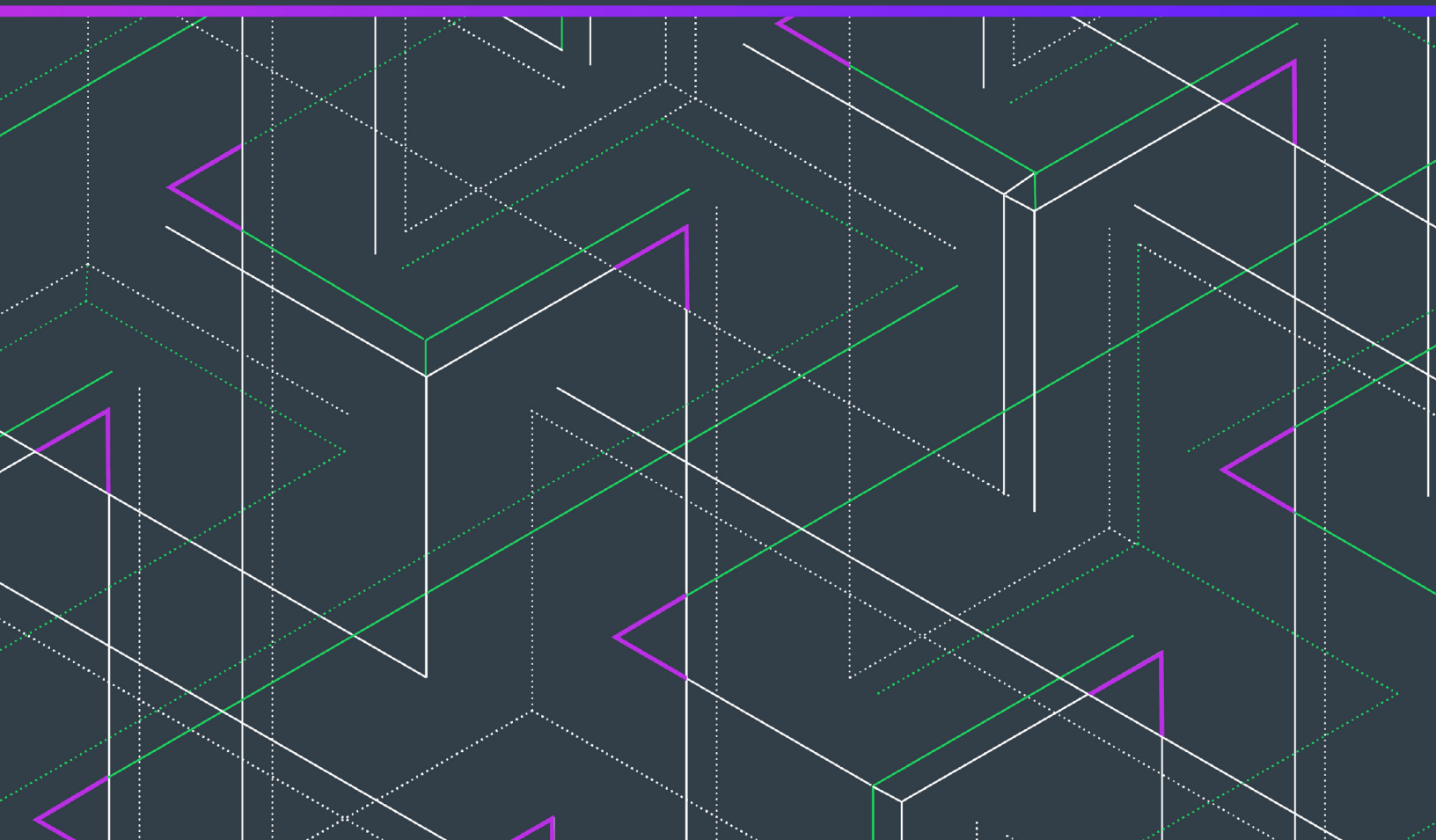
revenera™

Usage Intelligence 5.5.2

Java SDK Developer Guide



Legal Information

Book Name: Usage Intelligence 5.5.2 Java SDK Developer Guide

Part Number: FUI-0552-JAVAUG00

Product Release Date: 16 March 2020

Copyright Notice

Intellectual Property

Restricted Rights Legend

Contents

1

Usage Intelligence 5.5.2 Java SDK Developer Guide

Usage Intelligence 5.5.2—a software usage analytics solution designed for distributed C/C++, .NET, Obj-C and native Java applications on Windows, Macintosh, and Linux—provides deep insight into application usage. It enables you to see which of your application's features are used most and least often. Advanced reporting lets you filter by properties including region, version, OS platform, and architecture to focus your roadmap development.

The Usage Intelligence 5.5.2 Java SDK Developer Guide explains how to implement the Java SDK.

Table 1-1 • Usage Intelligence 5.5.2 Java SDK Developer Guide

Section	Description
Getting Started with the Usage Intelligence Java SDK	Explains how to get started using Usage Intelligence Java SDK.
SDK Configuration	Explains how to create the Usage Intelligence SDK instance, initialize the configuration, and other configuration tasks.
Basic SDK Controls	Describes how to start and stop the SDK, and start and stop a session.
Feature / Event Tracking	Explains how to track and log events.
ReachOut Direct-to-Desktop Messaging Service	Explains how to create ReachOut messaging campaigns to deliver messages or surveys directly to the desktop of users who are running your software.
Exception Tracking	Describes how to collect runtime exceptions from your application.
License Management	Explains how to maintain a license key registry on the Usage Intelligence server in order to track license key usage and verify the status/validity of license keys used on your clients.
Custom Properties	Explains how to collect any custom value that is relevant to your specific application.

Table 1-1 • Usage Intelligence 5.5.2 Java SDK Developer Guide (cont.)

Section	Description
SDK Status Checks	Describes how to collect custom values that are relevant to your specific application.
Common Function Return Values	Lists common return values for Usage Intelligence functions.

Product Support Resources

The following resources are available to assist you with using this product:

- Revenera Product Documentation
- Revenera Community
- Revenera Learning Center
- Revenera Support

Revenera Product Documentation

You can find documentation for all Revenera products on the Revenera Product Documentation site:

https://docs.revenera.com

Revenera Community

On the Revenera Community site, you can quickly find answers to your questions by searching content from other customers, product experts, and thought leaders. You can also post questions on discussion forums for experts to answer. For each of Revenera's product solutions, you can access forums, blog posts, and knowledge base articles.

https://community.revenera.com

Revenera Learning Center

The Revenera Learning Center offers free, self-guided, online videos to help you quickly get the most out of your Revenera products. You can find a complete list of these training videos in the Learning Center.

https://learning.revenera.com

Revenera Support

For customers who have purchased a maintenance contract for their product(s), you can submit a support case or check the status of an existing case by making selections on the **Get Support** menu of the Revenera Community.

https://community.revenera.com

Contact Us

Revenera is headquartered in Itasca, Illinois, and has offices worldwide. To contact us or to learn more about our products, visit our website at:

http://www.revenera.com

You can also follow us on social media:

- Twitter
- Facebook
- LinkedIn
- YouTube
- Instagram

2

Getting Started with the Usage Intelligence Java SDK

This section explains how to get started using the Usage Intelligence Java SDK:

System Requirements

The Usage Intelligence Java SDK can be used with Java 8 and later versions.

Registering Your Product

Before you can use the Usage Intelligence service or integrate the Usage Intelligence SDK with your software, you must first create an account by visiting https://info.revenera.com/SWM-EVAL-Usage-Intelligence.

Once you have a user name and register a new product account for tracking your application, you can get your Product ID, CallHome URL, and AES Key from the Administration page (within the Usage Intelligence dashboard). From here you can also download the latest version of the SDK.

API Overview

When using the Usage Intelligence API, there is one main class that you will have to integrate with: `SDKImpl()`. You need to instantiate this class in the application. All Usage Intelligence methods use this class.

In order to download the Usage Intelligence Java SDK, go to the following page in the Revenera Community:

Usage Intelligence SDK Download Links and API Documentation

Importing the SDK Files

Upon downloading the Java SDK, you find one `.jar` file:

`rui-sdk-<version>.jar`

This `.jar` file contains the Java SDK implementation and supporting dependencies.

`<version>` is the SDK release version number in the form of `N.N.N.N` where `N` is a number.

This JAR file can be incorporated into any Java 8 or later application.

Basic Integration Steps

The most basic Usage Intelligence integration can be accomplished by following the steps below. It is however recommended to read the more advanced documentation as Usage Intelligence can do much more than the basic functionality that can be achieved by following these steps.

Task ***To perform basic integration:***

1. Download the latest SDK from the following page in the Revenera Community, and extract it to your preferred project location.

   Usage Intelligence SDK Download Links and API Documentation

2. Include the `rui-sdk-<version>.jar` file in the build environment for your Java application.

3. Add the directives `import com.revulytics.rui.sdk.*` and import com.revulytics.rui.sdk.core.* to any file that will be using the Java SDK.

4. Create an instance of the `RUISDK` object.

   ```
   boolean registerDefaultReachOut = false;
   RUISDK mySDK = new SDKImpl(registerDefaultReachOut); // note no default graphical ReachOut available
   in Java SDK. Must be false. Value true is ignored.
   ```

5. Create the configuration point to the directory where the Usage Intelligence SDK will create and update files. The application using Usage Intelligence will need read and write access rights to this directory.

   ```
   String myPath = "<path to directory for RUI SDK logging>";
   String myProductId = "<Product ID>";
   String myAppName = "<AppName>";
   String myURL = "<CallHome URL without protocol prefix>";
   String myKey = "<Your AES HEX Key>";
   RUIProtocol myProtocol = RUIProtocol.HTTP_PLUS_ENCRYPTION;
   boolean myMultiSessionSetting = false;
   boolean myReachOutAutosyncSetting = false;

   mySDK.createConfig(myPath, myProductId, myAppName, myURL, myProtocol, myKey, myMultiSessionSetting,
   myReachOutAutosyncSetting);
   ```

   Note the following:

- The Call Home URL, the Product ID and the AES Key can be retrieved from the Administration page (within the Usage Intelligence Dashboard).
- The protocol choice is based on the application and environment needs. Setting protocol to `RUIProtocol.HTTP_PLUS_ENCRYPTION` (port 80) will give applications the greatest chance of success in most environments.
- The Multiple Session flag is a boolean value where you specify whether your application can have multiple user sessions per runtime session. This is normally false.

Note • *For further details, refer to Single vs. Multiple Session Modes.*

- The ReachOut Auto Sync flag indicates whether or not a ReachOut should be requested as part of each SDK Automatic Sync. A ReachOut request will be made only if a ReachOut handler has been set by registering the default graphical handler, `SDKImpl()`, or a custom handler, `setReachOutHandler()`.

6. Initialize the SDK with your product information. This is most conveniently done via the `setProductData()` call. This must be done BEFORE calling `startSDK()`.

```
String myProductEdition = "Professional";
String myLanguage = "US English";
String myVersion = "5.0.0";
String myBuildNumber = "17393";

RUIResult result = mySDK.setProductData(mySDK, myProductEdition, myLanguage, myVersion,
myBuildNumber);
```

7. Initialize the SDK with any optional custom properties by calling the function `setCustomProperty()`.
8. Call the function `startSDK()`.

Note • *You must set all known values for product data and custom properties BEFORE calling `startSDK()` otherwise you risk having null values for fields not specified. Once these calls are completed, you can safely call `startSDK()`.*

Before making any other API tracking calls, you **MUST** call `startSDK()`. It is recommended that you place this call at the entry point of your application so the SDK knows exactly at what time your application runtime session was started. If using multi-session mode, you also need to call `startSession()` when a user session is started, and also provide a unique user session ID that you will then also use for closing the session or for Feature / Event Tracking.

9. Call `stopSDK()` when closing your application so the SDK knows when your application runtime session has been closed.

Important • *You must allow at least 5 seconds of application runtime to allow event data to be written to the log file and synchronized with the Server. If necessary, add a sleep of 5 seconds before calling `stopSDK()`.*

If using multi-session mode, when user sessions are closed, you should call `stopSession()` and send the ID of the session that is being closed as a parameter.

10. Package your application by including the `rui-sdk-<version>.jar` file.

The following is an example of the basic integration outlined below. This example uses single-session mode.

```
import com.revulytics.rui.sdk.*;
import com.revulytics.rui.sdk.core.*;


class MyApplication {
   RUISDK mySDK;
   public MyApplication()
   {
     //Create instance of RUISDK
     boolean registerDefaultReachOut = false;  // Java SDK does not support Default ReachOut
     mySDK = new SDKImpl(registerDefaultReachOut);
     //Set the file path and connection information
     String myPath = "<path to directory for RUI SDK logging>";
     String myProductId = "<Product ID>";
     String myAppName = "<Your App Name>";
     String myURL = "<CallHome URL without protocol prefix>";
     String myKey = "<Your AES HEX Key>";
     RUIProtocol myProtocol = RUIProtocolType.HTTP_PLUS_ENCRYPTION;
     boolean myMultiSessionSetting = false;
     boolean myReachOutAutosyncSetting = false;

     mySDK.createConfig(myPath, myProductId, myAppName, myURL, myProtocol, myKey, myMultiSessionSetting,
myReachOutAutosyncSetting);

     // Set your product information

     String myProductEdition = "Professional";
     String myLanguage = "US English";
     String myVersion = "5.0.0";
     String myBuildNumber = "17393";

     mySDK.setProductData(mySDK, myProductEdition, myLanguage, myVersion, myBuildNumber);

     // If you have any custom properties set them here

     //Inform Revulytics Usage Intelligence that the application has been started.
     mySDK.startSDK();

     //Your program logic...
   }

   private void close()
   {
    //Program closing - inform Revulytics Usage Intelligence that this runtime session is closing down
and sync.
    //Must allow at least 5 seconds between ruiStartSDK() and this call. If less than that, add a
TimeUnit.SECONDS.sleep()
    // or Thread.sleep() or other mechanism to provide enough time to send captured events to RUI Server

            // values to pass stopSDK include :  -1 - No manual sync as part of stop;
            //                                    0 - perform manual sync and wait indefinitely for stop
to finish
            //                                   >0 - perform manual sync and wait x seconds to
completion
    mySDK.stopSDK(0);        // sync and wait indefinitely
```

```
    //Your program logic...
       }
}
```

Next Steps

In the above section, we covered the basic integration steps. While these steps would work for most software products, it is recommended to do some further reading in order to get the most of what Usage Intelligence has to offer. Refer to the following sections for more information: SDK Configuration and Basic SDK Controls. Once you are familiar with the SDK, you may look at the advanced features.

Advanced Features

By following the Basic Integration Steps above, the Usage Analytics SDK will be able to collect information about how often users run your product, how long they are engaged with your software as well as which versions and builds they are running. The SDK also collects information on what platforms and architectures your software is being run (i.e. OS versions, language, screen resolution, etc.). Once you have implemented the basic features, you may choose to use Usage Intelligence for more advanced features that include:

- Feature / Event Tracking
- ReachOut Direct-to-Desktop Messaging Service
- Exception Tracking
- License Management
- Custom Properties

3

SDK Configuration

Before an application can start reporting usage to the Revulytics Usage Intelligence SDK, it must first provide some basic information such as the location of where the SDK will create and save its working files, the application Product ID and the CallHome URL.

You should always attempt to fill in as much accurate and specific details as possible since this data will then be used by the Usage Intelligence Analytics Server to generate the relevant reports. The more (optional) details you fill in about your product and its licensing state, the more filtering and reporting options will be available to you inside the Usage Intelligence dashboard.

SDK Object Initialization

Before beginning any operation, you must first create an instance of the RUISDK object using the `SDKImpl()` function.

The `SDKImpl()` constructor creates an instance of the SDK. The constructor does not configure the SDK (`createConfig()`) nor start the SDK (`startSDK()`).

SDKImpl()

```
SDKImpl(boolean registerDefaultGraphicalReachOutHandler)
```

Parameters

The `SDKImpl()` constructor has the following parameters.

Table 3-1 • SDKImpl() Parameters

Parameter	Description
`registerDefaultGraphicalReachOutHandler (boolean)`	For Java SDK, value must be false. There is no default Graphical ReachOut Handler for the Java SDK.

Getting SDK Version Information

The `getSDKVersion()` function returns the version information for the Usage Intelligence SDK instance.

The `getSDKVersion()` function can be called more than once.

The `getSDKVersion()` function is a synchronous function, returning when all functionality is completed.

getSDKVersion()

```
String getSDKVersion()
```

Returns

The `getSDKVersion()` function returns a formatted string containing the RUISDK version information.

Getting the Client ID

The `getClientID()` function returns the client ID for the RUI SDK instance.

The `getClientID()` function can be called more than once.

The `getClientID()` function is a synchronous function returning when all functionality is completed.

getClientID()

```
String getClientID()¶
```

Returns

The `getClientID()` function returns a formatted string containing the RUISDK version information. It returns a string of "0" if the client ID is not available.

Initializing the Configuration

The `createConfig()` method must be called in order to initialize the configuration.

The `createConfig()` method creates a configuration for the SDK instance. A configuration is passed in a file, specified by `configFilePath`, `productID`, and `appName`. If two or more SDK instances, in the same process or in different processes, use the same values for these three parameters, then those SDK instances are bound together through a shared configuration. When multiple different executables are being used, such usage is generally not desirable nor recommended. Instead, each executable should use a different `appName` value.

The SDK has two communications modes: HTTPS or HTTP + AES-128 encryption. The mode is configured by `protocol`. When protocol is set to `HTTP_PLUS_ENCRYPTION` or `HTTPS_WITH_FALLBACK`, the AES key must be supplied as a 128-bit hex-encoded string (`aesKeyHex`). When protocol is HTTPS, then `aesKeyHex` must be empty.

On first execution of a client application, no SDK configuration file will exist. This situation is detected by the SDK and will result in a New Registration message to the Server at `startSDK()`. Once the configuration is received from the Server, the SDK writes the configuration file, that is then used for subsequent client application executions.

`createConfig()` must be called before most other APIs and must only be successfully called once.

`createConfig()` is a synchronous function, returning when all functionality is completed.

createConfig()

```
RUIResult createConfig(String configFilePath, String productID, String appName, String serverURL,
RUIProtocol protocol, String aesKeyHex, boolean multiSessionEnabled, boolean reachOutOnAutosync)
```

Parameters

The `createConfig()` method has the following parameters.

Table 3-2 • CreateConfig() Parameters

Parameter	Description
`configFilePath (String)`	The directory to use for the SDK instance's configuration file. Cannot be empty; must exist and be writable.
`productID (String)`	The Revenera-supplied product ID for this client; 10 digits. You obtain this ID after registering with Revenera.
`appName (String)`	The customer-supplied application name for this client, to distinguish suites with the same `productID`. Cannot be empty or contain white space; at most 16 UTF-8 characters. More information about the purpose of the `appName` parameter can be found in the Knowledge Base article: What is the purpose of the appName parameter when creating config in the SDK?

Table 3-2 • CreateConfig() Parameters

Parameter	Description
`serverUrl (String)`	Every product registered with Usage Intelligence has its own unique URL usually in the form `xxxxx.tbnet1.com`. This URL is generated automatically on account creation and is used by the SDK to communicate with the Usage Intelligence server. You can get this URL from the Developer Zone once you login to the Usage Intelligence dashboard. If you have a Premium product account, you may opt to use your own custom server URL (such as `http://updates.yourdomain.com`) that must be set up as a CNAME DNS entry pointing to your unique Usage Intelligence URL.  ***Note*** *• Before you can use your own custom URL, you must first inform Usage Intelligence support (support@revenera.com) to register your domain with the Usage Intelligence server. If you fail to do this, the server will automatically reject any incoming calls using `yourdomain.com` as a server URL. The URL should not contain a protocol prefix.*

`protocol (RUIProtocol)`	Indicates whether HTTP + AES, HTTPS with fall-back to HTTP + AES, or HTTPS only is used to communicate with the Server. Valid choices can be found in the `RUIProtocol` enum and are:
	
	`HTTP_PLUS_ENCRYPTION (1)`
	`HTTPS_WITH_FALLBACK (2)`
	`HTTPS (3)`

`aesKeyHex (String)`	AES Key to use when protocol includes encryption (`HTTP_PLUS_ENCRYPTION` or `HTTPS_WITH_FALLBACK`); 32 hex chars (128 bit) key.
`multiSessionEnabled (boolean)`	Indicates whether or not the client will explicitly manage `sessionIDs` via `startSession()` and `stopSession()`, and supply those `sessionIDs` to the various event tracking APIs. Refer to Single vs. Multiple Session Modes.
`reachOutOnAutosync (boolean)`	Indicates whether or not a ReachOut should be requested as part of each SDK Automatic sync. A ReachOut request will be made only if a ReachOut handler has been set by registering the default graphical handler (SDKImpl()) or a custom handler (`setReachOutHandler()`). This value may be changed at runtime using the call `setReachOutOnAutoSync()`.

Returns

The `createConfig()` method returns one of the return status constants below.

Table 3-3 • CreateConfig() Returns

Return	Description
OK	Function successful.
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
CONFIG_ALREADY_CREATED	Configuration has already been successfully created.
INVALID_PARAMETER_EXPECTED_NON_EMPTY	Some API parameter is expected to be non-empty, and is not.
INVALID_PARAMETER_EXPECTED_NO_WHITESPACE	Some API parameter is expected to be free of white space, and is not.
INVALID_PARAMETER_TOO_LONG	Some API parameter violates its allowable maximum length.
INVALID_CONFIG_PATH	The `configFilePath` is not a well-formed directory name.
INVALID_CONFIG_PATH_NONEXISTENT_DIR	The `configFilePath` identifies a directory that does not exist.
INVALID_CONFIG_PATH_NOT_WRITABLE	The `configFilePath` identifies a directory that is not writable.
INVALID_PRODUCT_ID	The `productID` is not a well-formed Product ID.
INVALID_SERVER_URL	The `serverURL` is not a well-formed URL.
INVALID_PROTOCOL	The protocol is not a legal value.
INVALID_AES_KEY_EXPECTED_EMPTY	The AES Key is expected to be NULL/empty, and it's not.
INVALID_AES_KEY_LENGTH	The AES Key is not the expected length (32 hex chars).
INVALID_AES_KEY_FORMAT	The AES Key is not valid hex encoding.

Code Example

The following example shows how to initialize the Usage Intelligence configuration:

```
String myPath = "<path to directory for RUI SDK logging>";
String myProductId = "<Product ID>";
String myAppName = "<AppName>";
String myURL = "<CallHome URL without protocol prefix>";
String myKey = "<Your AES HEX Key>";
RUIProtocol myProtocol = RUIProtocol.HTTP_PLUS_ENCRYPTION;
boolean myMultiSessionSetting = false;
```

```
boolean myReachOutAutosyncSetting = false;

mySDK.createConfig(myPath, myProductId, myAppName, myURL, myProtocol, myKey, myMultiSessionSetting,
myReachOutAutosyncSetting);
```

Single vs. Multiple Session Modes

In desktop software, a single application instance would normally have only one single user session. This means that such an application would only show one window (or set of windows) to a single user and interaction is done with that single user. If the user would like to use two different sessions, two instances of the application would have to be loaded that would not affect each other. In such cases, you should use the single session mode, which handles user sessions automatically and assumes that one process (instance) means one user session.

The multiple session mode needs to be used in multi-user applications, especially applications that have web interfaces. In such applications, a number of users might be using the same application process simultaneously. In such cases, you need to manually tell the Usage Intelligence SDK must be notified when user sessions start and stop, and also how to link events (see Feature / Event Tracking) to user sessions.

To do this, when starting or stopping a user session, the methods `startSession()` and `stopSession()` should be used, and when tracking events on a per user basis, a session ID needs to be passed as a parameter.

Opt-Out Mechanism

Starting from version 5.1.0, a new opt-out mechanism was introduced. Using this mechanism, if a user does not want to send tracking information to Revenera, the method `optOut()` must be called after calling `createConfig()` and before `startSDK()`.

The `optOut()` method instructs the SDK to send a message to the server to indicate that this user is opting-out (if not already sent in previous sessions) and disables all further functionality and communication with the server.

When `optOut()` is called, the SDK sends a message to the Server after startup (`startSDK()`). This message informs the Server that this client has opted-out and the Server will register the opt-out. This message is only sent to the Server once. The opt-out flag on the Server will be used for reporting opt-out statistics only.

If `optOut()` is called before a new registration, the Server will never have any data about that installation. If `optOut()` is called for an installation that was already being tracked, the Server will still contain the data that had been collected in the past and no past data is deleted.

The SDK will send no further information to the server as long as the user is opted-out. The application must keep calling `optOut()` before every startup as long as the user wants to stay opted-out. If this method is not called, then the SDK assumes that the user is opting-in again and will start tracking normally.

Note • *When an installation is not opted-out, it communicates with the Server immediately on calling `startSDK()`. At this point, the SDK attempts to sync data regarding past application and event usage that had not been synced yet, and also system and product information such as OS version, CPU, GPU, product version, product edition, etc.*

optOut()

```
RUIRESULT optOut()
```

Returns

The `optOut()` function returns a `RUIResult` enum value with the following possible values:

Table 3-4 • ruiOptOut() Returns

Return	Description
OK	Function successful.
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is permitted.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STARTED	SDK has already been successfully started.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

Providing Product Data

The Usage Intelligence SDK V5 requires that the application provide product data every time the SDK instance is run. In addition you can optionally set License data for the application. Finally, if you are using proxies to access the Internet, there is a function to set up the required information for connecting through that proxy.

- Product Details
- License Management

Product Details

The following functions are available to set product data:

- Setting Product Data
- Setting Product Edition
- Setting Product Language
- Setting Product Version
- Setting Product Build Number

Setting Product Data

The `setProductData()` function sets or clears the product data.

Note • *The product data must be set every time the SDK instance is run.*

Note • *This is different than V4 of the Usage Intelligence (Trackerbird) SDK where the supplied product data was stored in the SDK configuration file and if it was not supplied, the values in the configuration file were used.*

`setProductData()` can be called between `createConfig()` and `stopSDK()` and can be called zero or more times.

`setProductData()` is a synchronous function returning when all functionality is completed.

setProductData()

```
RUIResult setProductData(String productEdition, String productLanguage, String productVersion, String productBuildNum)
```

Parameters

The `setProductData()` function has the following parameters.

Table 3-5 • `setProductData()` Parameters

Parameter	Description
`productEdition (String)`	The product edition that is to be set. Maximum length of 128 characters.
`productLanguage (String)`	The product language that is to be set. Maximum length of 128 characters.
`productVersion (String)`	The product version that is to be set. Maximum length of 128 characters.
`productBuildNumber (String)`	The product build number that is to be set. Maximum length of 128 characters.

Returns

The `setProductData()` function returns one of the return status constants below.

Table 3-6 • `setProductData()` Returns

Return	Description
`OK`	Function successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.

Table 3-6 • `setProductData()` Returns

Return	Description
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

Setting Product Edition

The `setProductEdition()` function sets or clears the product data.

Note • *The product data must be set every time the RUI SDK instance is run. This is different than V4 of the RUI (Trackerbird) SDK where the supplied product data was stored in the SDK configuration file and if it was not supplied, the values in the configuration file were used.*

`setProductEdition()` can be called between `createConfig()` and `stopSDK()`and can be called zero or more times.

`setProductEdition()`is a synchronous function returning when all functionality is completed.

setProductEdition()

```
RUIResult setProductEdition(String productEdition)¶
```

Parameters

The `setProductEdition()` function has the following parameters.

Table 3-7 • setProductEdition() Parameters

Parameter	Description
productEdition (String)	The product edition that is to be set. Maximum length of 128 characters.

Returns

The `setProductEdition()` function returns a `RUIResult` enum value with the following possible values:

Table 3-8 • ruiSetProductEdition() Returns

Return	Description
OK	Function successful.
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.

Table 3-8 • ruiSetProductEdition() Returns

Return	Description
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

Setting Product Language

The `setProductLanguage()` function sets or clears the product data.

Note • *The product data must be set every time the RUI SDK instance is run. This is different than V4 of the RUI (Trackerbird) SDK where the supplied product data was stored in the SDK configuration file and if it was not supplied, the values in the configuration file were used.*

`setProductLanguage()`can be called between `createConfig()` and `stopSDK()` and can be called zero or more times.

`setProductLanguage()` is a synchronous function returning when all functionality is completed.

setProductLanguage()

```
RUIResult setProductLanguage(String productLanguage)
```

Parameters

The `setProductLanguage()` function has the following parameters.

Table 3-9 • setProductLanguage() Parameters

Parameter	Description
productLanguage (String)	The product language that is to be set. Maximum length of 128 characters.

Returns

The `setProductLanguage()` function `RUIResult` enum value with the following possible values.

Table 3-10 • setProductLanguage() Returns

Return	Description
OK	Function successful.

Table 3-10 • setProductLanguage() Returns

Return	Description
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. stopSDK() is possible.
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

Setting Product Version

The setProductVersion() function sets or clears the product data.

***Note** • The product data must be set every time the RUI SDK instance is run. This is different than V4 of the RUI (Trackerbird) SDK where the supplied product data was stored in the SDK configuration file and if it was not supplied, the values in the configuration file were used.*

setProductVersion() can be called between createConfig() and stopSDK() and can be called zero or more times.

setProductVersion() is a synchronous function returning when all functionality is completed.

setProductVersion()

```
RUIResult setProductVersion(String productVersion)
```

Parameters

The setProductVersion() function has the following parameters.

Table 3-11 • setProductVersion() Parameters

Parameter	Description
productVersion (String)	The product version number that is to be set. Maximum length of 128 characters.

Returns

The `setProductVersion()` function returns `RUIResult` enum value with the following possible values:

Table 3-12 • setProductVersion() Returns

Return	Description
OK	Function successful.
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

Setting Product Build Number

The `setProductBuildNumber()` function sets or clears the product data.

Note • *The product data must be set every time the RUI SDK instance is run. This is different than V4 of the RUI (Trackerbird) SDK where the supplied product data was stored in the SDK configuration file and if it was not supplied, the values in the configuration file were used.*

`setProductBuildNumber()` can be called between `createConfig()` and `stopSDK()` and can be called zero or more times.

`setProductBuildNumber()` is a synchronous function returning when all functionality is completed.

setProductBuildNumber()

```
RUIResult setProductBuildNumber(String productBuildNum)
```

Parameters

The `setProductBuildNumber()` function has the following parameters.

Table 3-13 • ruiSetProductBuildNumber() Parameters

Parameter	Description
productBuildNumber (String)	The product build number that is to be set. Maximum length of 128 characters.

Returns

The `setProductBuildNumber()` function returns `RUIResult` enum value with the following possible values:

Table 3-14 • ruiSetProductBuildNumber() Returns

Return	Description
OK	Function successful.
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

License Management

The `setLicenseData()` function sets or clears the license data. The legal parameter values include unchanged (-1).

Note • *Different from the V4 of the Usage Intelligence SDK, a `sessionID` parameter can be supplied.*

The `setLicenseData()` function can be called between `createConfig()` and `stopSDK()` and can be called zero or more times. However, the usage requirements of the `sessionID` parameter are different if `setLicenseData()` is called before `startSDK()` or called after `startSDK()`. See `sessionId` for more information.

The `setLicenseData()` function can be called while a New Registration is being performed (`createConfig()`, `startSDK()`). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

The `setLicenseData()` function is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

setLicenseData()

```
RUIResult setLicenseData(RUIKeyType keyType, RUIKeyStatus keyExpired, RUIKeyStatus keyActivated,
RUIKeyStatus keyBlacklisted, RUIKeyStatus keyWhitelisted)
```

Parameters

The `setLicenseData()` function has the following parameters.

Table 3-15 • setLicenseData() Parameters

Parameter	Description

keyType (RUIKeyType)	One of the key types from the RUIKeyType enum. Note that custom types can be used for application specific license types:
	`UNCHANGED (-1)`
	`EVALUATION (0)`
	`PURCHASED (1)`
	`FREEWARE (2)`
	`UNKNOWN (3)`
	`NFR (4) // Not for Resale`
	`CUSTOM1 (5)`
	`CUSTOM2 (6)`
	`CUSTOM3 (7)`

keyExpired (RUIKeyStatus)	Indicates whether the client license has expired. Use one of the RUIKeyStatus enum choices for the values:
	`UNCHANGED (-1)`
	`NO (0)`
	`YES (1)`

keyActivated (RUIKeyStatus)	Indicates whether the client license has been activated. Use one of the RUIKeyStatus enum choices for the values:
	`UNCHANGED (-1)`
	`NO (0)`
	`YES (1)`

keyBlacklisted (RUIKeyStatus)	Indicates whether the client license key has been blacklisted. Use one of the RUIKeyStatus enum choices for the values:
	`UNCHANGED (-1)`
	`NO (0)`
	`YES (1)`

keyWhitelisted (RUIKeyStatus)	Indicates whether the client license key has been whitelisted. Use one of the RUIKeyStatus enum choices for the values:
	`UNCHANGED (-1)`
	`NO (0)`
	`YES (1)`

Table 3-15 • setLicenseData() Parameters

Parameter	Description
`sessionId (String)`	An optional session ID complying with above usage (content conditioning and validation rules in `startSession()`. The usage requirements of the `sessionID` parameter are different if `setLicenseData()` is called before `startSDK()` or called after `startSDK()`. • **Called before startSDK (regardless of multiSessionEnabled)**—Use method without `sessionID`. • **Called after startSDK and multiSessionEnabled is set to false**—Use method without `sessionID`. • **Called after startSDK and multiSessionEnabled is set to true**—`sessionID` must be a current valid value used in `startSession()`, or it can be empty. This is different than normal event tracking APIs, whereby a empty value is not allowed.

Returns

The `setLicenseData()` function returns `RUIResult` enum value with the following possible values:

Table 3-16 • setLicenseData() Returns

Return	Description
`OK`	Function successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`INVALID_SESSION_ID_EXPECTED_EMPTY`	The `sessionID` is expected to be empty, and it was not.
`INVALID_SESSION_ID_EXPECTED_NON_EMPTY`	The `sessionID` is expected to be non-empty, and it was not.
`INVALID_SESSION_ID_TOO_SHORT`	The `sessionID` violates its allowable minimum length.

Table 3-16 • setLicenseData() Returns

Return	Description
INVALID_SESSION_ID_TOO_LONG	The sessionID violates its allowable maximum length.
INVALID_SESSION_ID_NOT_ACTIVE	The sessionID is not currently in use.

Changing ReachOut on Autosync Setting

The flag to determine whether or not a ReachOut should be requested as part of each SDK Automatic Sync is initially set in the createConfig() call. There may be certain cases when the application wants to either enable or disable this functionality during the application lifetime.

The setReachOutOnAutoSync() function allows the application to enable or disable this capability after createConfig() has been called.

The setReachOutOnAutoSync() function enables (true) or disables (false) the ReachOut on Autosync capability. Note if the call does not change the existing setting, the API will still return OK.

The setReachOutOnAutoSync() function can be called between createConfig() and stopSDK() and can be called zero or more times.

setReachOutOnAutoSync()

```
RUIResult setReachOutOnAutoSync(boolean reachOutOnAutoSyncSetting)
```

Parameters

The setReachOutOnAutoSync() function has the following parameters.

Table 3-17 • setReachOutOnAutoSync() Parameters

Parameter	Description
reachOutOnAutoSyncSetting (boolean)	Enable (true) or disable (false) the ReachOut on Autosync capability.

Returns

The setReachOutOnAutoSync() function returns RUIResult enum value with the following possible values:

Table 3-18 • setReachOutOnAutoSync() Returns

Return	Description
OK	Function successful.
INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.

Table 3-18 • setReachOutOnAutoSync() Returns

Return	Description
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

Proxy Support

The Usage Intelligence SDK V5 library supports communications through proxies on all major operating system types: Windows, Linux, and macOS. Application developers are responsible for obtaining proxy credentials (if the proxy requires it) and setting those credentials in the SDK so communications can use the credentials for the proxy. The function `setProxy()` handles setting and clearing the proxy related information.

The `setProxy()` function sets or clears the data to be used with a proxy. If there is no proxy between the SDK and the Server, there is no need to use this function. The address can be either empty (for transparent proxy servers) or non-empty. The username and password must both be empty (non-authenticating proxy) or both be non-empty (authenticating proxy). The port is only used for non-transparent proxy servers, hence port must be zero if address is empty, otherwise port must be non-zero.

`setProxy()` can be called between `createConfig()` and `stopSDK()`, and can be called zero or more times.

`setProxy()` is a synchronous function, returning when all functionality is completed.

setProxy()

```
RUIResult setProxy(String address, short port, String username, String password)
```

Permitted Parameter Combinations

The SDK uses the proxy data in multiple ways to attempt to communicate via a proxy. The allowed parameter combinations and their usage are as follows:

Table 3-19 • setProxy() Permitted Parameter Combinations

address	port	username	password	Description
empty	0	empty	empty	Resets the proxy data to its initial state, no proxy server is used, and the Server is contacted directly.

Table 3-19 • setProxy() Permitted Parameter Combinations

address	port	username	password	Description
non-empty	not 0	empty	empty	Identifies a non-authenticating non-transparent proxy that will be used unless communications fails, then falling back to using no proxy.
empty	0	non-empty	non-empty	Identifies an authenticating transparent proxy that will be used unless communications fails, then falling back to using no proxy.
non-empty	not 0	non-empty	non-empty	Identifies an non-transparent authenticating proxy that will be used unless communications fails, then falling back to using an authenticating transparent proxy, then falling back to using no proxy.

Parameters

The `setProxy()` function has the following parameters.

Table 3-20 • setProxy() Parameters

Parameter	Description
`address (String)`	The server name or IP address (dot notation) for the named proxy.
`port (short)`	The port for the proxy server; only used with named proxy, port `!= 0` if and only if address non-empty.
`username (String)`	The named proxy username; username and password must both be empty or both be non-empty.
`password (String)`	The named proxy password; username and password must both be empty or both be non-empty.

Returns

The `setProxy()` function returns RUIResult enum value with the following possible values:

Table 3-21 • setProxy() Returns

Return	Description
`OK`	Function successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.

Table 3-21 • setProxy() Returns

Return	Description
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_SUSPENDED	The Server has instructed a temporary back-off
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
INVALID_PROXY_CREDENTIALS	The proxy username and password are not an allowable combination.
INVALID_PROXY_PORT	The proxy port was not valid.

4

Basic SDK Controls

Once the required configuration is initialized (explained in SDK Configuration) and set according to the needs of your application, you may inform the SDK that the application has started. This will allow you to use further functions that expect the application to be running such as `checkLicenseKey()` and `checkForReachOut()`.

- Starting the SDK
- Stopping the SDK
- Starting a Session
- Stopping a Session
- Caching and Synchronizing

Starting the SDK

The startSDK() function starts the SDK. `startSDK()` must be paired with a call to `stopSDK()`.

After the SDK is started, the various event tracking APIs are available to be used. If `createConfig()` did not detect a configuration file, `startSDK()` will perform a New Registration with the Server. Until a New Registration is complete, the SDK will not be able to save event data to a log file or perform synchronization with the Server. A successful New Registration (or presence of a configuration file) will put the SDK into a normal running state, whereby events are saved to a log file, automatic and manual synchronizations with the Server are possible, and getting ReachOut campaigns from the Server are possible. A failed New Registration will put the SDK into an aborted state, not allowing further activity.

`startSDK()` must be called after `createConfig()`, and must be called only once.

`startSDK()` is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

If `optOut()` is called before a new registration has been done for a user, the SDK will not sync any system and product information and no data is recorded for the user. The SDK will inform the server once that there is an opted out user for reporting opt-out statistics only.

startSDK()

```
RUIResult startSDK()
```

Returns

The startSDK() function returns a `RUIResult` enum value with the following possible values:

Table 4-1 • startSDK() Returns

Return	Description
`OK`	Synchronous functionality successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`SDK_ALREADY_STARTED`	SDK has already been successfully started.
`SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.

Stopping the SDK

The `stopSDK()` function stops the SDK that was started with `startSDK()`.

If explicit sessions are allowed (`multiSessionsEnabled = true` in `createConfig()`), then any sessions that have been started with `startSession()` that have not been stopped with `stopSession()` are automatically stopped. A manual synchronization with the Server, `sync()`, will be performed at stop depending on the value of doSync (as described in Parameters).

`stopSDK()` must be called after `startSDK()` and must be called only once. After `stopSDK()` is called, the various event tracking APIs are no longer available. The only API available is `getState()`. The SDK cannot be re-started with a subsequent call to `startSDK()`.

`stopSDK()` is a synchronous function, including the manual synchronization with the Server (if requested), returning when all functionality is completed.

stopSDK()

```
RUIResult stopSDK(int dosync)
```

Parameters

The `stopSDK()` function has the following parameters.

Table 4-2 • stopSDK() Parameters

Parameter	Description
`doSync (int)`	Indicates whether to do a manual synchronization as part of the stop, and if so, the maximum wait limit in seconds. A manual synchronization with the Server, `sync()`, will be performed at stop depending on the value of doSync: • **1**—Do not perform a manual synchronization with the Server as part of the stop. • **0**—Perform a manual synchronization with the Server as part of the stop; wait indefinitely for completion. • **>0**—Perform a manual synchronization with the Server as part of the stop; wait only doSync seconds for completion.

Returns

The `stopSDK()` function returns `RUIResult` enum value with the following possible values:

Table 4-3 • stopSDK() Returns

Return	Description
`OK`	Synchronous functionality successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`SDK_NOT_STARTED`	SDK has not been successfully started.
`SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`INVALID_DO_SYNC_VALUE`	The doSync manual sync flag/limit violates its allowable range.

Code Example

This example shows `stopSDK()` being called in the closing event of a form.

```
// Create instance
boolean registerDefaultReachOut = false; // no default ReachOut exists for Java SDK
```

```
mySDK = new SDKImpl(registerDefaultReachOut);
// Other initialization.....


private void close()
{
    mySDK.stopSDK(5);     // wait maximum of 5 seconds
}
```

Starting a Session

The `startSession()` function starts an explicit session for event tracking in the SDK. It must be paired with a call to `stopSession()`.

Explicit sessions are allowed only if `createConfig()` was called with `multiSessionEnabled = true`. When explicit sessions are enabled, a valid `sessionID` becomes a required parameter to the event tracking APIs, as described in Parameters.

`startSession()` can be called between `startSDK()` and `stopSDK()`, and can be called zero or more times.

`startSession()` is a synchronous function, returning when all functionality is completed.

startSession()

```
RUIResult startSession(String sessionID)
```

Parameters

The `startSession()` function has the following parameters.

Table 4-4 • startSession() Parameters

Parameter	Description
`sessionID (String)`	This parameter should contain a unique ID that refers to the user session that is being started. This same ID should later be used for event tracking. `sessionID` must be at least 10 characters long and no longer than 64 characters. The content of a `sessionID` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Validation**—Cannot be shorter than 10 UTF-8 characters. • **Validation**—Cannot be longer than 64 UTF-8 characters. The resulting conditioned and validated `sessionID` must be unique (i.e. not already in use).  ***Note*** • *With the above conditioning, two `sessionIDs` that differ only by white space or after the 64th character, will not be unique. A `sessionID` should not be re-used for different sessions.*

Returns

The `startSession()` function returns one of the return status constants below.

Table 4-5 • startSession() Returns

Return	Description
`OK`	Function successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`SDK_SUSPENDED`	The Server has instructed a temporary back-off.

Table 4-5 • startSession() Returns

Return	Description
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_NOT_STARTED	SDK has not been successfully started.
FUNCTION_NOT_AVAIL	Function is not available.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
INVALID_SESSION_ID_EXPECTED_NON_EMPTY	The sessionID is expected to be non-empty, and it was not.
INVALID_SESSION_ID_TOO_SHORT	The sessionID violates its allowable minimum length.
INVALID_SESSION_ID_TOO_LONG	The sessionID violates its allowable maximum length.
INVALID_SESSION_ID_ALREADY_ACTIVE	The sessionID is already currently in use.

Stopping a Session

The stopSession() function stops an explicit session started with startSession().

Explicit sessions are allowed only if createConfig() was called with multiSessionEnabled = true. Any explicit sessions not ended with a call to stopSession() are automatically ended when stopSDK() is called. In case this method is never called, eventually this session will be considered as “timed-out”, and the time of the last recorded event will be assumed to be the time when the last event was recorded.

stopSession() can be called between startSDK() and stopSDK(), and can be called zero or more times.

stopSession() is a synchronous function, returning when all functionality is completed.

stopSession()

```
RUIResult stopSession(String sessionID)
```

Parameters

The stopSession() function has the following parameters.

Table 4-6 • stopSession() Parameters

Parameter	Description
sessionID (String)	This parameter should contain a unique ID that refers to the user session that is being stopped. This must be the same ID that was used earlier when calling startSession().

Returns

The `stopSession()` function returns an `RUIResult` enum value with the following possible values:.

Table 4-7 • stopSession() Returns

Return	Description
`OK`	Synchronous functionality successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`SDK_NOT_STARTED`	SDK has not been successfully started.
`SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`INVALID_SESSION_ID_EXPECTED_NON_EMPTY`	The `sessionID` is expected to be non-empty, and it was not.
`INVALID_SESSION_ID_TOO_SHORT`	The `sessionID` violates its allowable minimum length.
`INVALID_SESSION_ID_TOO_LONG`	The `sessionID` violates its allowable maximum length.
`INVALID_SESSION_ID_NOT_ACTIVE`	The `sessionID` is not currently in use.

Caching and Synchronizing

The Usage Intelligence SDK was designed to minimize network traffic and load on the end user's machine. In order to do this, all the collected architecture info and runtime tracking data is cached locally and then compressed and sent to the Usage Intelligence server in batches at various intervals whenever appropriate. Log data is usually sent at least once for every runtime session (during `startSDK()`), however this may vary based on the type of application and usage activity.

Data may be sent via HTTP (port 80) or HTTPS (port 443) depending on application preference. When data is sent over HTTP, AES encryption is used to encrypt the data payload. When data is sent on HTTPS, normal HTTPS security measures are used. The application may also choose to start with HTTPS communication and if blocked or unsuccessful the SDK will fall back to using encrypted HTTP.

- Forced Synchronization

Forced Synchronization

Under normal conditions, you do not need to instruct the Usage Intelligence SDK when to synchronize with the cloud server, since this happens automatically and is triggered by application interaction with the API. In a typical runtime session, the SDK will always attempt to synchronize with the server at least once whenever the application calls `startSDK()`. For long running applications, the SDK will periodically sync with the server every 20 minutes.

For applications that require a more customized synchronization, the API also provides an option to request manual synchronization of all cached data. This is done by calling the `sync()` function.

The sync() function performs a manual synchronization with the Server. The SDK periodically performs automatic synchronizations with the Server. `sync()` provides the client an ability to explicitly synchronize with the Server at a specific time. The manual synchronization can request a ReachOut with `getReachOut`.

Note • *Similar to the parameter `reachOutOnAutoSync` (on function `createConfig()`), the ReachOut will not be requested if there is no registered handler (SDK constructor and `setReachOutHandler()`).*

`sync()` can be called between `startSDK()` and `stopSDK()` and can be called zero or more times.

Note • *`sync()` will not be successful if a New Registration is in progress (i.e. `createConfig()` and `startSDK()`). A manual synchronization with the Server can be associated with `stopSDK()`.*

`sync()` is an asynchronous function returning immediately with further functionality executed on separate thread(s).

sync()

```
RUIResult sync(boolean getReachOut)
```

Parameters

The `sync()` function has the following parameters.

Table 4-8 • sync() Parameters

Parameter	Description
`getReachout (boolean)`	This optional parameter instructs the server whether to send a ReachOut message during this particular sync if available.

Returns

The `sync()` function returns one of the return status constants below.

Table 4-9 • sync() Returns

Return	Description
`OK`	Synchronous functionality successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.

Table 4-9 • sync() Returns

Return	Description
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_NOT_STARTED	SDK has not been successfully started.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
TIME_THRESHOLD_NOT_REACHED	The API call frequency threshold (set by the Server) has not been reached.

5

Feature / Event Tracking

Through event tracking, Usage Intelligence allows you keep track of how your clients are interacting with the various features within your application, potentially identifying how often every single feature is being used by various user groups. Apart from monitoring feature usage, you can also keep track of how often an event happens - such as how often an auto save has been made on average for every hour your application was running. This is accomplished through `trackEvent()`.

You may also keep a numeric value, a text value, or group of name-value pairs every time an event is reported. This can be used, for example in the case of `trackEventNumeric()`, to keep track of the length of time it took to save a file, or the file size that was saved, etc. These events can be recorded using the functions `trackEventNumeric()`, `trackEventText()`, and `trackEventCustom()` respectively.

Once event-related data has been collected, you will be able to identify trends of what features are most used during evaluation and whether this trend changes once users switch to a freeware or purchased license or once they update to a different version/product build. You will also be able to compare whether any UI tweaks in a particular version or build number had any effect on exposing a particular feature or whether changes in the actual functionality make a feature more or less popular with users. This tool provides excellent insight for A/B testing whereas you can compare the outcome from different builds to improve the end user experience.

Note • *Event Tracking should NOT be used to track the occurrence of exceptions since there is another specific API call for this purpose. If you need to track exceptions, refer to Exception Tracking.*

Tracking an Event

The `trackEvent()` feature logs a normal event with the supplied data.

`trackEvent()` can be called between `startSDK()` and `stopSDK()`, and can be called zero or more times.

`trackEvent()` can be called while a New Registration is being performed (`createConfig()`, `startSDK()`). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

`trackEvent()` is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

trackEvent()

```
RUIResult trackEvent(String eventCategory, String eventName)
RUIResult trackEvent(String eventCategory, String eventName, String sessionID)
```

Parameters

The `trackEvent()` function has the following parameters.

Table 5-1 • trackEvent() Parameters

Parameter	Description
`eventCategory (String)`	The name of the category that this event forms part of. This parameter is optional (send empty string if not required). Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.
`eventName (String)`	The name of the event to be tracked. Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.

Table 5-1 • trackEvent() Parameters

Parameter	Description
`sessionID (String)`	If multiple user sessions are supported within the application (`multiSessionEnabled = true`), this should contain the unique ID that refers to the user session in which the event occurred. It must be a current valid value used in `ruiStartSession`. If the application supports only a single session (`multiSessionEnabled = false`), then this value should be empty.

Returns

The `trackEvent()` function returns one of the return status constants below.

Table 5-2 • trackEvent() Returns

Return	Description
`OK`	Synchronous functionality successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`SDK_NOT_STARTED`	SDK has not been successfully started.
`FUNCTION_NOT_AVAIL`	Function is not available.
`INVALID_SESSION_ID_EXPECTED_NON_EMPTY`	The `sessionID` is expected to be non-empty, and it was not.
`INVALID_SESSION_ID_TOO_SHORT`	The `sessionID` violates its allowable minimum length.
`INVALID_SESSION_ID_TOO_LONG`	The `sessionID` violates its allowable maximum length.
`INVALID_SESSION_ID_NOT_ACTIVE`	The `sessionID` is not currently in use.
`INVALID_PARAMETER_EXPECTED_NON_EMPTY`	Some API parameter is expected to be non-empty, and is not.

Logging a Normal Event with a Numeric Field

The `trackEventNumeric()` function logs a normal event with the supplied data, including a custom numeric field.

The `trackEventNumeric()` function can be called between `startSDK()` and `stopSDK()`, and can be called zero or more times.

The `trackEventNumeric()` function can be called while a New Registration is being performed (`createConfig()`, `startSDK()`). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

The `trackEventNumeric()` function is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

trackEventNumeric()

```
RUIResult trackEventNumeric(String eventCategory, String eventName, double customValue)
RUIResult trackEventNumeric(String eventCategory, String eventName, double customValue,
String sessionID)
```

Parameters

The `trackEventNumeric()` function has the following parameters.

Table 5-3 • trackEventNumeric() Parameters

Parameter	Description
`eventCategory (String)`	The name of the category that this event forms part of. This parameter is optional (send empty string if not required). Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.

Table 5-3 • trackEventNumeric() Parameters

Parameter	Description
`eventName (String)`	The name of the event to be tracked. Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.
`customValue (double)`	A numeric custom value related to this particular event.
`sessionID (String)`	If multiple user sessions are supported within the application (`multiSessionEnabled = true`), this should contain the unique ID that refers to the user session in which the event occurred. It must be a current valid value used in `ruiStartSession`. If the application supports only a single session (`multiSessionEnabled = false`), then this value should be empty.

Returns

The `trackEventNumeric()` function returns one of the return status constants below.

Table 5-4 • trackEventNumeric() Returns

Return	Description
`OK`	Synchronous functionality successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`CONFIG_NOT_CREATED`	Configuration has not been successfully created.

Table 5-4 • trackEventNumeric() Returns

Return	Description
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
SDK_NOT_STARTED	SDK has not been successfully started.
FUNCTION_NOT_AVAIL	Function is not available.
INVALID_SESSION_ID_EXPECTED_NON_EMPTY	The sessionID is expected to be non-empty, and it was not.
INVALID_SESSION_ID_TOO_SHORT	The sessionID violates its allowable minimum length.
INVALID_SESSION_ID_TOO_LONG	The sessionID violates its allowable maximum length.
INVALID_PARAMETER_EXPECTED_NON_EMPTY	Some API parameter is expected to be non-empty, and is not.
INVALID_SESSION_ID_NOT_ACTIVE	The sessionID is not currently in use.

Logging a Normal Event with a String Field

The `trackEventText()` function logs a normal event with the supplied data, including a custom string field.

The `trackEventText()` function can be called between `startSDK()` and `stopSDK()`, and can be called zero or more times.

The `trackEventText()` function can be called while a New Registration is being performed (`createConfig()`, `startSDK()`). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

The `trackEventText()` function is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

trackEventText()

```
RUIResult trackEventText(String eventCategory, String eventName, String customValue)
RUIResult trackEventText(String eventCategory, String eventName, String customValue, String sessionID)
```

Parameters

The `trackEventText()` function has the following parameters.

Table 5-5 • trackEventText() Parameters

Parameter	Description
`eventCategory (String)`	The name of the category that this event forms part of. This parameter is optional (send empty string if not required). Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.
`eventName (String)`	The name of the event to be tracked. Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.
`customValue (String)`	Custom text data associated with the event, cannot be empty. Trimmed to a maximum length determined by the Server. Current default maximum is 4096 UTF-8 characters.
`sessionID (String)`	If multiple user sessions are supported within the application (`multiSessionEnabled = true`), this should contain the unique ID that refers to the user session in which the event occurred. It must be a current valid value used in `ruiStartSession`. If the application supports only a single session (`multiSessionEnabled = false`), then this value should be empty.

Returns

The `trackEventText()` function returns one of the return status constants below.

Table 5-6 • trackEventText() Returns

Return	Description
OK	Synchronous functionality successful.
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_NOT_STARTED	SDK has not been successfully started.
FUNCTION_NOT_AVAIL	Function is not available.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
INVALID_SESSION_ID_EXPECTED_NON_EMPTY	The `sessionID` is expected to be non-empty, and it was not.
INVALID_SESSION_ID_TOO_SHORT	The `sessionID` violates its allowable minimum length.
INVALID_SESSION_ID_TOO_LONG	The `sessionID` violates its allowable maximum length.
INVALID_SESSION_ID_NOT_ACTIVE	The `sessionID` is not currently in use.
INVALID_PARAMETER_EXPECTED_NON_EMPTY	Some API parameter is expected to be non-empty, and is not.

Logging a Custom Event

The `trackEventCustom()` function logs a normal event with the supplied data, including an array of custom name/value pairs.

The `trackEventCustom()` function can be called between `startSDK()` and `stopSDK()`, and can be called zero or more times.

The `trackEventCustom()` function can be called while a New Registration is being performed (`createConfig()`, `startSDK()`). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

The `trackEventCustom()` function is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

Note • *Custom data will be logged in the format `(Key1,Value1)&&(Key2,Value2)...&&(KeyN,ValueN)`.*

trackEventCustom()

```
RUIResult trackEventCustom(String eventCategory, String eventName, List<RUINameValuePair> customValues)
RUIResult trackEventCustom(String eventCategory, String eventName, List<RUINameValuePair> customValues,
String sessionID)
```

Parameters

The `trackEventCustom()` function has the following parameters.

Table 5-7 • trackEventCustom() Parameters

Parameter	Description
`eventCategory (String)`	The name of the category that this event forms part of. This parameter is optional (send empty string if not required). Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.

Table 5-7 • trackEventCustom() Parameters

Parameter	Description
`eventName (String)`	The name of the event to be tracked. Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.
`customValues (List<RUINameValuePair>)`	A List of name/value String pairs related to this particular event. The `RUINameValuePair` class has two members: • **public final String name**—Custom data associated with the event. A given name and/or value can be empty. A given name cannot contain white space. All names and values are trimmed to a maximum length configured on the RUI Server. Both names and values have a default maximum of 128 UTF-8 characters. • **public final String value**—Custom text data associated with the event, cannot be empty. Trimmed to a maximum length determined by the RUI Server. Both names and values have a default maximum of 128 UTF-8 characters.
`sessionID (String)`	If multiple user sessions are supported within the application (`multiSessionEnabled = true`), this should contain the unique ID that refers to the user session in which the event occurred. It must be a current valid value used in `ruiStartSession`. If the application supports only a single session (`multiSessionEnabled = false`), then this value should be empty.

Returns

The `trackEventCustom()` function returns one of the return status constants below.

Table 5-8 • ruiTrackEventCustom() Returns

Return	Description
`OK`	Synchronous functionality successful.

Table 5-8 • ruiTrackEventCustom() Returns

Return	Description
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
SDK_NOT_STARTED	SDK has not been successfully started.
FUNCTION_NOT_AVAIL	Function is not available.
INVALID_SESSION_ID_EXPECTED_NON_EMPTY	The `sessionID` is expected to be non-empty, and it was not.
INVALID_SESSION_ID_TOO_SHORT	The `sessionID` violates its allowable minimum length.
INVALID_SESSION_ID_TOO_LONG	The `sessionID` violates its allowable maximum length.
INVALID_PARAMETER_EXPECTED_NON_EMPTY	Some API parameter is expected to be non-empty, and is not.
INVALID_SESSION_ID_NOT_ACTIVE	The `sessionID` is not currently in use.

6

ReachOut Direct-to-Desktop Messaging Service

From the Usage Intelligence dashboard, you can create ReachOut messaging campaigns that are used to deliver messages or surveys directly to the desktop of users who are running your software. You may choose a specific target audience for your message by defining a set of delivery filters so that each message will be delivered only to those users who match the specified criteria (such as geographical region, edition, version, build, language, OS, license status, runtime duration, days since install, etc.)

When building a ReachOut campaign you can choose between two message delivery options.

- **Automated HTML pop-up messages** (handled entirely by the Usage Intelligence library and requires absolutely NO coding.) For more information, see Automated Message Retrieval.

  Important • *Currently this is only available on Windows and macOS. It is not available in the Java SDK.*

- **Manually retrieving the message** (plain text or URL) through code by using the `checkForReachOut()` or `checkForReachOutOfType()` functions. For more information, see Manual Message Retrieval.

Automated Message Retrieval

The Usage Intelligence V5 SDK provides a default, automated ReachOut handler that works on Windows and macOS. The Java SDK does not have a default ReachOut handler. Developers can override this handler by implementing the ReachOut handler functions with their own code and providing this handler to the `setReachOutHandler()` function after the creation of the RUISDK object. See the `RUIReachOutHandler` interface class for more details. In brief, the `RUIReachOutHandler` interface contains three methods that must be implemented:

Table 6-1 • Functions Required to Support Custom ReachOut

Function	Description
`public void handle(String width, String height, int position, String message)`	Responsible for handling the display of the ReachOut message. The parameters are: • **width**—Width of the window as configured in the ReachOut campaign. Value will be suffixed by P for pixels or % for percentage. • **height**—Height of the window as configured in the ReachOut campaign. Value will be suffixed by P for pixels or % for percentage. • **position**—Position where the window should be displayed (1 = top-left, 2 = top-center, 3 = top-right, 4 = middle-left, 5 = middle-center, 6 = middle right, 7 = bottom-left, 8 = bottom-center, 9 = bottom-right). • **message**—URL to display.
`public boolean readyForNext()`	Called by the SDK to determine if the handler is ready for the next ReachOut Message. A false return means not ready; true means ready.
`public void close()`	Called by the SDK on stop or shutdown.

The `setReachOutHandler()` function sets a custom ReachOut handler. Any previously registered handler, including the default graphical ReachOut handler that may have been registered (`SDKImpl()`). If handler is NULL, then all parameters are considered to be NULL. Setting a handler to NULL effectively removes the current handler, if any, without setting a new handler.

`setReachOutHandler()` can be called more than once.

`setReachOutHandler()` is a synchronous function, returning when all functionality is completed.

setReachOutHandler()

```
RUIResult setReachOutHandler(RUIReachOutHandler handler)
```

Parameters

The `setReachOutHandler()` function has the following parameters.

Table 6-2 • setReachOutHandler() Parameters

Parameter	Description
`handler (RUIReachOUtHandler)`	An instance implementing the `RUIReachOutHandler` interface.

Returns

The `setReachOutHandler()` function returns a `RUIResult` enum value with the following possible values

Table 6-3 • setReachOutHandler() Returns

Parameter	Description
`OK`	Function successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`SDK_ALREADY_STARTED`	The SDK has already been successfully started.
`SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.

Manual Message Retrieval

When you want full control on when and where in your application to display a ReachOut message to your users, you can define ReachOut messages of the type plain text or URL. From within the application call one of the below functions to check with the Usage Intelligence server whether there are any pending messages (of this type) waiting to be delivered.

You may choose to display plain text messages anywhere in the application such as in a status bar or information box. URL type messages can either be opened in a browser or else rendered it in a HTML previewer embedded within the application.

The difference between `checkForReachOut()` and `checkForReachOutOfType()` is that `checkForReachOut()` takes an 'empty' `messageType` parameter and fills it with the type of message that is sent by the server. In the case of `checkForReachOutOfType()`, the message type is specified by the developer and the server would then only send messages of that type.

The message type (plain text or URL) can be one of the types from the following `RUIMessageType` enum:

```
ANY  (0)
TEXT (1)
URL  (2)
```

For more information, see the following:

- Checking for Manual ReachOut Messages of Any Type
- Checking for Manual ReachOut Messages of a Specified Type

Checking for Manual ReachOut Messages of Any Type

The `checkForReachOut()` function explicitly checks for manual ReachOut messages on the Server. `checkForReachOut()` will check for any manual ReachOut message type, whereas `checkForReachOutOfType()` will check for ReachOut messages of a specified type.

Note • *`MutableInt` and `MutableObject` are from `org.apache.commons.lang.mutable`.*

`checkForReachOut()` can be called between `startSDK()` and `stopSDK()`, and can be called zero or more times.

`checkForReachOut()` is a synchronous function, returning when all functionality is completed.

checkForReachOut()

```
RUIResult checkForReachOut(MutableObject<String> message, MutableInt messageCount,
MutableObject<RUIMessageType> messageType)
```

Parameters

The `checkForReachOut()` function has the following parameters.

Table 6-4 • checkForReachOut() Parameters

Parameter	Description
`message (MutableObject<String>)`	A string of maximum length 256 that will be filled-in by the SDK with the message that is sent by the server.
`messageCount (MutableInt)`	Receives the message count (including returned message).
`messageType (MutableObject<RUIMessageType>)`	Receives the type of returned message. Can be either `RUIMessageType.TEXT` or `RUIMessageType.URL`.

Returns

The `checkForReachOut()` function returns one of the return status constants below.

Table 6-5 • checkForReachOut() Returns

Return	Description
`OK`	Function successful.

Table 6-5 • checkForReachOut() Returns

Return	Description
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. stopSDK() is possible.
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_NOT_STARTED	SDK has not been successfully started.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
TIME_THRESHOLD_NOT_REACHED	The API call frequency threshold (set by the Server) has not been reached.
NETWORK_CONNECTION_ERROR	Not able to reach the Server.
NETWORK_SERVER_ERROR	Error while communicating with the Server.
NETWORK_RESPONSE_INVALID	Message format error while communicating with the Server.

Code Example

The the following is an example to get all of the messages on the server, check if it is a text or URL message and display the appropriate message box.

```
// Create instance
boolean registerDefaultReachOut = false;  // No default ReachOut
RUISDK mySDK = new SDKImpl(registerDefaultReachOut);
// Other initialization including application specific ReachOut handler

MutableObject<String> message = new MutableObject<>("");
MutableObject<RUIMessageType> msgType = new MutableObject<>(RUIMessageType.ANY);
MutableInt msgCount = new MutableInt(0);

while (mySDK.checkForReachOut(message, msgCount, msgType) == RUIResult.OK &&
       msgCount > 0) {
                MutableObject<RUIMessageType> textMsg = new MutableObject<>(RUIMessageType.TEXT);
                MutableObject<RUIMessageTYpe> urlMsg = new MutableObject<>(RUIMessageType.URL);
    if (msgType.equals(testMsg)) {
        //  code to display Text message
    } else if (msgType.equals(urlMsg)){
        // code to display URL
    }
}
```

Checking for Manual ReachOut Messages of a Specified Type

The `checkForReachOutOfType()` function explicitly checks for manual ReachOut messages on the Server. `checkForReachOutOfType()` will check for ReachOut messages of a specified type, whereas `checkForReachOut()` will check for any manual ReachOut messages of any type.

Note • *`MutableInt` and `MutableObject` are from `org.apache.commons.lang.mutable`.*

`checkForReachOutOfType()` can be called between `startSDK()` and `stopSDK()`, and can be called zero or more times.

`checkForReachOutOfType()` is a synchronous function, returning when all functionality is completed.

checkForReachOutOfType()

```
RUIResult checkForReachOutOfType(MutableObject<String> message, MutableInt messageCount,
MutableObject<RUIMessageType> messageTypeExpected)
```

Parameters

The `checkForReachOutOfType()` function has the following parameters.

Table 6-6 • checkForReachOutOfType() Parameters

Parameter	Description
`message (MutableObject<String>)`	A string of maximum length 256 that will be filled-in by the SDK with the message that is sent by the server.
`messageCount (MutableInt)`	Receives the message count (including returned message).
`messageType (MutableObject<RUIMessageType>)`	This value is filled by the developer and should contain the type of message that is being requested. This must be one of the `RUIMessageType` enum values.

Returns

The `checkForReachOutOfType()` function returns a `RUIResult` enum value with the following possible values

Table 6-7 • checkForReachOutOfType() Returns

Return	Description
`OK`	Function successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.

Table 6-7 • checkForReachOutOfType() Returns

Return	Description
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
SDK_NOT_STARTED	SDK has not been successfully started.
INVALID_MESSAGE_TYPE	The messageType is not an allowable value.
TIME_THRESHOLD_NOT_REACHED	The API call frequency threshold (set by the Server) has not been reached.
NETWORK_CONNECT_ERROR	Not able to reach the Server.
NETWORK_SERVER_ERROR	Error while communicating with the Server.
NETWORK_RESPONSE_INVALID	Message format error while communicating with the Server.

Code Example

The following is an example to get all text messages from the server and display them inside of a message box.

```
boolean registerDefaultReachOut = false;  // Java SDK has no default handler
RUISDK mySDK = new SDKImpl(registerDefaultReachOut);
// Other initialization.....

MutableObject<String> = new MutableObject<>("");
MultableInt msgCount = new MutableInt(0);
MutableObject<RUIMessageType> msgType = new MutableObject<>(RUIMessageType.TEXT);

while (mySDK.checkForReachOutOfType(message, msgCount, msgType) == RUIResult.OK &&
       msgCount > 0) {
    // code to display TEXT message
}
```

7

Exception Tracking

Usage Intelligence is able to collect runtime exceptions from your application and then produce reports on the exceptions that were collected. Once an exception is tracked, Usage Intelligence will also save a snapshot of the current machine architecture so that you can later (through the on-line exception browser within the Usage Intelligence dashboard) investigate the exception details and pinpoint any specific OS or architecture related information that are the cause of common exceptions. Collection of exception data is done through the `trackException()` method.

The `trackException()` function logs an exception event with the supplied data.

`trackException()` can be called between `startSDK()` and `stopSDK()`, and can be called zero or more times.

`trackException()` can be called while a New Registration is being performed (`createConfig()`, `startSDK()`). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

`trackException()` is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

trackException()

```
RUIResult trackException(RUIExceptionEvent exceptionData)¶
RUIResult trackException(RUIExceptionEvent exceptionData, String sessionID)
```

Parameters

The `trackException()` function has the following parameters.

Table 7-1 • trackException() Parameters

Parameter	Description
`exceptionData (RUIExceptionEvent)`	The exception data to log: • `className`—Class catching exception. • `methodName`—Method catching exception. • `exceptionMessage`—Exception message. No content restrictions, but trimmed to a maximum length determined by the Server. • `stackTrace`—Exception stack trace. No content restrictions, but trimmed to a maximum length determined by the Server. The content of `exceptionData.className` and `exceptionData.methodName` are conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (‘ ‘) are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—Cannot be empty.
`sessionID (String)`	An optional session ID. If not used, use method without `sessionID`. Different than V4 of the Usage Intelligence (Trackerbird) SDK, `trackException()` accepts a `sessionID` parameter. The usage requirements of the `sessionID` parameter are the following: • If multiple user sessions are supported within the application (`multiSessionEnabled = true`), `sessionID` must be a current valid value used in `startSession()`, or it can be empty. This is different than normal event tracking APIs, whereby an empty value is not allowed. • If the application supports only a single session (`multiSessionEnabled = false`), then sessionID must be empty.

Returns

The `trackException()` function returns one of the following RUIState enum values:

Table 7-2 • trackException() Returns

Return	Description
`OK`	Synchronous functionality successful.

Table 7-2 • trackException() Returns

Return	Description
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. stopSDK() is possible.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
SDK_NOT_STARTED	SDK has not been successfully started.
FUNCTION_NOT_AVAIL	Function is not available.
INVALID_SESSION_ID_EXPECTED_NON_EMPTY	The sessionID is expected to be empty, and it was not.
INVALID_SESSION_ID_TOO_SHORT	The sessionID violates its allowable minimum length.
INVALID_SESSION_ID_TOO_LONG	The sessionID violates its allowable maximum length.
INVALID_SESSION_ID_NOT_ACTIVE	The sessionID is not currently in use.

Code Example

The following is an example of placing ExceptionTrack inside of a try catch statement so that Usage Intelligence can record it.

```
// Create instance
boolean registerDefaultReachOut = false;  // No default ReachOut handler for Java SDK
mySDK = new SDKImpl(registerDefaultReachOut);
// Other initialization.....

private void btnSave_Click()
{
    try
    {
        //Save Button Logic
    }
    catch (Exception ex)
    {
        RUIExceptionEvent myRUIExceptionData = new RUIExceptionEvent("MyApplication", "btnSave_Click",
ex.getMessage(), ex.getStackTrace().toString());
        mySDK.trackException(myRUIExceptionData);
```

```
        }
    }
```

8

License Management

Usage Intelligence allows you to maintain your own license key registry on the Usage Intelligence server in order to track license key usage and verify the status/validity of license keys used on your clients.

There are multiple ways that the key registry is populated with license keys:

- Keys are collected automatically from your clients whenever you call the `setLicenseKey()` function.
- You can add/edit keys manually via the Usage Intelligence dashboard.
- You can add/edit keys directly from your CRM by using the Usage Intelligence Web API.

For more information, see:

- Client vs. Server Managed Licensing
- Checking the License Data of the Supplied License Key
- Setting the Current License to the Supplied License Key

Client vs. Server Managed Licensing

Usage Intelligence gives you the option to choose between managing your license key status (i.e. Blacklisted, Whitelisted, Expired or Activated) and key type on the server (server managed) or managing this status through the application (client managed). Applications can individually set whether each license status or license type is either Sever Managed or Client Managed by visiting the **License Key Management Settings** page on the Usage Intelligence dashboard. The major difference is outlined below:

Client Managed

The server licensing mechanism works in reporting-only mode and your application is expected to notify the server that the license status has changed through the use of `setLicenseData()`.

When to Use

Use client managed when you have implemented your own licensing module/mechanism within your application that can identify whether the license key used by this client is blacklisted, whitelisted, expired or activated. In this case you do not need to query the Usage Intelligence server to get this license status. However you can simply use this function to passively inform Usage Intelligence about the license status used by the client. In this case:

- Usage Intelligence will use this info to filter and report the different key types and statuses and their activity.
- Usage Intelligence licensing server will operate in passive mode (i.e. reporting only).
- Calling `checkLicenseKey()` will return the license type and flags as Unknown (-1).

Server Managed

You manage the key status on the server side and your application queries the server to determine the status of a particular license key by calling `checkLicenseKey()` or `setLicenseKey()`.

When to Use

Use server managed if you do not have your own licensing module/mechanism within your application and thus you have no way to identify the license status at the client side.

In this mode, whenever a client changes their license key your application can call `setLicenseKey()` to register the new license key. In reply to this API call, the server will check if the license key exists on the key register and in the reply it will specify to your application whether this key is flagged as blacklisted, whitelisted, expired or activated, along with the type of key submitted. If you want to verify a key without actually registering a key change for this client you can use `checkLicenseKey()` which returns the same values but does not register this key with the server. In this case:

- The key register is maintained manually on the server by the software owner
- Usage Intelligence licensing server will operate in active mode so apart from using this key info for filtering and reporting, it will also report back the key status (validity) to the SDK whenever requested through the API.
- Calling `checkLicenseKey()` or `setLicenseKey()` will return the 4 status flags denoting whether a registered key is: Blacklisted, Whitelisted, Expired and Activated and the key type.
- If the key does not exist on the server, all 4 status flags will be returned as false (0).

Checking the License Data of the Supplied License Key

The `checkLicenseKey()` function checks the Server for the license data for the supplied `licenseKey`. Whereas `checkLicenseKey()` is a passive check, `setLicenseKey()` changes the license key.

The `checkLicenseKey()` function can be called between `startSDK()` and `stopSDK()`, and can be called zero or more times.

The `checkLicenseKey()` function is a synchronous function returning when all functionality is completed.

checkLicenseKey()

```
RUIResult checkLicenseKey(String licenseKey, List<RUIKeyData> licenseArray)
```

Parameters

The `checkLicenseKey()` function has the following parameters.

Table 8-1 • checkLicenseKey() Parameters

Parameter	Description

`licenseKey (String)`	The license key to be checked. This value cannot be empty.
	
	The function accepts a String parameter that is the license key itself a `List<RUIKeyData>` array of length 5 that it fills with the returned result. You may use the following constants to refer to the required value by its index from the `RUIKeyIndex` enum:
	
	`LICENSE_ARRAY_INDEX_KEY_TYPE (0)`
	`LICENSE_ARRAY_INDEX_KEY_EXPIRED (1)`
	`LICENSE_ARRAY_INDEX_KEY_ACTIVE (2)`
	`LICENSE_ARRAY_INDEX_KEY_BLACKLISTED (3)`
	`LICENSE_ARRAY_INDEX_KEY_WHITELISTED (4)`
	
	Each of the values 1 through 4 will be set to either 0 or 1 that refers to false or true respectively. The first value (`RUIKeyIndex.TYPE_INDEX`) will be set to a number between 0 and 7 (inclusive) that refers to the 8 possible license types listed below. The values may also be -1 that means “Unknown”. The following are the possible license types:
	
	`UNCHANGED (-1)`
	`EVALUATION (0)`
	`PURCHASED (1)`
	`REEWARE (2)`
	`UNKNOWN (3)`
	`NFR (4) - Key Type is Not For Resale`
	`CUSTOM1 (5)`
	`CUSTOM2 (6)`
	`CUSTOM3 (7)`
	
	The following are the possible key status values:
	
	• `UNCHANGED (-1)`—Key Status is Unchanged (when in parameter) or Unknown (when out parameter).
	
	• `NO (0)`—Key Status is No.
	
	• `YES (1)`—Key Status is Yes.

`licenseArray (List<RUIKeyData> length of 5)`	The vector that will be filled to contain the license status flags.

Returns

The `checkLicenseKey()` function returns one of the RUIState enum values below:

Table 8-2 • checkLicenseKey() Returns

Return	Description
OK	Function successful.
SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
SDK_SUSPENDED	The Server has instructed a temporary back-off.
SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
CONFIG_NOT_CREATED	Configuration has not been successfully created.
SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
SDK_NOT_STARTED	SDK has not been successfully started.
INVALID_PARAMETER_EXPECTED_NON_EMPTY	Some API parameter is expected to be non-empty, and is not.
TIME_THRESHOLD_NOT_REACHED	The API call frequency threshold (set by the Server) has not been reached.
NETWORK_CONNECT_ERROR	Not able to reach the Server.
NETWORK_SERVER_ERROR	Error while communicating with the Server.
NETWORK_RESPONSE_INVALID	Message format error while communicating with the Server.

Code Example

```
//Test a license key
mySDK.startSDK(); //...; //Creation and initialization shown in other snippets.
String myProductKey = "xyz";
List<RUIKeyData> licenseResult = new ArrayList<>(5);

RUIResult rc = mySDK.checkLicenseKey(myProductKey, licenseResult);
if(rc == RUIResult.OK)
{
    if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_TYPE) == RUIKeyType.UNCHANGED) {
        System.out.println("License Key is unchanged");
    } else {
```

```
        String myType = "License type = " +
licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_TYPE).getId();
        System.out.println(myType);
    }
    //Check if the license key is activated
    if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_ACTIVE) == RUIKeyStatus.YES){
        System.out.println("License Active");
    } else if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_ACTIVE) == RUIKeyStatus.NO) {
        System.out.println("License Inactive");
    } else {
        System.out.println("License status unknown");
    }

    //check if license key is blacklisted
    if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_BLACKLISTED) == RUIKeyStatus.YES){
        System.out.println("Key is black listed");
    } else if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_BLACKLISTED) == RUIKeyStatus.NO) {
        System.out.println("Key is NOT black listed");
    } else {
        System.out.println("Key blank listed status is unknown");
    }

    //Check if license key is expired
    if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_EXPIRED) == RUIKeyStatus.YES){
        System.out.println("Key is expired");
    } else if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_EXPIRED) == RUIKeyStatus.NO) {
        System.out.println("Key is NOT expired");
    } else {
        System.out.println("Key expiration status is unknown");
    }

    //Check if license key is white listed
    if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_WHITELISTED) == RUIKeyStatus.YES){
        System.out.println("Key is white listed");
    } else if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_WHITELISTED) == RUIKeyStatus.NO) {
        System.out.println("Key is NOT white listed");
    } else {
        System.out.println("Key white listed status is unknown");
    }
} else {
    System.out.println("Failed to invoke function checkLicenseKey()");
}
```

Setting the Current License to the Supplied License Key

The `setLicenseKey()` function checks the Server for the license data for the supplied `licenseKey` and sets the current license to `licenseKey`.

Whereas `checkLicenseKey()` is a passive check, `setLicenseKey()` changes the license key. The Server always registers the `licenseKey` even if the Server knows nothing about the `licenseKey`.

When a new (unknown) `licenseKey` is registered, the Server sets the license data to `RUIKeyType.unknown` and the four status flags (blacklisted, whitelisted, expired, activated) to `RUIKeyStatus.no`. The license array has size, indexes and values.

The order of the license array data has changed from the Usage Intelligence SDK V4.

The `setLicenseKey()` function can be called between `startSDK()` and `stopSDK()`, and can be called zero or more times.

The `setLicenseKey()` function is primarily a synchronous function, returning once the check with Server has completed. Some post- processing functionality is performed asynchronously, executed on separate thread(s).

The `setLicenseKey()` function should be called when an end user is trying to enter a new license key into your application and you would like to confirm that the key is in fact valid (i.e. blacklisted or whitelisted), active, or expired. The function is very similar to the `checkLicenseKey()` function, however rather than just being a passive license check, it also registers the new key with the server and associates it with this particular client installation.

setLicenseKey()

```
RUIResult setLicenseKey(String newKey, List<RUIKeyData> licenseArray)
RUIResult setLicenseKey(String newKey, List<RUIKeyData> licenseArray, String SessionID)
```

Parameters

The `setLicenseKey()` function has the following parameters.

Table 8-3 • ruiSetLicenseKey() Parameters

Parameter	Description

`licenseKey (String)`	The license key to be checked. This value cannot be empty.
	
	The function accepts a String parameter that is the license key itself and a `List<RUIKeyData>` array of length 5 that it fills with the returned result. You may use the following constants to refer to the required value by its index from the `RUIKeyIndex` enum:
	
	`LICENSE_ARRAY_INDEX_KEY_TYPE (0)`
	`LICENSE_ARRAY_INDEX_KEY_EXPIRED (1)`
	`LICENSE_ARRAY_INDEX_KEY_ACTIVE (2)`
	`LICENSE_ARRAY_INDEX_KEY_BLACKLISTED (3)`
	`LICENSE_ARRAY_INDEX_KEY_WHITELISTED (4)`
	
	Each of the values 1 through 4 will be set to either 0 or 1 that refers to false or true respectively. The first value (`RUIKeyIndex.typeIndex`) will be set to a number between 0 and 7 (inclusive) that refers to the 8 possible license types listed below. The values may also be -1 that means “Unknown”. The following are the possible license types from the `RUIKeyType` enum:
	
	`UNCHANGED (-1)`
	`EVALUATION (0)`
	`PURCHASED (1)`
	`FREEWARE (2)`
	`UNKNOWN (3)`
	`NFR (4) - Key Type is Not For Resale`
	`CUSTOM1 (5)`
	`CUSTOM2 (6)`
	`CUSTOM3 (7)`
	
	The following are the possible key status values from the `RUIKeyStatus` enum:
	
	• `UNCHANGED (-1)`—Key Status is Unchanged (when in parameter) or Unknown (when out parameter).
	• `NO (0)`—Key Status is No.
	• `YES (1)`—Key Status is Yes.

`licenseArray (List<int> of length 5)`	The vector that will be filled to contain the license status flags.

Table 8-3 • ruiSetLicenseKey() Parameters

Parameter	Description
`sessionID (String)`	An optional session ID complying with above usage (content conditioning and validation rules in `startSession()`). Different from the V4 of the Usage Intelligence SDK, a `sessionID` parameter can be supplied (based on `createConfig()` multi session value): • **If multiSessionEnabled is set to false**—`sessionID` must be empty. This is similar to event tracking APIs. • **If multiSessionEnabled is set to true**—`sessionID` must be a current valid value used in `startSession()`, or it can be empty. This is different than normal event tracking APIs, whereby a empty value is not be allowed.

Returns

The `setLicenseKey()` function returns one of the `RUIState` enum values below:

Table 8-4 • setLicenseKey() Returns

Return	Description
`OK`	Function successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`SDK_NOT_STARTED`	SDK has not been successfully started.
`INVALID_PARAMETER_EXPECTED_NON_EMPTY`	Some API parameter is expected to be non-empty, and is not.
`TIME_THRESHOLD_NOT_REACHED`	The API call frequency threshold (set by the Server) has not been reached.
`NETWORK_CONNECT_ERROR`	Not able to reach the Server.
`NETWORK_SERVER_ERROR`	Error while communicating with the Server.

Table 8-4 • setLicenseKey() Returns

Return	Description
NETWORK_RESPONSE_INVALID	Message format error while communicating with the Server.

Code Example

```
//Register a new license key
booleanuseDefaultReachOutHandler = false;
RUISDK mySDK = new SDKImpl(useDefaultReachOutHandler); //...; //Creation and initialization shown in
other snippets.
String myProductKey = "xyz";
List<RUIKeyData> licenseResult = new ArrayList<>(5);

RUIResult rc = mySDK.setLicenseKey(myProductKey, licenseResult);
if(rc == RUIResult.OK)
{
    if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_TYPE) == RUIKeyType.UNCHANGED) {
        System.out.println("License Key is unchanged");
    } else {
        String myType = "License type = " +
licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_TYPE).getId();
        System.out.println(myType);
    }
    //Check if the license key is activated
    if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_ACTIVE) == RUIKeyStatus.YES){
        System.out.println("License Active");
    } else if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_ACTIVE) == RUIKeyStatus.NO) {
        System.out.println("License Inactive");
    } else {
        System.out.println("License status unknown");
    }

    //check if license key is blacklisted
    if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_BLACKLISTED) == RUIKeyStatus.YES){
        System.out.println("Key is black listed");
    } else if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_BLACKLISTED) == RUIKeyStatus.NO) {
        System.out.println("Key is NOT black listed");
    } else {
        System.out.println("Key blank listed status is unknown");
    }

    //Check if license key is expired
    if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_EXPIRED) == RUIKeyStatus.YES){
        System.out.println("Key is expired");
    } else if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_EXPIRED) == RUIKeyStatus.NO) {
        System.out.println("Key is NOT expired");
    } else {
        System.out.println("Key expiration status is unknown");
    }

    //Check if license key is white listed
    if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_WHITELISTED) == RUIKeyStatus.YES){
        System.out.println("Key is white listed");
    } else if (licenseResult.get(RUISDK.LICENSE_ARRAY_INDEX_KEY_WHITELISTED) == RUIKeyStatus.NO) {
```

```
            System.out.println("Key is NOT white listed");
        } else {
            System.out.println("Key white listed status is unknown");
        }
    } else {
        System.out.println("Failed to invoke function setLicenseKey()");
    }
```

9

Custom Properties

Apart from the pre-set values that Usage Intelligence collects, such as OS version, product version, edition, language, and license type, you also have the ability to collect any custom value that is relevant to your specific application.

Typical examples where you can benefit from custom properties include storing the download source or marketing campaign from where the user downloaded your software or some other status in your application. These custom properties will then be available inside the filters panel on every report so you may use them as part of your report filtering criteria.

Custom properties are intended to store values that are not very dynamic for a particular installation since the reporting granularity provided by Usage Intelligence is on a daily basis. This means if you use this API to register multiple values inside the same custom property (for the same user), Usage Intelligence will only store the latest known property value for that user on that particular day.

The `setCustomProperty()` function sets the value of a custom property. For more information, see Setting Custom Property Data.

Setting Custom Property Data

The `setCustomProperty()` function sets the value of a custom property.

setCustomProperty()

```
RUIResult setCustomProperty(int customPropertyID, String customValue)
```

Parameters

The `setCustomProperty()` function has the following parameters.

Table 9-1 • setCustomProperty() Parameters

Parameter	Description
`customPropertyId (int)`	This is a numeric index between 1 and 20. On the Usage Intelligence dashboard, custom properties are given an ID ranging from C01 to C20. This ID is used to identify which of the 20 possible custom properties is being set.
`customValue (String)`	The custom property value to use in Server reports; null clears the value.

Returns

The `setCustomProperty()` function returns a `RUIResult` enum value with the following possible values:

Table 9-2 • setCustomProperty() Returns

Return	Description
`OK`	Function successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`INVALID_CUSTOM_PROPERTY_ID`	The 1-based `customPropertyID` violates its allowable range.

10

SDK Status Checks

You can perform SDK status checks using the `getState()` and `testConnection()` functions.

- Getting the State of the Usage Intelligence Instance
- Testing the Connection Between the SDK and the Server

Getting the State of the Usage Intelligence Instance

The `getState()` function returns the current state of the SDK instance. The SDK state can change asynchronously..

The `getState()` function can be called more than once.

The `getState()` function is a synchronous function, , returning when all functionality is completed.

getState()

```
RUIState getState()¶
```

Returns

The `getState()` function returns one of the `RUIState` enum values below:

Table 10-1 • getState() Returns

Return	Description
`FATAL_ERROR`	Irrecoverable internal fatal error. No further API calls should be made
`UNINITIALIZED`	Instance successfully created but not yet successfully configured (`createConfig()`).

Table 10-1 • getState() Returns

Return	Description
CONFIG_INITIALIZED_NOT_STARTED	Successfully configured (createConfig()) and not yet started (startSDK()). Will be normal start.
CONFIG_MISSING_OR_CORRUPT_NOT_STARTED	Successfully configured (createConfig()) and not yet started (startSDK()). Will be a New Registration start.
STARTED_NEW_REG_RUNNING	Running startSDK() with a New Registration in progress but not yet completed.
RUNNING	Running startSDK() with no need for New Registration or with successfully completed New Registration.
ABORTED_NEW_REG_PROXY_AUTH_FAILURE	Aborted run startSDK() due to failed proxy authentication on New Registration (createConfig()).
ABORTED_NEW_REG_NETWORK_FAILURE	Aborted run (startSDK()) due to failed New Registration (createConfig()).
ABORTED_NEW_REG_FAILED	Aborted run (startSDK()) due to failed New Registration (createConfig()).
SUSPENDED	Instance has been instructed by Server to back-off. Will return to running once back-off cleared.
PERMANENTLY_DISABLED	Instance has been instructed by Server to disable. This is a permanent, irrecoverable state.
OPTED_OUT	Instance has been instructed by the application to opt-out.
STOPPING_NON_SYNC	Stop in progress stopSDK(). Stopping non-Sync-related threads.
STOPPING_ALL	Stop in progress stopSDK(). Stopping Sync-related threads.
STOPPED	Stop completed stopSDK()

Testing the Connection Between the SDK and the Server

The testConnection() function tests the connection between the SDK and the Server. If a valid configuration file exists from createConfig(), the URL used for the test will be the one in that file, set by the Server. Otherwise, the URL used for the test will be the one set by the client in the call to createConfig() . The proxy is used during the test if set by calling setProxy().

This method allows you to test your application's connectivity with the Usage Intelligence server and to confirm that your CallHome URL is active and operational (for debugging purposes when using a custom CallHome URL). You do NOT need to call this method before other API calls since this would cause unnecessary traffic on your clients' machines. Instead, you should check the return types by each API call since every API call that requires server communication does its own connection status check and returns any connection errors as part of its return type.

The `testConnection()` functioncan be called between `createConfig()` and `stopSDK()` and can be called zero or more times.

The `testConnection()` function is a synchronous function and only returns with all functionality is completed.

testConnection()

```
RUIResult testConnection()¶
```

Returns

The `testConnection()` function returns one of the `RUIResult` enum values below::

Table 10-2 • testConnection() Returns

Return	Description
`OK`	Function successful.
`SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.
`SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`CONFIG_NOT_CREATED`	Configuration from `createConfig()` has not been successfully created.
`SDK_NOT_STARTED`	SDK has not been successfully started.
`SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`NETWORK_CONNECT_ERROR`	Not able to reach the Server.
`NETWORK_SERVER_ERROR`	Error while communicating with the Server.
`NETWORK_RESPONSE_INVALID`	Message format error while communicating with the Server.
`TEST_CONNECTION_INVALID_PRODUCT_ID`	Invalid Product ID.
`TEST_CONNECTION_MISMATCH`	Mismatch between Product ID and URL.

11

Common Function Return Values

Most methods return an enum value of type `RUIResult`. This is used to show whether the method call was accepted and executed successfully.

OK(0)

Function (which may be synchronous or asynchronous), fully successful during synchronous functionality.

SDK_INTERNAL_ERROR_FATAL(-999)

Irrecoverable internal fatal error. No further API calls should be made.

SDK_ABORTED(-998)

A required New Registration has failed, and hence the SDK is aborted. `stopSDK()` is possible.

INVALID_PARAMETER_EXPECTED_NON_NULL(-110)

Some API parameter is expected to be non-NULL, and is not.

INVALID_PARAMETER_EXPECTED_NON_EMPTY(-111)

Some API parameter is expected to be non-empty, and is not.

INVALID_PARAMETER_EXPECTED_NO_WHITESPACE(-113)

Some API parameter is expected to be free of white space, and is not.

INVALID_PARAMETER_TOO_LONG(-114)

Some API parameter violates its allowable maximum length.

INVALID_CONFIG_PATH(-120)

The `configFilePath` is not a well-formed directory name.

INVALID_CONFIG_PATH_NONEXISTENT_DIR(-121)

The `configFilePath` identifies a directory that does not exist.

INVALID_CONFIG_PATH_NOT_WRITABLE(-122)

The `configFilePath` identifies a directory that is not writable.

INVALID_PRODUCT_ID(-130)

The `productID` is not a well-formed Product ID.

INVALID_SERVER_URL(-140)

The serverURL is not a well-formed.

URL.INVALID_PROTOCOL (-144)

The protocol is not a legal value. Must be one of the following:

Table 11-1 • Protocol Values

Protocol	Description
`HTTP_PLUS_ENCRYPTION (1)`	Protocol to the Server is HTTP + AES-128 Encrypted payload.
`HTTPS_WITH_FALLBACK (2)`	Protocol to the Server is HTTPS, unless that doesn't work, falling back to HTTP + Encryption.
`HTTPS (3)`	Protocol to the Server is HTTPS with no fall-back.

INVALID_AES_KEY_EXPECTED_EMPTY(-145)

The AES Key is expected to be NULL/empty, and it is not. This occurs if `https` is used at the protocol selection and an AES Key is supplied.

INVALID_AES_KEY_LENGTH(-146)

The AES Key is not the expected length (32 hex chars). An AES key is required if using `HTTP_PLUS_ENCRYPTION` or `HTTPS_WITH_FALLBACK` as the protocol choice.

INVALID_AES_KEY_FORMAT(-147)

The AES Key is not valid hex encoding. String passed must only include hexadecimal characters.

INVALID_SESSION_ID_EXPECTED_EMPTY(-150)

The `sessionID` is expected to be empty, and it was not. This occurs if a session ID is passed to functions that accept a session ID but no `startSession()` is active.

INVALID_SESSION_ID_EXPECTED_NON_EMPTY(-151)

The `sessionID` is expected to be non-empty, and it was empty.

INVALID_SESSION_ID_TOO_SHORT(-152)

The `sessionID` violates its allowable minimum length. Minimum length is 10.

INVALID_SESSION_ID_TOO_LONG(-153)

The `sessionID` violates its allowable maximum length. Maximum length is 64.

INVALID_SESSION_ID_ALREADY_ACTIVE(-154)

The `sessionID` is already currently in use.

INVALID_SESSION_ID_NOT_ACTIVE(-155)

The `sessionID` is not currently in use.

INVALID_CUSTOM_PROPERTY_ID(-160)

The `customPropertyID` violates its allowable range. By default the range is 1 to 20.

INVALID_DO_SYNC_VALUE(-170)

The doSync manual sync flag/limit violates its allowable range.

INVALID_MESSAGE_TYPE(-180)

The `messageType` is not an allowable value.

INVALID_LICENSE_DATA_VALUE(-185)

The license data is not an allowable value.

INVALID_PROXY_CREDENTIALS(-190)

The proxy username and password failed proxy authentication.

INVALID_PROXY_PORT(-191)

The proxy port was not valid.

CONFIG_NOT_CREATED (-200)

Configuration has not been successfully created. The function `createConfig()` must be called before performing this operation.

CONFIG_ALREADY_CREATED (-201)

Configuration has already been successfully created. A previous `createConfig()` was successful and the subsequent calls to this function are not allowed.

SDK_NOT_STARTED (-210)

SDK has not been successfully started. The function `startSDK()` must be called before using this function.

SDK_ALREADY_STARTED (-211)

SDK has already been successfully started. A previous `startSDK()` was successful and subsequent calls to this function are not allowed.

SDK_ALREADY_STOPPED (-213)

SDK has already been successfully stopped. A previous `stopSDK()` was successful and subsequent calls to this function are not allowed.

FUNCTION_NOT_AVAIL (-300)

This indicates that this particular API call is not currently available. Possible causes include:

- **This feature is disabled from the server side**. If this is an optional feature you might need to turn it on from the Usage Intelligence dashboard.
- **You have called this function too many times** in quick succession from the same client. In order to prevent abuse the server might impose a minimum interval (time threshold) before you can call this function again. This interval can vary from seconds to minutes.
- **There has been a time out on this request** to the Usage Intelligence server.

SYNC_ALREADY_RUNNING (-310)

A sync with the Server is already running. Only one sync operation may be running at a time.

TIME_THRESHOLD_NOT_REACHED (-320)

The API call time frequency threshold (set by the Server) has not been reached. In other words, the application is generating too many requests per time period.

SDK_SUSPENDED (-330)

The Server has instructed a temporary back-off. No events are logged but future communication with the Server is possible if the server allows it.

SDK_PERMANENTLY_DISABLED (-331)

The Server has instructed a permanent disable. No communication with the server is possible and events will not be logged.

SDK_OPTED_OUT (-332)

Instance has been instructed by the application to opt-out.

NETWORK_CONNECTION_ERROR (-400)

Communication attempts were not able to reach the Server. This means there was a problem communicating with the Usage Intelligence server due to:

- Network connectivity problems
- Incorrect proxy settings
- HTTP or HTTPS traffic is blocked by a gateway or firewall

In some cases, you can use `testConnection()` to help diagnose the issue.

NETWORK_SERVER_ERROR (-401)

Error while communicating with the Server. Communication with the server was successful but the server response indicates a Server error.

Login to the Usage Intelligence dashboard to make sure your account is active and there are no critical warnings. Test using `testConnection()` function.

NETWORK_RESPONSE_INVALID (-402)

The response from the Server was returned with a message format error.

TEST_CONNECTION_INVALID_PRODUCT_ID (-420)

The `testConnection()` function had an invalid `ProductID` supplied. Check the Product ID provided to you for accuracy.

TEST_CONNECTION_MISMATCH (-421)

The `testConnection()` function had a mismatch between URL and Product ID. Check the Product ID and URL provided to you for accuracy.