InfoReach

Engineered for what's next

TMS / Trade Management System

TMS C++ Programmer's Guide

Version 8.3

Contents

1 Introduction

TMS API allows users to interact programmatically with TMS server components. The API consists of two sets of methods where methods of the first ("subscription") set can be used to subscribe custom application component to the TMS event flow, and methods of the other ("instruction") set can be used to directly instruct the TMS to perform an action. The event flow exposed to the custom application in real time includes all order-related information, portfolio-related information, market data events, custom record events, position-related information, and report data. The events notify the application components of any changes taking place in the system and based on that information the instructions can be given to the TMS to perform a set of actions. Sending/modifying/canceling an order, creating new portfolio, modifying set of portfolio's targets, starting/pausing an analytic that automatically sends order waves are some of the actions available to the users' applications.

2 Initializing TMSClientSystem

2.1 Initialization

To use C++ TMS API the TMSClientSystem should be initialized:

```
TMSClientSystem::initialize(Url("java:ELTClientProxy:./cppcfg/cppbootstrap.xml"));
```

The string "java:ELTClientProxy:./cppcfg/cppbootstrap.xml" contains the path to the XML file containing the system settings. The cppbootstrap.xml contains the settings for the JVM and also process type and process name that are used to read the process' configuration form the config files/

The current implementation of the TMS C++ interface is based on the JNI Specification which allows C/C++ processes to start JVM inside and communicate with it via the JNI API. This approach has been chosen because it has been proven to be faster than other inter-process communication techniques, such as CORBA. Also it gives C++ applications all the flexibility that the JAVA applications have and allows the use of the same configuration for both Java and C++ processes.

Example of cppbootstrap.xml:

```
<Bootstrap>

    <Default>
        <JVMSettings>
            <property
                 name="-Djava.class.path"
                 value="../../../$(OS)/bin;"
                 append="true" />

            <property
                 name="-Djava.class.path"
                 value="$(LIB)/inforeachInfra.jar;"
                 append="true" />

            <property
                 name="-Djava.class.path"
                 value="$(LIB)/inforeachView.jar;"
```

```
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/inforeachApp.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/eltraderEngine.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/eltraderView.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/eltradertms.jar;"
     append="true" />
<property
     name="-Djava.class.path"
     value="$(LIB)/eltradertmsView.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/jgl3.1.0.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/jce1_2-do.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/jaas.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/jsse.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/jcert.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/jnet.jar;"
```

```
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/oromatcher.zip;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/jgl3.1.0.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/staticdata.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/vbjorb.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/MSSQL3111.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/msutil.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/mssqlserver.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/msbase.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/jms.jar;"
     append="true" />

<property
     name="-Djava.class.path"
     value="$(LIB)/jai_codec.jar;"
     append="true" />

<property
     name="-Djava.class.path"
```

```
            value="$(LIB)/jai_core.jar;"
            append="true" />

        <property
            name="-Djava.class.path"
            value="$(LIB)/mlibwrapper_jai.jar;"
            append="true" />

        <property
            name="-Djava.class.path"
            value="$(LIB)/vantagePoint.jar;"
            append="true" />

        <property
            name="-Djava.class.path"
            value="$(LIB)/bsh-1.2b5.jar;"
            append="true" />

        <property
            name="-Djava.class.path"
            value="$(LIB)/jsfc.zip;"
            append="true" />

        <property
            name="-Djava.class.path"
            value="$(LIB)/jhall.jar;"
            append="true" />

        <property
            name="-Djava.class.path"
            value="$(LIB)/js.jar;"
            append="true" />

        <property
            name="-Djava.class.path"
            value="$(LIB)/classes111.zip"
            append="true" />

        <property
            name="-DuseRMIRegistry"
            value="false" />
    </JVMSettings>

</Default>

<TMSProxyList>
    <TMSProxy
        kind="java"
        bootstrapDataFile="../../../cfg/BackendBootstrap.xml"
        type="ELTClientApp"
        name="ELTClientApp"
    >
        <JVMSettings>
            <property
```

```
                    name="-DmaxMemorySizeForMonitoring"
                    value="500" />
                <option
                    name="-Xms"
                    value="128m" />
                <option
                    name="-Xmx"
                    value="512m" />
                <property
                    name="-DsecurityHome"
                    value="/ElTrader/TMS/Security" />
                <property
                    name="-DobjectsHome"
                    value="/ElTrader/TMS/Objects" />
                <property
                    name="-DuseRMIRegistry"
                    value="false" />
                <property
                    name="-DdirectoryUserHomeBase"
                    value="/ElTrader/TMS/Users" />
                <property
                    name="-DdirectoryUserHomeDefault"
                    value="/ElTrader/TMS/Users/all"/>
                <property
                    name="-DVM_THREAD_BUG"
                    value="1" />
                <property
                    name="-Dsun.rmi.dgc.client.gcInterval"
                    value="300000" />
                <property
                    name="-Dsun.rmi.dgc.server.gcInterval"
                    value="300000" />
                <property
                    name="-DeventQueueDebugParams"
                    value="true,1000,500,1000,100" />
            </JVMSettings>
        </TMSProxy>
    </TMSProxyList>

</Bootstrap>
```

To call API methods the handle to ITMSRemoteClientPtr must be obtained by calling the TMSClientSystem::getRemoteClient() method (see sample app).

2.2 Uninitialization

To uninitialize TMSClientSystem you should call this method:

```
TMSClientSystem::uninitialize();
```

NO TMS API CALLS CAN BE MADE AFTER TMS IS UNINITIALIZED.

3 Using "subscription" set of ITMSRemoteClient and ITMSRemoteMarketPortfolio interfaces

3.1 Subscribing for order data through ITMSRemoteClient

- *void subscribeForOrderData(const UserTicket &userTicket, ITMSOrderEventRemoteListenerPtr listener, bool includePortfolioOrders) throw(TMSException)*

  Subscribes the listener module for events carrying the information about new orders or changes in the existing orders' fields. For example, each time a fill arrives for an order the FillQty field of an order will change and the method onOrderUpdate() of the application's listening module will be called. The listening module must implement interface

  ITMSOrderEventRemoteListener. See "Listener interfaces" section for more details. This method can be used to only monitor orders that were NOT sent as portfolio target slices if parameter includePortfolioOrders is set to "false". If it is set to "true" then events with information about all orders in the system will be sent to the listener.

- *void unsubscribeFromOrderData(const UserTicket &userTicket, ITMSOrderEventRemoteListenerPtr listener) throw(TMSException)*

  Unsubscribes the module from order-related event flow.

3.2 Subscribing for order data through ITMSRemoteMarketPortfolio

- *void subscribeForOrderData(UserTicket userTicket, ITMSOrderEventRemoteListener listener, boolean includePortfolioOrders) throws TMSException*

  Subscribes the listener module for events carrying the information about new orders or changes in the existing orders' fields. For example, each time a fill arrives for an order the FillQty field of an order will change and the method onOrderUpdate() of the application's listening module will be called. The listening module must implement interface

  ITMSOrderEventRemoteListener. See "Listener interfaces" section for more details. This is used to monitor orders that were sent as this portfolio's target slices.

- *void unsubscribeFromOrderData(UserTicket userTicket, ITMSOrderEventRemoteListener listener) throws TMSException*

  Unsubscribes the module from order-related event flow.

3.3 Subscribing for portfolio data through ITMSRemoteClient

- *void subscribeForPortfolioData (const UserTicket &userTicket, ITMSMarketPortfolioEventRemoteListenerPtr listener) throw(TMSException)*

  Subscribes the module for events carrying the information about field changes of any system portfolio. For example, every time a fill arrives for an order released from target of a portfolio the field FillQty of the portfolio will change and the listener will be notified. The listening module must implement interface *ITMSMarketPortfolioEventRemoteListener*. See "Listener interfaces" section for more details.

- *void unsubscribeFromPortfolioData(const UserTicket &userTicket, ITMSMarketPortfolioEventRemoteListenerPtr listener) throw(TMSException)*

  Unsubscribes the module from portfolio-related event flow.

3.4 Subscribing for portfolio data through ITMSRemoteMarketPortfolio

- *void subscribeForPortfolioData (const UserTicket &userTicket, ITMSMarketPortfolioEventRemoteListenerPtr listener) throw(TMSException)*

  Subscribes the module for events carrying the information about changes in this portfolio fields. For example, every time a fill arrives for an order released from target of this portfolio the field FillQty of the portfolio will change and the listener will be notified. The listening module must implement interface *ITMSMarketPortfolioEventRemoteListener*. See "Listener interfaces" section for more details.

- *void unsubscribeFromPortfolioData(const UserTicket &userTicket, ITMSMarketPortfolioEventRemoteListenerPtr listener) throw(TMSException)*

  Unsubscribes the module from portfolio-related event flow.

3.5 Subscribing for target data through ITMSRemoteMarketPortfolio

- *void subscribeForTargetData (const UserTicket &userTicket, ITMSMarketTargetEventRemoteListenerPtr listener) throw(TMSException)*

  Subscribes the module for events carrying the information about changes in this portfolio's targets fields. For example, every time a fill arrives for an order released from target of this portfolio the field FillQty of the target will change and the listener will be notified. The listening module must implement interface *ITMSMarketTargetEventRemoteListener*. See "Listener interfaces" section for more details.

- *void unsubscribeFromTargetData(const UserTicket &userTicket, ITMSMarketTargetEventRemoteListener listener) throws TMSException*

  Unsubscribes the module from portfolio-related event flow.

3.6 Subscribing for market data through ITMSRemoteClient

- *void subscribeForMarketData(const UserTicket &userTicket, const String &instrumentId, ITMSMarketDataEventListenerPtr listener) throw(TMSException)*

  Subscribes the listener module for market data events for the given instrument. For example, each time a tick arrives for instrument *instrumented* the method onMarketDataUpdate() of the application's listening module will be called. The listening module must implement interface ITMSMarketDataEventListener. See "Listener interfaces" section for more details.

- *void unsubscribeFromMarketData(const UserTicket &userTicket, const String &instrumentId, ITMSMarketDataEventListenerPtr listener) throw(TMSException)*

  Unsubscribes the module from market data events for the given instrument.

- *void subscribeForMarketData(const UserTicket &userTicket, const StringVector &instrumentIds, ITMSMarketDataEventListenerPtr listener) throw(TMSException)*

  Subscribes the listener module for market data events for the given list of instruments.

- *void unsubscribeFromMarketData(const UserTicket &userTicket, const StringVector &instrumentIds, ITMSMarketDataEventListenerPtr listener) throw(TMSException)*

  Unsubscribes the module from market data events for the given list of instruments.

3.7 Subscribing for custom data through ITMSRemoteClient

When custom record data source is plugged into TMS server backend it is possible to subscribe remote applications to this source's event flow

- *void subscribeForCustomRecordData(const UserTicket &userTicket, const String &datasourceName, const String &recordId, ITMSCustomRecordDataEventListenerPtr listener) throw(TMSException)*

  Subscribes the listener module for custom record events from the specified source for the given record id.

- *void unsubscribeFromCustomRecordData(const UserTicket &userTicket, const String &datasourceName, const String &recordId, ITMSCustomRecordDataEventListenerPtr listener) throw(TMSException)*

  Unsubscribes the module from custom record events for the given record id.

- *void subscribeForCustomRecordData(const UserTicket &userTicket, const String &datasourceName, const StringVector &recordIds, ITMSCustomRecordDataEventListenerPtr listener) throw(TMSException)*

  Subscribes the listener module for custom record events from the specified source for the given record ids.

- *void unsubscribeFromCustomRecordData(const UserTicket &userTicket, const String &datasourceName, const StringVector &recordIds, ITMSCustomRecordDataEventListenerPtr listener) throw(TMSException)*

  Unsubscribes the module from custom record events for the given list of record ids.

3.8 Subscribing for system-wide position data through ITMSRemoteClient

- *void subscribeForGlobalPositionData(const UserTicket &userTicket, const String &instrumentId, ITMSPositionEventRemoteListenerPtr listener) throw(TMSException)*

  Subscribes the module for events carrying the information about changes in instrument position. For example, every time a fill arrives for an order for a specific instrument the instrument's position will change and the listener will be notified. The listening module must implement interface *ITMSPositionRemoteListener*. See "Listener interfaces" section for more details.

- *void unsubscribeFromMarketData(const UserTicket &userTicket, const String &instrumentId, ITMSPositionEventRemoteListenerPtr listener) throw(TMSException)*

Unsubscribes the module from position events for specific instrument.

- *`void subscribeForGlobalPositionData(const UserTicket &userTicket, ITMSPositionEventRemoteListenerPtr listener) throw(TMSException)`*

  Subscribes the module for events carrying the information about changes in ANY instrument position. For example, every time a fill arrives for an order for a specific instrument the instrument's position will change and the listener will be notified. The listening module must implement interface *ITMSPositionRemoteListener*. See "Listener interfaces" section for more details.

- *`void unsubscribeFromMarketData(const UserTicket &userTicket, ITMSPositionEventRemoteListenerPtr listener) throw(TMSException)`*

  Unsubscribes the module from position events.

3.9 Listener interfaces

3.9.1 Interface ITMSOrderEventRemoteListener

The listening component's methods from this interface will be called by the TMS if subscribeForOrderData() call was issued by the application.

- *`void onOrderAdded(const String &orderId, IRecordPtr record)`*

  This method is called when a new order is issued in the TMS. Use methods of IRecord interface to examine values of the record representing this order.

- *`void onOrderUpdate(const String &orderId, IRecordUpdatePtr recordUpdate)`*

  This method is called when any of the order fields change (e.g. status or fillQty). Use methods of *IRecordUpdate* interface to examine changed values of the record representing this order.

- *`void onOrderDataFeedDisconnected()`*

  This method is called when the listener is disconnected from the remote event source for any reason. The listener would try to reconnect automatically.

- *`void onOrderDataFeedReconnected()`*

  This method is called when the listener is automatically reconnected to the remote source after the disconnect. **NOTE: after the reconnect the *onOrderAdded()* method will be called for all orders in TMS.**

- *`void onInitialStateReceived()`*

  When application issues a subscribe() call it first will be notified of all existing market orders in the system(or portfolio) through the *onOrderAdded ()* method and then it will continue to receive additional events as orders get added/removed/modified. The *onInitialStateReceived()* method of the listener will be called immediately after the last *onOrderAdded* () method for an existing order is called to mark the end of the initial state.

3.9.2 Interface ITMSMarketPortfolioEventRemoteListener

The listening component's methods from this interface will be called by the TMS if subscribeForPortfolioData() call was issued by the application.

- `void onPortfolioAdded (const String &portfolioName, IRecordPtr record)`

  This method is called when a new portfolio is added to TMS. Use methods of IRecord interface to examine values of the record representing this portfolio.

- `void onPortfolioUpdate(const String &portfolioName, IRecordUpdatePtr recordUpdate)`

  This method is called when any of the portfolio fields change (e.g. PnL or fillQty). Use methods of *IRecordUpdate* interface to examine changed values of the record representing this portfolio.

- `void onPortfolioRemoved(const String &portfolioName)`

  This method is called when portfolio is removed.

- `void onPortfolioDataFeedDisconnected()`

  This method is called when the listener is disconnected from the remote event source for any reason. The listener would try to reconnect automatically.

- `void onPortfolioDataFeedReconnected()`

  This method is called when the listener is automatically reconnected to the remote source after the disconnect. ***NOTE: after the reconnect the onPortfolioAdded() method will be called for all portfolios in TMS.***

- `void onInitialStateReceived()`

  When application issues a subscribe() call it first will be notified of all existing market portfolios in the system through the *onPortfolioAdded()* method and then it will continue to receive additional events as portfolios get added/removed/modified. The *onInitialStateReceived()* method of the listener will be called immediately after the last *onPortfolioAdded*() method for an existing portfolio is called to mark the end of the initial state.

3.9.3 Interface ITMSMarketTargetEventRemoteListener

The listening component's methods from this interface will be called by the TMS if subscribeForTargetData() call was issued by the application.

- `void onTargetAdded (TargetId targetId, IRecordPtr record)`

  This method is called when a new target is added to the portfolio. Use methods of IRecord interface to examine values of the record representing this portfolio.

- `void onTargetUpdate(TargetId targetId, IRecordUpdatePtr recordUpdate)`

  This method is called when any of the target's fields change (e.g. PnL or fillQty). Use methods of *IRecordUpdate* interface to examine changed values of the record representing this target.

- `void onTargetRemoved(TargetId targetId)`

  This method is called when target is removed.

- `void onTargetPaused(TargetId targetId)`

This method is called when target is paused.

- `void onTargetResumed(TargetId targetId)`

  This method is called when target is resumed.

- `void onTargetTerminated(TargetId targetId)`

  This method is called when target is terminated.

- `void onTargetDataFeedDisconnected()`

  This method is called when the listener is disconnected from the remote event source for any reason. The listener would try to reconnect automatically.

- `void onTargetDataFeedReconnected()`

  This method is called when the listener is automatically reconnected to the remote source after the disconnect. **NOTE: after the reconnect the *onTargetAdded()* method will be called for all targets in the portfolio.**

- `void onInitialStateReceived()`

  When application issues a subscribe() call it first will be notified of all existing targets in the portfolio through the *onTargetAdded()* method and then it will continue to receive additional events as targets get added/removed/modified. The *onInitialStateReceived()* method of the listener will be called immediately after the last *onTargetAdded*() method for an existing target is called to mark the end of the initial state.

3.9.4 Interface ITMSMarketDataEventListener

The listening component's methods from this interface will be called by the TMS if one of the subscribeForMarketData() calls was issued by the application.

- `void onMarketDataUpdate(const String &instrumentId, IRDRecordPtr record)`

  called when the TMS processes a market data tick for an instrument for which there was a subscription.

- `void onMarketDataFeedDisconnected()`

  called when the TMS detects the disconnect from the market data feed.

- `void onMarketDataFeedReconnected ()`

  called when the TMS detects the connect from the market data feed.

3.9.5 Interface ITMSCustomRecordDataEventListener

The listening component's methods from this interface will be called by the TMS if one of the subscribeForCustomRecordData() calls was issued by the application.

- `void onCustomRecordDataUpdate(const String &dataSourceName, const String &recordId, IRDRecordPtr record)`

  called when the TMS processes a custom record update for a record id for which there was a subscription.

- `void onCustomRecordDataFeedDisconnected()`

called when the TMS detects the disconnect from the custom data feed.

- `void onCustomRecordDataFeedReconnected()`

  called when the TMS detects the connect from the custom data feed.

3.9.6 Interface ITMSPositionEventRemoteListener

The listening component's methods from this interface will be called by the TMS if one of the subscribeForGlobalPositionData() calls was issued by the application.

- `void onPositionAdded(IRecordPtr positionRecord)`

  This method is called when position for a new instrument is recorded in TMS. Use methods of IRecord interface to examine values of the record representing this position.

- `void onPositionUpdate(IRecordUpdatePtr positionRecordUpdate)`

  This method is called when any of the position fields change (e.g. BuyFillQty). Use methods of *IRecordUpdate* interface to examine changed values of the record representing this position.

- `void onPositionDataFeedDisconnected()`

  This method is called when the listener is disconnected from the remote event source for any reason. The listener would try to reconnect automatically.

- `void onPositionDataFeedReconnected()`

  This method is called when the listener is automatically reconnected to the remote source after the disconnect. **NOTE: after the reconnect the *onPositionAdded()* method will be called for all portfolios in TMS.**

- `void onInitialStateReceived()`

  When application issues a subscribe() call it first will be notified of all existing positions in the system through the onPositionAdded () method and then it will continue to receive additional events as positions get added/removed/modified. The *onInitialStateReceived()* method of the listener will be called immediately after the last *onPositionAdded ()* method for an existing position is called to mark the end of the initial state.
  This method is called when the listener is automatically reconnected to the remote source after the disconnect. **NOTE: after the reconnect the *onPositionAdded()* method will be called for all portfolios in TMS.**

4 Using "instruction" set of ITMSRemoteClient API

4.1 Getting resources

- *ITMSRemoteMarketPortfolioPtr getMarketPortfolio(const String &name) throw(TMSException)*

  Get an instance of the portfolio with the given name

- *ITMSRemoteMarketPortfolioPtr addMarketPortfolio(const UserTicket &userTicket,*
  const String &name, int type) throw(TMSException)

  Add new portfolio with of the specified type and name. There are currently two portfolio types that can be used: 0 – index portfolio (qauntity and side of the target is not defined), 1 – market portfolio (quantity and side of the target is defined). This method returns a handle to the newly created portfolio

- *IRDRecordPtr getMarketDataRecord(const String &instrumentId)*

  Get market data record for the specified instrument

- *ITMSStaticDataAccessorPtr getStaticDataSource()*

  Get a handle to the security master resource (see Static Data section below)

- *IRDRecordPtr getCustomDataRecord(const String &dataSouceName, const String &recordId)*

  Get custom data record from the specified source for the specified record id

4.2 Stand-alone order instructions

- *ActionId sendOrder(const UserTicket &userTicket,*
  TMSNewOrderMessagePtr orderMessage) throw(TMSException)

  Send an order through the FIX connection with the field specified in the TMSNewOrderMessage object. The transaction destination will be one of the fields in the orderMessage

- *ActionId sendOrders(const UserTicket &userTicket,*
  const TMSNewOrderMessageVector &orderMessages) throw(TMSException)

  Send multiple orders through the FIX connection(s) with the fields for each order specified in the TMSNewOrderMessage object from the array. The transaction destination will be one of the fields in each orderMessage

- *ActionId modifyOrder(const UserTicket &userTicket, const String &orderId,*
  const TMSModifyOrderMessageVector &fields) throw(TMSException)

  Modify order with id orderId. The new fields are specified in the TMSModifyOrderMessage object

- *ActionId modifyOrders(const UserTicket &userTicket, const StringVector &orderIds,*
  const TMSModifyOrderMessageVector &fields) throw(TMSException)

  Modify multiple orders.

- *ActionId modifyOrdersToMarketOrdType(const UserTicket &userTicket,*
  const StringVector &orderIds) throw(TMSException)

  Modify multiple orders to order type "Market"

- *ActionId cancelOrder(const UserTicket &userTicket, const String &orderId,*
  TMSCancelOrderMessagePtr orderMessage) throw(TMSException)

  Cancel order with id orderId. The additional fields of the Cancel request may be specified in the TMSCancelOrderMessage object

- *ActionId cancelOrders(const UserTicket &userTicket, const StringVector &orderIds,*
  const TMSCancelOrderMessageVector &orderMessages) throw(TMSException)

  Cancel multiple orders

- *ActionId rejectOrder(const UserTicket &userTicket,*
  TMSRejectOrderMessage orderMessage) throw(TMSException)

  Reject order with id orderId. The fields of the Reject report will be specified in the TMSRejectOrderMessage object

- *ActionId rejectOrders(const UserTicket &userTicket,*
  const TMSRejectOrderMessageVector &orderMessages) throw(TMSException)

  Reject multiple orders

- *ActionId confirmOrder(const UserTicket &userTicket,*
  TMSConfirmOrderMessagePtr orderMessage) throw(TMSException)

  Confirm order with id orderId. The fields of the Confirm report will be specified in the TMConfirmOrderMessage object

- *ActionId confirmOrders(const UserTicket &userTicket,*
  const TMSConfirmOrderMessageVector &orderMessages) throw(TMSException)

  Confirm multiple orders

- *ActionId fillOrder(const UserTicket &userTicket,*
  TMSFillOrderMessagePtr orderMessage) throw(TMSException)

  Fill order with id orderId. The fields of the Fill report will be specified in the TMSFillOrderMessage object

- *ActionId fillOrders(const UserTicket &userTicket,*
  const TMSFillOrderMessageVector &orderMessages) throw(TMSException)

  Fill multiple orders.

4.3 Portfolio instructions

In this section all methods of ITMSRemoteMarketPortfolio interface are described. The handle to ITMSRemoteMarketPortfolio is obtained either by getMarketPortfolio or addMarketPortfolio methods described above.

- *const String &getName()*

  Returns portfolio name.

- *int getType()*

  Returns portfolioType. One of the

```
INDEX_MARKET_PORTFOLIO = 0;
   PURE_MARKET_PORTFOLIO = 1;
   LINKED_MARKET_PORTFOLIO = 2;
   CORRELATED_MARKET_PORTFOLIO = 3;
```

4.3.1 Portfolio targets instructions

- *ActionId addTargets(const UserTicket &userTicket, const FieldsContainerVector &fields) throw(TMSException)*

  Adds new targets to a portfolio. The fields of the targets are given in the FieldsContainer array.

- *TargetId addTargetAndGetId(const UserTicket &userTicket, FieldsContainerPtr fields) throw(TMSException)*

  Adds new target to a portfolio. The fields of the target are given in the FieldsContainerPtr. This method returns the ID of added target that can be used by the application later.

- *void addTargetsAndGetIds(TargetIdVector &ids, const UserTicket &userTicket, const FieldsContainerVector &fields) throw(TMSException)*

  Adds new targets to a portfolio. The fields of the targets are given in the FieldsContainer array. This method fills the ids array (1st parameter) by the ID's of added targets (in the same order as the order of target fields) that can be used by the application later to do some operations on the targets.

- *IRecordPtr addAndGetTarget(const UserTicket &userTicket, FieldsContainerPtr fields) throw(TMSException)*

  Adds new target to a portfolio. The fields of the target are given in the FieldsContainerPtr. This method returns the added target as IRecordPtr.

- *void addAndGetTarget(IRecordVector &targets, const UserTicket &userTicket, FieldsContainerPtr fields) throw(TMSException)*

  Adds new targets to a portfolio. The fields of the targets are given in the FieldsContainerVector. This method fills targets array (1st parameter) with the targets added.

- *IRecordPtr getTarget(const UserTicket &userTicket, TargetId targetId) throw(TMSException)*

  This method can be used to synchronously obtain the record for target of the portfolio with target ID targetId.

- *void getTargets(IRecordVector &targets, const UserTicket &userTicket, const TargetIdVector &targetIds) throw(TMSException)*

  This method can be used to synchronously obtain the records for targets of the portfolio with target IDs targetIds.

- *void getTargets(IRecordVector &targets, const UserTicket &userTicket, const String &filterExpression) throw(TMSException)*

  This method can be used to synchronously obtain the records for all targets of the portfolio. The filterExpression can be used to filter out targets; in case it is empty all targets will be returned.

- *ActionId modifyTargets(const UserTicket &userTicket, const TargetIdVector &targetIds,*
  const FieldsContainerVector &fields) throw(TMSException)

  Modifies targets with given ids. The new targets' fields are given in the FieldsContainer array.

- *ActionId removeTargets(const UserTicket &userTicket, const TargetIdVector &targetIds) throw(TMSException)*

  Removes targets with given ids from the portfolio

- *ActionId removeAllTargets(const UserTicket &userTicket) throw(TMSException)*

  Removes all targets from the portfolio

4.3.2 Portfolio orders instructions

- *ActionId sendOrder(const UserTicket &userTicket, TargetId targetId,*
  TMSNewOrderMessagePtr orderMessage) throw(TMSException)

  Send an order through the FIX connection with the field specified in the TMSNewOrderMessage object. The remaining fields will be set in the order from the target with id targetId. Basically, the target has a role of a template from which the orders are sliced. Target's released/unreleased qty will change.

- *ActionId sendOrders(const UserTicket &userTicket, const TargetIdVector &targetIds,*
  const TMSNewOrderMessageVector &orderMessages) throw(TMSException)

  Same as sendOrder, only sends multiple.

- *ActionId modifyOrdersToMarketOrdType(const UserTicket &userTicket,*
  int waveNumber) throw(TMSException)

- *ActionId cancelOpenOrders(const UserTicket &userTicket, TargetId targetId) throw(TMSException)*

  Cancel open orders released from the target with id targetId

- *ActionId cancelOpenOrders(const UserTicket &userTicket) throw(TMSException)*

  Cancel all open orders released from the targets of this portfolio

4.3.3 Portfolio wave instructions

- *ActionId startWave(const UserTicket &userTicket) throw(TMSException)*

  Set the wave marker. After this call any orders released from the portfolio targets will have new wave id

- *ActionId sendWave(const UserTicket &userTicket, const TargetIdVector &targetIds,*
  const TMSNewOrderMessageVector &orderMessages,
  FieldsContainerPtr targetFields) throw(TMSException)

  Starts a new wave and sends orders as part of that wave. Each targetId from the targetIds array has a corresponding orderMessage from orderMessages array

- *ActionId sendWave(const UserTicket &userTicket) throw(TMSException)*

  Starts new wave and sends an order for all targets of the portfolio based on field values and instructions in the targets

- `ActionId cancelWave(const UserTicket &userTicket, int waveNumber) throw(TMSException)`

  Cancel all orders with given wave id

4.4 Accessing Market Data

Call method getMarketDataRecord*(const String &instrId)* of the ITMSRemoteClient interface to get market data record for the given instruments. Then the fields of the record may be examined.

4.5 Accessing Static Data

Call method *getStaticDataSource()* of the ITMSRemoteClient interface to get a handle to *ITMSStaticDataAccessor* object. Then use methods of *ITMSStaticDataAccessor* to get the necessary static data.

5 Entity Objects and Interfaces

The methods of the ITMSRemoteClient and ITMSRemoteMarketPortfolio API often return objects to the caller or require objects to be passed as parameters. The interfaces to these objects that represent orders/targets/portfolios/messages in the TMS are described below.

5.1.1 Interface IRecord

- `double getNumericFieldValue(const String &fieldId)`
- `const String &getStringFieldValue(const String &fieldId)`
- `char getCharFieldValue(const String &fieldId)`
- `bool isFieldNumeric(const String &fieldId)`
- `bool getBooleanFieldValue(const String &fieldId)`
- `long getTimeFieldValue(const String &fieldId)`

5.1.2 TMSNewOrderMessage

This object extends ELTFieldGroup and has the same API as ELTFieldGroup.

5.1.3 TMSModifyOrderMessage

This object extends ELTFieldGroup and has the same API as ELTFieldGroup.

5.1.4 TMSCancelOrderMessage

This object extends ELTFieldGroup and has the same API as ELTFieldGroup.

5.1.5 TMSConfirmOrderMessage

This object extends ELTFieldGroup and has the same API as ELTFieldGroup.

5.1.6 TMSFillOrderMessage

This object extends ELTFieldGroup and has the same API as ELTFieldGroup.

5.1.7 TMSRejectOrderMessage

This object extends ELTFieldGroup and has the same API as ELTFieldGroup.

5.1.8 ELTFieldGroup

See "InfoReach FIX CPP Programmers Guide.doc" section 2.2.1

5.1.9 FieldsContainer

- *setFieldValue(const STDNAMESPACE string &fieldId, const STDNAMESPACE string &value)*
- *void setFieldValue(const STDNAMESPACE string &fieldId, double value)*
- *void setFieldValue(const STDNAMESPACE string &fieldId, bool value)*
- *void setFieldValue(const STDNAMESPACE string &fieldId, char value)*
- *void setFieldValue(const STDNAMESPACE string &fieldId, long value)*
- *void clear()*
- *bool containsStringField(const STDNAMESPACE string &fieldId)*
- *bool containsNumericField(const STDNAMESPACE string &fieldId)*
- *const String &getStringFieldValue(const STDNAMESPACE string &fieldId)*
- *double getNumericFieldValue(const STDNAMESPACE string &fieldId)*
- *StringFieldMap::iterator getStringFieldsBegin()*
- *StringFieldMap::iterator getNumericFieldsEnd()*

6 Using ITMSRemoteClient API for report data monitoring

6.1 Requesting data snapshots

The user's application can request data from the report residing on the server in a form of a snapshot. Once the snapshot request is made the application's listener component will receive report data for all records/nodes of the report corresponding to the request's query. The returned snapshot may come in multiple events. The first and last events will be marked with special flags.

The following method of ITMSRemoteClient can be used to request data from any report including any user-defined report.

- ```
  void requestReportData (const UserTicket &userTicket,
                          const String &domainManagerName,
                          const String &reportName,
                          const String &queryString,
                          IRPTEventListenerPtr listener)
          throw(RPTException)
  ```

The client application calls this method whenever it needs to obtain information about nodes from arbitrary levels of a given tree report. The first two parameters identify the report by name and by the name of the domain where this report was instantiated. By default, there is only one domain for reports in the TMS: "Portfolio report manager". Depending on the TMS configuration more domain managers may be present.

queryString parameter is used to specify data in which listener is interested. Current API accepts only one form of the query, which is created by method:

- ```
  String createQuery(
        const String& levelId,
        const String& filterExpression,
        const String& fieldMetaDataName);
  ```

This method takes as parameters the level id, filtering criteria and name of the metadata containing field descriptions used in the report . The name of the metadata is necessary because each report can have its own metadata. In most common cases reports that group order transactions have "Single order tree report levels and fields" metadata. createQuery method covers the most common use case when information can be requested from all nodes on a certain level satisfying filtering criteria. In the example below information is requested from all nodes of the report 'Instrument Trade PnL' that reside on the 'OrdTrnId' level with an instrument 'IBM':

```
//ITMSRemoteClientPtr client
client->requestReportData(
           UserTicket::getSystemTicket(),
           "Portfolio report manager",
           "Instrument Trade PnL",
           client->createQuery(
               "OrdTrnId",
               "\"Instrument\" = 'IBM'",
               "Single Order hierarchical report fields"),
           listener);
```

Again, the requested data may come in multiple *RPTStateEvent* events. The first and last event will be specially marked. Each event will contain multiple records (one per transaction) and each record will contain the field values that can be examined. See Listing 1 for an example of the implementation of the IRPTEventListener interface.

```
class SampleListener : public IRPTEventListener
{
private:
    STDNAMESPACE string name_;
public:
    SampleListener(const STDNAMESPACE string &name) : name_(name) {}
    virtual ~SampleListener() {}

    virtual void processEvent(const RPTStateEvent& event) const
    {
        STDNAMESPACE ostringstream output;
        output << "\n\n(" << name_ << ") ";
        if (event.isLast())
        {
            // On request state, we will know when we have all the
            // data when we recieve the state event where isLast() is
            // true!
            output << "last ";
        }
        output << "stateEvent (" << event.getEventCount() << ") ---> " <<
event.stringValue();

        for (int i=0; i < event.getEventCount(); i++)
        {
            processEvent(dynamic_cast<const
RPTRecordEvent&>(*event.getEventAt(i)));
        }

        STDNAMESPACE cout << output.str() << STDNAMESPACE flush;
    }

    virtual void processEvent(const RPTRecordEvent& event) const
    {
        STDNAMESPACE ostringstream output;
        output << "\n\n(" << name_ << ") processing record ";
        if(event.getAction() == RPTRecordEvent::EVENT_RECORD_CHANGED)
        {
            output << "changed";
        }
        else if(event.getAction() == RPTRecordEvent::EVENT_RECORD_INSERTED)
        {
            output << "inserted";
        }
        else if(event.getAction() == RPTRecordEvent::EVENT_RECORD_REMOVED)
        {
            output << "removed";
        }
        else if(event.getAction() == RPTRecordEvent::EVENT_RECORD_UPDATED)
```

```
        {
            output << "updated";
        }

        IRPTRecordPtr record = event.getNewRecord();
        IRPTRecordContextPtr recordContext = record->getRecordContext();
        output << " ID=" << record->getIdentifier();

        // Extract information from the record by Id.
        for(int i = 0; i < record->getRecordContext()->getStringFieldCount()
+
                        record->getRecordContext()->getNumericFieldCount();
i++)
        {
            output << ", " << recordContext->getFieldIdentifierAt(i) << "=";
            if(recordContext->isFieldNumericAt(i))
            {
                if(record->isFieldSetAt(i))
                    output << record->getNumericFieldValueAt(i);
                else
                    output << "NaN";
            }
            else
            {
                if(record->isFieldSetAt(i))
                    output << record->getStringFieldValueAt(i);
                else
                    output << "null";
            }
        }
        output << STDNAMESPACE endl;

        // Extract information from the record by field name.
        int fieldIndx;

        for (int i=0; FIELD_IDS[i] != ""; i++)
        {
            fieldIndx = recordContext->getFieldIndex(FIELD_IDS[i]);
            if (fieldIndx >= 0)
            {
                if(recordContext->isFieldNumericAt(fieldIndx))
                {
                    if(record->isFieldSetAt(fieldIndx))
                        output  <<  FIELD_IDS[i]  <<  "  =  "  <<  record-
>getNumericFieldValueAt(fieldIndx) << STDNAMESPACE endl;
                    else
                        output << "NaN";
                }
                else
                {
                    if(record->isFieldSetAt(fieldIndx))
                        output  <<  FIELD_IDS[i]  <<  "  =  "  <<  record-
>getStringFieldValueAt(fieldIndx)  << STDNAMESPACE endl;
                    else
```

```
                        output << "null";
                    }
                }
            }

            STDNAMESPACE cout << output.str() << STDNAMESPACE flush;
        }
};
```

Listing 1.

6.2 Subscribing for data flow

The user's application can subscribe for the data flow from the report residing on the server. When the subscribe call is made the data currently existent in the report will be sent to the calling component in a form of a snapshot (in multiple state events) and then the record events will continuously arrive from the report until unsubscribe call is made. The following method of ITMSRemoteClient can be used to subscribe for data from any report including any user-defined report.

- ```
  void subscribeForReportData (const String& domainManagerName,
                               const String& reportName,
                               const String& queryString,
                               IRPTReconnectableEventListenerPtr listener)
      throw(RPTException)
  ```

The client application calls this method whenever it needs the data flow for nodes from arbitrary levels of a given report. The first two parameters identify the report by name and by the name of the domain manager where this report was instantiated. By default, there is only one domain for reports in the TMS: "Portfolio report manager". Depending on the TMS configuration more domain managers may be present.

The queryString parameter should be created via createQuery method, which covers the most common use case when information can be requested from all nodes on a certain level satisfying filtering criteria. In the example below information is requested from all nodes of the report 'Instrument Trade PnL' that reside on the 'OrdTrnId' level with an instrument.

```
client->subscribeForReportData(UserTicket::getSystemTicket(),
                               "Portfolio report manager",
                               "Instrument Trade PnL",
                                client->createQuery(
                               "OrdTrnId",
                               "\"Instrument\" = 'IBM'",
                               "Single Order hierarchical report fields"),
                                listener);
```

The createQuery method takes as parameter the level id, filtering criteria (can be empty string "") and name of the metadata containing field descriptions used in the report. The name of the metadata is necessary because each report can have its own metadata. In most common cases reports that group order transactions have "Single order tree report levels and fields" metadata.

After the request is made all data from the report nodes currently satisfying the criteria will be sent to the listener in multiple *RPTStateEvent* events. The first and the last events will be specially marked. Each
```

event will contain multiple records (one per transaction) and each record will contain the field values that can be examined.

After the initial state is sent to the subscribing component the record events corresponding to the changes in the report will be sent continuously to the subscribing component until an unsubscribe call is made. Each record event will contain information on what exactly happened to the report's nodes and the nodes' fields values.

There are three possible changes that can be reflected in the record event:

```
RPTRecordEvent::EVENT_RECORD_CHANGED --- sent when fields of the node change
RPTRecordEvent::EVENT_RECORD_INSERTED --- sent when new node is added
RPTRecordEvent::EVENT_RECORD_REMOVED --- sent when node is removed
```

The change indicator is the record event through the *getAction()* call. See Listing 1 for an example of the *processEvent(const RPTRecordEvent &event)* implementation. Appendix B lists the default set of fields that can be used for specifying the request criteria. Users can expand this list by developing their own fields and placing them in the reports.

6.3 Examining received record events

The record events arriving to the listening component contain records of values that can be accessed and examined. The record object is obtained from the event by calling getNewRecord() method. Once the record is obtained the fields of the record are retrieved by calling either getNumericFieldValueAt(int index) or getStringFieldValueAt(int index) methods. The type of the field dictates which method should be called. Appendix B contains description of fields that can be received inside the record event and their type and 'representation'. For fields with NUMERIC representation the getNumericFieldValueAt method should be called and for fields with STRING representation the getStringFieldValueAt method should be called. The index of the field in the record that can be passed to the get…FieldValueAt() methods is obtained from the *record context* by supplying the field's name to the record context's getFieldIndex() method. For example, to retrieve value of the Instrument field from the record the following sequence of calls should be made:

```
IRPTRecordPtr rec = recEvent.getNewRecord();
IRPTRecordContextPtr recContext = rec->getRecordContext();
fieldIndex = recContext->getFieldIndex("Instrument");
if (fieldIndex >= 0)
    STDNAMESPACE string instr = rec->getStringFieldValueAt(fieldIndex));
```

Even when representation of the field is not known it is still possible to call the right get…() method if one extra call is made to discover the representation of a given field:

```
IRPTRecordPtr rec = recEvent->getNewRecord();
IRPTRecordContextPtr recContext = rec->getRecordContext();
fieldIndex = recContext->getFieldIndex("fieldName");
if (fieldIndex >= 0)
{
   if (recContext->isFieldNumericAt(fieldIndex))
      double d = rec->getNumericFieldValueAt(fieldIndex));
   else
      String s = rec->getStringFieldValueAt(fieldIndex));
}
```

It is important to note that multiple field types have NUMERIC representation and are accessed via getNumericFieldValueAt() method. Types *double, int, long, UniLong, float, FIX Date, FIX UTCTimestamp, FIX Boolean, char* all have Numeric representation and therefore require extra manipulation of the number returned by getNumericFieldValueAt() method in order to get a final value of the right type. The samples below show how the value returned by getNumericFieldValueAt() method has to be handled in order to obtain result of the right type:

```
// for chars:
char c = getNumericFieldValueAt(index);

// for UTCTimestamp:
time_t t = getNumericFieldValueAt(index));

struct tm *stamp = gmtime(&t);

// for Date (will have hour  = 0, min = 0, sec = 0):
time_t t = getNumericFieldValueAt(index));
struct tm *stamp = gmtime(&t);

// for Boolean (FIX Boolean is actually a char with values 'Y' or 'N'):
char b = getNumericFieldValueAt(index);

// for int:
int i = getNumericFieldValueAt(index);

// for long:
long l = getNumericFieldValueAt(index);

// for float:
float f = getNumericFieldValueAt(index);
```

6.4 Unsubscribing from data flow

The following method takes as a parameter the listener object that was subscribed to the report previously and unsubscribes it.

- ```
  public void unsubscribeFromReportData(const UserTicket &userTicket,
         const STDNAMESPACE string &domainManagerName,
         const STDNAMESPACE string &reportName,
         IRPTReconnectableEventListenerPtr listener)
     throw(RPTException)
  ```

6.5 Creating report

It is possible to create completely new report on the server through by calling this method

- ```
  void createReport(const UserTicket &userTicket,
           const String &reportSpecResource) throw(RPTException)
  ```

The reportSpecResource parameter is a name of the file residing in Directory containing XML-formatted description of the report. See "

Portfolio target fields (used when working with ITMSMarketTarget, or IRecord objects representing TMS targets or TMS portfolios)

File [inforeach_home]/backend/Directory/ElTrader/TMS/Config/ElementSettings/TableElement/PortfolioTableFieldsMetaData.xml contains definition of all TMS fields for portfolio targets or portfolio summary records. When examining target/portfolio events received in the external applications or inside the analytics the exact field names and their types may be looked up in this file.

6.6 Market quote fields (used when working with IRDRecord objects)

File [inforeach_home]/backend/Directory/ElTrader/TMS/Config/MarketData/MarketQuoteFieldsMetaData.xml contains definition of all TMS fields for market data records. When examining market data events received in the external applications or inside the analytics the exact field names and their types may be looked up in this file.

Appendix C: Sample report resource" for an example of the sample report resource.

6.7 Removing report

It is possible to remove report on from the server:

- `void removeReport(const UserTicket &userTicket,`
  `const String &domainManagerName,`
  `const String &reportName) throw(RPTException)`

6.8 Checking for report

It is possible to check whether the report is available:

- `bool isReportAvailable (const String &domainManagerName,`
  `const String &reportName) throw(RPTException)`

7 Build instructions

WIN32 platform.

1. Add to the INCLUDE path of your project path to %TMSROOT%/backend/include
2. Link your application with libraries:

   portfolio.lib
   report.lib
   eltraderEngine_d.lib
   infra.lib
   eltradertms.lib

from %TMSROOT%/common/win32/native folder.

To start your application make sure %TMSROOT%/common/win32/native is in your PATH environment variable.

8 Appendix A: Sample TMS accessing application

The sample file ELTClientApp.java is located in the folder backend/samples/cpp/ELTClientApp. In the same folder the MSVC project can be found. To run the sample run backend/samples/cpp/ELTClientAppWin32.bat (the TMS server has to be running)

9 Appendix B: TMS fields

9.1 Order fields (used when working with ITMSOrder or IRecord objects representing TMS orders)

File [inforeach_home]/backend/Directory/ElTrader/TMS/Config/ElementSettings/TreeElement/SingleOrderTreeFieldsMetaData.xml contains definition of all TMS fields for Single Order Domain reports. When examining order events received in the external applications or inside the analytics the exact field names and their types may be looked up in this file. The table below describes some (but not all) fields. The fields marked by * are available only on order request level. The records from this level are only accessible to components subscribing to this order request level in the reports (see section "Using ITMSRemoteClient API for report data monitoring" of this document).

Field Name	Type	Representation	Description
OrdTrnId	String	String	The id assigned by TMS to every single order transaction
OrdRqstId*	String	**STRING**	**THE ID ASSIGNED BY TMS TO EVERY SINGLE ORDER REQUEST (ONE SINGLE ORDER TRANSACTION CONSISTS OF MULTIPLE SINGLE ORDER REQUESTS: THE ORIGINAL REQUEST AND ALL CONSEQUENT MODIFY, CANCEL REQUESTS)**
ConnName*	String	String	FIX engine connection name where request was originated
OrdRec*	int	Numeric	Index of the order message in the request node
AckRec*	int	Numeric	Index of the acknowledgement message in the request node
AckTime*	UTCTimestamp	Numeric	Time of the acknowledgement of the request
RecErrors*	int	Numeric	Errors detected in execution reports if consistency checking is turned on.
OrdStatus	char	Numeric	Valid values: A = UnAck 0 = New 1 = Partial 2 = Filled 3 = Done 4 = Canceled 5 = Replaced 6 = Pending C/R 7 = Stopped 8 = Rejected 9 = Suspended B = Calculated C = Expired D = Accepted for bidding X = Implicitly Rejected

IsOrdActive*	Boolean	Numeric	Specifies whether this request is the current active request in the transaction (usually latest modify)
ClientOrdId*	String	String	Request's ClOrdID from the order message
Side	char	Numeric	Request's Side from the order message Valid values: 1 = Buy 2 = Sell 3 = Buy minus 4 = Sell plus 5 = Sell short 6 = Sell short exempt 7 = Undisclosed (valid for IOI and List Order messages only) 8 = Cross (orders where counterparty is an exchange, valid for all messages except IOIs) 9 = Cross short A = Cross short exempt
OrdQty	float	Numeric	Request's OrdQty from the order message. Is an aggregate on levels above the transaction level
ActiveOrdQty	float	Numeric	Part of request's order qty that was not modified or canceled. Is an aggregate on levels above the transaction level
FillQty	float	Numeric	Fill qty of the request. Is an aggregate on levels above the transaction level
FillVal	double	Numeric	Fill qty * avg. fill price. Is an aggregate on levels above the transaction level
1stFillRec*	int	Numeric	Index of the first arrived fill message
1stFillTime*	UTCTimestamp	Numeric	Time of the first arrived fill message
LastFillRec*	int	Numeric	Index of the last arrived fill message
LastFillTime	UTCTimestamp	Numeric	Time of the last arrived fill message
Leaves	float	Numeric	Unfilled qty of the request. Is an aggregate on levels above the transaction level

RqstState*	int	Numeric	Internal state if the request. Only used by auxiliary TMS components like validator, router, etc. Valid values: -1 = NA 1 = PendingValidation 2 = FailedValidationProcess 3 = PendingToBeSent 4 = Internalized 5 = PendingRelease 6 = BeingSent 7 = Sent 8 = UnderSpecialHandling
LatestActiveRqstId*	String	String	Id of the latest active request of the transaction
OrdType	char	Numeric	Request's OrdType from the order message
OrdPx	double	Numeric	Request's OrdPx from the order message
StopPx	double	Numeric	Request's StopPx from the order message
OrdVal	double	Numeric	OrdQty * OrdPx. Is aggregate on higher levels. Is an aggregate on levels above the transaction level
AvgOrdPx	double	Numeric	Has meaning on higher levels: OrdVal / OrdQty
OpenOrdVal	double	Numeric	Same as OrdVal only calculated based on leaves. Is an aggregate on levels above the transaction level
OrdTime	UTCTimestamp	Numeric	Time of the processing of the order request
OrdInternalTime	UTCTimestamp	Numeric	
NetFillQty	float	Numeric	FillQty adjusted by Side. Is an aggregate on levels above the transaction level
NetFillVal	float	Numeric	FillVal adjustedby Side. Is an aggregate on levels above the transaction level
AutoTrdId	String	Numeric	Id of the AutoTrader that initiated the order
SliceNumber	int	Numeric	Sif this order was a slice of a basket target then it will have the slice number field
Custodian	String	String	Request's Custodian from the order message
DestID	String	String	Request's DestID from the order message
SenderID	String	String	Request's SenderID from the order message

LastOrdTime	UTCTimestamp	Numeric	Time of the last request of the transaction
ExecBroker	String	String	Request's ExecBroker from the order message
AvgFillPx	double	Numeric	Avg. fill price.
NetLeaves	float	Numeric	Leaves adjusted by Side. Is an aggregate on levels above the transaction level
NetOrdQty	float	Numeric	OrdQty adjusted by Side. Is an aggregate on levels above the transaction level
NetOrdVal	float	Numeric	OrdVal adjusted by Side
NetOpenOrdVal	float	Numeric	OpenOrdVal adjusted by Side. Is an aggregate on levels above the transaction level
OpenOrClose	char	Numeric	For options. Request's OpenOrClose from the order message
ExpTime	UTCTimestamp	Numeric	For options. Request's ExpTime from the order message
OrdSender	String	String	ID of the originator of the order
MsgType	String	String	Request's MsType from the order message ('D' or 'G' or 'F')
OrigClientOrdId	String	String	Request's OrigClOrdID from the order message
LeavesSlippage	float	Numeric	Leaves * (LastPx - OrdPx). Is an aggregate on levels above the transaction level
FilledSlippage	float	Numeric	FillQty * (LastPx - OrdPx). Is an aggregate on levels above the transaction level
OrdSlippage	float	Numeric	LeavesSlippage + FilledSlippage. Is an aggregate on levels above the transaction level
TargetQty	float	Numeric	If order was originated by AutoTrader or Basket this field shows original target qty
PctTargetQtyFilled	float	Numeric	FillQty / TargetQty

** - fields marked by * are only accessible from level OrdRqstId through methods subscribeForReportData or requestReportData.*

9.2 Portfolio target fields (used when working with ITMSMarketTarget, or IRecord objects representing TMS targets or TMS portfolios)

File [inforeach_home]/backend/Directory/ElTrader/TMS/Config/ElementSettings/TableElement/PortfolioTableFieldsMetaData.xml contains definition of all TMS fields for portfolio targets or portfolio summary records. When examining target/portfolio events received in the external applications or inside the analytics the exact field names and their types may be looked up in this file.

9.3 Market quote fields (used when working with IRDRecord objects)

File [inforeach_home]/backend/Directory/ElTrader/TMS/Config/MarketData/MarketQuoteFieldsMetaData.xml contains definition of all TMS fields for market data records. When examining market data events received in the external applications or inside the analytics the exact field names and their types may be looked up in this file.

10 Appendix C: Sample report resource

The following XML structure contains specification of a simple report based on the Single Order Domain base report. This tree report aggregates all single order transactions from the base report into a tree structure grouped by Account, SubAccount, and Industry. On each level the set of fields is specified. On Account, SubAccount, and Industry levels most field values are aggregations of the field values of the nodes from the levels below.

```
<?xml version='1.0'?>
<Report>
  <ReportSpecification
      type="Chain Report"
      name="Manual Orders by Account by Industry"
  >

    <ReportElement
      type="Grouping Tree"
      isEventModelReusable="false"
      purgingLevel="-1"
      reusingLevel="-1"
    >
      <Level grouper="Account">
        <Field id="OrdQty" />
        <Field id="FillQty" />
        <Field id="Leaves" />
        <Field id="NetOrdQty" />
        <Field id="NetFillQty" />
        <Field id="FillVal" />
        <Field id="NetOrdVal" />
        <Field id="NetFillVal" />
        <Field id="NetLeaves" />
      </Level>
      <Level grouper="SubAccount">
        <Field id="OrdQty" />
        <Field id="FillQty" />
        <Field id="Leaves" />
```

```
  <Field id="NetOrdQty" />
  <Field id="NetFillQty" />
  <Field id="FillVal" />
  <Field id="NetOrdVal" />
  <Field id="NetFillVal" />
  <Field id="NetLeaves" />
</Level>
<Level grouper="Industry">
  <Field id="OrdQty" />
  <Field id="FillQty" />
  <Field id="Leaves" />
  <Field id="NetOrdQty" />
  <Field id="NetFillQty" />
  <Field id="FillVal" />
  <Field id="NetOrdVal" />
  <Field id="NetFillVal" />
  <Field id="NetLeaves" />
</Level>
<Level grouper="OrdTrnId">
  <Field id="OrdTime" />
  <Field id="OrdStatus" />
  <Field id="Side" />
  <Field id="OrdQty" />
  <Field id="OrdPx" />
  <Field id="OrdType" />
  <Field id="FillQty" />
  <Field id="AvgFillPx" />
  <Field id="Leaves" />
  <Field id="FillVal" />
  <Field id="OrdVal" />
  <Field id="AcctId" />
  <Field id="TrnDest" />
  <Field id="OrdSender" />
  <Field id="Instrument" />
  <Field id="SideType" />
  <Field id="DestID" />
  <Field id="ExecBroker" />
  <Field id="Custodian" />
  <Field id="LastOrdTime" />
  <Field id="ActiveOrdQty" />
  <Field id="NetFillVal" />
  <Field id="NetOrdQty" />
  <Field id="NetFillQty" />
  <Field id="NetLeaves" />
  <Field id="NetOpenOrdVal" />
  <Field id="NetOrdVal" />
  <Field id="OpenOrdVal" />
  <Field id="Symbol" />
</Level>
<Level grouper="OrdRqstId">
  <Field id="MsgType" />
  <Field id="OrdTime" />
  <Field id="OrdStatus" />
  <Field id="Side" />
```

```
        <Field id="OrdQty" />
        <Field id="OrdPx" />
        <Field id="OrdType" />
        <Field id="FillQty" />
        <Field id="AvgFillPx" />
        <Field id="Leaves" />
        <Field id="FillVal" />
        <Field id="OrdVal" />
        <Field id="LastFillTime" />
        <Field id="AcctId" />
        <Field id="TrnDest" />
        <Field id="OrdSender" />
        <Field id="Instrument" />
        <Field id="SideType" />
        <Field id="OrigClientOrdId" />
        <Field id="ClientOrdId" />
        <Field id="SenderID" />
        <Field id="DestID" />
        <Field id="ExecBroker" />
        <Field id="Custodian" />
        <Field id="AckRec" />
        <Field id="RecErrors" />
        <Field id="OrdRec" />
        <Field id="1stFillRec" />
        <Field id="1stFillTime" />
        <Field id="LastFillRec" />
        <Field id="LastOrdTime" />
        <Field id="ActiveOrdQty" />
        <Field id="NetFillVal" />
        <Field id="NetOrdQty" />
        <Field id="NetFillQty" />
        <Field id="NetLeaves" />
        <Field id="NetOpenOrdVal" />
        <Field id="NetOrdVal" />
        <Field id="OpenOrdVal" />
        <Field id="IsOrdActive" />
        <Field id="Symbol" />
        <Field id="ConnName" />
      </Level>
    </ReportElement>
  </ReportSpecification>

  <ReportReferenceList>
    <ReportReference
       reportName="Single Order Domain Tree Report"
       managerNames="Single Order Domain Manager"
    />
  </ReportReferenceList>

</Report>
```