



.NET (C#) Plugin

November 2008

Overview

This document is a complementary reference to the integration guide [3scale-integration](#)¹. In that document you will find information on the whole integration process and the generic connection using the 3scale API. The current document just provides specific details for connecting to the 3scale backend using the .NET-C# plugin. This plugin encapsulates the call to the 3scale API.

C# (.NET) Plugin

To ease the integration task of your service with the 3scale backend, 3scale offers a library (plugin) that encapsulates the communication actions with 3scale, taking complete control of the data connection, data encryption, and data caching to ensure reliability and performance.

The C# plugin is available from the repository at the following address:

http://github.com/3scale/3scale_ws_api_for_dotnet/tree/master

Get the CS_threescale.dll (Windows version)

Alternatively, you can rebuild the dll using Mono or other versions of Visual Studio (the library source is in `threescale.net.src.tar.gz`)

Import the 3scale lib to your C sharp code:

```
using CS_threescale;
```

3scale library depends on:

```
using System;  
using System.Collections.Generic;  
using System.Text;  
using System.Collections;  
using System.Net;
```

Within your application code you need to initialize a 3scale Api object as shown in the following example. You need to provide the 3scale url, and the private provider key for your service (see key descriptions in step 4):

```
_3ScaleAPI = new Api(3SCALE_SERVER, PROVIDER_KEY);
```

Please review section 4 and 5 on document **3scale-connection** to learn more on where and when to use the functions provided in the API. The syntax of the functions for this plugin is described as follows:

¹ Available at <http://www.3scale.net/home/downloads>

Start function

Parameters:

- `:user_key` (to identify the user that is requesting the service)
- `:usage` (optional parameter to indicate Predicted resource consumption of this transaction. This should be a Hash table of the form:

```
Hashtable costs = new Hashtable();  
-costs.Add(METRIC, VALUE);-  
costs.Add("hits", 1);  
costs.Add("storage", "20480");
```

where *metric* is name of the metric that measures particular resource (for example "*storage*" or "*hits*") and *value* is any amount of corresponding resource that is expected to be spend. This parameter may contain only approximate values, or it can be left out entirely. In that case, the actual usage should be reported in **Confirm** operation, described in the next section.

Result:

If 3scale grants access to the user indicated, this function will return a `TransactionData` object with three getters:

- `ID: String tdata.ID`

(Identifier of the transaction. This is needed for confirmation/cancellation of the transaction later)

- `Contract name: String tdata.ContractName;`

(Name of contract the user has signed for. The provider can use this information for example to send different results according to contract types, if that is desired)

- `Provider verification key: String tdata.ProviderVerification`

(Provider service public key. It can be send back to the user to let him verify the authenticity of the provider, although this is not mandatory)

If the access is not granted by whatever reason, this function will issue an exception indicating the nature of the problem. In the start operation the possible problems are

- `MetricInvalid` (if the "usage" parameter contains some undefined metric. The error identifier in this case is: `provider.invalid_metric`)
- `UserKeyInvalid` (if the user key is invalid)
- `ProviderKeyInvalid` (if the provider private key is invalid)
- `ContractNotActive` (when the contract referenced is pending, sespended or canceled)
- `LimitsExceeded` (if the usage limits of the contract are exceeded)
- `SystemError` (if there has been some unexpected error)

The plugin captures the error codes defined in the 3scale API (see document **3scale-connection**). The error codes can be captured in the following manner:

```
Console.WriteLine("Error occured : " + err.GetType().Name + "\n");  
Console.WriteLine("Error type : " + err.ErrorReturn.ID + "\n");  
Console.WriteLine("Description : " + err.ErrorReturn.ServerMessage + "\n");
```

Example Request:

```
class Program
{
    static Api _3ScaleAPI;
    static Hashtable costs;

    static void Main(string[] args)
    {
        _3ScaleAPI = new Api("http://www.3scale.net", "5b20da34365d584...");
        costs = new Hashtable();
        costs.Add("hits", "1");
        costs.Add("storage", "20480");

        try
        {
            tdata = _3ScaleAPI.Start("526843c1e7f45003646ed13b6c885a61", costs);
        }
        catch (ApiException err)
        {
            ...
        }
        catch (WebException netEx)
        {
            Console.WriteLine("Network exception : " + netEx.Message);
        }
        catch (Exception genEx)
        {
            Console.WriteLine("General exception : " + genEx.Message);
        }
    }
}
```

Confirm Function

Parameters:

- :transaction_id (identifier of the transactions, obtained by previous "Start" operation)
- :usage (final resource usage. This parameter is only needed when the predicted resource usage was missing or different from the initial prediction) -see start function to check the syntax

Result:

If there was no problem with the reported usage operation, no exception will be thrown. Otherwise, this call might issue two different types of exceptions:

- MetricInvalid (if the "usage" parameter contains some undefined metric)
- ProviderKeyInvalid (if the provider private key is invalid)
- TransactionNotFound (if the transaction id is not correct)
- SystemError (if there has been some unexpected error)

Example requests:

```
_3ScaleAPI.Confirm(tdata.ID);
```

or

```
_3ScaleAPI.Confirm(tdata.ID, costs);
```

Cancel Function

Parameters:

- :transaction_id (identifier of the transactions, obtained by previous "Start" operation)

Result:

If there was no problem with the reported usage operation, no exception will be thrown. Otherwise, this call will issue two different types of exceptions:

- ProviderKeyInvalid (if the provider private key is invalid)
- TransactionNotFound (if the transaction id is not correct)
- SystemError (if there has been some unexpected error)

Example requests:

```
_3ScaleAPI.Cancel(tdata.ID);
```

Support

Email support is available from support@3scale.net. A developers forum is also available from <https://www.3scale.net/forums> and phone support will be soon available. 3scale is also constantly trying to improve it's service, so if you have any feedback whatsoever, please don't hesitate to let us know at the same address.