
Slingshot JavaScript Client API

Slingshot WDS Version 2.4.1 and above

Document Version 1.0.1
May 2009

This Developer's Guide refers to
The JavaScript Client API version 1.0.1 and above

For more information contact:

Swissrisk Financial Systems GmbH
Technical Support
Holzhausenstrasse 44
D-60322 Frankfurt
Germany

Phone +49 69 50952 111
Fax: +49 69 50952 199
Email hotline@sr-financial-systems.com
Website www.sr-financial-systems.com

Copyright Information

This document is protected by copyright law and may not be reproduced or distributed either in part or in total. The licensee is not allowed to pass on the software or the accompanying written materials to third parties or make them otherwise available without prior written agreement of the licensor. Information in this document that refers to possible product extensions or to available accessories is not legally binding, especially because the products are subject to continuous adaptation and because the information may also relate to future development. The contents of this document can change without prior notice and does not represent any legal obligation on the part of Swissrisk Financial Systems GmbH.

Swissrisk Financial Systems GmbH cannot be made liable for the correctness of information in this document or for damages resulting from the use of this information or the impossibility of using this information. All other legal regulations for using the software and the corresponding documentation are set in the applicable license agreement.

InVision is a trademark of Swissrisk Financial Systems GmbH. All other product and company names mentioned in this manual are trademarks of their respective companies.

Published by:

Swissrisk Financial Systems GmbH
Holzhausenstrasse 44
D-60322 Frankfurt
Germany

Phone: +49 69 50952-0
Fax: +49 69 50952-333
Website: www.sr-financial-systems.com

Copyright © Swissrisk Financial Systems GmbH
All rights reserved

Slingshot JavaScript Client API

Introduction..... 1

Requirements..... 1

Accessing the API..... 1

Cross-Domain issue..... 1

Sharing a single connection between browser windows..... 2

JavaScript API request functions..... 2

JavaScript API callback functions..... 7

Introduction

The Slingshot JavaScript Client API offers developers of web-applications functionality to request and receive real-time data from Slingshot Web Data Server (WDS) using JavaScript only without the need for any external plug-in.

In a nutshell the API has the following functionality:

- Automatic connection handling including recovery from broken connections.
- Request and response handling for data. Requests consist of a service name, a record name and the desired set of fields.
- Re-requesting of request data if the connection has to be renegotiated (broken connection) or if no response to the original request is received.
- Client can unmonitor all the requested fields for a record or a subset of the requested fields.
- Partial field updates are supported.
- Client can contribute data to the server.

Requirements

The API requires WDS version 2.4.1 or greater. JavaScript has to be enabled in the browser. Also, so far only Firefox and Internet Explorer are officially supported.

Accessing the API

The API is contained in a single JavaScript file `sls_js_api.js`. This file should be included in user's HTML file. For example,

```
<script type="text/javascript" src="sls_js_api.js"></script>
```

Cross-Domain issue

There is no special configuration required if all the HTML files and scripts are being served by the WDS that is also the source of real-time data. But if the HTML files and scripts are being served by a server other than the WDS then, it is important that they have the same top-level sub-domain for the API to work.

Let us say, the HTML files and scripts are being served by `slingshot.swissrisk.com` and the WDS URL is `ss3web.de.swissrisk.com`. In this case the **common** top-level sub-domain is `swissrisk.com`. The WDS ini/configuration file has to be configured to use the top-level sub-domain in its response by setting the `JS_DOMAIN` parameter in WDS section. For our example

JS_DOMAIN = swissrisk.com

Also, the user has to specify his document's domain to be the same as the common top-level sub-domain. This could be done as follows for our example,

```
<script type="text/javascript">document.domain="swissrisk.com"</script>
```

Sharing a single connection between browser windows

To simulate a static `HttpConnection` like in Slingshot Java Applets seems difficult in JavaScript. One partial solution can be as discussed in the following link.

<http://bugs.directwebremoting.org/bugs/browse/DWR-353>

The first page (parent/control window/tab) that is opened is deemed to be the control window/tab and actually makes the connection to the WDS and gets all the callbacks. This parent window/tab can then open child windows/tabs which register themselves as listeners of data with the parent. The parent window/tab can pass update objects to child windows following some strategy. One would have to make sure that when a child window/tab is closed, it unregisters itself with the parent. When the parent window/tab is closed it should close the child windows/tabs as well.

With this partial solution one could in principle have multiple tabs which share a single JavaScript API connection to WDS.

Any suggestions and ideas in this area are welcome.

JavaScript API request functions

Please note that all public functions use the `sls` name space.

1. *function create_http_connection(baseUri, callbacks, auth, connFlag)*

Description:

This is the first function that needs to be called. This starts the API.

Parameters:

baseUri (string) - This is the URI of WDS to connect to.

callbacks (object) - This contains callback functions for receiving connection status information and data. At present a maximum of three callback objects are expected - `record_update` for receiving record data, `connection_status` for receiving connection status information and `insert_status` for receiving the status of an insert (Currently to determine if server returned an error or if there was a timeout, i.e., the API did not receive an insert status within a specified amount of time. An insert timeout does not necessarily indicate that there was a problem with the insert/contribution.).

connFlag (number) – This optional parameter can be set to 1 to prevent updates to the connection to be merged.

auth (string) – This optional parameter is the authorization identifier (ID) in case the WDS in question is configured to use authentication ID authorization.

connFlag (number) – This optional parameter can be set to 1 to prevent updates to the connection to be merged.

Return Value:

If the `HttpConnection` has not been created, creates the `HttpConnection` and returns 0. If the `HttpConnection` exists, returns 1.

Example:

```
var c = sls.create_http_connection(  
  "http://garfield.de.swissrisk.com:8080",  
  { record_update: record_update,  
    connection_status: connection_status,  
    insert_status: insert_status  
  },  
  "my_auth_id"  
);
```

2. function destroy_http_connection()**Description:**

This function will destroy the function created by the `create_http_connection` function call.

Example:

```
sls.destroy_http_connection();
```

3. function request_record(request)**Description:**

This function is used for requesting data and is passed a `RecordRequest` object that contains information about the service, the record and the desired set of field identifiers. An important point to note here is that positive identifiers should start with a P and negative identifiers with an M. For example, P22, P23, M999 for 22, 23 and -999 respectively.

Parameters:

request (object) – the `RecordRequest` object

The `RecordRequest` object is constructed as follows

```
var r = new sls.RecordRequest(service, name);
```

where `service` and `name` are the desired service and record/instrument names respectively.

The `RecordRequest` offers the following functionality:

```
setFields(fids);
```

where `fids` is an array of fields for which data is desired.

```
setType(type);
```

where `type` is the type of request and can be either `sls.RequestType.Monitor` or `sls.RequestType.Snapshot`. The default value for type of request is `sls.RequestType.Monitor`.

```
setPriority(p);
```

where `p` is a number specifying the priority of the request. The default value of priority is 0.

```
setNoMerge();
```

This function can be called if the updates for this request should not be merged. Merging is default behaviour.

```
setWhenAvailable();
```

This function can be called if the updates should be sent when the desired request becomes available. Please note that this functionality has not been fully tested.

Return value:

If the request was queued up successfully, returns 0. If the `HttpConnection` does not exist, returns 1. If there is a request validation error, returns 2.

Example:

```
var r = new sls.RecordRequest("SwissriskFeed", "EUR="); //create the request
object
r.setFields(["P22", "P23"]); //set the fields
sls.request_record(r); //make the request
```

4. function unmonitor_record(request)

Description:

This function is used to unmonitor all the requested fields for a record.

Parameters:

request (object) – the RecordRequest object

The RecordRequest object is constructed as follows:

```
var r = new sls.RecordRequest(service, name);
```

where *service* and *name* are the desired service and record/instrument names respectively.

The type of the record can be optionally set to `sls.RequestType.Unmonitor` by calling `setType` on the request object.

Return value:

If the request was queued up successfully, returns 0. If the `HttpConnection` does not exist, returns 1. If there is a request validation error, returns 2.

Example:

```
var r = new sls.RecordRequest("SwissriskFeed", "EUR=");  
r.setType(sls.RequestType.Unmonitor);  
sls.unmonitor_record(r);
```

5. function unmonitor_fields(request)**Description:**

This function is used to unmonitor the specified fields for a record. An important point to note here is that positive identifiers should start with a P and negative identifiers with an M. For example, P22, P23, M999 for 22, 23 and -999 respectively.

Parameters:

request (object) – The RecordRequest object in the sls namespace.

The RecordRequest object is constructed as follows:

```
var r = new sls.RecordRequest(service, name);
```

where *service* (string) and *name* (string) are the desired service and record/instrument names respectively.

The type of the record can be optionally set to `sls.RequestType.Unmonitor` by calling `setType` on the request object.

The array of fields to be unmonitored can be set by calling *setFields* on the request object *s*.

Return value:

If the request was queued up successfully, returns 0. If the *HttpConnection* does not exist, returns 1. If there is a request validation error, returns 2.

Example:

```
var r = new sls.RecordRequest("SwissriskFeed", "EUR=");
r.setFields(["P22", "P25"]);
r.setType(sls.RequestType.Unmonitor);
sls.unmonitor_fields(r);
```

6. function insert_request(insert)**Description:**

This function is used for contributing data for a set of fields for a particular service and record combination (this information is contained in the insert object passed to this function). An important point to note here is that positive identifiers should start with a P and negative identifiers with an M. For example, P22, P23, M999 for 22, 23 and -999 respectively. Also, the field data to be contributed to the server should be UTF-8 encoded. There is a helper function *sls.Base64._utf8_encode* contained in the API that can be used for this such as

```
string = sls.Base64._utf8_encode(string);
```

where *string* is the string to be encoded.

Parameters:

insert (object) – the *InsertRequest* object

The *InsertRequest* object is created as follows:

```
var ins = new sls.InsertRequest(srv, rec, seqNo);
```

where the parameters are,

srv (string) – the name of the service.

rec (string) – the name of the record.

seqNo (number) – the sequence number to identify the contribution.

The *InsertRequest* object offers the following functionality:

The contribution fields can be set by calling

```
setField(field, offset, data, partial)
```

where,

field (string) - field identifier

offset (number) – offset of the update

data (string) – UTF-8 encoded data to be contributed

partial (boolean) – to indicate if the update is partial or not

Currently for record updates offset should be set to 0 and partial to false.

The type of the insert can be set using

```
setType(type);
```

where type can be `sls.InsertType.Record` or `sls.InsertType.Page`. Default value is `sls.InsertType.Record`.

Return value:

If the request was queued up successfully, returns 0. If the `HttpConnection` does not exist, returns 1. If there is a request validation error, returns 2.

Example:

```
var ins = new sls.InsertRequest("SwissriskFeed", "EUR=", 66);  
ins.setType(sls.InsertType.Record);  
ins.setField("P1", 0, "123456", false);  
ins.setField("P2", 0, "ABCDEFGH", false);  
sls.insert_request(ins);
```

JavaScript API callback functions

Please note that all public functions use the `sls` name space.

1. *connection_status(status)*

Description:

When supplied as a callback object to the API, connection related information will be received through this callback. The possible values of status are:

```

sls.ConnectionStatus = {
  Init: 0,
  Active: 1,
  Down: 2,
  Reconnecting: 3,
  Unauthorised: 4,
  MaxClientsExceeded: 5
};

```

Example:

```

connection_status(status)
{
  if (o == sls.ConnectionStatus.Active)
  {
    //Successfully connected
  }
  else
  {
    //Check the status and respond accordingly
  }
}

```

2. function record_update(o)**Description:**

When supplied as a callback object to the API, data and status information for requests will be received through this callback.

Parameters:

o (object) - the callback data object.

The callback data object has the following structure:

```

{s:"service name", k:"record name", f:flag, n:"Name change", d:delayed,
fids[{i:"P22",v:"data",f:"flag"}, {i:"P22",v:"data",f:"flag"}, ...]}

```

s (string) – the service name

k (string) – the record name

f (number) – flag.

Parameter *f* (flag) for update object *o* can have the following values:

```

sls.RecFidState = {
  OK: 0,

```

```

RequestPending: 1, //currently not sent
Stale: 2,
NotAvailable: 3,
RequestError: 4,
NotPermissioned: 5,
ServiceStale: 6,
ServiceUnkown: 7,
Dropped: 8, // or Unmonitored
RecordPaused: 9,
ErrorRecordPaused: 10
};

```

n (string) - Old name in case of a name change (**TO BE IMPLEMENTED**)

d (number) - 0 - normal, 1 - delayed.

fids (array) - array of field objects

The field objects have the following structure:

```
{i:"P22",v:"data",f:"flag}
```

where *i* is the field identifier, *v* is the data and *f* is a field level flag. *f* with bitwise operations can be used to determine offset (in case of partial data) or if the data was merged. For example:

```

if (f & 0x100) { //partial data}
if (f & 0x200) { //merged data}

```

Example:

```

record_update(o)
{
var rdisp = "Service: " + o.s + " Record: " + o.k + " Flag: " + o.f + " Name change:
" + o.n + " Delayed: " + o.d + "\n";
for (var f = 0; f < o.fids.length; f++)
{
rdisp += "Field: " + o.fids[f].i + " Data: " + o.fids[f].v + " Flag: " + o.fids[f] +
"\n";
}
}
}

```

3. function insert_status(o)

Description:

When supplied as a callback object to the API, data and status information for requests will be received through this callback. This function is automatically called if

a status response is received from the server. If no response is received for a certain amount of time, then a timeout response is internally generated. This does not necessarily mean that there was an error.

Parameters:

o (object) - the callback data object.

The callback data object has the following structure:

```
{s:"Service name", k:"Record name", i:Sequence number , t:"Text of status message"}
```

s (string) - service name

k (string) - record name

i (number) – sequence number that was sent with the contribution request

t (string) – text of the status message

Example:

```
function insert_status(o)  
{  
var rdisp = "Insert status " + o.s + " " + o.k + " Seqno: " + o.i + " " + o.t;  
}
```