Introduction

PlayHaven is a real-time mobile game marketing platform to help you take control of the business of your games. Acquire, retain, re-engage and monetize your players with the help of PlayHaven's powerful marketing platform. Integrate once and embrace the flexibility of the web as you build, schedule, deploy and analyze your in-game promotions and monetization in real-time through PlayHaven's easy-to-use, web-based dashboard.

An API token and secret are required to use this SDK. These identify your game to PlayHaven and prevent others from making requests to the API on your behalf. To get a token and secret, please visit the PlayHaven developer dashboard at:

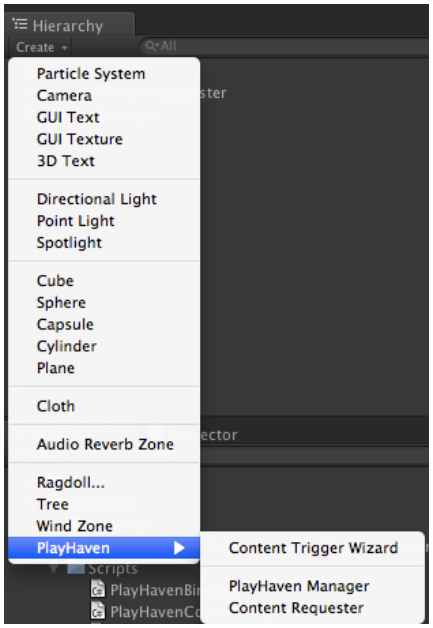
https://dashboard.playhaven.com

The PlayHaven Unity SDK is a Unity3 in-editor extension and iOS/Android plug-in to make it possible to efficiently integrate the PlayHaven iOS and Android SDKs into your Unity games with minimal effort and code production.

Getting Started Quickly

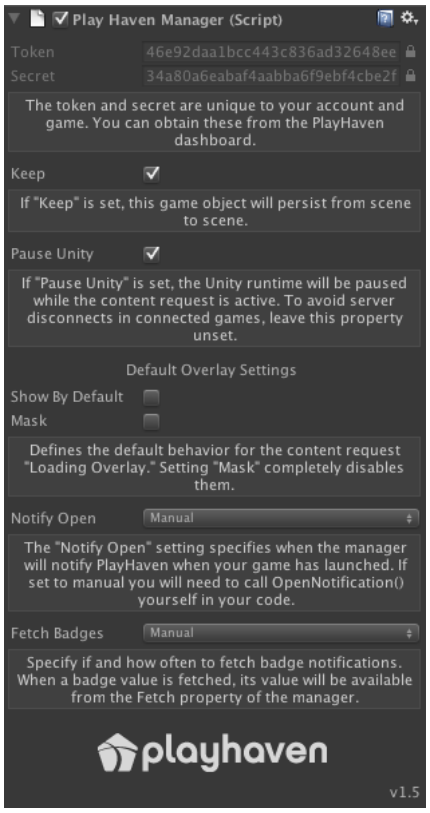
To get started using PlayHaven, you must first be a registered developer and create a new game in the developer dashboard. You will then have the required token and secret strings necessary in order to perform communications with the PlayHaven servers. All instructions contained in this document assume that you have done this and have created placements and rewards in the dashboard. Navigate your browser to https://dashboard.playhaven.com to get started using the PlayHaven system.

When your game project is open within the Unity3 editor you must import the PlayHaven SDK package into your project. You can obtain this package in one of two ways: (1) from the Unity Asset Store as a free download or (2) from the PlayHaven dashboard. Importing this package will result in over a dozen imported files and several directories (if they don't exist). However, you will only need to become minimally familiar with two of these files: `PlayHavenManager` and `PlayHavenContentRequester`. Both of these scripts are located in the `Plugins/PlayHaven/Scripts` directory.



Communications with the PlayHaven servers is facilitated through the `PlayHavenManager`. Your game will need this manager attached to a game object in your scene. To add one, activate the "Create" menu in the Hierarchy pane in your opened scene. Select "PlayHaven > PlayHaven Manager" from the drop down panel. This will automatically create a game object in your scene called PlayHavenManager with the `PlayHavenManager` script attached to it.

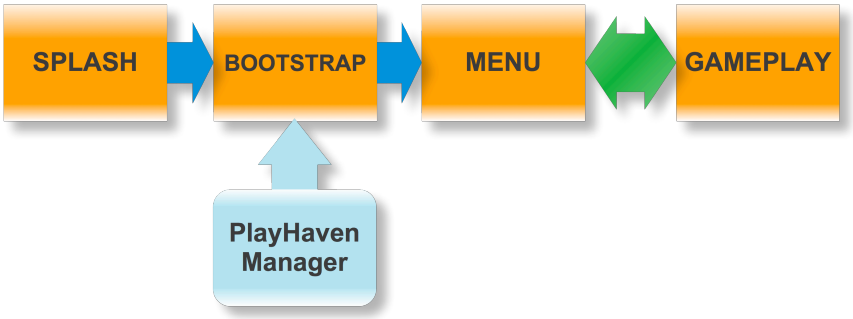
There are several configurable fields in the PlayHaven Manager component.



The first fields that you should fill out are the **Token** and **Secret** fields; these values give your game permission to use the PlayHaven API and ensure that only your game is reporting data on your behalf. Once you copy/paste these values from the PlayHaven developer dashboard, click the lock buttons to the right of these fields to ensure they aren't accidentally changed during development.

The **Keep** flag indicates if the game object that the manager is attached to should persist from level to level changes (i.e. "don't destroy on load"). This is *on* by default. So that you do not have to deal with adding multiple managers to your project (and therefore having to keep the Token and Secret values in sync) it is recommended that you keep this flag on. Then, as different scenes are loaded the manager is still available in each of those scenes without any additional work on your part. The diagram below shows an example level flow pattern that is commonly used to properly handle objects

that are persisted across scenes.



When requests to PlayHaven are made, an overlay view can be displayed on top of your game. This overlay can be a visual queue to indicate that some activity is happening and it can also be used to keep the player from interacting with the game while the communication to PlayHaven is occurring. This behavior is generally specified per content request, but the default behavior can be set with the manager by setting the **Default Overlay Settings**. If **Show By Default** is on, then the overlay will be shown during the request when the content request doesn't specifically specify the setting. If **Mask** is set, then an overlay will never be shown while the request is in progress, regardless of what is requested. If you regularly plan to specify placements that will sometimes be configured to not return anything, then you will likely want to mask overlays during the request.

The **Notify Open** setting specifies when PlayHaven is notified when your game is launched. The default setting is *Awake*, though you can change it to *Start* or *Manual*. If you change it to Manual you will need to call

```
PlayHavenManager.instance.OpenNotification()
```

somewhere in your codebase.

Finally, the **Fetch Badges** setting specifies when the badge value is requested from PlayHaven. If you want to take advantage of badges, then you need to ensure these are not requested until after the **Notify Open** request has been made. If you do not want to use badges you can set this value to *Disabled*. More information about badges will be explained below.

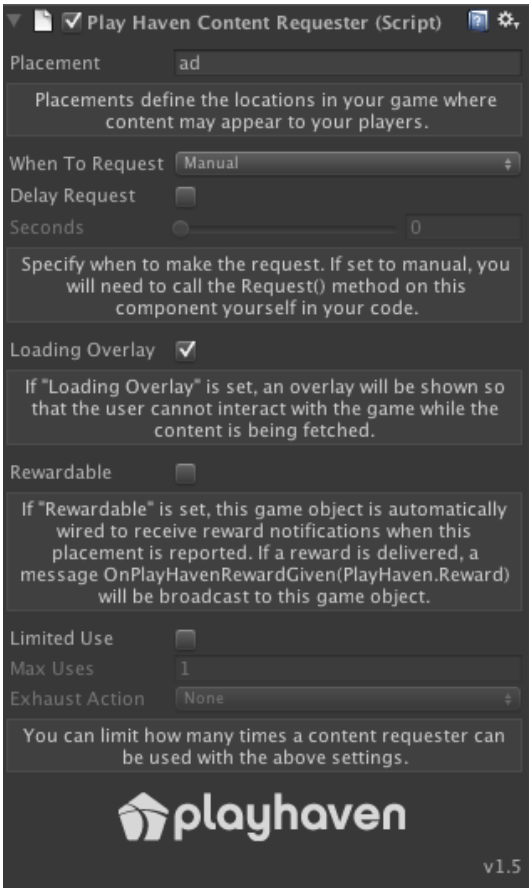
At this point your game is configured to take advantage of all of the features of the PlayHaven platform. With only the PlayHaven Manager in your scene you can invoke the "More Games" feature in your game's GUI by calling the

```
PlayHavenManager.instance.ShowCrossPromotionWidget()
```

method.

However, if you want to display advertisements and provide rewards to your players then you will need to utilize the PlayHaven Content Requester object. Any number of these content requesters can be located in your scene. They can be invoked automatically upon scene load or manually (perhaps due to the interaction of the player with a trigger!). To add a content requester to your scene, activate the “Create” menu in the Hierarchy pane in your opened scene. Select “PlayHaven > Content Requester” from the drop down panel. This will automatically create a game object in your scene called PlayHavenContentRequester with the `PlayHavenContentRequester` script attached to it.

The figure to the right shows a newly created content requester. Every requester needs to have a **Placement** value set. This is one of the placements that you have created in the PlayHaven web dashboard; if you have not yet created any placements, create some now! Some default suggestions are available to you on the dashboard.



By default, a content request is made *Manual*ly. This is the default value because you can have as many content requester objects in your scene as you need. Practically, you’d only want at most one of them to invoke itself automatically in *Start* or *Awake* (for example, to show advertisements when the scene loads). For example, if you created a placement called “ads” in the PlayHaven dashboard and configured that placement to show advertisements, you configure a content requester with a Placement value of “ads” and **When To Request** set to *Awake* or *Start*. But since most of your content requesters will be set to *Manual*, then you will need to invoke the request yourself somewhere in your code. This can be done in one of two ways when using a Content Requester component:

1. Send a parameterless message `RequestPlayHavenContent` to the game object with the content requester component, or
2. Execute the method `Request()` on the component directly.

You can optionally configure the component to perform the actual call to the PlayHaven system after a delay. Enable this feature by checking the “Delay Request” checkbox and setting the slider value to the desired delay value (in seconds).

Which way you do it is totally up to you, depending on what coding style or game architecture you use.

For example, if you want to have your players presented with PlayHaven-configured content when they sit on a chair in one of your game's taverns, then you could configure a content requester with the placement "chair" and in the code that invokes the logic to animate the player sitting in the chair and send the following message to the content requester when the animation completes:

```
contentRequesterGameObject.SendMessage("RequestPlayHavenContent");
```

You can even reward your players with content and currency that you have configured in the PlayHaven dashboard! If your placement is configured to possibly provide a reward, then you should set the **Rewardable** flag to *yes*. You will then need to attach a custom script to the game object (or a game object that is a child of) that holds the Content Requester component. This custom script needs to implement a method with the following signature:

```
void OnPlayHavenRewardGiven(PlayHaven.Reward reward)
```

The `Reward` object contains the *name* and *quantity* value that you would have previously set for the reward in the PlayHaven dashboard. With this data you can implement whatever means you require for your reward system.

For example:

```
void OnPlayHavenRewardGiven(PlayHaven.Reward reward)
{
      // implementation to handle the delivery of the reward
      // to the player of your game
      // ...
      Debug.Log("Reward! "+reward.name);
}
```

Building the Player

Whether you are building a player for iOS or Android, there is minimal action necessary when building your game with the PlayHaven Unity SDK. The plugin takes advantage of the procedure recommended by Unity by placing plugin assets in the Plugins/iOS and Plugins/Android directories; Unity automatically knows how to associate the files in these directories with the appropriate player build.

There is one manual action that you should do, however. Notice that there is an asset `PlayHavenIntegrationSkin.guiskin` in the `Assets/Plugins/PlayHaven/Resources` folder. This skin is used for in-editor-only display of content units, useful in assisting with in-editor integration testing. Since this skin uses image assets that you most definitely will not want to be included in your production builds, you will want to

relocate this skin to a non-`Resources` directory for a production build (and, unfortunately, move it back when the build is done being exported).

Fortunately, this action can be automated, but to do so you will need to create a special editor script that performs your builds. Below is an example; this example specifically shows the skin being moved out of Resources before the build and placed back afterwards.

```
[MenuItem("Build/iOS Player")]
static void BuildForIDevice()
{
      string[] scenes = new string[]
      {
            "Assets/scene.unity"
      };

      // Temporarily move the PlayHavenIntegrationSkin out of Resources
      AssetDatabase.MoveAsset("Assets/Plugins/PlayHaven/Resources/
PlayHavenIntegrationSkin.guiskin", "Assets/Plugins/PlayHaven/Integration/
PlayHavenIntegrationSkin.guiskin");
      AssetDatabase.Refresh();

      // Build
      BuildPipeline.BuildPlayer(scenes, "Player-iOS", BuildTarget.iPhone,
BuildOptions.None);

      // Put the PlayHavenIntegrationSkin back into Resources
      AssetDatabase.MoveAsset("Assets/Plugins/PlayHaven/Integration/
PlayHavenIntegrationSkin.guiskin", "Assets/Plugins/PlayHaven/Resources/
PlayHavenIntegrationSkin.guiskin");
      AssetDatabase.Refresh();
}
```

Advanced Integration

As described in the previous section, you won't need very much custom code at all to integrate PlayHaven into your game and in some cases you won't need any at all. However, it is possible to invoke the capabilities of the PlayHaven Unity Extension by invoking the manager's methods on your own as well as listen to various message handlers (including errors).

When interacting with PlayHaven manually, you only need to be concerned with the `PlayHavenManager` behaviour. The `PlayHavenContentRequester` component serves as a good example on how to use the manager to request content (which is what you'd be doing almost exclusively).

PlayHaven Manager API

This section describes the methods and event handlers that are available with the PlayHavenManager. They are listed in alphabetical order.

Badge

```
public string Badge { get; }
```

This is the accessor for the current badge value. The badge value will be set after a `BadgeRequest()` has been made.

BadgeRequest

```
public void BadgeRequest()
```

To indicate to your players that new content is available, you can request a badge from the system using this method. When successful, an `OnBadgeUpdate(string badge)` event will be fired. The current badge value can also be obtained with the `Badge` accessor at any time.

ClearBadge

```
public void ClearBadge()
```

Call this method to clear the badge value.

ContentRequest

```
public void ContentRequest(string placement)
public void ContentRequest(string placement, bool
showsOverlayImmediately)
public void ContentRequest(string placement, bool
showsOverlayImmediately, bool pauseUnity)
```

This method performs a content request. A valid placement must be supplied. Placements are defined and configured in the PlayHaven dashboard. By default, an overlay is shown immediately, which ensures that the player cannot interact with the game while the content is being requested (interaction could result in unexpected behavior). When successful the content is shown on top of the Unity canvas. If there was an error with the request an `OnErrorContentRequest` event is fired.

instance

```
public static PlayHavenManager instance { get; }
```

This accessor returns the singleton instance of the `PlayHavenManager`. There must be one and only one `PlayHavenManager` object in the currently loaded scene. All interaction with the manager is done through this instance.

OnBadgeUpdate

```
public event BadgeUpdateHandler OnBadgeUpdate
public delegate void BadgeUpdateHandler(string badge)
```

This event is fired when a badge has been successfully returned due to making a badge request. The badge value (as a *string*) is provided in the event. It can also be obtained from the `Badge` accessor (see above).

OnDismissContent

```
public event DismissHandler OnDismissContent
public delegate void DismissHandler()
```

This parameterless event is fired when the player has dismissed content that was previously presented as a result of a content request.

OnDismissCrossPromotionWidget

```
public event DismissHandler OnDismissCrossPromotionWidget
public delegate void DismissHandler()
```

This parameterless event is fired when the player has dismissed the cross promotion widget.

OnErrorContentRequest

```
public event ErrorHandler OnErrorContentRequest
public delegate void ErrorHandler(PlayHaven.Error error)
```

This event is fired if an `ContentRequest()` call has failed. Failure is typically because the supplied *token* and/or *secret* are incorrect or the *placement* value is invalid. The provided `Error` object will have details.

OnErrorCrossPromotionWidget

```
public event ErrorHandler OnErrorCrossPromotionWidget
public delegate void ErrorHandler(PlayHaven.Error error)
```

This event is fired if an `BadgeRequest()` call has failed. Failure is typically because the supplied *token* and/or *secret* are incorrect. The provided `Error` object will have details.

OnErrorMetadataRequest

```
public event ErrorHandler OnErrorMetadataRequest
```

```
public delegate void ErrorHandler(PlayHaven.Error error)
```

This event is fired if an `ShowCrossPromotionWidget()` call has failed. Failure is typically because the supplied *token* and/or *secret* are incorrect. The provided `Error` object will have details.

OnErrorOpenRequest

```
public event ErrorHandler OnErrorOpenRequest
public delegate void ErrorHandler(PlayHaven.Error error)
```

This event is fired if an `OpenNotification()` call has failed. Failure is typically because the supplied *token* and/or *secret* are incorrect. The provided `Error` object will have details.

OnRewardGiven

```
public event RewardTriggerHandler OnRewardGiven
public delegate void RewardTriggerHandler(PlayHaven.Reward reward)
```

This event is fired if a content request results in a reward being returned by the PlayHaven system. The reward object includes the *name* and *quantity* values as specified in the PlayHaven dashboard.

OnSuccessOpenRequest

```
public event SuccessHandler OnSuccessOpenRequest
public delegate void SuccessHandler()
```

This event is fired when the Open request initiated by the `OpenNotification` method is successful. Every game that uses PlayHaven should inform the PlayHaven servers that the game has been launched. You may want to listen to this event if you want to ensure that you do not make any additional requests unless you have confirmed that the `OpenNotification` was registered as successful.

OpenNotification

```
public void OpenNotification()
```

This method notifies PlayHaven that your game has been launched. PlayHaven requires that an Open request is made with each game launch. If successful, an `OnSuccessOpenRequest` even will be fired; otherwise an `OnErrorOpenRequest` even will be fired.

ShowCrossPromotionWidget

```
public void ShowCrossPromotionWidget()
```

This method requests that the cross-promotion (or "More Games") widget be displayed. If the request is successful, an overlay showing more games that can be downloaded will be automatically displayed. The `OnErrorCrossPromotionWidget` event will be fired if an error occurs with this request.

PlayHaven Content Requester API

The `PlayHavenContentRequester` component helps take away some of the burden of fully implementing content requests yourself. This `MonoBehaviour` can automatically request content for you as well as listen to reward notifications if the specified placement is configured to possibly give a reward (this configuration is performed in the PlayHaven web-based dashboard).

If a content request can give a reward, you will want to set `rewardMayBeDelivered` (shown as `Rewardable` in the editor) to *`true`*. Then if the content request does indeed return a reward an `OnPlayHavenRewardGiven(PlayHaven.Reward)` message is broadcast (the default setting) to this game object and its children. You must attach a script that will respond to this message and handle the reward as your game desires.

Note: If you are of the school that has a deep aversion to using the reflection messaging features (e.g. `BroadcastMessage` or `SendMessage`) in your Unity projects, then it is up to you to write your code to get a handle on the `OnRewardGiven` event. For example:

```
void Awake()
{
      PlayHavenManager.instance.OnRewardGiven += OnRewardGiven;
}

void OnDestroy()
{
      PlayHavenManager.instance.OnRewardGiven -= OnRewardGiven;
}

void OnRewardGiven(PlayHaven.Reward reward()
{
      // implementation to handle the delivery of the reward
      // to the player of your game
      // ...
}
```

When interacting with the `PlayHavenContentRequester` component, you only need to be concerned with the methods listed below.

OnPlayHavenRewardGiven

```
OnPlayHavenRewardGiven(PlayHaven.Reward reward)
```

This is the message that is broadcast to the game object and to its children if a content request results in a reward being given. If the placement you've defined can deliver a reward, you must attached a script that implements the following:

```
void OnPlayHavenRewardGiven(PlayHaven.Reward reward)
{
      // implementation to handle the delivery of the reward
      // to the player of your game
      // ...
      Debug.Log("Reward! "+reward.name);

}
```

Request

```
public void Request()
```

This method performs a content request using the specified placement value.

RequestPlayHavenContent

```
void RequestPlayHavenContent()
```

This method, only executable via a Unity `SendMessage()`, performs the same functionality as `Request()`. Technically you can also call `Request()` with a `SendMessage()`; however, two methods are provided so that you can make your own code more readable depending on what type of programming/architecture paradigm you chose to use.