (re-import) the UnityIAP package to version 1.14.0 or newer to avoid incompatibility issues.

Upgrading to version 1.1.0 or newer

If you're upgrading from an older version to version 1.1.0 or newer, you'll need to:

1. Remove the EasyMobile/Demo folder
2. Remove the EasyMobile/Script folder
3. Import the new version

Troubleshooting

This section describes known issues and common solutions for them.

App crashes when using Google Play Game Plugin with Unity 2018.2.0 or newer.

Symptoms

- You are using the Google Play Games Plugin for Unity (GPGS) version 0.9.50 or older (0.9.50 is the latest version as of this writing, this may or may not apply to future versions).
- You are building the app with Unity 2018.2.0 or newer.
- The app crash while attempting to login to Google Play Games.
- The crashlog includes this line "Application ID (xxxxxxxxxxxx) must be a numeric value."

Resolution

This is a known issue of GPGS. Please follow this instruction to solve it.

Google Play Services Resolver gets stuck after importing OneSignal SDK

Symptoms

- Your project is using Google Play Services Resolver 1.2.88 or newer.
- The resolver worked correctly before importing OneSignal plugin, but gets stuck after importing it.

Resolution

Please follow OneSignal instruction on solving this.

Can't sign in to Google Play Games with development builds created by Gradle build system

Symptoms

- You have setup your app to work with Google Play Games via the Game Services module.
- If you create a non-development build using Gradle and try to sign in to Google Play Games in the app, the authentication succeeds.
- If you create a non-development or development build using the Unity's internal build system, the authentication also succeeds.
- If you create a development build using the Gradle build system, the authentication always fails.

Resolution

The root cause of the problem is the default gradle config of Unity doesn't use the keystore specified in Play Settings (in other words, the debug keystore is always used), causing Google Play Games authentication to fail. To address this you need to:

1. Enable the 'Custom Gradle Template' option in Player Settings > Android Tab > Publishing Settings.
2. Add a **SIGNCONFIG** line to the custom gradle config (normally it is at Assets/Plugins/Android/mainTemplate.gradle) at the debug buildTypes config like below, so that the release keystore can be used in development builds.

```
**SIGN**
buildTypes {
    debug {
        minifyEnabled **MINIFY_DEBUG**
        useProguard **PROGUARD_DEBUG**
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-unity.txt'**USER_PROGUARD**
                jniDebuggable true
        **SIGNCONFIG**   <--- ADD THIS LINE
    }
    release {
        minifyEnabled **MINIFY_RELEASE**
        useProguard **PROGUARD_RELEASE**
         proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-unity.txt'**USER_PROGUARD**
         **SIGNCONFIG**
     }
}
```

Easy Mobile settings are not saved

Symptoms

You open the Easy Mobile global settings interface (Window > Easy Mobile > Settings) and made some changes, then close the interface. When you re-open it, the changes are gone instead of being saved.

Resolution

Normally, this is because there are more than one Inspector tabs being opened at the same time. So double check that and make sure you only have one Inspector tab in which the Easy Mobile settings interface is shown.

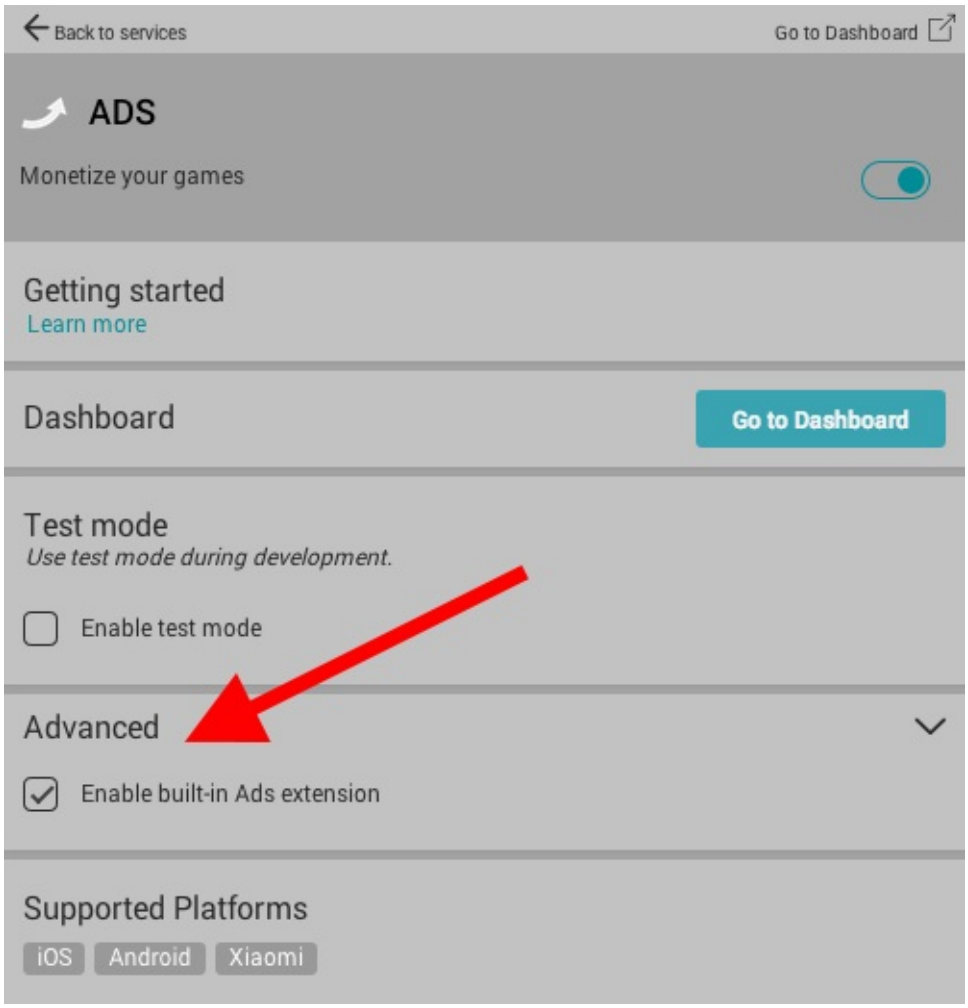
UnityAds Service is enabled, but shown as unavailable in Advertising module settings

Symptoms

You have enabled UnityAds from the Services tab in Unity, but the Advertising module still shows that it is unavailable.

Resolution

1. First of all make sure that your current active platform is iOS or Android, since UnityAds is not available on other platforms. To switch platform go to File > Build Settings, then select the target platform and hit the Switch Platform button.
2. In UnityAds settings panel (Window > Services > Ads), make sure the "Enable built-in Ads extension" option in the Advanced section is checked.



If UnityAds is still not detected as available, perform the following steps:

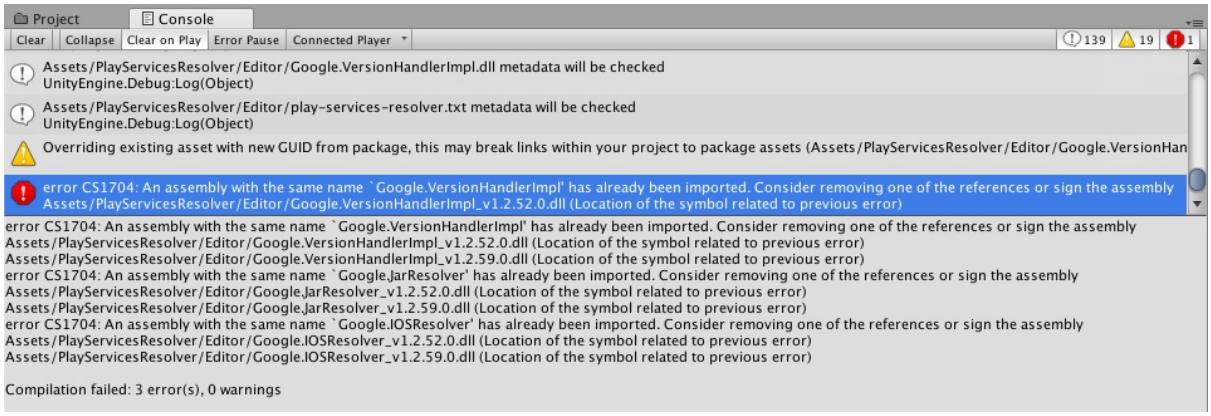
1. Disable UnityAds service in the Services tab.
2. Exit Unity.
3. Go to the project folder, locate the Project Settings folder and open it.
4. Delete the UnityConnectSettings.asset file and the UnityAdsSettings.asset file (if any).
5. Open Unity and enable UnityAds service again. It should be detected now.

Errors upon importing Easy Mobile due to conflicting versions of the Google Play Services Resolver plugin

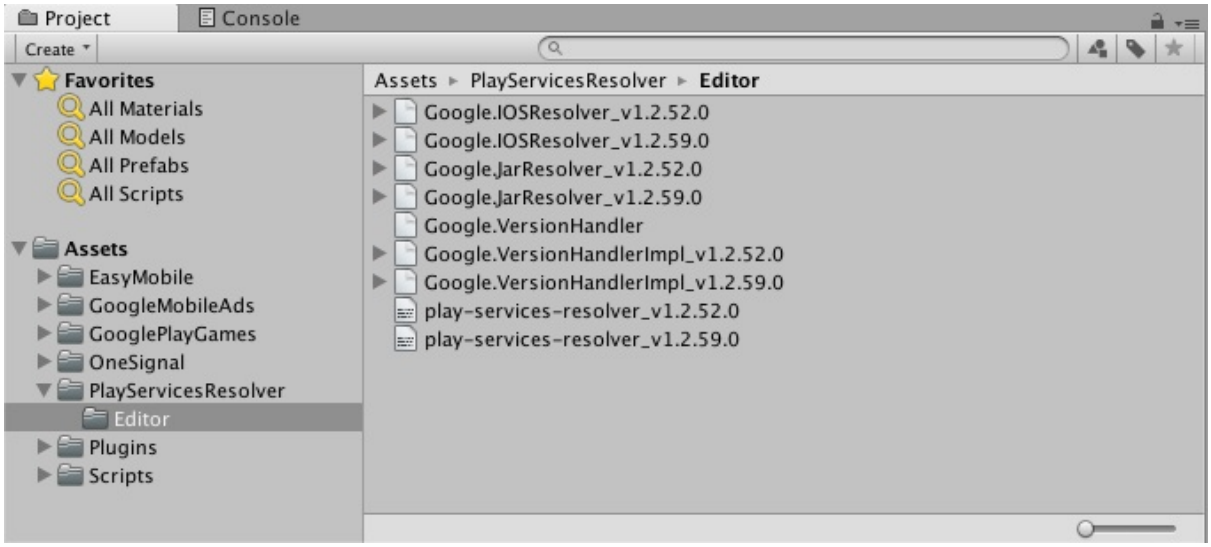
Symptoms

After importing/upgrading Easy Mobile in a project that already contains the Google Play Services Resolver plugin (the folder Assets/PlayServicesResolver exists):

- You get an error in the console starting with **"An assembly with the same name 'Google.VersionHandlerImpl' has already been imported..."**.



- The Assets/PlayServicesResolver/Editor contains multiple files with same names but different versions.



Resolution

Use menu *Assets* > *PlayServicesResolver* > *Version Handler* > *Update*. The Version Handler of the Google Play Services Resolver will automatically resolve the conflict, pick the appropriate version and remove the redundant files.