
Slingshot Developer's Guide

Slingshot Version 2

**Document Version 1.8
January 2009**

**This Developer's Guide refers to
the Java Client API version 2.2.0 and above,
the C++ Client API version 2.2.0 Build 8 and above,
the C# API version 2.0 and above**

For more information contact:

Swissrisk Financial Systems GmbH
Holzhausenstrasse 44
D-60322 Frankfurt am Main
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.

It is not permitted to pass on the software or the accompanying written materials to third parties or make them otherwise available without prior written agreement of Swissrisk Financial Systems GmbH.

Information in this document is not legally binding, especially since Swissrisk products and professional services are subject to continuous adaptation and future development of products. 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 liable for the correctness of information in this document nor 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, Slingshot, I-Trader, I-Pricer, X-Gen are registered trademarks of Swissrisk Financial Systems GmbH. All other product and company names mentioned in this document are trademarks of their respective companies.

Published by:

Swissrisk Financial Systems GmbH
Holzhausenstrasse 44
D-60322 Frankfurt am Main

Germany

Slingshot Developer's Guide

Client API	1
Operation.....	1
Functionality.....	1
Update Pausing.....	1
Response Priority.....	2
Request Batching.....	2
Recovery	2
Java	2
Class SingleHttpConnection.....	2
Class HttpConnection.....	2
Class RecordRequest.....	3
Class RequestFidCallback.....	3
Class UpdateRecord.....	3
Class InsertCallBack.....	4
Class HttpConnectionStatus.....	4
Making a Request.....	4
Making an Insert.....	5
Example Applet Code.....	6
C++	7
Class HttpConnection.....	7
Class RecordRequest.....	8
Class RequestFidCallback.....	8
Class UpdateRecord.....	9
Class InsertCallBack.....	9
Class HttpConnectionStatus.....	9
Class UserAuthorization.....	9
Making a Request.....	10
Making an Insert.....	11
C#	12
HttpConnection.....	12
RecordRequest.....	12
RequestFidCallback.....	12
UpdateRecord.....	13
InsertCallback.....	13
HttpConnectionStatus.....	13
Making a Request.....	14
Making an Insert.....	14

Client API

Operation

The Java and C++ client API operates using a call back method. A request is made for a field of a record and a class is supplied to be called when an event occurs for that field. A connection to a host consists of an instance of the `HttpConnection` class. Each connection can contain any number of services with any number of fields in records monitored. A `HttpConnection` consists of a cache, a pending queue and an insert queue. All monitored fields are cached and all requests for a field are queued. With only one request made to the server for multiple client requests. The `HttpConnection` class is responsible for establishing the connection and for maintaining it. It can handle redirections and after a redirection it will try to continue to communication with this redirected server (not implemented yet for C# API). If this fails it will revert back to the original server in the request. The `HttpConnection` class will only notify the callback class of a failure to reconnect to server after a timeout period. If a `HttpConnection` class has no activity, that is, if its cache, pending queue or insert queue are empty it will close the host connection and stop internal threads. Any new request made at this stage will result in an error.

Functionality

Update Pausing

The API has the ability to pause updates to a record that it has monitored. The record has to be currently monitored by the API. That is the API must have received an update for that record. The `HttpConnection` class has two functions for pausing and resuming updates for a record:-

```
int pauseUpdate( string srv, string key);  
int resumeUpdate( string srv, string key);
```

The two functions take the service and record key of the record you want paused or to resume.

Pausing a record will work in the following sequence.

When a record is paused all the fields in that record will get a callback with the current value but with a state `recordStatePaused` (9). No more updates will be received for that record until it is resumed. If you request a field of a record that is paused and the field is in the cache, the request will receive a callback with the value of the field in cache and a state `recordStatePaused` (9). If a field of a paused record is requested that is not in the local cache, the callback will be called with a value of zero length string and a state of `recordStateErrorPaused` (10).

When a paused record is resumed, all the fields of the record will get a callback with the current value and state.

If the API goes into recovery or the connection to the WDS is lost, all paused records are reset to normal.

Response Priority

Requests can set the priority of the response messages. This option has to be enabled in the WDS. See WDS Parameter:- USE_PRIORITY_QUEUEING

The RecordRequest class has a function setPriority(short) (for C# API this a property – Priority). The priority is a short value 0 to 32767, with 0 as the normal or lowest priority. By default all requests will get a priority of 0. Messages are prioritized on a connection basis. So a message with a priority of 1 will have priority over all messages of priority zero for that connection.

Request Batching

The API will attempt to batch all requests together for sending to the WDS. This is done to reduce traffic and request delays between the API and the WDS.

Recovery

After a successful connection has been established and a successful response has being received for a monitor request. The client considers that a valid object until it is either dropped by the server or is unmonitored by the client application. If the WDS fails or a network failure occurs, the client will first try to reconnection to the old server. In this time period the application is not notified of any failure. If the reconnection fails the API will notify the application that all records are stale. The API will continue to try to reconnect to server. After all records are marked stale a successful reconnection to the server will result in all fields being requested. If a connection is made to a server that is not its original server (incorrect server signature or connection id), the client will consider this a new connection and request all fields also.

Java

The main classes of the user of the interface are.

Class SingleHttpConnection

This class is used to ensure that only one connection is maintained to the WDS per archive, per Virtual Machine. The use of this class is not mandatory, but it ensures that a browser's maximum number of connections to the host is not exceeded (this figure is 4 in Internet Explorer). This class extends HttpConnection so no HttpConnection class is needed if a SingleHttpConnection class is used.

Class HttpConnection

An instance of this class has to be created in the Applet or Application for each host connection.

Class RecordRequest

This class is used for making requests. The information required is a host, a service name, a record name and the fields required. Also a call back class is required.

Class RequestFidCallback

This class is an interface class. It has one function that has to be implemented. The function will be called when an event occurs to the requested field.

```
public void FidUpdate( String Key,
                    short Fid,
                    String Value,
                    short State,
                    short Offset,
                    long Flags);
```

Key – the name of the record

Fid – the field identifier of the request field

Value – the value of the field(can be null depending on the state)

State – the current state of the field

(The current supported states)

Normal or OK	0
Request pending	1
Stale	2
Not Available	3
Request Error	4
Not Permissioned	5
Service Stale	6
Service Unknown	7
Unmonitored or Dropped	8
Record Paused	9
Error Record Paused	10

Offset – the offset into the field for this update

Flags – misc information

(currently)

Full update	0
Partial update	1
Delayed data	0x20000000
Updates missed or merged	0x40000000
Name Change bit	0x80000000

Class UpdateRecord

This class is used to insert field data into a record. The information required is Host, a Service name, a record name, fields with data, a sequence number and the length of time to wait in

milliseconds for an insert response. A call back class can be supplied to handle the insert response.

Class InsertCallback

This interface class is used to handle the insert responses. It has only one function that must be implemented.

```
public void insertStatus( short fid, short resp, long seq, String txt);
```

Fid – the field identifier of the inserted field

Resp – the response code for the insert.

(currently supported)

Success 0

Failed 1

Timed Out 2

Not Permissioned 5

Seq – the sequence supplied with the insert.

Txt – text stating the reason for insert failure

Class HttpConnectionStatus

This interface class is used to handle status event in the HttpConnection. It is used with the call setStatus. It is one function that must be implemented.

```
public void status( int State, String stateTxt)
```

State – the state of the connection

Initialising 0

Connection Active 1

Connection Stale or Reconnecting 2

Connection Failure 3

Connection Closed 4

Maximum Clients reached in Server 5

Connection to this Server is Unauthorised 6

StateTxt – a text string representing the state

All other classes in the API are not used or visible to the user.

Making a Request

A request for a field of a record can be made in the following manner.

Implement RequestFidCallback class.

```

class FidCallback implements RequestFidCallback
{
    public void FidUpdate( String Rec,
                          short Fid,
                          String Value,
                          short State,
                          short Offset,
                          int Flags)
    {
        System.out.println("Monitor Record(" + Rec
                           + " Fid(" + Fid
                           + ") VCalue(" + Value
                           + ") State(" + State
                           + ") Offset(" +Offset
                           + ") Flags(" + Flags + ")");
    }
}

```

Create an instance of the `HttpConnection` class. This has to be done for each host connection.

```

HttpConnection c = new HttpConnection (new
URL("http://www.request.com/MarketLink/DAV="));

```

Create a request. In this case we are requesting DAV= from service Marketlink.

```

RecordRequest monReq = new RecordRequest
("http://www.request.com/MarketLink/DAV=");

```

or

```

RecordRequest monReq = new RecordRequest();
monReq.setService("Marketlink");
monReq.setkey("DAV=");

```

Create an instance of the call back class.

```

FidCallback monTest = new FidCallback();

```

Set the fields that will be watched.

```

monReq.setFid( new Short((short)22), monTest);
monReq.setFid( new Short((short)25), monTest);

```

Connect to the host.

```

c.connect();

```

Request fields.

```

c.request( monReq);

```

The events that will occur to the call back class are as follows.

After the request has been successful sent, the call back routine will be called with the requested field id, Value set to null, state set to pending, offset set to 0 and flags set to 0. If a successful connection to the host can not be made, the call back class will be called with the state of request error.

If the request is unsuccessful the call back function will be call with the requested field id, Value set to null, state set to unavailable, offset set to 0 and flags set to 0.

If the request was successful the call back function will be called with the request field id, a Value, state set to normal or stale, offset set to 0 and flags set to 0. It can then receive updates where the flag is 0 or 1 and the offset can be 0 or a value.

Making an Insert

An insert of a field for a record can be made in the following sequence.

Implement the InsertCallback class.

```
class TestInsert implements InsertCallback
{
    public void insertStatus( String Rec, short fid, short resp, long
seq, String txt)
    {
        System.out.println("Insert Status Callback");
        System.out.println("Fid "+fid+" Resp: "+resp+" Seq: "+seq+"
Txt ("+txt+")");
    }
};
```

Create an instance of the HttpURLConnection class. This has to be done for each host connection.

```
HttpURLConnection c = new HttpURLConnection (new
URL("http://www.request.com/MarketLink/DAV="));
```

Create an insert class. In this case we are going to insert into record DAV= in service MarketLink.

```
UpdateRecord Insert = new UpdateRecord
("http://www.request.com/MarketLink/DAV=");
```

or

```
RecordRequest monReq = new RecordRequest()
MonReq.setService("Marketlink");
MonReq.setkey("DAV=");
```

Create an instance of the insert call back class.

```
TestInsert Test = new TestInsert( );
```

Set the field value pairs to be inserted. We are making insert requests to field ids 22 and 25. The values being inserted are 9999 for each with offset 0. The timeout for each insert is 5000 milliseconds. A timeout value of 0 is to wait forever and a timeout value of -1 is to not wait for response. Setting the call back class to null is the same as setting the timeout to -1.

```
Insert.setFid( new Short((short)22), "9999", 0, 1234, Test,
5000);
Insert.setFid( new Short((short)25), "9999", 0, 1235, Test,
5000);
```

Connect to the host.

```
c.connect();
```

Make the request.

```
c.request(Insert);
```

The call back routine will be called with either a successful insertion, an insert failure or a timeout. If the insert was a failure and txt is not null, it will contain a reason for the failure.

Example Applet Code

A more complete sample applet is available in the Slingshot Demo Package in the directory, "Slingshot2\WDS\public_html\QuoteViewer".

This is just a basic outline in the uses of the Java Client API in an applet. An instance of the Controller class should be created in the start routine and the applet should be registered in the start routine if it is already started.

```
import java.applet.Applet;
import java.awt.*;
import net.Slingshot.ClientAPI.*;

public class TestApplet extends Applet
{
    /** call back class for displaying updates */
```

```

private class AppletCallback implements RequestFidCallback
{
    /** The call back routine called by the API */
    public void FidUpdate( short Fid,
                          String Value,
                          short State,
                          short Offset,
                          int Flags)
    {
        //
        // Handle update
    }
};

/** Applet stop */
public void stop() {
    if( myC != null)
        myC.close();
    myC = null;
}

/** Applet destroy routine */
public void destroy() {
    if( myC != null)
        myC.close();
}

/** Applet start routine */
public void start() {
    if( myC == null) {
        myC = new HttpConnection();
        // Connect to the host.
        myC.connect();
        //
        // Make requests
    }
}

/** Applet paint routine */
public void paint(Graphics g) {
    //
    // paint display
}

HttpConnection myC = null;
}

```

C++

The main difference between the java and C++ versions of the Client API software is in using the Secured Socket Layer and User Authorization. The HttpConnection class has functions for setting the Certificate Authority information and for supplying a user certificate if required. It also has a member function for setting the user name and password for basic authorization or a callback class can be set.

The main classes of the user of the interface are.

Class HttpConnection

An instance of this class has to be created in the Application for each host connection.

Class RecordRequest

This class is used for making requests. The information required is a host, a service name, a record name and the fields required. Also a call back class is required. Request can also be made directly to the HttpConnection class

Class RequestFidCallback

This class is an virtual class. It has one function that has to be implemented. The function will be called when an event occurs to the requested field.

```
virtual void FidUpdate( string key,
                      short Fid,
                      string Value,
                      short State,
                      short Offset,
                      unsigned long Flags);
```

Key – the name of the record

Fid – the field identifier of the request field

Value – the value of the field(can be null depending on the state)

State – the current state of the field

(The current supported states)

Normal or OK	0
Request pending	1
Stale	2
Not Available	3
Request Error	4
Not Permissioned	5
Service Stale	6
Service Unknown	7
Unmonitored or Dropped	8
Record Paused	9
Error Record Paused	10

Offset – the offset into the field for this update

Flags – misc information

(currently)

Full update	0
Partial update	1
Delayed data	0x20000000
Updates missed or merged	0x40000000
Name Change bit	0x80000000

Class UpdateRecord

This class is used to insert field data into a record. The information required is Host, a Service name, a record name, fields with data, a sequence number and the length of time to wait in milliseconds for an insert response. A call back class can be supplied to handle the insert response.

Class InsertCallBack

This interface class is used to handle the insert responses. It has only one function that must be implemented.

```
public void insertStatus( short fid, short resp, long seq, String txt);
```

Fid – the field identifier of the inserted field

Resp – the response code for the insert.

(currently supported)

Success	0
Failed	1
Timed Out	2
Not Permissioned	5

Seq – the sequence supplied with the insert.

Txt – text stating the reason for insert failure

Class HttpConnectionStatus

This virtual class is used to handle status event in the HttpConnection. It is used with the call setStatus. It has one function that must be implemented.

```
virtual void insertStatus( string key, short fid, short resp, long seq, string txt) ;
```

State – the state of the connection

Initialising	0
Connection Active	1
Connection Stale or Reconnecting	2
Connection Failure	3
Connection Closed	4
Maximum Clients reached in Server	5
Connection to this Server is Unauthorised	6

StateTxt – a text string representing the state

Class UserAuthorization

This class can be used as a callback class to obtain the Users name and password. It has one function that must be implemented.

```
virtual void UserInfo( string &user, string &password ) ;
```

```
    User          - User Name
    Password      - User Password
```

All other classes in the API are not used or visible to the user.

Making a Request

A request for a field of a record can be made in the following manner.

Implement RequestFidCallback class.

```
class callback : public RequestFidCallback
{
public:

callback( void) {}
~callback( void) {}

void FidUpdate( string key,
                short Fid,
                string Value,
                short State,
                short Offset,
                unsigned long Flags)
{
char *stateStrings[] = {
    "Normal",
    "Pending",
    "Stale",
    "NotAvailable",
    "Error",
    "PermissionDenied",
    "Service Stale",
    "Service Unknown",
    "Unmonitored"};

printf("Updated: Key(%s) Fid(%d) Value(%s) State(%s) Offset(%d) Flags(%ld)\n",
       key.data(),Fid, Value.data(),
       (State < 9) ? stateStrings[State] : "Error Unknown State",
       Offset, Flags);
}
};
```

Create an instance of the HttpConnection class. This has to be done for each host connection.

```
HttpConnection *c =
    new HttpConnection (new URL("http://www.request.com/MarketLink/DAV="));
```

Create a request. In this case we are requesting DAV= from service Marketlink.

```
RecordRequest *monReq = new RecordRequest ("MarketLink", "DAV=");
```

Create an instance of the call back class.

```
callback *monTest = new callback();
```

Set the fields that will be watched.

```
monReq->setFid(22, monTest);
```

```
monReq->setFid( 25, monTest);
```

Connect to the host.

```
c->connect();
```

Request fields.

```
c->request( monReq);
```

The events that will occur to the call back class are as follows.

After the request has been successful sent, the call back routine will be called with the requested field id, Value set to zero length string, state set to pending, offset set to 0 and flags set to 0. If a successful connection to the host can not be made, the call back class will be called with the state of request error.

If the request is unsuccessful the call back function will be call with the requested field id, Value set to null, state set to unavailable, offset set to 0 and flags set to 0.

If the request was successful the call back function will be called with the request field id, a Value, state set to normal or stale, offset set to 0 and flags set to 0. It can then receive updates where the flag is 0 or 1 and the offset can be 0 or a value.

Making an Insert

An insert of a field for a record can be made in the following sequence.

Implement the InsertCallback class.

```
class insert : public InsertCallback
{
public:

insert() {}
~insert() {}

void insertStatus( string key, short fid, short resp, long seq, string txt)
{
printf("Insert: Key(%s) Fid(%d) resp(%d) seq(%ld) txt(%s)\n",
key.data(),fid,resp,seq,(txt == "") ? "No Text" : txt.data());
}
};
```

Create an instance of the HttpConnection class. This has to be done for each host connection.

```
HttpConnection *c =
new HttpConnection (new URL("http://www.request.com/MarketLink/DAV="));
```

Create an insert class. In this case we are going to insert into record DAV= in service MarketLink.

```
UpdateRecord *Insert = new UpdateRecord ("MarketLink","DAV=");
```

Create an instance of the insert call back class.

```
insert *Test = new insert( );
```

Set the field value pairs to be inserted. We are making insert requests to field ids 22 and 25. The values being inserted are 9999 for each with offset 0. The timeout for each insert is 5000 milliseconds. A timeout value of 0 is to wait forever and a timeout value of -1 is to not wait for response. Setting the call back class to null is the same as setting the timeout to -1.

```
Insert->setFid( 22, "9999", 0, 1234, Test, 5000);
Insert->setFid( 25, "9999", 0, 1235, Test, 5000);
```

Connect to the host.

```
c->connect();
```

Make the request.

```
c->request(Insert);
```

The call back routine will be called with either a successful insertion, an insert failure or a timeout. If the insert was a failure and txt is not a zero length string, it will contain a reason for the failure.

C#

HttpConnection

An instance of this class has to be created in the Application for each host connection.

RecordRequest

This class is used for making requests. The information required is a host, a service name, a record name and the fields required. Also a call back class is required.

RequestFidCallback

This class is an interface class. It has one function that has to be implemented. The function will be called when an event occurs to the requested field.

```
void fidUpdate(
    String      key,
    Int16 fid,
    String      value,
    Int16 state,
    Int16 offset,
    Int64 flags);
```

key – the name of the record

fid – the field identifier of the request field

value – the value of the field(can be null depending on the state)

state – the current state of the field

(The current supported states)

Normal or OK	0
Request pending	1
Stale	2
Not Available	3
Request Error	4
Not Permissioned	5
Service Stale	6
Service Unknown	7
Unmonitored or Dropped	8
Record Paused	9
Error Record Paused	10

Offset – the offset into the field for this update

flags – misc information

(currently)	
Full update	0
Partial update	1
Delayed data	0x20000000
Updates missed or merged	0x40000000
Name Change bit	0x80000000

UpdateRecord

This class is used to insert field data into a record. The information required is Host, a Service name, a record name, fields with data, a sequence number and the length of time to wait in milliseconds for an insert response. A call back class can be supplied to handle the insert response.

InsertCallback

This interface class is used to handle the insert responses. It has only one function that must be implemented.

```
void insertStatus( Int16 fid, Int16 resp, Int32 seq, String txt);
```

fid – the field identifier of the inserted field

resp – the response code for the insert.

(currently supported)

Success	0
Failed	1
Timed Out	2
Not Permissioned	5

seq – the sequence supplied with the insert.

txt – text stating the reason for insert failure

HttpConnectionStatus

This interface class is used to handle status event in the HttpConnection. It is used with the call setStatus. It is one function that must be implemented.

```
void status( Int32 State, String stateTxt);
```

State – the state of the connection

Initialising	0
Connection Active	1
Connection Stale or Reconnecting	2
Connection Failure	3
Connection Closed	4
Maximum Clients reached in Server	5
Connection to this Server is Unauthorised	6

stateTxt – a text string representing the state

Making a Request

A request for a field of a record can be made in the following manner.

Implement RequestFidCallback class.

```
public class MyCallback : RequestFidCallback
{
    public void fidUpdate(String key, Int16 fid, String value,
        Int16 state, Int16 offset, Int64 flags)
    {
        System.Console.WriteLine("RequestFidCallback :: FidUpdate
            {0} {1}={2}", key, fid, value);
    }
}
```

Create an instance of the HttpConnection class. This has to be done for each host connection.

```
Uri url = new Uri("http://wellington.Swissrisk.co.uk");
HttpConnection con = new HttpConnection(url);
con.connect();
```

Create a request, the callback class and set the fields.

```
RecordRequest req = new RecordRequest();
req.Service = "SWISSRISKDataFeed";
req.Key = "EUR=";
req.monitor();
MyCallback cb = new MyCallback();
req.setFid(22, cb);
req.setFid(23, cb);
```

Request fields.

```
con.request( req);
```

The events that will occur to the call back class are as follows.

After the request has been successful sent, the call back routine will be called with the requested field id, Value set to null, state set to pending, offset set to 0 and flags set to 0. If a successful connection to the host can not be made, the call back class will be called with the state of request error.

If the request is unsuccessful the call back function will be call with the requested field id, Value set to null, state set to unavailable, offset set to 0 and flags set to 0.

If the request was successful the call back function will be called with the request field id, a Value, state set to normal or stale, offset set to 0 and flags set to 0. It can then receive updates where the flag is 0 or 1 and the offset can be 0 or a value.

Making an Insert

An insert of a field for a record can be made in the following sequence.

Implement the InsertCallback class.

```
public class MyInsertCallback : InsertCallback
{
    public void insertStatus( Int16 fid, Int16 resp, Int32 seq,
        String txt)
    {
```

```
        System.Console.WriteLine("Fid: {0}, Resp: {1}, Seq: {2},  
        Txt: {3}", fid, resp, seq, txt);  
    }  
};
```

Create an instance of the `HttpConnection` class. This has to be done for each host connection.

```
Uri url = new Uri("http://wellington.Swissrisk.co.uk");  
HttpConnection con = new HttpConnection(url);  
con.connect();
```

Create an insert class.

```
UpdateRecord Insert = new UpdateRecord  
("http://wellington.Swissrisk.co.uk/SWISSRISKDataFeed/DAV=");
```

Create an instance of the insert call back class.

```
MyInsertCallback Test = new MyInsertCallback ();
```

Set the field value pairs to be inserted. We are making insert requests to field ids 22 and 25. The values being inserted are 9999 for each with offset 0. The timeout for each insert is 5000 milliseconds. A timeout value of 0 is to wait forever and a timeout value of -1 is to not wait for response. Setting the call back class to null is the same as setting the timeout to -1.

```
Insert.setFid( 22, "9999", 0, 1234, Test, 5000);  
Insert.setFid( 25, "9999", 0, 1235, Test, 5000);
```

Make the request.

```
con.request(Insert);
```

The call back routine will be called with either a successful insertion, an insert failure or a timeout. If the insert was a failure and `txt` is not null, it will contain a reason for the failure.