**Countly Unity SDK Guide**

**Assumptions:**

Following are some of the key assumptions being considered while developing the SDK. Please take into account following considerations before integrating this SDK:

* This SDK is developed in Unity 2018 2.7f1
* Scripting version is based on .NET 4.x Equivalent
* API Compatibility Level is based on .NET 4.x

**Integration:**

* Import the package (right click on **Assets** => **Import Package** => **Custom Package** => **Path\_To\_Package**)
* Leave all the files as checked because we need to import all the files in the package.
* Attach **AppInitScript.cs** file to the main scene of your Unity application.

**Samples:**

A script named “**Testing.cs**” under folder “Assets\Scripts\Main\Testing” is added. This script contains examples to test all the features of the SDK. You can take reference from this script to know how to utilize each feature of the SDK.

**Initialization:**

To initialize Countly Unity SDK, use following two methods with appropriate parameters.

**NOTE-** The following two methods should be added inside your application start event, in the exact order with method “Begin” being called before method “SetDefaults”.

**\* Begin:**

Countly.Begin(string serverUrl, string appKey, string deviceId = null);

* serverUrl: Required  
  Type: string

Desc: The URL of the Countly Server where you are going to post our requests.   
**Ex:** <https://us-try.count.ly/>

* appKey: Required  
  Type: string  
  Desc: The “App Key” for the app that you created on Countly Server.  
  **Ex:** 1234qwer5678qwef88d6123456789qwertyui123
* deviceId: Optional  
  Type: string  
  Desc: Your Device ID. It is an optional parameter.   
  **Ex:** f16e5af2-8a2a-4f37-965d-qwer5678ui98

If you do not provide **deviceId** during initailization, SDK will get device's unique ID as Device ID and use it for all future requests made to the Countly server. This Device ID is persisted by the SDK per device.

If you provided a Device ID to the SDK and later you are initializing the SDK again with a different Device ID, the old Device ID will be used by the SDK and not the new one because the priority is as follows:

[Cached Device ID] **>** [Provided Device ID Upon Initialization] **>** [SDK Generated Device ID]

**\* SetDefaults:**  
This method takes as input an instance of **CountlyConfigModel** class. The constructor for CountlyConfig model takes following list of parameters (the default values are mentioned alongside each one of them):

* string salt = NULL
* bool enablePost = false
* bool enableConsoleErrorLogging = false
* bool ignoreSessionCooldown = false
* bool enableManualSessionHandling = false
* int sessionDuration = 60
* int eventThreshold = 100
* int storedRequestLimit = 1000
* int totalBreadcrumbsAllowed = 100
* TestMode? notificationMode = NULL

**Description:**

* salt:  
  Type: string  
  Desc: Used to prevent parameter tampering.
* enablePost:  
  Type: bool  
  Desc: When set to true, all requests made to the Countly Server will be done using HTTP POST. Otherwise, SDK sends all requests using HTTP GET method. In some cases, if the data to be sent exceeds 1800 Characters limit then, SDK uses POST method.
* enableConsoleErrorLogging:  
  Type: bool  
  Desc: This parameter is only useful when you are debugging your application in Unity Editor. When set to true, it basically turns on Error Logging on Unity Console window.
* ignoreSessionCooldown:  
  Type: bool  
  Desc: To turn on/off session cooldown behavior as mentioned in the [development-guide](https://resources.count.ly/docs/sdk-development-guide).
* enableManualSessionHandling:   
  Type: bool  
  Desc: To turn on/off manual session handling in the application.
* sessionDuration:  
  Type: bool  
  Desc: To set the interval (in seconds) after which the application will automatically extend the session provided manual session is disabled. This interval is also used to process requests in queue. The default value is 60 (seconds).
* eventThreshold:  
  Type: int  
  Desc: To set a threshold value that limits the number of events that can be recorded internally by the system before they all can be sent together in one request. Once the threshold limit is reached, the system groups all recorded events and send them to the server. The default value is 100 (events).
* eventThreshold:  
  Type: int  
  Desc: To set a threshold value that limits the number of events that can be recorded internally by the system before they all can be sent together in one request. Once the threshold limit is reached, the system groups all recorded events and send them to the server. The default value is 100 (events).
* totalBreadcrumbsAllowed:  
  Type: int  
  Desc: To set a threshold value that limits the number of requests that can be stored internally by the system. The system processes these requests after every **sessionDuration** interval has passed. The default value is 1000 (requests).
* notificationMode: Optional  
  Type: Enum of type TestMode  
  Desc: When null, SDK disables Push Notification for the device. Otherwise, when provided an appropriate value from the enum TestMode, SDK uses the supplied mode for sending Push Notifications.

**Ex:**

var configObj = new CountlyConfigModel(null, false, false, false, false, 60, 100, 1000,   
 100, TestMode.TestToken);

await Countly.SetDefaults(configObj);

**Sessions:**

**\* Automatic Session Handling:**

**Begin Session-** SDK is responsible for automatically handling Countly session in your app. As soon as you call the initialization methods (Begin and SetDefaults) in your app start event, SDK will start the session automatically (only when you set **enableManualSessionHandling** to true during initialization).

**Update Session-** SDK is responsible for automatically extending the session after every 60 seconds (default value). This value is configurable during initialization using parameter **sessionDuration**. It cannot be modified any time after initialization.

**NOTE-**

In iOS, session will not extend when the app is in background. As soon as user switches back to the app, session extension will resume.

Whereas in Android, session will extend on both occasions- Foreground/Background.

**End Session-** SDK is responsible for automatically ending the session whenever User quits the application.

**NOTE-** Session will be ended automatically when user calls Application.Quit() method available in Unity.

**\* Manual Session Handling:**

**Begin Session-** To Start the session manually, you need to call the following method:

Parameters: None

Countly.BeginSessionAsync();

**Update Session-** To extend the session manually, you need to call following method:

Parameters: None

Countly.ExtendSessionAsync();

**End Session-** To end the session manually, you need to call the following method:

Parameters: None

Countly.EndSessionAsync()

**Optional Parameters:**

Although, Countly servers recognize the location of the user’s device from their IP address. However, you can specify user’s location manually using the following methods.

**NOTE-** You need to use the following methods in conjunction to specify the exact location manually. Also, IPAddress can be left out intentionally when you want to specify a location but don’t have access to user’s IPAddress. It is because Countly server gives priority to IPAddress. So, if you’ve specified County, City and Coordinates of a different location and IPAddress of a different location, Countly server will choose the location from the IPAddress.

**\* Set Country Code:**

You can set choose to set the country. To set the country, you need to call method SetCountryCode.

Syntax: Countly.SetCountryCode(string country\_code);

Parameter(s):

* country\_code: Required.  
  Type: string  
  Desc: It takes an ISO Country Code in string format, as parameter.

**Ex:** Countly.SetCountryCode(“au”);

**\* Set City:**

You can set choose to set the city. To set the city, you need to call method SetCity.

Syntax: Countly.SetCity(string cityname);

Parameter(s):

* cityname: Required.   
  Type: string  
  Desc: It takes name of the city in string format, as parameter.

**Ex:** Countly.SetCity(“adelaide”);

**\* Set Location:**

You can choose to set the coordinates of the user’s location. To set the coordinates, you need to use method SetLocation.

Syntax: SetLocation(double latitude, double longitude);

Parameter(s):

* latitude: Required.   
  Type: double  
  Dec: It takes latitude value of the device location.
* longitude: Required.   
  Type: double  
  Desc: It takes longitude value of the device location.

**Ex:** Countly.SetLocation(34.9285,138.6007);

**\* Set IPAddress:**

You can choose to set the IP Address of the user’s device. To set the IP Address, you need to first get user’s device IP Address and then use method SetIPAddress.

Syntax: Countly.SetIPAddress(string ip\_address);

Parameter(s):

* ip\_address: Required.   
  Type: string  
  Desc: It takes IP Address of the user’s device.

**Ex:** Countly.SetIPAddress(“SOME\_IP\_ADDRESS”);

**\* Disable Location:**

You can disable location altogether. This will reset everything that has been previously set in context of location like- Country, City, Location and IPAddress.

Parameter(s): None

**Ex:** Countly.DisableLocation();

**Crash Reporting:**

**\* Automatic Crash Reporting:**

Countly Unity SDK automatically reports regarding uncaught exceptions/crashes, in the application, to the Countly server.

**\* Manual Crash Reporting:**

Apart from automatically reporting crashes/uncaught exceptions to the Countly server, the SDK allows you to report your custom errors by using method SendCrashReportAsync.

Syntax:

Countly.SendCrashReportAsync(string message, string stackTrace, LogType type,

IDictionary<string, object> segments = null)

Parameters:

* message: Required.   
  Type: string  
  Desc: Complete error message.
* stackTrace: Required.   
  Type: string  
  Desc: Complete Stack Trace of the exception.
* type: Required.   
  Type: A value of enum LogType, defined under UnityEngine namespace.   
  Desc: You can choose from the various values defined in the enum LogType.
* segments: Optional.   
  Type: IDictionary<string, object>  
  Desc: Custom data in key-value pairs of string and object. This data will be posted to the Countly Server along with the exception details.

**\* Breadcrumbs:**

Additionally, SDK allows you to leave breadcrumbs that would be submitted together with the crash reports. These breadcrumbs are added automatically and sent along with the crash report (for both automatically caught and manually caught exceptions).

You can add breadcrumbs in the SDK using method AddBreadcrumbs. A breadcrumb is a string with at most 1000 characters. This cannot be modified. We can add a maximum of 100 breadcrumbs (as it is default value) in the system. However, we can modify this during initialization with parameter **totalBreadcrumbsAllowed**.

Syntax:

Countly.AddBreadcrumbs(string breadcrumb);

Parameter(s):

* breadcrumb: A string value   
  Type: String

**Events:**

SDK allows you to report events to the Countly server. An event can be anything from a user action like- click, hover, to your custom events “purchase made”, “user passed X level”, etc. You can report any type of events to the Countly server.

SDK allows you to record as many events as you can (you can set a threshold limit during initialization), and the system will send them automatically to the server once the threshold limit is reached.  
  
**NOTE-** You need to first set the threshold value that decides the number of events that can be recorded by the system, i.e., EventSendThreshold value, during initialization. The default value is 100. Once we have recorded events till the threshold value, SDK will send all events at once in a single request.

* **Record Event:**You can record an event by providing the event name. Use following method to record a event. The event will not be reported to the Countly server immediately and will be reported to the server once the threshold limit is reached.  
  Syntax:  
  Countly.RecordEventAsync(string key);  
  Parameter(s):   
  i) key: Required.   
  Type: string  
  Desc: The name of the event.  
  **Ex:** Countly.RecordEventAsync(“Game\_Level\_X\_Started”);
* **Record Event (Overload):**This is an overload to the method we defined above. We can provide other information related to the particular event with this overload method.

Syntax:  
Countly.RecordEventAsync(string key, IDictionary<string, object> segmentation,

int? count = 1, double? sum = 0, double? duration = null)

Parameter(s):

* key: Required  
  Type: string  
  Desc: The name of the event.
* segmentation: Optional  
  Type: IDictionary<string, object>  
  Desc: Payload data to be sent along with the event
* count: Optional  
  Type: integer  
  Desc: Default value is 1
* sum: Optional  
  Type: double
* duration: Optional  
  Type: double

**Ex:** Countly.RecordEventAsync("Game\_Level\_X\_Started",  
 new Dictionary<string, object>

{   
 { "Time Spent", "1234455"},

{ "Retry Attempts", "10"}  
 });

**Views:**

**\* Manual View (Screen) Tracking:**

You can report what views did user view and for how long. Countly Unity SDK allows you to manage view (screen) tracking manually. So, whenever switches screen in your app, you can report it to the Countly server by using following method.

Syntax:

Countly.ReportViewAsync(string name, bool hasSessionBegunWithView = false);

Parameter(s):

* name: Required  
  Type: string  
  Desc: The name of the view to be reported.
* hasSessionBegunWithView: Optional  
  Type: bool  
  Desc: Set it to true to indicate that the session started with this view

**Ex:** Countly.ReportViewAsync(“LoginScreen”);

**View Actions:**

Additionally, it is possible to report actions taken on views to display on heat maps or any other purpose. For that, you need to use method ReportActionAsync.

Syntax:

Countly.ReportActionAsync(string type, int x, int y, int width, int height);

Parameter(s):

* type: action type, as click, touch, longpress,etc
* x: x coordinate of action
* y: y coordinate of action
* width: width of the screen
* height: height of the screen

**Ex:** Countly.ReportActionAsync("Touch", 0, 0, 50, 50);

**Star Rating:**

When a user rates your application, you can report it to the Countly server using method ReportStarRatingAsync.

Syntax:

Countly.ReportStarRatingAsync(string platform, string app\_version, int rating);

Parameter(s):

* platform: platform on which application runs
* app\_version: application's version number
* rating: user's 1-to-5 rating

**Ex:** Countly.ReportStarRatingAsync("android", "0.1", 3);

**User Details:**

Countly Unity SDK allows you to upload specific data related to a User on to the Countly server. SDk allows you to set following data for a user:

* Name: Full name of the user
* Username: Username of the user
* Email: Email address of the user
* Organization: Organization the user is working in
* Phone: Phone number
* Picture Url: Web based Url for user’s profile
* Gender: Gender of the user (use only single char like ‘M’ for Male and ‘F’ for Female)
* Birth Year: Birth year of the user

Apart from the above data, you can also add your own custom data for a user. SDK allows you to upload user details using following different methods. You can choose one of the following methods as per your requirement.

**\* Set User Details:**

You can set all the details (specified above) of a user along with some custom data using method SetUserDetailsAsync.

Syntax:

SetUserDetailsAsync();

Parameter(s): None

This method is an instance method and not a static method. It doesn’t take any parameters. You first need to create an instance of class **CountlyUserDetailsModel**. The constructor for this class takes following number of parameters:

* Name: String
* Username: String
* Email: String
* Organization: String
* Phone: String
* PictureUrl: String
* Gender: String
* Birth Year: String
* Custom: String. It is a json string containing key-value pairs. It is used to send custom data.

**Ex:**var userDetails = new CountlyUserDetailsModel(

"Full Name", "username", "useremail@email.com", "Organization",

"222-222-222",   
 "<http://webresizer.com/images2/bird1_after.jpg>",

"M", "1986",

new Dictionary<string, object>

{

{ "Hair", "Black" },

{ "Race", "Asian" },

});

await userDetails.SetUserDetailsAsync();

**\* Set Custom User Details:**

SDk allows you the flexibility to send only custom data to the COuntly servers when you don’t want to send other user related data. Similar to the above method, it is also an instance method and not a static method. So, you first need to create an instance of class **CountlyUserDetailsModel**. All the parameters expected in the constructor remain the same. You can leave all parameters as NULL and just provide the custom data segment for sending custom data to the Countly server.

Syntax:

SetCustomUserDetailsAsync();

Parameter(s): None

This method doesn’t take any parameters.

**Ex:**

var userDetails = new CountlyUserDetailsModel(

new Dictionary<string, object>

{   
 { "Nationality", "Indian" },

{ "Height", "5.8" },

{ "Mole", "Lower Left Cheek" }   
 });

await userDetails.SetCustomUserDetailsAsync();

**Modifying Custom Data Properties:**

Apart from setting user details (**SetUserDetailsAsync)** and custom user details (**SetCustomUserDetailsAsync)**, SDK allows you to set/modify specific custom data properties. The system records multiple properties at once and reports them (updates to the server) when you want.

Following are some of the methods available in the SDK. All these methods are static methods and defined in the class “**CountlyUserDetailsModel**”.

* **Set:** Sets a value to the provided key.  
  Syntax: CountlyUserDetailsModel.SetOnce(string key, string value);  
  Parameter(s):
* key: Name of the property to be updated
* value: Value to be updated with

**Ex:** CountlyUserDetailsModel.Set("Weight", "80");  
 CountlyUserDetailsModel.Save();

* **Set Once:** Sets value to key, only if property was not defined before for this user  
  Syntax:   
  CountlyUserDetailsModel.SetOnce(string key, string value);  
  Parameter(s): Already defined above.

**Ex:**CountlyUserDetailsModel.SetOnce("Weight", "90");  
CountlyUserDetailsModel.Save();  
In the above example, Weight will not be updated with value “90” if you’ve already updated Weight with a certain value.

* **Increment:** To increment value on server by 1 (if no value on server, assumes it is 0)  
  Syntax:  
  CountlyUserDetailsModel.Increment(string key);  
  Parameter(s): Already defined above.

**Ex:**CountlyUserDetailsModel.Increment("Weight");  
CountlyUserDetailsModel.Save();

* **Increment By:** To increment value on server by provided value (if no value on server, assumes it is 0)  
  Syntax:  
  CountlyUserDetailsModel.IncrementBy(string key, double value);  
  Parameter(s): Already defined above.

**Ex:**CountlyUserDetailsModel.IncrementBy("Weight", 1);  
CountlyUserDetailsModel.Save();

* **Multiply:** To multiply value on server by provided value (if no value on server, assumes it is 0)

Syntax:  
CountlyUserDetailsModel.Multiply(string key, double value);  
Parameter(s): Already defined above.

**Ex:**CountlyUserDetailsModel.Multiply("Weight", 2);  
CountlyUserDetailsModel.Save();

* **Max:** To store maximal value from the one on server and provided value (if no value on server, uses provided value)

Syntax:  
CountlyUserDetailsModel.Max(string key, double value);  
Parameter(s): Already defined above.

**Ex:**CountlyUserDetailsModel.Max("Weight", 90);  
CountlyUserDetailsModel.Save();

* **Min:** To store minimal value from the one on server and provided value (if no value on server, uses provided value)

Syntax:  
CountlyUserDetailsModel.Min(string key, double value);  
Parameter(s): Already defined above.

**Ex:**CountlyUserDetailsModel.Min("Weight", 40);  
CountlyUserDetailsModel.Save();

* **Push:** Add one or many values to array property (can have multiple same values, if property is not array, converts it to array)

Syntax:  
CountlyUserDetailsModel.Push(string key, string[] value);  
Parameter(s):

* value: A string array containing list of values that you want to set for the provided   
   key.

**Ex:**CountlyUserDetailsModel.Push("Mole", new string[] { "Left Cheek", "Back", "Toe" });  
CountlyUserDetailsModel.Save();  
  
**Push Unique:** Add one or many values to array property (will only store unique values in array, if property is not array, converts it to array)

Syntax:  
CountlyUserDetailsModel.PushUnique(string key, string[] value);  
Parameter(s): Already defined above.

**Ex:**CountlyUserDetailsModel.PushUnique("Mole", new string[] { "Right Leg", "Right Leg" });  
CountlyUserDetailsModel.Save();

* **Pull:** Remove one or many values from array property (only removes value from array properties)

Syntax:  
CountlyUserDetailsModel.Pull(string key, string[] value);  
Parameter(s): Already defined above.

**Ex:**CountlyUserDetailsModel.Pull("Mole", new string[] { "Right Leg", "Back" });  
CountlyUserDetailsModel.Save();

**\* Recording Multiple Update Requests:**

Apart from updating a single property in one request, you can modify multiple (unique) properties in one single request. So, you can increment Weight and multiply Height in the same request. Similarly, you can record N number of modify requests and Save them all together in one single request instead of multiple requests.

To record a “modify custom user detail” request, you just need to call the particular methods but DO NOT call “Save” method immediately after each method.

CountlyUserDetailsModel.Save() method immediately posts all modify requests recorded yet.

**NOTE-** If you are going to modify multiple properties in one request, make sure you properties are unique, i.e., a property shouldn’t be modified more than once in a single request. However, if you record a property more than once, only the latest modifier will be posted to the server.

**Ex:**

CountlyUserDetailsModel.Max("Weight", 190);

CountlyUserDetailsModel.Multiply("Weight", 190);

CountlyUserDetailsModel.Min("Height", 5.5);

CountlyUserDetailsModel.Push("Mole", new string[] { "Left Cheek", "Back", "Toe" });

CountlyUserDetailsModel.Save();

You can record N number of modify requests and call “Save” when you want to post all events recorded before calling “Save” method.

**NOTE-** In the above example, you can see Max and Multiply are modifying the same property “Weight”. Therefore in such a scenario, only Multiply request will pushed to the server and Max request will not.

**Push Notifications:**

This section requires setting up either APNS (Apple Push Notification Services) or FCM (Firebase Cloud Services). For APNS, you need to get push credentials and upload to Countly. [This document](https://resources.count.ly/docs/countly-sdk-for-ios-and-os-x#section-push-notifications) explains how to retrieve and upload push credentials for APNS.

If you are developing for Android, please read [this documentation](https://resources.count.ly/docs/countly-sdk-for-android#section-setting-up-push-notifications) and follow instructions to setup for GCM/FCM.

After setting up push notification credentials, you must initialize Push Notification for Countly Unity SDK. However, SDK already does it for you so, you don’t have to initialize it separately. At the beginning of this documentation, go to “initialization” process (method **SetDefaults**) where we init the Countly Unity SDK. In that method, you can see we have a parameter (last one) named TEST\_MODE.

* If you don’t want to enable Push Notification for your application, you can pass NULL to this parameter during initialization.
* If you want to enable Push Notification for your application, you can provide any of the suitable value from the enum **TestMode** under namespace Assets.Scripts.Enums.

So, the SDK already initializes Push Notification for you. But, if you’ve disabled Push Notification during initialization and you want to enable anytime after that, you can do that by using following method **EnablePush**.

Syntax:

Countly.EnablePush(TestMode mode);

Parameter(s): Explained above.

**Key Points:**

* Push Notification with banner image in the background is only supported for Android platform. Banner image is not supported on iOS platform.
* Push Notifications will only appear when the app is in Foreground.   
  As soon as user switches to another application and your app is in background, your session will end and you’ll have to initialize Push Notification again on your AppStart or OnFocus event, if you haven’t initialized it in the SetDefaults method of the SDK by passing TestMode parameter.
* For platforms Android and iOS, the notification icon can be set as follows:  
  **Android**- Please place the notification icons under folder “Assets\Plugins\Android\res” for different screen densities. The name of the icon must be “notification\_icon”. A sample PNG file is placed under the folder “**drawable**”.  
  **iOS**- iOS platform will basically pick your app icon as the default push notification icon. So, you don’t need to do anything for iOS platform.

**Changing Device ID:**

Countly Unity SDK persists Device ID which you provide during initialization or generates a random ID, if you don’t provide it. This Device ID will be used persistently for all future requests made from a device until you change that.

The SDK allows you to change the Device ID at any point of time. You can use following any of the following two methods for changing Device ID, depending upon your scenario.

**\* Change Device ID & End Current Session:**

This method changes the Device ID and does following other operations:

* Ends all the events that have been recorded till now.
* Ends current session.
* Updates Device ID and starts new session with new Device ID.

Syntax:

Countly.ChangeDeviceIDAndEndCurrentSessionAsync(string deviceId);

Parameter(s):

* deviceId: Required.

Type: string  
Desc: The new device id

**Ex:** Countly.ChangeDeviceIDAndEndCurrentSessionAsync(“NEW\_DEVICE\_ID”);

**\* Change Device ID And Merge Session Data:**

This method changes the Device ID and merges data for both Device IDs on the Countly server.

Syntax:

Countly.ChangeDeviceIDAndMergeSessionDataAsync(string deviceId);

Parameter(s): Explained above.

**Ex:**

Countly.ChangeDeviceIDAndMergeSessionDataAsync(“NEW\_DEVICE\_ID”);