Trackerbird
Software Analytics

API Reference Guide

.NET SDK

version 0.9 beta

Last updated: 11 March 2012

www.trackerbird.com

This document is a quick reference guide intended to help software developers integrate the Trackerbird .NET SDK with a .NET application. Before using this SDK you must first register a Trackerbird account by visiting http://register.trackerbird.com

You may also visit our support Helpdesk at: http://helpdesk.trackerbird.com

Contents

Introduction

Trackerbird Software Analytics is a cloud service that anonymously tracks installations and user activity of your software product. Once you register a product account on the trackerbird website, and integrate this SDK into your application, your product will call home to the Trackerbird servers everytime it is run. The Trackerbird server is powered by a unique and highly advanced analytics engine which crunches away on the collected data to provide you with real-time Business Intelligence reports. These reports can potentially answer a number of critical product management question and provide you with the added benefit of interactive drilldown filters, so that you can breakdown and analyze your data by various fields such as geographical region, edition, version, build, language, OS, license type, etc.

Trackerbird is also used as a powerful Sales and Marketing tool that can help you stay in sync with your users and boost your conversion trends by pushing direct-to-desktop messages or surveys to end-users running your software. This is achieved through ReachOut™ Message delivery framework.

Getting Started

Before you can use the Trackerbird Software Analytics service or integrate the Trackerbird SDK with your software, you must first register an account by visiting: http://register.trackerbird.com

Once you register a username and create a new product account for tracking your application, you can get your Product ID and callhome URL from the Developer Zone (within the login area). From here you can also download the latest version of the SDK.

When using the Trackerbird API there is one main class which you will have to call into that contains all of the major functions. This is the *TrackerBird.Tracker.App class*. All the methods inside this class are declared static so creating an instance of the class is NOT required.

Supported Platforms

The Trackerbird .NET SDK can be used with .NET Framework versions 2.0, 3.0, 3.5 and 4.0. Please note that the SDK makes use of system.web library which is not available on .NET 4 Client Profile, so in order to run on .NET 4 you must build your project using .NET 4 FULL framework.

App.Start

This method must be called before utilizing any of the other functionality of the Trackerbird SDK (except for *App.SetProxy* which is explained later on in the document).

The *App.Start* method has only one parameter which is an object of type *TBConfig*. This object is explained in detail in the next section, but for the most basic implementation of *App.Start* the constructor for the TBConfig class expects the following 4 required properties.

1. **Callhome-URL** : Every product registered with Trackerbird has its own unique callhome URL usually in the form '*http://xxxxx.tbnet1.com'*. This URL is generated automatically on account creation and is used by the SDK to communicate with the Trackerbird server. You can get this URL from the Developer Zone once you login to the customer area. If you have a Premium product account you may opt to use your own custom callhome URL (such as *http://updates.yourdomain.com)* which must be setup as a CNAME DNS entry pointing to your unique Trackerbird URL. Please note that before you can use your own custom URL you must first inform Trackebird support (support@trackerbird.com) to register your domain with the Trackerbird server. If you fail to do this, the server will automatically reject any incoming calls using *yourdomain.com* as a callhome URL.
2. **[Product-ID]**: This is a unique 10-digit Product ID number which identifies your product with the Trackerbird server. You can get this ID from the Developer Zone once you login to the customer area.
3. **[Product-Version]**: The version number of the application being run.
4. **[Product-Build-number]**: The build number of the application being run.

Code Example:

Create an instance of TBConfig

```
TBConfig tbConfig = new TBConfig("http://12345.tbnet1.com", "1234567890",
                                 "1.0.0.1", "10");
```

Then pass the instance to the *App.Start* method

```
App.Start(tbConfig);
```

Note on UAC: If you are using Trackerbird with MS Windows Vista or higher and your application runs out of the 'Program Files' directory, be sure to use the *SetFilePath* method of *TBConfig* in order to set the file path to a location where your application has write permissions. This will prevent issues with UAC which could block Trackerbird from saving log & configuration files to disk. In the example below we are instructing the SDK to store its files in C:\myappdata.

```
ITBConfig tbConfig = new TBConfig("http://12345.tbnet1.com", "1234567890", "1.0.0.1",
"10").SetFilePath(@"c:\myappdata");


App.Start(tbConfig);
```

Trackerbird SDK working folder: The first time your Trackerbird-enabled application is run, the 4 files below are created in the application folder (default) or in the folder you indicated when calling *SetFilePath*. The filenames are: tbinfo.xml, tbconfig.xml, tblog.log, tbdebug.log. The latter is only created when debugging is enabled.

Using Class TBConfig

The TBConfig class is used to set some basic information about your product as well as some settings for the Trackerbird SDK. You should always attempt to fill in as much accurate and specific details as possible inside the TBConfig object since this data will then be used by the Trackerbird 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 optoins will be available to you inside the Trackerbird Analytics portal.

This object utilizes a fluent interface (please refer to: http://en.wikipedia.org/wiki/Fluent_interface).

When creating an instance of the TBConfig class there is only one constructor which can be called and this requires four mandatory string parameters (outlined earlier in App.Start):

```
public TBConfig(string url, string productID,
                string productVersion, string productBuildNumber)
```

Besides the parameters inside the constructor, you can use the list of methods outlined below that will set additional (optional) information about your running application and its licensing state.

- **`ITBConfig SetProductEdition(string productEdition);`**

This method allows you to set the edition of your product. An example of this would be when a single product can be licensed/run in different modes such as as “Home” and “Business”.

- **`ITBConfig SetProductLanguage(string productLanguage);`**

This method allows you to set the language in which the client is viewing your product. This is useful for products which have been internationalized, so you can determine how many installations are running your software in a particular language. Please note this is different than the OS language which is collected automatically by the Trackerbird SDK. We suggest that if your product supports only a single language (such as English), then you simply call SetProductLanguage("English") rather than leaving this property undefined.

- **`ITBConfig SetKeyType(KeyType keyType);`**

Set the type of license key being used by the client. You can choose any of the enumerations below. Please note that the 3 custom values may be used freely to denote your own custom license types. From the online portal you can then apply friendly labels to these custom license types for better visualization inside reports.

```
public enum KeyType
    {
        Evaluation=0,
        Purchased,
        Freeware,
        Unknown,
        NFR,
        Custom1,
        Custom2,
        Custom3
    }
```

- **`ITBConfig SetKeyActivated(bool? keyActivated);`**

Set whether the license key being used by the client has been activated. Trackerbird does not place any restrictions on what you mean by an activated key. Therefore you may use this flag as you deem fit for your particular software.

- **`ITBConfig SetKeyBlacklisted(bool? keyBlacklisted);`**

Set whether the license key being used by the client has been blacklisted. This property is only used if your client software has its own means to check whether a key is blacklisted and you wish to inform the Trackerbird server that this client is using a blacklisted key (for reporting purposes only).

If on the otherhand, you choose to use the license key blacklist functionality on the Trackerbird server (i.e. you want to maintain a list of blacklisted keys on the Trackerbird server so your software can validate keys using *App.KeyCheck* API) then any value you save in this property will be ignored by the server.

- **`ITBConfig SetKeyExpired(bool? keyExpired);`**

Set if the license key being used is expired.

- **`ITBConfig SetKeyWhitelisted(bool? keyWhitelisted);`**

Set whether the license key being used by the client has been whitelisted. This property is only used if your client software has its own means to check whether a key is whitelisted and you wish to inform the Trackerbird server that this client is using a whitelisted key (for reporting purposes only).

If on the otherhand, you choose to use the license key whitelist functionality on the Trackerbird server (i.e. you want to maintain a list of whitelisted keys on the Trackerbird server so your software can validate keys using *App.KeyCheck* API) then any value you save in this property will be ignored by the server.

- **`ITBConfig SetPrivacyMode(TBPrivacyMode privacyMode);`**

Set the privacy level of the SDK, which basically determines what information should or should not be collected from the clients' machines. Please refer to the section *App.SetPrivacyMode* for more details on this property.

- **`ITBConfig SetFilePath(string filePath);`**

Set the file path where the Trackerbird SDK will create and save its working files. If this property is not set, the SDK will place the files in the same folder location from where the dll is being called. It is important to remember that the calling process should have read/write accessibility to the location.

For practicality, you can use all of the methods together in one simple call as follows:

```
ITBConfig tbc = new TBConfig("URL", "ProductID", "10", "45")
.SetFilePath(@"C:\temp\tb").SetKeyActivated(true).SetKeyBlacklisted(false)
.SetKeyExpired(false).SetKeyWhitelisted(false).SetKeyType(KeyType.Purchased)
.SetPrivacyMode(TBPrivacyMode.Off).SetProductEdition("Basic")
.SetProductLanguage("English");
```

The above call would set the following properties that will be reported to the Trackerbird Server:

- File path - C:\temp\tb
- Key Activated
- Key NOT blacklisted
- Key NOT expired
- Key NOT whitelisted
- Key Type = purchased
- Privacy Mode = Off
- Edition = Basic
- Product Language = English

App.Stop

The *App.Stop* method should always be called on the exit point of your application. This will help the Trackerbird server keep track of when your application was closed to accurately calculate session runtime duration and provide you with reports based on application usage statistics. Calling *App.Stop* before your application terminates will also tell the Trackerbird server that your application was terminated gracefully without any unhandled exceptions.

Code Example:

This example shows *App.Stop* being called in the closing event of a form.

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
        App.Stop();
}
```

Recommendation: If you are using a windows forms application, the best location to place the *App.Stop* call would be inside of the entry point method of your application, on the immediate line following the *Application.Run* call which creates your main form as seen below. This will allow the Trackerbird SDK to execute its final server synchronization procedure AFTER your form has been closed. Although *App.Stop* usually takes just a few milliseconds to execute, in case of a slow network connection, the user could experience a small lag from when he hits your application close button until the time the form actually closes. By placing *App.Stop* in the location below, this delay is completely invisible to the user since the form would have been already closed.

```
        [STAThread]
         static void Main()
         {
             Application.EnableVisualStyles();
             Application.SetCompatibleTextRenderingDefault(false);
             Application.Run(new Form1());

             App.Stop();
         }
```

App.ConnectionCheck

This method allows you to test your application's connectivity with the Trackerbird 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 which requires server communication does it's own connection status check and returns any connection errors as part of it's return type.

This method is typically used in conjunction with *App.SetDefaultProxyCredentials* and *App.SetProxy* in order to confirm whether proxy authentication is required before calling *App.Start*. It is also used to test whether a generic internet connection is available for the application or whether the internet connection is down or possibly blocked by some firewall or webfilter at the gateway. The SDK will attempt to use cached credentials from IE to log onto the proxy, however in case this method reports an authentication failure, it means your application cannot make use of cached credentials so you will need to ask the user to enter credentials.

The method requires a single parameters which is your're product's callhome URL and returns an enum with values below:

```
public enum ConnectionReturn
    {
        ServerError = -4,
        AuthenticationFailure = -3,
        ConnectionError = -2,
        ConnectionOK = 1
    }
```

Code Example:

Show a message box to see if the SDK is able to connect with the server:

```
if (App.ConenctionCheck("http://12345.tbnet1.com") == 1)
       {
         MessageBox.Show("Server available");
       }
       else
              {
                MessageBox.Show("Cannot connect to the Trackerbird server.");
              }
```

App.SetDefaultProxyCredentials

This method can be called before *App.Start* and is used in case the application is running behind an HTTP proxy which requires authentication (usually verified through *App.ConnectionCheck*). It enables you to set the proxy username and password that will be used by the SDK to authenticate with the default Windows proxy server defined in IE settings. If the end-user does not have a default proxy server defined, then these credentials are simply ignored.

Conventionally, web-enabled applications usually ask the user to enter these credentials by displaying a dialogue or having a proxy settings section inside the application. Once you get the username and password from the user you can pass them on to the Trackerbird SDK through this method.

This method requires 2 string parameters and has no return value. The parameters are:

1. **Username**: The username to be used to authenticate with the proxy.
2. **Password**: The password to be used to authenticate with the proxy. Note that the password is not saved to disk and must be passed again once the application is restarted.

```
App.SetDefaultProxyCredentials("Testuser", "Pass123");

TBConfig tbConfig = new TBConfig("http://12345.tbnet1.com", "12345", "1.0.0.1",
"10");

App.Start(config);
```

App.SetProxy

This method can be called before *App.Start*. This method basically allows you to set specific proxy settings which the SDK will use from the end user's machine to connect to the Trackerbird server. Please note that unless a proxy server is specified using this method, the Trackerbird SDK will always use the default Windows settings as defined in IE.

This method requires 3 string parameters and has no return value. The three parameters are:

1. **Address**: The address of the HTTP proxy server from which any SDK communication must pass. You may set this value to an empty string in order to discard a previously set proxy address and use the default Windows settings as defined in IE.
2. **Username**: If authentication is required pass the username here, otherwise use an empty string.
3. **Password**: If a password is required for the connection it should be passed here. If a password is not required you can set this value to an empty string. Note that the password is not saved to disk and must be passed again once the application is restarted.

Code Example:

Setting the proxy before calling App.Start:

```
App.SetProxy("192.168.1.250:8080", "TestUser", "TestPassword");

TBConfig tbConfig = new TBConfig("http://12345.tbnet1.com", "12345", "1.0.0.1", "10");

App.Start(config);
```

App.IsConfigLoaded

This is a Boolean property which will return true if *App.Start* has been called and was able to successfully load the configuration. If *App.Start* was not called or if an error occurred during the initialization of the Trackerbird SDK, the property will return false.

```
public static bool IsConfigLoaded
        {
            get
            {
                return isConfigLoaded;
            }
        }
```

Code Example:

A typical *if* condition to check the status of the Trackerbird SDK inside your application:

```
if (App.IsConfigLoaded)
       {
         //Configuration is loaded and initialization was successful
       }
       else
              {
                //App.Start not yet called or an error occured on init.
              }
```

App.FeatureTrack

Through the *App.FeatureTrack* method, Trackerbird 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.

App.FeatureTrack is typically used to check how many times a specific method is called, how often a particular button or menu item is used, how often a user accesses a specific part of the UI, or even to record when a specific important event occurs in your application.

Note: this method should not be used to track the occurrence of exceptions since there is another specific API call for this purpose. The method is also not intended

Through the online analytics reports you will then be able to identify trends of which 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 expierence.

Using *App.FeatureTrack* is very simple since it requires only one string parameter, being the name of the feature or event you want to track. The method has no return value. This function should not be called in a separate thread as it already spawns its own, allowing your application to continue processing asynchronously and seemlessly.

Note on Feature Naming Conventions: Whatever feature names you use as a parameter will be visible in the analytics reports. Therefore we recommend you use a meaningful and structured naming convention to make it easier for you to identify the feature/event in the analytics reports. A popular naming convention is to use a hierarchical format (for example Export.PDF, Export.HTML, Export.TXT) whereas the first part of the name identifies the feature category/subset and the following part identifies the specific function. This will make feature grouping easier when browsing reports.

Naming restrictions: Feature names are limited to 20 characters and cannot contain the colon ":" character. Longer feature names will be truncated and ":" will be replaced by "_".

Code Example:

Track a specific button click event:

```
protected void btnExportPDF_Click(object sender, EventArgs e)
{
        //Track when a button is being clicked
        App.FeatureTrack("Export.PDF");

        //Button Click Logic
}
```

App.ExceptionTrack

This method is used to track and report on any exceptions that are generated by your application on the end users' machine. This function should always be called from within a *try..catch* statement as shown in the example below. Once an exception is tracked, Trackerbird will also save a snapshot of the current machine architecture so that you can later (through the online exception browser within the customer area) investigate the exception details and pinpoint any specific OS or architecture related specs which are the cause of common exceptions.

The method requires 3 parameters. The first 2 strings should be used to store the class name and the method name so that you can tell exactly where the exception happened. The last parameter should be passed the exception object itself.

Code Example:

Placing *App.ExceptionTrack* inside of a try catch statement so that Trackerbird can record it:

```
private void btnSave_Click(object sender, EventArgs e)
       {
            try
            {
                //Save Button Logic
            }
            catch (Exception ex)
            {
                App.ExceptionTrack("Form1", "btnSave_Click", ex);
            }
        }
```

App.ConfigChange

Use the *App.ConfigChange* method if you need to signal the SDK that a configuration change has been made, which in turn will also signal the changes to the server. This method, like the *App.Start* method accepts only one parameter which is of type *ITBConfig*. This method is typically used in events such as when users change or activate their license or when a product build is updated through some live-update without requiring an application restart.

Code Example:

In this code sample we are changing the product id, product version, file path and the key activated status from the original *App.Start* call.

```
ITBConfig tbConfig = new TBConfig("http://91305.tbnet1.com", "2375158964", "1.0.0.1",
"10").SetFilePath(@"c:\temp");

App.Start(tbConfig);

...

//following some event which caused an update to the properties in TBConfig

...

tbConfig = new TBConfig("http://91305.tbnet1.com", "123456789", "1.0.0.2",
"10").SetFilePath(@"c:\temp2").SetKeyActivated(false);

App.ConfigChange(tbConfig);
```

Caching and Synchronizing

The Trackerbird 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 Trackerbird server in batches, at various intervals whenever appropriate. Log data is usually sent at least once for every runtime session (during app.stop), however this may vary based on the type of application and usage activity.

All data is sent over HTTP (port 80) using a proprietary Trackerbird protocol. Using HTTP port 80 is crucial for callhome requests not to be blocked by gateway firewalls especially when running in corporate networks which are sometimes configured to block HTTPS and other unknown traffic.

Only a minor portion of traffic (containing authentication IDs) is encrypted by the protocol. Log data is stored in plaintext and transferred unencrypted. This was designed purposely for the sake of transparency so that security-conscious users can freely sniff whatever is being sent out of their machine so they can confirm that no user-identifiable information is being collected or transmitted.

Forced Synchronization

Under normal conditions, you do not need to instruct the Trackerbird SDK when to synchronize with the cloud server, since this happens automatically at various intervals, and is triggered by your interaction with the API. In a typical runtime session, the SDK will always attempt to synchronize with the server at least once whenever your application calls *App.Stop*.

In order to cater for custom requirements, the API also provides you with the option to forcefully request the SDK to sync all cached data. This is done by calling the *App.Sync* method. In the case where your application runs as a background service or as a webserver where *App.Stop* is rarely called due to the *always-on* nature of the application, you may opt to use *App.StartAutoSync* and *App.StopAutoSync* which launch a background process that will call *App.Sync* on a regular schedule.

App.Sync

This method will forcefully synchronize all cached data from the client machine with the Trackerbird server. The method has no return value and has no parameters. It runs asynchronously so there is no need to place this method inside of a separate thread .

As explained in the previous section, this method is not required and in fact should be avoided. Both the SDK and the server can reject a sync request from occuring even if this is forced by the developer. This is done to prevent abuse and unnecessary server load if *App.Sync* is called too frequently.

Code Example:

```
private void btnsync_Click(object sender, EventArgs e)
      {
          App.Sync();
      }
```

App.StartAutoSync

The Trackerbird SDK gives you the option to launch a separate thread that will perform an automatic sync with the server at a regular intervals. This is usually only required for applications running as background services or web services whose running cycle (session runtime duration) spans over 6 hours. Before using this function please make sure you read the section on *Caching and Synchronizing*.

Once you call *App.StartAutoSync* you may use *App.StopAutoSync* to stop the automatic synchronization process. (See the next section)

Code Example:

Call on *App.start* followed by *App.StartAutoSync*:

```
ITBConfig tbConfig = new TBConfig("http://91305.tbnet1.com", "123456789", "1.0.0.2",
"10").SetFilePath(@"c:\temp").SetKeyActivated(false);

App.Start(tbConfig);

App.StartAutoSync();
```

App.StopAutoSync

Use this method to stop the automatic synchronization with the server which can be started by calling on the StartAutoSync method. (See previous section)

Code Example:

Stopping Autosync on a button click event:

```
private void btnAutoSstop_Click(object sender, EventArgs e)
        {
            App.StopAutoSync();
        }
```

App.KeyCheck

Trackerbird allows you to maintain your own license key register on the Trackerbird server. During BETA, the license key register only accepts Blacklisted keys, however support for other key types will be added soon. By using the *App.Keycheck* method your software can validate a license key (entered by your client) with the blacklist stored on the Trackerbird server.

The method accepts a string parameter which is the license key itself and it returns an object of type *TBLicenseInfo*. This object is a simple class which contains properties that represent the status of your key.

The properties are as follows:

- LicenseBlacklisted – int
- LicenseExpired – int (not available in BETA)
- LicenseActivated – int (not available in BETA)
- LicenseWhiteListed – int (not available in BETA)
- Status – LicenseReturn (Enum) where 1 = OK, < 0 = error (see enum defn. below)

You can use the *TBLicenseInfo* returned from the method to check the various states of your key. This method can only be called a limited amount of times within a specific time frame to help prevent abuse.

Note: Whenever a license key is sent to the server using this method, the key is automatically encrypted by the Trackebird SDK before being sent to the server.

```
public enum LicenseReturn
    {
        ServerError = -4,
        AuthenticationFailure = -3,
        ConnectionError = -2,
        FunctionNotAvailable = -1,
        ConnectionOK = 1
    }
```

Code Example:

Passing “Test Key” as the key and checking the return:

```
TBLicenseInfo kCheck = App.KeyCheck("TEST KEY");

            if (kCheck.Status == LicenseReturn.OK)
            {
                //Check if the license is activated
                if (kCheck.LicenseActivated > 0)
                {
                    MessageBox.Show("License Active");
                }
                else
                {
                    MessageBox.Show("License Inactive");
                }

                //check if key is black listed
                if (kCheck.LicenseBlacklisted > 0)
                {
                    MessageBox.Show("Key is black listed");
                }
                else
                {
                    MessageBox.Show("Key is NOT black listed");
                }

                //Check if key is expired
                if (kCheck.LicenseExpired > 0)
                {
                    MessageBox.Show("Key is expired");
                }
                else
                {
                    MessageBox.Show("Key is NOT expired");
                }

                //Check if key is white listed
                if (kCheck.LicenseWhiteListed > 0)
                {
                    MessageBox.Show("Key is white listed");
                }
                else
                {
                    MessageBox.Show("Key is NOT white listed");
                }
            }
            else if (kCheck.Status == LicenseReturn.AuthenticationFailure)
            {
                MessageBox.Show("Authentication Failure");
            }


                                                        //continued on next page…
```

```
//continued from previous page…

            else if (kCheck.Status == LicenseReturn.FunctionNotAvailable)
            {
                MessageBox.Show("Function is not currently available");
            }
            else if (kCheck.Status == LicenseReturn.ServerError)
            {
                MessageBox.Show("Server Error");
            }
            else if (kCheck.Status == LicenseReturn.ConnectionError)
            {
                MessageBox.Show("Connection Error");
            }
```

App.KeyChanged

This method 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. not blacklisted). The method is very similar to the *KeyCheck* method, 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. The method accepts a string parameter which you should use to pass the key and returns an object of type *TBLicenseInfo*. For more information on the *TBLicenseInfo* class please read the section about *App.KeyCheck*.

Note: Whenever a license key is sent to the server using this method, the key is automatically encrypted by the Trackebird SDK before being sent to the server.

Code Example:

Changing the key to “Test Key Number 2” and checking the return:

```
TBLicenseInfo kCheck = App.KeyChanged("Test Key Number 2");

  if (kCheck.CallStatus < 0)
            {
                MessageBox.Show("Error in keycheck");
            }
            else
            {
                if (kCheck.Status == LicenseReturn.OK)
                {
                    //Check if the license is activated
                    if (kCheck.LicenseActivated > 0)
                    {
                        MessageBox.Show("License Active");
                    }
                    else
                    {
                        MessageBox.Show("License Inactive");
                    }

                    //check if key is black listed
                    if (kCheck.LicenseBlacklisted > 0)
                    {
                        MessageBox.Show("Key is blacklisted");
                    }
                    else
                    {
                        MessageBox.Show("Key is NOT blacklisted");
                    }

                    //Check if key is expired
                    if (kCheck.LicenseExpired > 0)
                    {
                        MessageBox.Show("Key is expired");
                    }
                    else
                    {
                        MessageBox.Show("Key is NOT expired");
                    }

                    //Check if key is white listed
                    if (kCheck.LicenseWhiteListed > 0)
                    {
                        MessageBox.Show("Key is whitelisted");
                    }
                    else
                    {
                        MessageBox.Show("Key is NOT whitelisted");
                    }
                }

                                                        //continued on next page…
```

```
//continued from previous page…

                else if (kCheck.Status == LicenseReturn.AuthenticationFailure)
                {
          MessageBox.Show("Authentication Failure");

                }
                else if (kCheck.Status == LicenseReturn.FunctionNotAvailable)
                {
                    MessageBox.Show("Function is not currently available");
                }
                else if (kCheck.Status == LicenseReturn.ServerError)
                {
                    MessageBox.Show("Server Error");
                }
                else if (kCheck.Status == LicenseReturn.ConnectionError)
                {
                    MessageBox.Show("Connection Error");
                }
            }
```

App.KeyStatusChange

By using *App.KeyStatusChange* you are able to signal the server that the type and state of a license key for a particular client has changed. An example is when a user moves from Evaluation to Purchased or else when a key is Activated or Expired. This will allow the Trackerbird server to note a license transition which can then be reported on through conversion funnel and license activity reports.

Trackerbird gives you the option to choose between managing your license key status (i.e. blacklisted, whitelisted, expired or activated) on the server (server managed) or managing this status through your client (client managed). This option can be set from the **License Key Management** node on the online customer portal. The major difference is outlined below:

1- **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 *App.KeyStatusChange*.

   When to use: 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 Trackerbird server to get this license status. However you can simply use this method to passively inform Trackerbird about the license status used by the client. In this case:

   a. Trackerbird will use this info filter and report the different key types and their activity.
   b. Trackerbird licensing server will operate in passive mode (i.e. reporting only).
   c. Calling *App.KeyCheck* will return *function not available* since the key status is client managed.

2- **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 *App.KeyCheck* or *App.KeyChanged.* In this mode, *App.KeyStatusChange* is only used to pass the key type (Evaluation, Purchased, Freeware, etc) but NOT the key status (Blacklisted, Whitelisted, Expired, Activated).

When to use: If you do not have your own licensing module/mechanism within your application and thus you have no way to to identify the license status at the client side. In this mode, whenever a client changes their license key your application can call *App.KeyChanged* 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. If you want to verify a key without actually registering a key-change for this client you can use *App.KeyCheck* which returns the same values but does register this key with the server. In this case:

a. The key register is maintained *manually* on the server by the software owner
b. Trackerbird 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.
c. Calling *App.KeyCheck* or *App.KeyChanged* will return the 4 status flags denoting whether a registered key is: blacklisted, whitelisted, expired and activated.
d. If the key does not exist on the server, all 4 status flags will be returned as false.

The method expects 4 parameters which are:

```
(       KeyType keyType,
        bool keyExpired,
        bool keyActivated,
        bool keyBlacklisted,
        bool keyWhitelisted
)
```

KeyType is an enumeration with the values below. You can use any of the 3 custom enumeration values in case your software supports non-standard license key types.

```
public enum KeyType
    {
        Evaluation =0,
        Purchased =1,
        Freeware =2,
        Unknown =3,
        NFR =4,
        Custom1 =5,
        Custom2 =6,
        Custom3 =7
    }
```

Code Example:

Setting a license key to purchased and is not expired, activated, not blacklisted and not whitelisted:

```
App.KeyStatusChange(KeyType.Purchased, false, true, false, false);
```

ReachOut™ direct-to-desktop messaging service

From the online customer area you can create ReachOut™ messaging campaigns which 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 2 message delivery options.

- Automated HTML popup messages (which is handled entirely by the Trackerbird library and requires absolutely NO coding.
- Manually retrieving the message (plaintext or URL) through code by using the *App.MessageCheck* API (described hereafter)

App.MessageCheck

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*. Then from within your application you can call *App.MessageCheck* to check with the Trackerbird server whether there are any pending messages (of this type) waiting to be delivered.

You may choose to display plaintext messages anywhere in your application such as in a status bar or information box. For the URL type messages you can either open the URL in a browser or else render it in some HTML previewer embedded within your application. In any case, just call on *App.MessageCheck* to retrieve the message contents.

There are two overloads for this method and we will go through each.

The first overload has the following signature:

```
public static int MessageCheck(out string message, MessageType messageTypeExpected)
```

As can be seen from the signature above, the method has two parameters. The first is an out string that will return the message itself which you can display to the end user. The second is an enumeration of *MessageType* for which you can see the values for below.

```
public enum MessageType
    {
        AllMessageType=0,
        TextMessageType=1,
        URLMessageType=2
    }
```

Basically the *MessageType* sets the type of messages that the SDK should get from the server. This would be ideal if for example you want to show a text message in a status bar and render a URL in some news section within your application.

The method returns an integer which signifies how many of messages are still left on the server. An ideal use would be to place the message check in a loop until 0 is returned from the method signaling that no messages are left on the server.

Code Example:

Get all text messages from the server and display them inside of a message box:

```
string message = "";

while (App.MessageCheck(out message, MessageType.TextMessageType) > 0)
{
	MessageBox.Show(message);
}
```

The second overload has this signature:

```
public static int MessageCheck(out string message, out MessageType messageType)
```

The main differences with this overload of *App.MessageCheck* is that all types of messages are retrieved from the server and that both of the parameters are out parameters and that the second parameter of type `MessageType` will actually signify what type of message is being returned from the server.

Code Example:

Get all of the messages on the server, check if it is a text or url message and display the appropriate message box:

```
string message = "";
MessageType msgType;

while (App.MessageCheck(out message, out msgType) > 0)
{
	if (msgType == MessageType.TextMessageType)
	{
		MessageBox.Show("TEXT Message" + message);
	}
	else if (msgType == MessageType.URLMessageType)
	{
		MessageBox.Show("URL Message" + message);
	}
}
```

App.VersionCheck

This method is used to implement a check for updates system for your software. By logging in to the customer area and accessing the *Builds Management* page you are able to add 1 or more build numbers that will be tagged as the '*latest builds*' for your software. Then from within your application you can call *App.VersionCheck* to confirm whether end users are using the latest build/version of your your application or whether there are any newer builds available for download. Every latest build can apply to either a specific edition/versions/build or else apply to all installations. Therefore when the server matches the latest build numbers for your application it will also take into consideration the current software Edition, Version and build number that were initially submitted through *App.Start*.

```
VersionCheckReturn VersionCheck(out string internalNewVersion, out string
userFriendlyNewVersion, out string downloadURL, out string changelogURL)
```

The four out parameters will give you the information about the new build which you can use to inform a user a new build is available, what the version is, where they can download it from and where to find a log which contains all of the changes for the new build.

This method returns an enumeration, *VersionCheckReturn* which is shown below:

```
public enum VersionCheckReturn
    {
        ServerError = -4,
        AuthenticationFailure = -3,
        ConnectionError = -2,
        FunctionNotAvailable = -1,
        UptoDate = 0,
        NewerVersionAvailable = 1
    }
```

You should only have to notify a user that a newer build is available when the method returns 1.

Code Example:

```
            string internalNewVersion = "";
            string userFriendlyNewVersion = "";
            string downloadURL = "";
            string changelongURL = "";

            VersionCheckReturn vcr = App.VersionCheck(out internalNewVersion,
                   out userFriendlyNewVersion, out downloadURL, out changelongURL);

            string msg = "";

            switch (vcr)
            {
                case VersionCheckReturn.ServerError:
                    msg = "Server error for version check";
                    break;
                case VersionCheckReturn.AuthenticationFailure:
                    msg = "Authentication Failure for version check";
                    break;
                case VersionCheckReturn.ConnectionError:
                    msg = "Connection error for version check";
                    break;
                case VersionCheckReturn.FunctionNotAvailable:
                    msg = "Version check function not available";
                    break;
                case VersionCheckReturn.NewerVersionAvailable:
                    msg = "A newer version is available";
                    msg += string.Format("\nInternal Version: {0}
                            \nUser Friendly Version: {1}
                            \nDownload URL: {2}
                            \nChangelog URL: {3}", internalNewVersion,
                             userFriendlyNewVersion, downloadURL, changelongURL);
                    break;
                case VersionCheckReturn.UptoDate:
                    msg = "The current version is up to date";
                    break;
            }
            MessageBox.Show(msg);
        }
```

App.SetPrivacyMode

The Trackerbird SDK supports 3 different Privacy settings that give you control on what type of anonymous data should be collected form the end user's machine, based on whether the user opted in or out of your Customer Experience Improvement Program. This method is used to set the level of privacy. It has no return value and has only one parameter which is the enumeration *TBPrivacyMode* as seen below. To read more about privacy and what data is collected by each privacy level, please refer to this Kbase article: http://helpdesk.trackerbird.com/knowledgebase.php?article=1

```
public enum TBPrivacyMode
    {
        /// <summary>
        /// Collects both architecture and usage data (default).
        /// </summary>
        Off = 0,
        /// <summary>
        /// Collect only architecture data but NOT usage data.
        /// </summary>
        Low,
        /// <summary>
        /// Does not collect any architecture or usage data.
        /// </summary>
        High
    }
```

Code Example:

To set the privacy to low:

```
App.SetPrivacyMode(TBPrivacyMode.Low);
```

App.GetPrivacyMode

This method takes no parameters and will return the current privacy level for the Trackerbird SDK. For more information about Privacy Mode please check out the previous section on *App.SetPrivacyMode*.

Code Example:

Check if the privacy mode is set to Off and if so set it to high:

```
if (App.GetPrivacyMode() == TBPrivacyMode.Off)
{
       App.SetPrivacyMode(TBPrivacyMode.High);
}
```

Technical Support

Should you encounter any difficulties or have queries about integrating Trackerbird into your application, please visit our support helpdesk at: http://helpdesk.trackerbird.com where you will find several useful Kbase articles and FAQs.

If you do not find what you're looking for, or you think you may have encountered a bug in the Trackerbird SDK or any other part of the Trackebird service, we encourage you to open a support ticket through our helpdesk.

Thank you!

Thank you for using Trackerbird and we hope you enjoy the experience. If you have any feedback on how we can improve the service, we would be delighted to hear from you. Feel free to send us your thoughts or comments to support@trackerbird.com.

Regards,

The Trackerbird Team.