InfoReach

Engineered for what's next

HiFREQ

HiFREQ Java Programmer's Guide

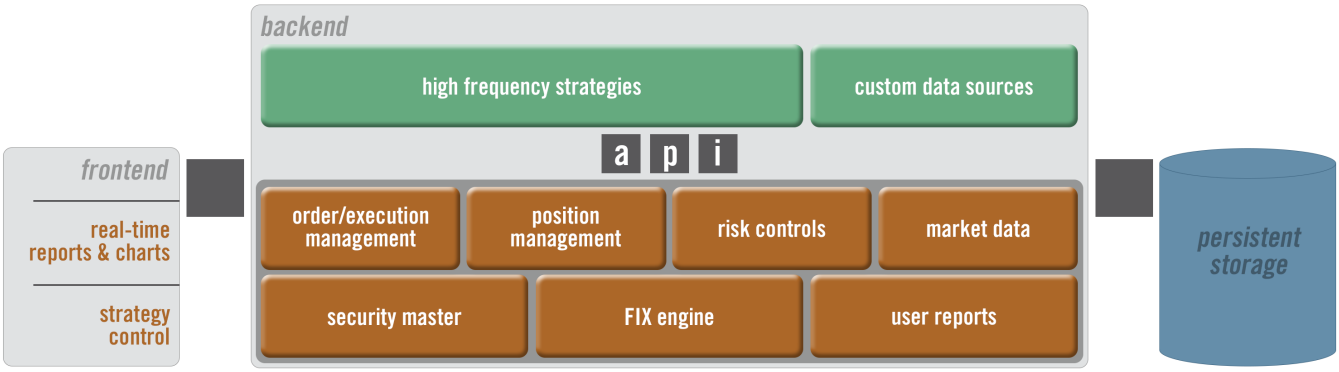
Version 9.0

Contents

Overview

HiFREQ is an event-driven algorithmic engine that gives traders the ability to employ HFT strategies for equities, futures, options, forex, and swaps trading. Its open, broker-neutral architecture allows users to create and deploy proprietary, complex trading strategies as well as access third-party algorithms. Orders can be routed to any global market destination via FIX Engine.

HiFREQ has a 3-tier architecture comprised of frontend (desktop), backend, and persistent storage. The frontend applications are used to monitor data and to enable users to control the backend (e.g. send, cancel or modify orders, change strategy parameters, etc.) InfoReach's proprietary highly optimal persistent storage allows making HiFREQ recoverable in case it has to be restarted. The backend components can reside in one or more processes and this tier is responsible for order, execution, and position management, FIX connectivity, market data feeds, strategies, risk controls, etc. Backend makes in-process and out-of-process APIs available to developers.



As part of the backend, HiFREQ core manages the state of the following objects: orders, targets (parent orders), target lists, position and strategy data. The state of these objects changes in real-time when the FIX-based order/execution information is exchanged with trading counterparties, when market data or other data events are processed, or when modifications are initiated from the GUI or via API.

The backend provides API access to order, target, target list, position, and strategy data, to FIX Engine and messages that flow through it, to market data, to information about tradable instruments (security master), to custom data sources, and to data from user defined reports. One can access and use core data objects or subscribe to the changes. For example, one can get a callback every time the backend receives a fill for an open order, or every time a frontend user selects some data and clicks a pre-defined or custom action, or whenever market data events are received for certain instruments, or when a user changes the parameters of a strategy from the GUI. API can be used to create additional data sources. For example, one might want to add statistical analysis of historical market data such as trade impact prediction, average trade size, spread and volatility into the HiFREQ backend, so it can be used in strategies' trade decision making logic. One can also enrich existing HiFREQ data sources with additional data. For example, adding calculation of the fair market value to the market data source.

Developing an HFT strategy with HiFREQ API is a straight forward process that can be thought of as the following. The strategy subscribes to the data changes that it's interested in (e.g. every time market data changes or an execution is received or a scheduled timer elapses). The backend notifies of the changes that the strategy is subscribed to and the strategy code evaluates what changed (e.g. the price of a stock has moved). Strategy makes the backend perform an action based on the results of the event evaluation and the decision making logic (e.g. modify the limit price of an open order).

HFT strategies can be coded and deployed in-process or out-of-process in relation to the HiFREQ backend.

In-process strategies have an advantage of higher performance capabilities due to direct access to the core objects, due to being in-process (if desired) with market data feed handlers, FIX Engine, and other backend components, as well as due to being plugged into the core's internal threading model. In-process strategies have to be developed in Java and depend on HiFREQ core's process lifecycle. To code in-process strategies one should use Local client API.

Out-of-process strategies have full access to all backend data, are able to control the backend, are fully decoupled from the core, have independent process lifecycle, and can be geographically distributed if needed. Out-of-process API can also be used to control and monitor multiple HiFREQ deployments from a single client application. Out-of-process strategies can be developed in Java, C/C++/C#, Python, or any other programming environment that can work with C API library. To code out-of-process strategies one should use Remote client API.

At a glance here is a list of all included services and components:

- inbound and outbound order management
- target (parent order) and target list management
- low latency FIX connectivity (via in-process or out-of-process FIX Engine)
- position management
- risk controls
- market data feed handlers

- commission management
- security master
- open concurrency model
- purging of data from memory
- persistence of data
- recovery
- real time data reporting
- GUI for centralized monitoring and control

In-process API - Local client

In-process strategies use Local client API to perform trading and control HiFREQ core, to directly access the core's data objects, and to as well as to subscribe to order/target changes, market data, custom data, position data, etc.

Base Terminology and Naming Convention

Local Client

Facade over HiFreq framework. Provides access to the framework data and services, also notifies about framework events.

Local Client Listener

Optional notification API. If listener is specified, the local client will notify the application about events.

FIX engine connection

Physical connection to or from a counter party.
Only one message at a time can be sent. Concurrency can be achieved by adding connections to the counter party.
Same is true for receiving messages.

FIX message

The message which is sent through the FIX connection. Local Client provides API for sending and receiving FIX messages.

Transaction destination

Transaction destination is a logical wrapper for the connection.
Destination can specify different validation rules, defaulting rules, etc.
More than one destination is possible for the connection, i.e. n..1 relationship

Outbound destination

Transaction destination for sending trade requests out and processing incoming trade reports.

Inbound destination

Transaction destination for processing incoming trade requests and sending trade reports back.

Order manager

Order manager is a unit of the framework, residing on top of a transaction destination. Manager calculates and keeps order state.
Implements API for processing FIX messages received from the destination, or to be sent to the destination.
Manager works with a single transaction destination, more than one manager can be used for a destination. I.e. n..1 relationship

Outbound order manager

Order manager for sending trade requests out and processing incoming trade reports.

Inbound order manager

Order manager for processing incoming trade requests and sending trade reports back.

Order

Order represents data related to a trading transaction. Framework processes messages and updates the order state accordingly. It also calculates important fields.

Target

Represents a trading target for a given symbol. Each target is fulfilled by sending 1..N orders.
It is represented as a record with a set of fields. It is linked to its orders, and orders are linked to the target.

Target List

Represents a set of targets. It is used to unite targets into the logical units like baskets/portfolios, pairs, etc.

Strategy Data

Keeps data relevant to an application strategy. This data is displayed and is modifiable from GUI. Using relevant notifications the application strategy code can react to GUI user actions, like changing strategy parameters or starting/pausing a strategy.

Thread Model

The framework is multi-threaded

The following threads can be involved in concurrently working with data objects:

- Framework threads
  - FIX message receiver thread
  - Report state requester thread
  - Order purger thread
- Client threads
  - Market Data events thread
  - Timer events thread
  - Any other application thread

Data access and synchronization

Every thread accessing a data object must obtain a respective lock object for it.
Lock object is determined with one of the following methods of IHFLocalClient:

```
public interface IHFLocalClient
{
 ...
    public Object getLockObject(IHFLocalOrder order);
    public Object getLockObject(IHFLocalTarget target);
    public Object getLockObject(IHFLocalTargetList targetList);
    public Object getLockObject(IHFLocalStrategyData strategyData);
```

Example of data access with synchronization:

```
synchronized(localClient_.getLockObject(order))
{
    double orderQty = order.getOrderQuantity()
    ...
}
```

Notifications

See IHFLocalClientListener interface javadoc for all available notifications.

The sample below demonstrates usage of the notification about received trade report message.
Note that no lock is held by the framework when notifying, so the application must obtain a lock if accessing framework's data objects.

```
static final void addListener()
{
    localClient_.addListener(new MyLocalClientListener());
}

private static class MyLocalClientListener extends HFLocalClientListenerAdapter
{
    @Override
    public void onTradeReportReceived(IHFLocalOrder order, IELTTradeReportMessage message)
    {
        synchronized(localClient_.getLockObject(order))
        {
            if (message instanceof IELTExecutionReport)
            {
                IELTExecutionReport executionReport = (IELTExecutionReport)message;
                if ((executionReport.isOrderFillReport() &&
                     order.getState() == ITMSConstants.ORDSTATE_DONE))
                {
                    /* Code reacting to fully filled order event. */
                }
            }
        }
    }
}
```

Order Operations

Order request operations

An application works with order related API for sending/modifying/canceling orders.

Sending a new order

IELTOrderSingle object represents a "D" new order single FIX Message. A reusable object is obtained from the pool to minimize the performance impact. Then this object is populated with field values and sent over the FIX connection.
Returned IHFLocalOrder object is a result of sending and represents given order transaction. It can be used for further operations on that order, e.g. modifying or cancelling the order.

```
static final void newOrder()
{
   try
   {
      String managerName = "Simulator-HiFreq";
      IELTOrderSingle orderSingleRequest = localClient_.
            getOrderSingleRequestFromPool(managerName);
      populateNewOrderSingle(orderSingleRequest);
      IHFLocalOrder newOrder = localClient_.sendOrder(managerName, orderSingleRequest);
   }
   catch (TMSException e)
   {
      HFSystem.getLogFacility().error().write("Exception when sending order");
      HFSystem.getLogFacility().error().write(e);
   }
}

static final void populateNewOrderSingle(IELTOrderSingle orderSingleRequest)
{
   orderSingleRequest.setStringFieldValue(ITMSConstants.MSGFLDTAG_SYMBOL, "IBM");
   orderSingleRequest.setStringFieldValue(ITMSConstants.MSGFLDTAG_INSTRUMENT, "IBM");
   orderSingleRequest.setNumericFieldValue(ITMSConstants.MSGFLDTAG_ORD_TYPE,
                                 ITMSConstants.ORDTYPE_LIMIT);
   orderSingleRequest.setNumericFieldValue(ITMSConstants.MSGFLDTAG_PRICE, 70);
   orderSingleRequest.setNumericFieldValue(ITMSConstants.MSGFLDTAG_HANDL_INST,
         ITMSConstants.HANDLINST_AUTOMATED_EXECUTION_ORDER_PRIVATE_NO_BROKER_INTERVENTION);
   orderSingleRequest.setNumericFieldValue(ITMSConstants.MSGFLDTAG_SIDE, ITMSConstants.SIDE_BUY);
   orderSingleRequest.setNumericFieldValue(ITMSConstants.MSGFLDTAG_TIME_IN_FORCE,
                                 ITMSConstants.TIMEINFORCE_DAY);
   orderSingleRequest.setNumericFieldValue(ITMSConstants.MSGFLDTAG_ORDER_QTY, 100);
   orderSingleRequest.setStringFieldValue(ITMSConstants.MSGFLDTAG_CURRENCY, "USD");
   orderSingleRequest.setNumericFieldValue(ITMSConstants.MSGFLDTAG_TIMESTAMP,
                                 System.currentTimeMillis());
}
```

Modifying an order

There are two ways to modify an order. The simple one allows modification of the order price and quantity:

```
static final void modifyOrder1(IHFLocalOrder orderToModify)
{
   synchronized (localClient_.getLockObject(orderToModify))
   {
      try
      {
          localClient_.modifyOrder(orderToModify,
                           orderToModify.getPrice() + 10, /* Increasing price */
                           orderToModify.getOrderQuantity());
      }
      catch (TMSException e)
      {
         HFSystem.getLogFacility().error().write("Exception when modifying order " +
                                    orderToModify);
         HFSystem.getLogFacility().error().write(e);
      }
   }
}
```

Another way allows modification of other fields.
IELTOrderCancelReplaceRequest object represents a "G" order cancel/replace request FIX message. A reusable object is kept for each order. It's populated with fields to modify and sent over the FIX connection.

```
static final void modifyOrder2(IHFLocalOrder orderToModify)
{
   synchronized (localClient_.getLockObject(orderToModify))
   {
      try
      {
         IELTOrderCancelReplaceRequest orderCancelReplaceRequest = localClient_.
               getPreparedOrderCancelReplaceRequest(orderToModify);
         orderCancelReplaceRequest.setNumericFieldValue(ITMSConstants.MSGFLDTAG_PRICE,
                                             orderToModify.getPrice() + 10);
         orderCancelReplaceRequest.setStringFieldValue(ITMSConstants.MSGFLDTAG_TEXT, "Modify me");
         localClient_.modifyOrder(orderToModify, orderCancelReplaceRequest);
      }
      catch (TMSException e)
      {
         HFSystem.getLogFacility().error().write("Exception when modifying order " +
                                          orderToModify);
         HFSystem.getLogFacility().error().write(e);
      }
   }
}
```

Canceling an order

There are two ways of canceling an order. The simple one:

```
static final void cancelOrder1(IHFLocalOrder orderToCancel)
{
   synchronized (localClient_.getLockObject(orderToCancel))
   {
      try
      {
         localClient_.cancelOrder(orderToCancel);
      }
      catch (TMSException e)
      {
         HFSystem.getLogFacility().error().write("Exception when canceling order " +
                                          orderToCancel);
         HFSystem.getLogFacility().error().write(e);
      }
   }
}
```

Another way allows to specify additional fields in the cancel request FIX message:

```
static final void cancelOrder2(IHFLocalOrder orderToCancel)
{
   synchronized (localClient_.getLockObject(orderToCancel))
   {
      try
      {
         IELTOrderCancelRequest orderCancelRequest = localClient_.
               getPreparedOrderCancelRequest(orderToCancel);
         orderCancelRequest.setStringFieldValue(ITMSConstants.MSGFLDTAG_TEXT, "Cancel me");
         localClient_.cancelOrder(orderToCancel);
      }
      catch (TMSException e)
      {
         HFSystem.getLogFacility().error().write("Exception when canceling order " +
                                    orderToCancel);
         HFSystem.getLogFacility().error().write(e);
      }
   }
}
```

Canceling multiple orders

The method takes optional filter and fields container. Orders which satisfy the filter are canceled. Fields from field container are put into cancel request message.
If filter is null all orders are canceled.
If fields container is null no fields are copied.

```
static final void cancelOrders()
{
   try
   {
      /* Cancel Buy orders. */
      localClient_.cancelOrders("Side = 'Buy'", null);

      /* Cancel Sell orders, with a Text field in a cancel message. */
      FieldsContainerById extraFields = new FieldsContainerById();
      extraFields.add("Text", "Cancel Me");
      localClient_.cancelOrders("Side = 'Sell'", extraFields);
   }
   catch (TMSException e)
   {
      HFSystem.getLogFacility().error().write("Exception when canceling multiple orders");
      HFSystem.getLogFacility().error().write(e);
   }
}
```

Trade report operations

All report operations are related to the request. The request message is used to populate necessary fields in a report message.

Confirming a request

```
static final void confirmOrder(IHFLocalOrder order, IELTTradeRequestMessage requestMessage)
{
   synchronized (localClient_.getLockObject(order))
   {
      try
      {
         IELTExecutionReport confirmationReport = localClient_.getPreparedReportMessage(order);
         localClient_.populateConfirmMessageForRequest(confirmationReport, requestMessage,
                                                       order, false);
         localClient_.confirmOrder(order, confirmationReport);
      }
      catch (TMSException e)
      {
         HFSystem.getLogFacility().error().write("Exception when sending execution report.");
         HFSystem.getLogFacility().error().write(e);
      }
   }
}
```

Rejecting an order request

```
static final void rejectOrder(IHFLocalOrder order, IELTTradeRequestMessage requestMessage)
{
   synchronized (localClient_.getLockObject(order))
   {
      try
      {
         IELTExecutionReport orderRejectReport = localClient_.getPreparedReportMessage(order);
         localClient_.populateRejectMessageForRequest(orderRejectReport, requestMessage,
                                                      order, "Too late");
         localClient_.rejectOrder(order, orderRejectReport);
      }
      catch (TMSException e)
      {
         HFSystem.getLogFacility().error().write("Exception when sending execution report.");
         HFSystem.getLogFacility().error().write(e);
      }
   }
}
```

Rejecting an order cancel request or an order cancel/replace request

```
static final void rejectOrderCancel(IHFLocalOrder order, IELTTradeRequestMessage requestMessage)
{
   synchronized (localClient_.getLockObject(order))
   {
      try
      {
         IELTOrderCancelReject orderCancelRejectMessage = localClient_.
               getPreparedOrderCancelRejectMessage(order);
         localClient_.populateOrderCancelReplaceRejectMessageForRequest(orderCancelRejectMessage,
                                                      requestMessage, order,
                                                      "Too Late to cancel");
         localClient_.rejectOrderCancel(order, orderCancelRejectMessage);
      }
      catch (TMSException e)
      {
         HFSystem.getLogFacility().error().write("Exception when sending order cancel reject.");
         HFSystem.getLogFacility().error().write(e);
      }
   }
}
```

Replacing an order

```
static final void replaceOrder(IHFLocalOrder order, IELTTradeRequestMessage replaceRequestMessage)
{
   synchronized (localClient_.getLockObject(order))
   {
      try
      {
         IELTExecutionReport orderReplaceReport = localClient_.getPreparedReportMessage(order);
         localClient_.populateReplaceForRequest(orderReplaceReport, replaceRequestMessage, order);
         localClient_.replaceOrder(order, orderReplaceReport);
      }
      catch (TMSException e)
      {
         HFSystem.getLogFacility().error().write("Exception when sending execution report.");
         HFSystem.getLogFacility().error().write(e);
      }
   }
}
```

Filling an order

```
static final void fillOrder(IHFLocalOrder order, IELTTradeRequestMessage requestMessage)
{
   synchronized (localClient_.getLockObject(order))
   {
      try
      {
         IELTExecutionReport executionReport = localClient_.getPreparedReportMessage(order);
         double fillQty = order.getOrderQuantity();
         double fillPrice = order.getPrice();
         localClient_.populateFillMessageForRequest(executionReport, requestMessage,
                                          order, fillQty, fillPrice);
         localClient_.fillOrder(order, executionReport);
      }
      catch (TMSException e)
      {
         HFSystem.getLogFacility().error().write("Exception when sending execution report.");
         HFSystem.getLogFacility().error().write(e);
      }
   }
}
```

Outing an order

```
static final void outOnOrder(IHFLocalOrder order, IELTTradeRequestMessage requestMessage)
{
   synchronized (localClient_.getLockObject(order))
   {
      try
      {
         IELTExecutionReport executionReport = localClient_.getPreparedReportMessage(order);
         localClient_.populateCancelForRequest(executionReport, requestMessage, order);
         localClient_.outOnOrder(order, executionReport);
      }
      catch (TMSException e)
      {
         HFSystem.getLogFacility().error().write("Exception when sending execution report.");
         HFSystem.getLogFacility().error().write(e);
      }
   }
}
```

Target Operations

Adding targets

Targets belong to a target list. The following sample adds a target list first if it does not yet exist and then adds a target to it.

```
static final void addTarget()
{
   try
   {
      String targetListName = "My Portfolio";
      IHFLocalTargetList targetList = localClient_.getTargetList(targetListName);
      if (targetList == null) /* if initial running, no portfolios & targets yet recovered */
      {
         targetList = localClient_.addTargetList(targetListName, null);
      }

      HFLocalTargetFieldsContainerById targetFieldsContainer =
            new HFLocalTargetFieldsContainerById();
      targetFieldsContainer.addSymbol("IBM");
      targetFieldsContainer.addInstrument("IBM");
      targetFieldsContainer.addSecurityType(ITMSConstants.SECURITYTYPE_COMMON_STOCK);
      targetFieldsContainer.addTargetQuantity(1000);
      targetFieldsContainer.addWaveSize(200);
      targetFieldsContainer.addSide(ITMSConstants.SIDE_BUY);
      targetFieldsContainer.addType(ITMSConstants.ORDTYPE_LIMIT);
      targetFieldsContainer.addPrice(11);
      IHFLocalTarget target = localClient_.addTarget(targetList, targetFieldsContainer);
   }
   catch (TMSException e)
   {
      HFSystem.getLogFacility().error().write("Exception when adding a target.");
      HFSystem.getLogFacility().error().write(e);
   }
}
```

Sending target orders

When sending an order of a target, sendTargetOrder(...) method should be used instead of sendOrder(...), so that the framework properly links orders with the parent target. Also the optional utility method populateOrderSingleRequestFromTarget(...) is used - it populates core important fields in a request message from the target.

```
static final void newTargetOrder(IHFLocalTarget target)
{
   try
   {
      synchronized (localClient_.getLockObject(target))
      {
         final String managerName = "Simulator-RemoteHiFreq";
         IELTOrderSingle newOrderMessage = localClient_.
               getOrderSingleRequestFromPool(managerName);
         localClient_.populateOrderSingleRequestFromTarget(target, newOrderMessage);
         localClient_.sendTargetOrder(managerName, target, newOrderMessage);
      }
   }
   catch (TMSException e)
   {
      HFSystem.getLogFacility().error().write("Exception when sending an order of a target " +
                                    target);
      HFSystem.getLogFacility().error().write(e);
   }
}
```

Order Purging

Purging of orders is a service responsible for cleaning closed orders out of memory.

Local client configuration provides purging parameters.
XML configuration example (all time parameters are in seconds):

```
<OrderPurger
 canceledOrderAgeBeforePurging="10"
 canceledOrderPurgingCheckInterval="30"
 rejectedOrderAgeBeforePurging="10"
 rejectedOrderPurgingCheckInterval="30"
 closedOrderAgeBeforePurging="10"
 closedOrderPurgingCheckInterval="30"
 maxPurgingCountInOneOperation="100000"
/>
```

Purging of orders happens on a separate thread.
See javadoc for the related API:

```
boolean isOrderPurged = localClient_.isOrderPurged(orderId);
IHFLocalOrder nonPurgedOrder = localClient_.getNonPurgedOrder(orderId);
IHFLocalOrder unpurgedOrder = localClient_.unpurgeOrder(orderId);

boolean canBePurged = order.canBePurged();
order.setCanBePurged(false);
```

Recovery

All target, order, and strategy data is recovered when localClient.initialize() is invoked.

A listener is notified about the recovery process:

```
public interface IHFLocalClientListener
{
...
    void onTargetListRecovered(IHFLocalTargetList recoveredTargetList);
    void onTargetRecovered(IHFLocalTarget recoveredTarget);
    void onOrderRecovered(IHFLocalOrder recoveredOrder, boolean goodForPurging);
    void onStrategyDataRecovered(IHFLocalStrategyData recoveredStrategyData);
    void onTargetsRecoveryFinished();
...
}
```

Replaying unprocessed messages

If the application uses individual incoming messages somehow, then a dedicated API can be used to retrieve unprocessed recovered messages.
See javadoc for this method:

```
public interface IHFLocalClient
{
...
    void setLastProcessedMessageId(@Nonnull String managerName, int lastProcessedMessageId);
...
}
```

If last processed message id is set for the manager, then messages from that manager with the id next to the specified one are replayed through li
stener.onUnprocessedMessageReplayed(order, message) API.

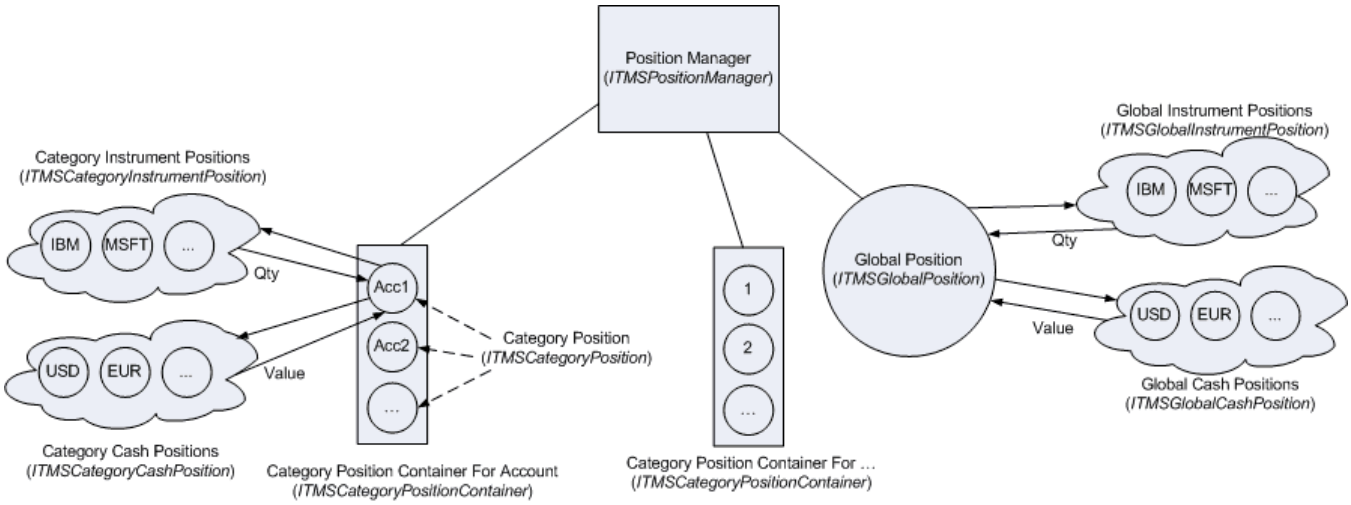
Positions Management

General Positions Management

Positions overview

General positions overview

Here is a basic positions structure and relationship diagram:



Key points:

- Positions types are: global, instrument, cash and category.
- Examples of category position type are: Account, Strategy, etc.
- Qtys are propagated from instrument positions to global position.
- Values are propagated from cash positions to global position.

Positions terms

- Category type + name pair - some criterion to classify positions category. Typically category type is name of some field in order messages or order records, and category name is value of given field. However categories might be compound and include few fields, in this case category position will be maintained for each unique combination of these fields. Category type cannot be free-form but should be one of registered types.
  Example of category type + name pair: "`Account`" (category type) + some account name (category name).
- Normalization currency - is a currency in which all category and global position values are expressed. Instrument and cash position values might be in different currencies, they are normalized to given currency and accounted in category/global position totals.
- Position info types:
  - Loaded - these values aren't accumulated in manager, new values will override previously loaded values;
  - Traded - accumulated values traded intraday;
  - Open - open values that aren't traded yet, i.e. expected delta for current position values.

    > ⚠ Positions manager doesn't use open positions at all:
    >
    > - Open positions are not used for calculating position totals (see below).
    > - Hence open positions are not exported.
    > - I.e. they are only reflected in position data structures.
    > - It's up to application only whether to maintain open positions or not.

- For each type of maintained values (loaded/traded/open) position manager maintains:
  - Long quantity/value - increase when something is bought (i.e. obtained or expected to be obtained);
  - Short quantity/value - increase when something is sold (i.e. removed or expected to be removed);
  - For each type (loaded/traded/open) always: `Total = Long - Short`;
  - For cash positions:
    - long value increases when getting some cash (as result of some FX trade or when selling some non-FX instrument);
    - short value increases when spending money (as result of some FX trade or when buying some non-FX instrument).
  - For category position (including global):
    - long/short values only reflect instrument position values;

        - i.e. these fields are regular sums of corresponding fields for all underlying instrument positions;
      - fields `LoadedCashVal`, `TradedCashVal` and `OpenCashVal` represent money position for each type loaded/open/traded;
        - i.e. these fields include FX and regular trades, and also loaded cash position values;
      - this way category/global position has pure instrument position related information (how many shares are sold/bought and how much money spent/obtained from this), and also total net cash values.
    - Both long and short quantities/values are non-negative numbers.
- Position totals are calculated for API user convenience.
  Loaded and traded values are summed up in totals per position. E.g. for category/instrument/cash positions:
    - `Qty = LongQty - ShortQty`
      - `LongQty = LoadedLongQty + TradedLongQty`
      - `ShortQty = LoadedShortQty + TradedShortQty`
    - `Value = LongValue - ShortValue`
      - `LongValue = LoadedLongValue + TradedLongValue`
      - `ShortValue = LoadedShortValue + TradedShortValue`
    - There are no reporting fields to keep these values, but there are API methods to access these values. I.e. position itself in this case takes care of calculating totals.

Custom category types and names

Application updating some position values in given manager should provide implementation of `ITMSPositionCategoryNameProvider` if at least one category type is registered in manager.

- Position manager will figure out category name (to update proper category position) for each registered category type.
- Typically category type is just name of some field in order records, in messages, etc. So typically implementation of such provider would just take value of given field from record/message/etc;
- It is very desirable that numeric fields are not used as category types since it would require conversion of numeric values to strings (since category name must be expressed as string);

Normalization currency

Category position total values (expressed in normalization currency) are calculated based on cash position values only (see diagram above).

- I.e. instrument position values are expressed in primary currency of given instrument. They are not normalized.
- Cash position values are in given currency, they are normalized and accounted in category position totals.
  - i.e. this way category position total quantities are calculated based on underlying instrument positions, and total values - based on normalized cash position values(see diagram above).
- Normalization currency is required for each position manager.
  - Even if all values will be expressed in the same currency, e.g. USD, then USD should be specified as normalization currency.
  - In this case normalization will not be really performed while currency for all positions is USD. Forex rates data feed will not be used.

Positions export/import

Example of positions export output

```
Instrument Currency CategoryType CategoryName LongQty LongValue ShortQty ShortValue CustomNum1 CustomNum2
<cash> USD Account test1  4000  8000 1 2
IBMEU EUR Account test2 1000 50000 500 100000 3 4
MSFT USD Account test2 200 4000 400 8000 5 6
<cash> EUR Account test2  50000  100000 7 8
IBM USD Account test3 0 0 0 0 9 10
<cash> USD Account test3  10000  0 11 12
<cash> JPY Account test3  0  100000 13 14
<cash> EUR Account test3  0  0 15 16
```

Exporting/importing of instrument and cash positions is supported:

- Export/import conditions.
  - Import happens either on start up (in case `externalDataResource` is set in server position manager XML configuration) or on API call.
  - Export only happens on API call.
- Instrument and cash positions are together exported to and imported from the same resource.
- For cash positions `Instrument` column is populated with "`<cash>`" and `Currency` column is populated with cash position currency.
- Notes on columns:
  - `Instrument, Currency, LongQty, LongVal, ShortQty, ShortVal` are required columns in external position data resource.

    - More columns with data that helps to identify instruments can be present (i.e. custom fields, alternate instrument ID columns, etc).
        - Obviously for cash positions these columns will stay empty.
    - Obviously for cash positions `*Qty` columns will stay empty.
    - Extra columns with externally generated data to be stored in positions can be present.
- Data columns are delimited by TABs.
- Global or category positions export is exclusive:
    - If only global instrument/cash positions are maintained - then global positions are exported/imported (`CategoryType` is either empty or `CategoryType="Global"` and `CategoryName` is empty).
    - Otherwise, i.e. if there is at least one category type registered - then only category positions are be exported/imported.
- Positions export result contains only totals of loaded and traded values: `LongQty, LongVal, ShortQty, ShortVal`.

Open positions
Open positions are not used for calculating these totals, i.e. ignored during export.

- Formatted numbers are also supported in imported data (since TMS_7.1.2009.02.09.r14013):
    - current locale's number format is assumed;
    - numbers in parentheses are considered negative;
    - Excel's export number format (`" 5,844.72 "`) is understood (since TMS_7.1.2009.02.09.r14015).
- Import granularity is "per position":
    - importing position data does not mean currently accumulated list of positions should be cleared and then replaced with new data;
    - in case data being imported has position already present in system, existing position is replaced with imported position.

Position manager distribution (optional)

- Overview
  Position management can be distributed in case it needs to be shared between processes and be centralized.
  A client position manager serves as a proxy representative of a server manager, which resides in the remote process.
  A server position manager is the actual position manager.
- Technical details
    - Position system report maintenance is only required if position manager is distributed.
    - Updating of positions is asynchronous. This includes programmatic loading of start-of-day positions as well.
        - Importing of start-of-day position happens in client position manager. I.e. file is loaded on client side and then loaded values are passed to server.
        - Exporting always happens on the backend. If invoked on client site, then exported data is taken from server and then can be saved to local resource.
    - Position structures on client side are replicated based on position system reports:
        - Client manager is always subscribed for category position system report. Hence per category position totals are replicated right away.
        - Underlying instrument and cash positions contributing to each category position are downloaded and maintained on demand, i.e. on first usage.
        - Client manager may optionally maintain only cash, only instrument or both positions for specified category types;
        - Client manager may pre-subscribed for instrument and cash positions for some category type (including Global) if needed.
    - Once category position container (for given category type) or category position (for given type+name), proxy will subscribe for needed data and will keep maintaining these positions.
    - I.e. application may get some [TS:category/instrument/cash] position and keep the reference, and this instance is kept updated all the time with latest received info.
- Local positions
    - With distributed position manager multiple clients may contribute to shared data on server, and client is optionally being updated asynchronously with server-side data
    - Even if there is only one client process updating given category (entire category or only some category positions within given category) - normally data is still flowing thru the server
    - This creates certain delay in how soon up-to-date totals are becoming available in the client, which might be a concern in certain very sensitive cases when trading limits, for instance, are very strict
    - In cases like this some position categories can be marked as local, and it means that client position manager will not be getting totals for given category from the server, but only use own totals
    - Updates to local categories are still aggregated on server side, so server would have all totals, the only difference is that category totals on client side are not picking up other changes, only updates from the same process.
    - This way when category is declared as local - position totals are immediately available in the local client process.
        - For instance multiple HiFreq order managers might be updating same local category position, and totals across all managers in this process would be immediately available for trading limit checks, for example.
- Caveats
    - Data is not available on client, though updates from client are processed on server.
        - Explanation:
          Client position manager uses reporting and hence event service for data replication. Hence there should be an even service connection (configured in `EventService.xml`) between processes hosting position manager server and clients. Thus if both client and server managers are hosted in separate processes of the the type, data will not be available on client, though updates from client will be processed on server.
        - Workarounds:
            - Host server manager in some separate process, and in both custom apps have client managers.

        - Route events through client hub, but that's not desirable due to performance considerations.
    - If local position categories are in use - LoadedXXX field values for such categories are also local.
        - Meaning that positions cannot be loaded in one central location (on server) and then just distributed across all clients.
        - Instead each process should load own portion of start-of-day positions so it's properly reflected in local data structures and also reported to server.

Positions recovery

Position manager does not facilitate its own recovery.
Technical details:

- Position manager does not provide any persistence for provided data.
    - Each application component that contributes some position updates must have recovery and provide same position values again on startup.
    - Though distributed position manager has some internal recovery to restore position data if some position manager client or server is restarted.
    - If position manager server process is restarted, it will gather current state from all connected position manager clients.

Other notes on position management

- Global position and instrument positions for global and each category position are always maintained.
- Cash positions are optional now. I.e. can be turned off. In this case only trades in currency same as normalizing one can be processed/accounted in position manager.
- Same interface `ITMSPositionManager` represents both server and client side of position manager, so usage by application should be absolutely transparent.
- `ITMSPositionDataAccessor` represents read-only part of position manager, i.e. it has only data accessing methods and doesn't allow to update data in the manager.

Positions configuration

Positions XML configuration

The easiest way to configure a position manager is through an XML resource.
Folder `backend\Directory\ElTrader\TMS\Config\Positions` contains samples of xml configuration.

- Server position manager XML configuration
  `HiFreqPositionManager.xml` contains sample of a server position manager configuration:

```
<!--
Server position manager configuration.

<PositionManager> attributes:
 - name - the name of the server position manager.
  Values: free-form case-sensitive string.
  Validation: not validated.
  Required: yes.

 - mode - must have "server" value

       - distributed - indicates whether position manager should be distributed.
  Values: true/false.
  Required: no.
  Default Value: true.

       - normalizingCurrency - indicates the normalization currency. Values will be normalized to specified currency.
  Values: USD/EUR/GBP/...
  Required: yes.

 - expectingOpenPositionUpdates - indicates whether position manager is expecting to receive position manage updates.
   I.e. this determines if open positions will be updated by users of this manager.
   Allows to control if open positions should be tracked for both long and short, or separately.
  Values: true/false/longOnly/shortOnly
  Required: no.
  Default Value: true.

 - maintainingCashPositions - indicates whether position manager will maintain cash positions.
   If cash positions aren't maintained, then it will allow only positions for currency same as normalizing one.
```

Values: true/false.
Required: no.
Default Value: true.

- maintainingIntradayTradedPositions - indicates whether position manager will expect traded positions to be intraday.
Not supported in HiFreq yet, i.e. traded positions are all positions currently present in HiFreq.
In portfolio domain if traded positions should be intraday, then orders system report should have intraday related fields.
Values: true/false.
Required: no.
Default Value: false.

- maintainingCategoryTotalsForFXInstrumentPositions - indicates whether position manager
will maintain include (roll up) FX instrument positions in category position totals.
Values: true/false.
Required: no.
Default Value: false.

- reportManagerName - report manager name where position reports will be kept.
Required: yes if distributed = "true"

- forexRatesDataFeedName - name of FX rates data feed.
Required: no.
Default Value: "Forex"

- metaDataName - name of fields metadata to be used for system reports.
Required: no.
Default Value: "Position table report fields"

- externalDataResource - name of resource with start-of-day positions. Will try to load this file on startup.
Required: no.

- externalDataAlternateInstrIdSources - comma-delimited list of alternate instrument IDs to be exported (or accounted when import)
in position data resource.
Values: RIC, CUSIP, ISIN, etc - will make data columns RIC.ID, CUSIP.ID,ISIN.ID, etc be exported/imported.
Required: no.

- externalDataExtraFieldsToResolveInstrId - comma-delimited list of custom fields that can be used during import to identify security/instrument. E.g. can be Symbol, SymbolSfx, SecurityExchange, Country, etc.
Required: no.

- externalDataCustomFields - comma-delimited list of custom fields to be imported/exported for instrument and cash positions
to/from position data resource.
Required: no.

- categoryPositionsReportName - name of category position system report.
Required: no.
Default Value: generated as "managerName - Category Position Report"

- instrumentPositionsReportName - name of instrument position system report. If not specified, then generated as "managerName - Instrument Position Report".
Required: no.
Default Value: generated as "managerName - Instrument Position Report"

- cashPositionsReportName - name of cash position system report.
Required: no.
Default Value: generated as "managerName - Cash Position Report"

<Categories> - list of category types to be maintained by manager. By default only Global positions maintained.

<Category> attributes:
- type - name of the type of position category.
E.g. "Account", "Portfolio", ...
- adjustCategoryPositionByCoef - rather exotic parameter that allows to adjust/scale instrument and category position quantities by certain coefficient that should be specified as custom instrument field (PositionCoef.CUSTOM) in security master.

```
  Values: true/false.
  Required: no.
  Default Value: false.
 - local - indicates if given category should be locally updated. Updates to this category
  are still reported to server, however totals are not taken from server, which means
  that such category doesn't have updating delay, totals in given category
  are immediately available in updating process.
  Values: true/false.
  Required: no.
  Default Value: false.
 - expectingOpenPositionUpdates - same optional true/false/longOnly/shortOnly parameter as for
  entire position manager, but allows to control to have open positions maintained
  only for some categories, but not others.
 - borrowTrackingCategory - allows to indicate that special handling of updates, that is
  suitable for "borrow tracking" should be enabled for this whole category,
  or only some specific position categories for this type.
  "borrow tracking" position means - traded long (buy cover portion) qty cannot exceed
  traded short qty at any given moment, i.e. cannot cover more than shorted.
  Values: true/false/<regexp>.
  Required: no.
  Default Value: false.

 <ReportBuffer> - spec of report buffering element to be used with position system reports.
  Default is 3 sec/500 events. Buffer is fired after timeout or if event count is reached.

 <ReportElement> attributes
       - type - must have "Buffered Records" value.
       - timeout - buffering timeout in milliseconds.
       - eventCount - event count -->
<PositionManager
 name="HiFreq Positions"
 mode="server"
 distributed="true"
 normalizingCurrency="USD"
 forexRatesDataFeedName="Forex"
 metaDataName="Position table report fields"
 reportManagerName="HiFreq report manager"
 expectingOpenPositionUpdates="true"
 maintainingCashPositions="true"
>
 <Categories>
  <Category type="Strategy"/>
 </Categories>
 <ReportBuffer>
  <ReportElement type="Buffered Records" timeout="100" eventCount="25" />
```

```
 </ReportBuffer>
</PositionManager>
```

- Client position manager XML configuration
  `HiFreqClientPositionManager.xml` contains sample of a client position manager configuration:

```
<!--
Client position manager configuration.

<PositionManager> attributes:
 - name - the name of the client position manager. Must be the same as a server position manager name.
  Values: free-form case-sensitive string.
  Validation: not validated.
  Required: yes.

 - mode - must be "client".

 - maintainInstrumentPositionsForCategoryTypes - comma-delimited list of category types (including 'Global' if needed)
   to maintain instrument positions for. Or * to maintain them for all types.
        Required: no.
        Default Value: "Global"

 - maintainCashPositionsForCategoryTypes - comma-delimited list of category types (including 'Global' if needed)
   to maintain cash positions for. Or * to maintain them for all types.
        Required: no.
        Default Value: "Global"

 - presubscribedCategoryTypes - comma-delimited list of category types (including 'Global' if needed) to pre-subscribe
   and maintain instrument/cash positions for. Or * to pre-subscribe for all types.
   By default doesn't pre-subscribe.
   Makes sense only if instrument/cash positions should be maintained at least for some category types or "Global".
       Required: no.
       Default Value: empty, so it does not presubscribe.

    - resetPositionsOnReconnect - true/false attribute indicating whether cached position data
                should be reset/removed when connection to server is lost and client position
                manager is restoring connection.
        Required: no.
        Default Value: "false"

 - waitForInitialPositionDataTimeoutInSec - timeout in seconds indicating for how long client
   position manager is allowed to wait for initial data to come and the time of
   initialization (when subscribing for category positions and optionally for
   instrument/cash positions for specified "presubscribe" category types).
   0 or negative values mean that position manager client doesn't wait for data
   to come, so it will be coming in asynchronously some time later after initialization.
        Required: no.
        Default Value: "10"-->
<PositionManager name="HiFreq Positions" mode="client" />
```

Access to positions data via API

Most important API methods

- `ITMSPositionDataAccessor` (note `ITMSPositionManager` extends `ITMSPositionDataAccessor`):
  - `ITMSGlobalPosition getGlobalPosition()` - returns container of global position info.
  - `ITMSCategoryPosition getCategoryPosition(String categoryType, String categoryName)` - returns category position for the specified category type and name.
  - `ITMSCategoryPositionsContainer getCategoryPositionsContainer(String categoryType)` - category positions container for specified category type.
  - `ITMSPositionManagerConfigProvider getConfigProvider()` - returns container with all configuration data for this position manager (e.g. available category types, whether open positions be updated etc).

- `ITMSCategoryPositionsContainer`
  - `ITMSCategoryPosition getCategoryPosition(String categoryName)` - returns category position for the specified category name.
  - `ITMSCategoryPosition ensureCategoryPositionExists(String categoryName)` - returns category position for the specified category name.
- `ITMSCategoryPosition`
  - `ITMSCategoryCashPosition getCashPosition(String currency)` - returns cash position in this category for specified currency.
  - `ITMSCategoryInstrumentPosition getInstrumentPosition(String instrumentId)` - returns instrument position in this category for specified instrument.
- `ITMSGlobalPosition`
  - `ITMSGlobalInstrumentPosition getInstrumentPosition(String instrumentId)` - returns global instrument position for specified instrument.
  - `ITMSGlobalCashPosition getCashPosition(String currency)` - returns global cash position for specified currency.
- `ITMSPositionManager`
  - `void importPositions(String resourceName)` - allows to import start-of-day positions from specified external resource.

Position synchronization notice

- Position manager implementation is thread safe.
- The only case when external synchronization might be required is synchronization by position object (category/instrument/cash position) when calling multiple `getXXX()` methods (to avoid data changes between `getXXX()` calls).
- For position listeners the updated position is locked while `on*()` calls.
- Access to another positions should not be done in this call because of that lock. It will help to prevent deadlock with another thread, accessing positions in a different order.
  - If access to other positions is needed in this `on*()` method, thread boundary should be introduced and another thread should access positions.

Positions GUI

Position reports specified in XML or by config provider are available from GUI.

HiFreq Positions Management

Configuration

Local client XML configuration lets specify 0..N position managers.
E.g.

```
<PositionManagers>
 <PositionManager resource="/ElTrader/TMS/Config/Positions/HiFreqPositionManager.xml"/>
</PositionManagers>
```

Then configuration of the participating order manager can refer to the position manager by name.
In this case orders of a manager will contribute to the positions of the named position manager.

```
<LocalClient>
      ...
 <Outbound>
  <OrderManager
   name="Timer Orders"
                ...
   positionManager="HiFreq Positions"
  />
 </Outbound>
```

Recovery

When orders are recovered they update related position managers. So positions are recovered based on orders.

Usage of positions through API

Obtain access to the position manager and then use one of its methods:

```
private void usePositions()
{
   ITMSPositionManager positionManager = localClient_.getPositionManager();
   try
   {
      ITMSGlobalInstrumentPosition ibmPosition = positionManager.getGlobalPosition().getInstrumentPosition("IBM");
      HFSystem.getLogFacility().debug().write("IBM global position = " + ibmPosition);
   }
   catch (TMSPositionException e)
   {
      e.printStackTrace();
   }
}
```

Commissions Management

Features

- Commission categories - represent homogeneous sets of entities (orders/targets/messages/etc) for which commissions/rebates are maintained.
    - Entities are categorized based on configured criteria (values of key fields - see below).
    - Same commission/rebate calculating rules apply for items of the same category (though there can be variations based on traded volume - see below).
    - Category is the only unit of commission/rebate info management, i.e. commission/rebate info from all entities is summarized into (and maintained only for) categories.
- Commission sub-categories - represent rules for refined (within category) fills commissions/rebates calculation based on configured criteria (values of sub-key fields - see below).
    - Only used to refine commissions/rebates for individual fills.
    - The idea is that each fill might be processed differently, i.e. provide liquidity or take it, be executed on different exchanges, etc. So different commissions are paid or different rebates are get for different fills.
    - Not present in reports or exported data as individual entities, but only affect numbers calculated for categories.
- Rebates are supported.
    - Total rebates are tracked per category.
- Total/intraday commission values and traded volumes per category are maintained.
    - Import/export is supported (invoked by console commands - see below).
- Normalization
    - For every given commission category all values are expressed in category's currency. I.e. all commission values, and also if "value" units are utilized for given category - then all traded volume values and commitment are also expressed in this currency.
    - However many different targets/orders may contribute to given commission category, potentially they could be for another currency.
    - All target/order values are normalized prior to contributing to commission category.
- Recovery
    - Commissions data recovery is facilitated by HiFreq order managers.

Configuration and deployment

- HiFreq commissions management is configured in `HFSystem` process section in processes.xml, e.g.:

directory://ElTrader/TMS/Config/processes.xml

```
<HFSystem logFacilityName="ELTHiFreqAppLogFacility">
 <Commissions
  maintainCommissions="true"
  keyFieldNames="SecurityType,TrnDestination"
  subKeyFieldNames="LastMkt,ExecScenario"
  categoryInfoResource="/ElTrader/TMS/Config/Commissions/MarketCommissions.txt"
  startOfDayInfoResource="/ElTrader/TMS/Config/Commissions/StartOfDayCurrentBillingCycleMarketCommissionsInfo.txt"
 >
  <ReportBuffer>
   <ReportElement
    eventCount="-1"
    timeout="2000"
    type="Buffered Records"
   />
  </ReportBuffer>
 </Commissions>
</HFSystem>
```

- "keyFieldNames" - comma-delimited list of key fields determining commission categories, required.
  - Unique key value combination defines an independent commission category.
  - These are fields from:
    - order messages;
    - order nodes (should be present in "HiFreq Order Transactions" system report, directory://ElTrader/TMS/Config/SystemReports/HifreqOrderReport.xml).
  - Only fields from "Portfolio commission table report fields" metadata are allowed (directory://ElTrader/TMS/Config/ElementSettings/TableElement/PortfolioCommissionTableFieldsMetaData.xml).
- "subKeyFieldNames" - comma-delimited list of key fields determining commission subcategories, optional.
  - These are fields from:
    - execution report messages. So typically execution reports would have to be processed using defaulting rules in order to properly populate these sub-key fields.
      - These two resources can be used to apply defaulting rules to execution reports:
        - directory://ElTrader/TMS/Config/Defaulters/ReceivedExecutionReportDefaulter.xml - applied to all execution reports.
        - directory://ElTrader/TMS/Config/TrnDestinations/CustomDefaults/TrnDestinationCustomDefaultsForReceived_<TrnDestName>.xml - applied to execution reports received for order previously sent to `TrnDestName` destination.
    - order nodes (in case relevant field is empty in execution report).
- "categoryInfoResource" - name of the resource with commission categories info, required;
- "maintainCommissions" - `true/false` attribute allowing to turn on/off commissions management, optional. By default commission maintenance is activated if `<Commissions>` section is present;
- "startOfDayInfoResource" - name of the resource with start-of-day ingo, optional. If not specified all start-of-day values assumed 0;
- <ReportBuffer> element is optional, default buffer is <ReportElement eventCount="500" timeout="3000" type="Buffered Records" />. "Market Commission System Report" will be created with specified buffer element;

- Basic commission configuration resources (category and start-of-day infos are located at directory://ElTrader/TMS/Config/Commissions);
- Format of commission category info resources (MarketCommissions.txt):
  - tab-delimited;
  - columns order is not arbitrary and should be exactly the same as shown below:

	Required? (for categories)	Required? (for sub-categories)	Description / Notes
KeyField $_1$	➖	➖	
KeyField $_2$	➖	➖	
...	➖	➖	
KeyField $_n$	➖	➖	

SubKeyField $_1$	not allowed	➖	- A row is only considered to be a category if it has no sub-key field value. - A row is only considered to be a subcategory if it has at least one value specified for sub-key.
...	not allowed	➖	(see above)
SubKeyField $_m$	not allowed	➖	(see above)
TradedVolumeBrackets	➖	➖	|-delimited list of traded volume ranges when commission rate changes for given category.
CommissionType	➕/➖ (required if traded volume brackets are configured)	➕	Valid values: ℹ CM is contract multiplier for given instrument, FillQty is in contracts - per share - CommVal=FillQty * CM * Comm - bps - CommVal=FillVal * Comm / 10000 - pct - CommVal=FillVal * Comm - absolute - CommVal=Comm, this is absolute commission for whole order/target, i.e. regardless of FillQty/Val - if there are no fills for order/target - CommVal=0, if FillQty > 0 - CommVal=Comm. - per contract - CommVal=FillQty * Comm (FillQty is in contracts) - per million - CommVal=FillVal * Comm / 1000000. Typically if CommissionType is per share, TradedVolumeUnits would be shares; per contract - contracts, per million - value... Though it supports any combinations.
Commissions	➖	➕	|-delimited list of commission/rebate rates corresponding to configured traded volume brackets. There always should be `N+1` rates configured where `N` is number of specified range traded volumes. E.g. if traded volume brackets is "500000,1000000" - then there should be 3 rates for ranges 0-500000, 500000-1000000 and 1000000-infinity. If no brackets configured - single commission/rebate rate is optional (meaning there is no default rate or default rate never changes based on traded volume). In order to configure rebate rather than commission have negative number in this column.
MinCommission	➖	not allowed	Define valid range for commission rate configured in Commissions field. This range is needed for preventing mistakes when users are adjusting commissions in live system. E.g. prevent from changing `.02` accidentally to `.3` instead of `.03`.
MaxCommission	(see above)	(see above)	(see above)
Commitment	➖	not allowed	Commitment for given category expressed in shares, contracts or value in configured currency.
Currency	➕	not allowed	It's mostly relevant if CommissionType is per share, per contract, per million, or absolute. Also, if TradedVolumeUnits is "value", then "Currency" is relevant for TradedVolumeBrackets. Though regardless of CommissionType/TradedVolumeUnits it's needed for maintaining of CommissionVal per category, hence required.
TradedVolumeUnits	➕	not allowed	For maintaining of total TradedVolume per category. Valid values: - shares - traded volume brackets and commitment are configured in shared, total traded volume for category should be expressed in shares; - contracts - very similar to shares (actually for equities it's same) but in contracts; - value - all traded volumes and commitments are expressed in specified currency values.
IsRequired	➖	not allowed	Not applicable for HiFreq.

- e.g.:

SecurityType	TrnDestination	CommissionType	TradedVolumeBrackets	Commissions	MinCommission	MaxComm

CS	LEHMAN	per share	5000000|10000000	0.05|0.04|0.02	0.02	0.07
CS	MS	per share		0.03	0.02	0.04
FOR	DB-FX	per million		10	8	15

- All key fields and all their combinations may have empty (meaning <any>), values.
- Once subcategories are used:
  - subcategories cannot be independent, but always relate to some category (having the same set of key field values);
    - key field values set should be exactly the same for category and it's subcategories, i.e. for subcategories empty key field names do not mean "match any value" (making this subcategory match several categories), but "match empty value";
  - subcategories should be defined in the same file below parent categories.
- Format of start-of-day resources - StartOfDayCurrentBillingCycleMarketCommissionsInfo.txt:

KeyField $_1$	KeyField $_2$	...	KeyField $_n$	TradedVolume	CommissionVal	RebateVal

  - Same set of key fields must be used in corresponding start-of-day and commissions resources;
  - Key field names and values in these resources should match fields/values configured in commission info resources - MarketCommissions.txt;
  - these StartOfDayCurrentBillingCycleMarketCommissionsInfo.txt files can be generated by exporting commissions info or be provided by clients based on their own accounting data;
  - e.g.:

SecType	TrnDestination	TradedVolume	CommissionVal	RebateVal
CS	LEHMAN	7324700	...	...

Import/export

- StartOfDayCurrentBillingCycleMarketCommissionsInfo.txt is automatically loaded on start-up.
- Console commands:
  - `hifreq cm esdi` - to export commissions data to StartOfDayCurrentBillingCycleMarketCommissionsInfo.txt.
  - `hifreq cm rsdi` - to import commissions data from StartOfDayCurrentBillingCycleMarketCommissionsInfo.txt.

Strategy Management

Strategy is custom application code working with the local client API.
The framework provides a way to keep and edit strategy data which application can associate with some particular code. Strategy data is a record that is visible and editable in GUI.

See simple.HFTimerSample for a working example of strategy data API utilization.

Initializing strategy data

This example creates a strategy data record when running on a clean DB. Sequential runs will have strategy data recovered.

```
private static final String STRATEGY_NAME = "Timer Strategy";
private static final String PX_INCREMENT_FIELD = "PxIncrement";
private static final String PERIOD_FIELD = "Period";

@SuppressWarnings("unused")
private static void addStrategyData() throws Exception
{
   if (localClient_.getStrategyData(STRATEGY_NAME) == null)
   {
      //No strategy data recovered from DB, so need to create a new one.
      HFLocalStrategyDataFieldsContainerById strategyFields = new HFLocalStrategyDataFieldsContainerById();
      strategyFields.addName(STRATEGY_NAME);
      strategyFields.addStatus(IHFLocalStrategyData.STATUS_ACTIVE);
      strategyFields.add(PX_INCREMENT_FIELD, 250);
      strategyFields.add(PERIOD_FIELD, 4000);
      localClient_.addStrategyData(strategyFields);
   }
}
```

Using strategy data

This example gets strategy data record and uses its values to schedule a timer task.

```
private void useStrategyData()
{
   IHFLocalStrategyData strategyData = localClient_.getStrategyData(STRATEGY_NAME);
   synchronized (localClient_.getLockObject(strategyData))
   {
      long period = (long)strategyData.getNumericFieldValue(PERIOD_FIELD);
      if (strategyData.getStatus() == IHFLocalStrategyData.STATUS_ACTIVE)
      {
         //schedule task with specified period
      }
   }
}
```

Listening to strategy data updates

This part demonstrates how to be notified about GUI-initiated changes to a strategy data record. The listener should be added to the local client in order to get notified.

```
private static final class LocalClientListener extends HFLocalClientListenerAdapter
{
   @Override
   public void onStrategyDataUpdated(IHFLocalStrategyData strategyData, IRecordUpdate strategyDataUpdate)
   {
      boolean restartTask = false;
      if (strategyDataUpdate.getUpdatedFieldIndex(PERIOD_FIELD) >= 0)
      {
         restartTask = true;
      }

      int statusFieldIndex = strategyDataUpdate.getUpdatedFieldIndex(IHFLocalStrategyData.STATUS_FIELD_NAME);
      if ( statusFieldIndex >= 0)
      {
         int newStatus = (int) strategyDataUpdate.getUpdatedNumericFieldValueAt(statusFieldIndex);
         if (newStatus == IHFLocalStrategyData.STATUS_ACTIVE)
         {
            restartTask = true;
         }
         else
         {
            //stop timer task here
         }
      }

      if (restartTask)
      {
         //restart timer task here
      }
   }
}
```

Audit Utility

Audit Utility is used to report necessary information to log and to GUI report view.
The following sample sends a message through the utility. Different severity levels can be used.

```
private void sendAuditInformation()
{
   localClient_.getAuditUtility().sendAuditRecord(IHFLocalClientAuditUtility.AUDIT_SEVERITY_INFORMATION, "Every HiFreq Sample
has started up successfully");
}
```

Market Data

Market data overview

There are two ways to access market data and quote montage data through InfoReach API:

- Using Quote Montage Data Server (QMDS)
  - Has "pull" model: due to performance considerations it is some (most) cases not affordable to have analytics subscribe for L2 data. So, instead of publisher/subscriber model (like in MDF) QMDS provides pull model (requests are made from analytics when timer wakes up and module needs this data)
  - Provides configured aggregating functionality
  - Implies huge event flow from market data provider to QMDS (and therefore should not be used for obtaining NBBO)
- Market Data Facility (MDF)
  - Has "push" (publisher/subscriber) model

Using Market Data Facility (MDF)

Subscribing for market data (NBBO)

Obtain `IRDRecordFacilityProxy` by calling `GlobalSystem.getRecordFacilityProxy()` (see code snippet below).
Then use methods of `IRDRecordFacilityProxy`:

- `subscribe(IRDRecordFacilityListener, String)`
- `subscribe(IRDRecordFacilityListener, String[])`

```
IRDRecordFacilityProxy marketDataFacilityProxy = GlobalSystem.getRecordFacilityProxy();
IRDRecordFacilityListener marketDataListener = new MarketDataUpdatesListener("MarketData listener", true);
marketDataFacilityProxy.subscribe(marketDataListener , "IBM");
```

`IRDRecordFacilityListener` implementation sample:

```
private static class MarketDataUpdatesListener implements IRDRecordFacilityListener
{
   private String name_;
   private boolean debugLevelMax_;

   public MarketDataUpdatesListener(String name, boolean debugLevelMax)
   {
      name_ = name;
      debugLevelMax_ = debugLevelMax;
   }

   @Override
   public void onUpdate(IRDRecord record)
   {
      String instrumentId = record.getStringFieldValue(ITMSConstants.RDFLDTAG_INSTRUMENT);
      double lastPrice = record.getDoubleFieldValue(ITMSConstants.RDFLDTAG_LAST_PRICE);
      System.out.println("### ("+name_+") market data update for '"+instrumentId+"' arrived"
            + (debugLevelMax_ ? (": " + record) : "")
            + ".\n"
            + "\tLast price for '"+instrumentId+"' is " + lastPrice + ".\n");
   }
}
```

Subscribing for L1/L2 data

Obtain `IRDRecordFacilityProxy` by calling `GlobalSystem.getRecordFacilityProxy("QuoteMontageFacility")` (see code snippet below).
Then use methods of `IRDRecordFacilityProxy`:

- `subscribe(IRDRecordFacilityListener, String)`
- `subscribe(IRDRecordFacilityListener, String[])`

```
IRDRecordFacilityProxy quoteMontageFacilityProxy =
   GlobalSystem.getRecordFacilityProxy("QuoteMontageFacility");
IRDRecordFacilityListener quoteMontageListener = new QuoteMontageDataUpdatesListener(
      "QuoteMontage",
      "IBM[NYSE]",
      100,
      false);
quoteMontageFacilityProxy.subscribe(quoteMontageListener , "IBM");
```

`IRDRecordFacilityListener` implementation sample:

```
private static class QuoteMontageDataUpdatesListener implements IRDRecordFacilityListener
{
   private String name_;
   private String recordId_;
   private double desiredPrice_;
   private boolean debugLevelMax_;

   public QuoteMontageDataUpdatesListener(
         String name,
         String recordId,
         double desiredPrice,
         boolean debugLevelMax)
   {
      name_ = name;
      recordId_ = recordId;
      debugLevelMax_ = debugLevelMax;
      desiredPrice_ = desiredPrice;
   }

   @Override
   public void onUpdate(IRDRecord record)
   {
      Object recordId = record.getIdentifier();
      if (recordId.equals(recordId_))
      {
         System.out.println("### ("+name_+") update for '"+recordId+"' arrived"+
               (debugLevelMax_ ? ": "+record+"\n" : "\n"));
         double bestBidPrice = record.getDoubleFieldValue(ITMSConstants.RDFLDTAG_BESTBID1_PRICE);
         if (bestBidPrice > desiredPrice_)
         {
            // ...
         }
      }
   }
}
```

Subscribing for FOREX data

- Obtain `ITMSForexTradingFacilityProxy` by calling `GlobalSystem.getRecordFacilityProxy("ForexTrading")`.
- Then use methods of `ITMSForexTradingFacilityProxy`:
    - `subscribe(IRDRecordFacilityListener, String)`
    - `subscribe(IRDRecordFacilityListener, String[])`
- Then subscribe to necessary FX quotes using
    - `subscribeForSpotQuote(String, String, String, double)`

```
// Getting Forex Trading proxy
ITMSForexTradingFacilityProxy fxFacility
    = (ITMSForexTradingFacilityProxy)GlobalSystem.getRecordFacilityProxy("ForexTrading");

// Getting available providers list
String[] providers = fxFacility.getProviderNames();
for (int i=0; i<providers.length; i++)
{
    System.out.println( "### "+providers[i]);
}

// Subscribing for record update events
IRDRecordFacilityListener fxRecordListener = new ForexTradingDataUpdatesListener(
    "Forex Trading",
    "SimDB-FX", // pass null for all providers
    FX_RECORD_NAME, // pass null for all ccy pairs
    false);
fxFacility.subscribe(fxRecordListener, FX_RECORD_NAME);

// Subscribing for quotes from Simulator FX trading provider
// Note: Many providers ignore Size and Currency parameters, offering full book of available quotes.
// Provider/Symbol/Currency/Size
fxFacility.subscribeForSpotQuote("SimDB-FX", FX_RECORD_NAME, HF_FX_CCY, 200000);
// Provider/Symbol/Currency/Size
//fxFacility.subscribeForSpotQuote("SimMS-FX", "AUD/USD", "USD", 500000);
```

`IRDRecordFacilityListener` implementation sample:

```
private static class ForexTradingDataUpdatesListener implements IRDRecordFacilityListener
 {
    private String name_;
    private String providerName_;
    private String recordId_;
    private boolean debugLevelMax_;

    public ForexTradingDataUpdatesListener(
       String name,
       String providerName,
       String recordId,
       boolean debugLevelMax)
    {
       name_ = name;
       providerName_ = providerName;
       recordId_ = recordId;
       debugLevelMax_ = debugLevelMax;
    }

    @SuppressWarnings("unused")
@Override
    public void onUpdate(IRDRecord rdRecord)
    {
       String provider = rdRecord.getStringFieldValue(ITMSConstants.RDFLDTAG_PROVIDER);
       String rootRecordName = rdRecord.getParentRecordName();
       if (recordId_ != null && !recordId_.equals(rootRecordName))
          return; // do not process irrelevant record
       if (providerName_ != null && !providerName_.equals(provider))
          return; // do not process records from irrelevant provider

       String instrumentName = TMSForexNamesUtil.getInstrId(rdRecord.getName());

       // To get all asks and bids (maximum 10 of each), use the loop like this:
       TMSFXSubscriptionParams subscrParams = TMSForexNamesUtil.parseParamsFromRecordName(rdRecord.getName());
       for (int i = 0; i < 10; i++)
       {
          // Ask[i]
          boolean isAskTradable = TMSForexUtil.isTradable(rdRecord, i, false);
          double askPx = rdRecord.getDoubleFieldValue(ITMSConstants.RDFLDTAG_BESTASK1_PRICE+i);
          String askQuoteId = rdRecord.getStringFieldValue(ITMSConstants.RDFLDTAG_BESTASK1_QUOTE_ID+i);
          long askSize = rdRecord.getLongFieldValue(ITMSConstants.RDFLDTAG_BESTASK1_SIZE+i);
          String askCcy = subscrParams.getCcy();
          // Process only valid best Ask quote (with valid price)
          // and only tradable quote
          if (!Double.isNaN(askPx) && askPx > 0 && isAskTradable)
          {
             System.out.println(
                   "### ("+name_+"): update for "+askSize+" "+askCcy+" on '"+instrumentName
                   +"' arrived from "+provider+
                   (debugLevelMax_ ? ": "+rdRecord+"\n" : "\n"));
             // ...
          }
          // ...
          // Bid[i]
          boolean isBidTradable = TMSForexUtil.isTradable(rdRecord, i, true);
          double bidPx = rdRecord.getDoubleFieldValue(ITMSConstants.RDFLDTAG_BESTBID1_PRICE+i);
          String bidQuoteId = rdRecord.getStringFieldValue(ITMSConstants.RDFLDTAG_BESTBID1_QUOTE_ID+i);
          long bidSize = rdRecord.getLongFieldValue(ITMSConstants.RDFLDTAG_BESTBID1_SIZE+i);
          String bidCcy = subscrParams.getCcy();
          // ...
       }
    }
 }
```

Lightweight market data proxy

⚠ This section is only applicable for in-process MarketData, FOREX data.

Lightweight market data proxy overview

The purpose of this proxy to provide the fastest access to data in market data facility and minimize related performance overheads.
From the implementation point of view it is achieved by not maintaining records cache (unlike standard proxy).

⚠ Warning

- `IRDRecord` that comes to `IRDRecordFacilityListener#onUpdate` from this proxy is shared data holder i.e. data consumer isn't allowed to cache reference to it or perform any modification. If this data has to be accessible from different thread then it should be copied to another data structure.
- Take into account that all listeners are notified it the same thread. Moreover for some providers it might be the thread getting data from remote source. That's why consumer has to return control as soon as possible.

Lightweight market data proxy configuration

1. Configure your process to use "`MarketDataLocal`" market data facility (it is a pre-configured sample that uses lightweight proxy). For this add inclusion on "`Inprocess-Light`" section from `RecordFacilities.xml` to relevant process:

processes.xml

```
<ELTHiFreqApp name="ELTHiFreqApp">
 <GlobalSystem... >
  ...
  <Include
   resource="/ElTrader/TMS/Config/MarketData/RecordFacilities.xml"
   datapath="/RecordFacilities/MarketData/Section[name=Inprocess-Light]"
  />
  ...
```

- Adjust "`MarketDataLocal`" market data facility if needed. Its configuration can be found inside "`Inprocess-Light`" section in `RecordFacilities.xml`:

RecordFacilities.xml

```
<Section name="Inprocess-Light">
 <RecordFacility
  ...
  name="MarketDataLocal"
  ...
 >
 </RecordFacility>
```

2. Set "`MarketDataLocal`" as default market data facility for your process. For this add `-DdefaultRecordDataFacility=MarketDataLocal` to script running relevant process:

ELTHiFreqMDSampleApp.pl

```
...
    _JAVA_OPTIONS() => {
        ...
 "-DdefaultRecordDataFacility" => "MarketDataLocal"
        ...
```

3. Remove unneeded market data proxies from your process(e.g. standard "`MarketData`" proxy):

processes.xml

```
<ELTHiFreqApp name="ELTHiFreqApp">
 <GlobalSystem... >
  ...
  <Include
   resource="/ElTrader/TMS/Config/MarketData/RecordFacilities.xml"
   datapath="/RecordFacilities/MarketData/Section[name=Inprocess-Light]"
  />
<!--
COMMENTED OUT unneeded proxies
  <Include
   resource="/ElTrader/TMS/Config/MarketData/RecordFacilities.xml"
   datapath="/RecordFacilities/MarketData/Section[name=Client-Remote-NoEntitlement]"
  />
  ...
-->
  ...
```

- Also comment out <`TargetingEventSource`> sections that depend on the removed items:

tms.xml

```
<ELTHiFreqApp ...>
 <TargetingEventSourceList>
<!--
COMMENTED OUT
  <TargetingEventSource
   coalesceRecords="true"
   specImpl="com.inforeach.recorddata.report.RDTargetingEventSourceSpec"
   threadPriority="3"
  >
   <CoalescingBin
    isCloningEvents="false"
    maxDelay="2000"
    minDelay="50"
   />
   <RecordDataSource recordFacilityName="MarketData"/>
  </TargetingEventSource>
-->
 ...
```

Lightweight forex trading proxy configuration

1. Configure your process to use lightweight "`ForexTrading`" market data facility (it is a pre-configured sample that uses lightweight proxy). For this add inclusion on "`Inprocess-Light`" section from `RecordFacilities.xml` to "`TMSystem`" section for relevant

process:

processes.xml

```
<ELTHiFreqApp name="ELTHiFreqApp">
 <TMSystem... >
  ...
  <Include
   resource="/ElTrader/TMS/Config/MarketData/RecordFacilities.xml"
   datapath="/RecordFacilities/ForexTrading/Section[name=Inprocess-Light]"
  />
  ...
```

2. Remove unneeded market data proxies from your process(e.g. standard "`ForexTrading`" proxy):

processes.xml

```
<ELTHiFreqApp name="ELTHiFreqApp">
 <GlobalSystem... >
  ...
<!--
COMMENTED OUT unneeded proxies
  <Include
   resource="/ElTrader/TMS/Config/MarketData/RecordFacilities.xml"
   datapath="/RecordFacilities/ForexTrading/Section[name=Client-Remote]"
  />
  ...
-->
  ...
```

Lightweight market data proxy usage

Lightweight proxy usage is very similar to usage of standard marked data proxy, but also these new methods are available:

- `addDirectListener(IRDRecordFacilityListener)` - adds to listener to be notified about ALL updates from facility;
- `removeDirectListener(IRDRecordFacilityListener)`.

Sample:

```
IRDRecordFacilityProxy marketDataFacilityLightweightProxy = GlobalSystem.getRecordFacilityProxy();
if (! (marketDataFacilityProxy instanceof RDInprocessPassthroughProxy))
{
    System.err.println("### Lightweight market data proxy is not properly configured.");
    return;
}
// using lightweight proxy like ordinary MDF proxy
IRDRecordFacilityListener marketDataListenerLightweightIBM =
    new MarketDataUpdatesListener("MarketData lightweight proxy listener for IBM", false);
marketDataFacilityLightweightProxy.subscribe(marketDataListenerLightweightIBM , "IBM");
IRDRecordFacilityListener marketDataListenerLightweightMSFT =
    new MarketDataUpdatesListener("MarketData lightweight proxy listener for MSFT", false);
marketDataFacilityLightweightProxy.subscribe(marketDataListenerLightweightMSFT , "MSFT");

// subscribing for ALL updates (only available with lightweight proxy)
IRDRecordFacilityListener marketDataListenerLightweightALL =
    new MarketDataUpdatesListener("MarketData lightweight proxy listener DIRECT", false);
((RDInprocessPassthroughProxy)marketDataFacilityLightweightProxy).addDirectListener(
        marketDataListenerLightweightALL);
```

Creating MDF record post-processor

MDF record post-processor purpose

Market Data Facility (MDF) provider's post-processor allows to examine/adjust records after an update is received from provider, but before the update is fired to subscribers (e.g. to client MDF proxies).

MDF record post-processor code sample

A component that examines/augments records published from MDF must implement `com.inforeach.recorddata.IRDProviderPostprocessor` interface:

```
/**
 * Sample MD provider postprocessor implementation. For updates with "IBM"
 * instrument ({@link #TRACKED_INSTRUMENT}) prints out updates and changes
 * custom field ("RecommendedOrdType") with tag=2000 (
 * {@link #CUSTOM_FIELD_VALUE_TO_SET}) to "Limit" (
 * {@link #CUSTOM_FIELD_VALUE_TO_SET}) in case new LastPx is bigger than 100 (
 * {@link #UPPER_LAST_PX}).
 */
public class HFMDProviderPostprocessorSample implements IRDProviderPostprocessor
{
    private static final String TRACKED_INSTRUMENT = "IBM";
    private static final int CUSTOM_FIELD_TAG = 2001;
    private static final String CUSTOM_FIELD_VALUE_TO_SET = "Limit";
    private static final double UPPER_LAST_PX = 100;

    public HFMDProviderPostprocessorSample()
    {
        log("created using default constructor");
    }

    public HFMDProviderPostprocessorSample(String configStr)
    {
        log("created using constructor taking single java.lang.String parameter;"
                + " parameter value: '" + configStr + "'");
    }

    @Override
    public void initialize(IRDRecordFacilityForProvider facility)
    {
    }

    @Override
    public void process(IRDRecord record,
            IRDRecordFieldChecker fieldChecker)
    {
        String instrument = record.getStringFieldValue(ITMSConstants.RDFLDTAG_INSTRUMENT);
        if (!TRACKED_INSTRUMENT.equals(instrument))
        {
            return; // only process updates for TRACKED_INSTRUMENT
        }

        IIntIterator iter = fieldChecker.getFieldIterator();

        boolean shouldSetRecommendedOrdType = false;
        String changedFieldsInfo = ""; // (use StringBuilder to increase performance)
        while (iter.hasNext())
        {
            int tag = iter.next();

            if (fieldChecker.hasFieldChanged(tag))
            {
                // append info on changed fields to buffer
                String fieldName = record.getContext().getFieldName(tag);
```

```
            changedFieldsInfo += "\n\t" + fieldName + " -> "
                + (record.getContext().isFieldNumeric(tag)
                      ? record.getDoubleFieldValue(tag)
                      : record.getStringFieldValue(tag));

            if (tag == ITMSConstants.RDFLDTAG_LAST_PRICE
                  && record.getDoubleFieldValue(tag) > UPPER_LAST_PX)
            {
               shouldSetRecommendedOrdType = true;
            }
         }

      }

      if (changedFieldsInfo.length() != 0)
      {
         log("fields changed in record " + record + ": " + changedFieldsInfo);
      }

      if (shouldSetRecommendedOrdType)
      {
         String customFieldName = record.getContext().getFieldName(CUSTOM_FIELD_TAG); // RecommendedOrdType
         record.setStringFieldValue(CUSTOM_FIELD_TAG, CUSTOM_FIELD_VALUE_TO_SET);
         log("!!! changed '" + customFieldName + "' to '" + CUSTOM_FIELD_VALUE_TO_SET
               + "' for record " + record);
      }

   }

   private void log(String message)
   {
      System.out.println(
            "### (" + getClass().getSimpleName() + ") RD provider postprocessor: " + message + ".");
```

```
    }
}
```

MDF record post-processor configuration

1. In order to plug in the custom post-processor into the Market Data Facility one has to add `postprocessorImplementationClass` attr ibute to MDF providers' configuration file:

   Directory/ElTrader/TMS/Config/MarketData/MarketQuoteProviders.xml

```
<?xml version="1.0"?>
  <RecordFacilityProviderList
    ...
    postprocessorImplementationClass = "HFMDProviderPostprocessorSample"
    postprocessorConfig = "===TEST==="
  >
...
```

   If `postprocessorConfig` attribute is specified in addition to `postprocessorImplementationClass` and post-processorimplementation class has constructor taking single `java.lang.String` parameter then post-processor is instantiated by this constructor, value of `postprocessorConfig` is passed as parameter. Otherwise default constructor is used.
2. In case some custom market data fields are to be added, then MDF metadata file should be changed like shown below:

   ⚠ Currently tags 1-2000 are reserved for the standard TMS set of fields. Therefore, the custom fields should have tag numbers greater than 2000. By adding the XML element describing the custom field to file one instructs TMS Market Data Facility to reserve a place for the field in each record so that the user post-processor can set the field's value.

   Directory/ElTrader/TMS/Config/MarketData/MarketQuoteFieldsMetaData.xml

```
<?xml version="1.0"?>
<MarketMetadata defaultFieldsMetaDataName="Marketdata MetaData">
 <FieldList
  defaultFieldMetaDataClassName="com.inforeach.recorddata.RDFieldMetaData"
  name="Marketdata MetaData"
 >
...
<!-- Added custom field for Recommended Order Type -->
      <Field
          identifier = "2001"
          name = "RecommendedOrderType"
          type = "alphaNumeric"
      />
```

Creating custom MD field

Implementing new MD field

ⓘ Code samples listed below provide same functionality as `com.inforeach.recorddata.fields.RDRecentAvgField` MD field.

1. Implement `IRDField`. It is recommended to extend `RDFieldBase` like shown below:

```
/**
```

```
 * Sample record data field for calculating average value of recent field values.
 */
public class HFRecentAvgCustomMDField extends RDFieldBase
{
   private final String fieldToAvg_;
   private final int numOfRecentValuesToAvg_;

   private int fieldToAvgIndex_;

   public HFRecentAvgCustomMDField(RDFieldMetaData metaData)
   {
      super(metaData);
      HFRecentAvgCustomMDFieldMetaData customMetaData = (HFRecentAvgCustomMDFieldMetaData) metaData;
      fieldToAvg_ = customMetaData.getFieldToAvg();
      numOfRecentValuesToAvg_ = customMetaData.getNumOfRecentValuesToAvg();
      fieldToAvgIndex_ = -1;
   }

   @Override
   public Object[] getDependencies()
   {
      return new Object[] {fieldToAvg_};
   }

   @Override
   public IRDFieldCalculationHelper createCalculationHelper()
   {
      return new HFRecentAvgCustomMDFieldCalcHelper(numOfRecentValuesToAvg_);
   }

   @Override
   public double getNumericValue(RDInternalRecord record, IRDFieldCalculationHelper helper)
   {
      try
      {
         HFRecentAvgCustomMDFieldCalcHelper h = (HFRecentAvgCustomMDFieldCalcHelper)helper;

         if (numOfRecentValuesToAvg_ > 0)
         {
            double newFieldToAvgValue = record.getNumericFieldValueAt(getFieldToAvgIndex(record));
            double avgValue = h.getAvgValue(); // get stored avg value

            // Only perform calculation in case field value has changed.
            // Return old avg value otherwise.
            if (Double.doubleToLongBits(h.getNewestValue()) != Double.doubleToLongBits(newFieldToAvgValue))
            {
               // Avg value calculation algorithm:
               // 1) Derive sum of old values by multiplying: (last avg value) * (number of averaged values)
               // 2) In case number of averaged values has reached numOfRecentValuesToAvg_,
               //    remover (by subtracting) the oldest value from sum of old values.
               // 3) Add new field value to sum of old values (deriving sum of values).
               // 4) New avgValue = (sum of values) / (values count).


               // 1)
               avgValue = avgValue * h.getListSize();

               // 2)
               if (h.getListSize() == numOfRecentValuesToAvg_)
               {
                  avgValue = avgValue - h.getOldestValue();
               }

               // 3)
               avgValue = avgValue + newFieldToAvgValue;

               // 4)
```

```
                h.addValue(newFieldToAvgValue); // add new value for increasing list size
                avgValue = avgValue / h.getListSize(); // divide by the number of values
                h.setAvgValue(avgValue); // store new avg value
            }
            return avgValue;
        }
        else
        {
            return 0.0;
        }
    }
    catch (Throwable ex)
    {
        System.out.println("### Caught exception when calculating field '" + getName() + "'.");
        ex.printStackTrace();
    }
    return Double.NaN;
}

/**
 * Get and cache field to avg index in record.
 */
private int getFieldToAvgIndex(RDInternalRecord record)
{
    if (fieldToAvgIndex_ == -1)
    {
        IRDRecordContext context = record.getContext();
        fieldToAvgIndex_ = context.getFieldIndex(context.getFieldTag(fieldToAvg_));
    }

    return fieldToAvgIndex_;
}
}
```

2. Implement field metadata.
    - If new field doesn't need any custom configuration params, `RDFieldMetaData` can be used as field metadata.
    - Otherwise `RDFieldMetaData` should be extended like shown below:

```
/**
 * Custom{@link RDFieldMetaData} child to hold 2 params: name of field to
 * average ({@link #getFieldToAvg()}) and number of recent values to average (
 * {@link #getNumOfRecentValuesToAvg()}). Setters for these properties will be
 * invoked automatically via reflection mechanism.
 */
public class HFRecentAvgCustomMDFieldMetaData extends RDFieldMetaData
{
   protected String fieldToAvg_;
   protected int numOfRecentValuesToAvg_;

   public HFRecentAvgCustomMDFieldMetaData()
   {
   }

   public void setFieldToAvg(String fieldToAvg)
   {
      fieldToAvg_ = fieldToAvg;
   }

   public void setNumOfRecentValuesToAvg(String numOfRecentValuesToAvg)
   {
      try
      {
         numOfRecentValuesToAvg_ = Integer.parseInt(numOfRecentValuesToAvg);
      }
      catch (Throwable ex)
      {
         numOfRecentValuesToAvg_ = 0;
         System.out.println("### Invalid string value '" + numOfRecentValuesToAvg
               + "' was specified in numOfRecentValuesToAvg.");
         ex.printStackTrace();
      }

      if (numOfRecentValuesToAvg_ < 0)
      {
         System.out.println("### numOfRecentValuesToAvg should be non-negative, but "
               + numOfRecentValuesToAvg_ + " was found. Defaulted to zero.");
         numOfRecentValuesToAvg_ = 0;
      }
   }

   public String getFieldToAvg()
   {
      return fieldToAvg_;
   }

   public int getNumOfRecentValuesToAvg()
   {
      return numOfRecentValuesToAvg_;
   }

   public String toString()
   {
      return getClass().getSimpleName() + "[fieldoAvg="+fieldToAvg_+", numOfRecentValuesToAvg=" +
            numOfRecentValuesToAvg_+", parent: "+super.toString()+"]";
   }
}
```

3. Implement calculation helper (`IRDFieldCalculationHelper`) (if needed) like shown below:

```
/**
 * Sample calculator helper for {@link HFRecentAvgCustomRDField}.
```

```
 */
public class HFRecentAvgCustomMDFieldCalcHelper implements IRDFieldCalculationHelper
{
    protected int oldestValueIndex_;
    protected int newestValueIndex_;
    protected int listSize_;
    protected double[] values_;
    protected double avgValue_;

    public HFRecentAvgCustomMDFieldCalcHelper(int size)
    {
        listSize_ = 0;
        oldestValueIndex_ = 0;
        newestValueIndex_ = -1;
        values_ = new double[size];
        Arrays.fill(values_, 0);
    }

    public Object clone()
    {
        HFRecentAvgCustomMDFieldCalcHelper obj;
        try
        {
            obj = (HFRecentAvgCustomMDFieldCalcHelper)super.clone();
        }
        catch (CloneNotSupportedException ex)
        {
            throw new InternalError();
        }
        obj.oldestValueIndex_ = oldestValueIndex_;
        obj.newestValueIndex_ = newestValueIndex_;
        obj.listSize_ = listSize_;
        obj.values_ = (double[])values_.clone();
        obj.avgValue_ = avgValue_;
        return obj;
    }

    public double getAvgValue()
    {
        return avgValue_;
    }

    public void setAvgValue(double avgValue)
    {
        avgValue_ = avgValue;
    }

    public void addValue(double value)
    {
        // move oldest value cursor
        if (listSize_ == values_.length)
        {
            // move oldest cursor only if value list is filled
            oldestValueIndex_ ++;
            if (oldestValueIndex_ >= values_.length)
            {
                oldestValueIndex_ = 0;
            }
        }

        // increase newest value cursor and set value
        newestValueIndex_++;
        if (newestValueIndex_ >= values_.length)
        {
            newestValueIndex_ = 0;
        }
        values_[newestValueIndex_] = value;
```

```
    // increase size if necessary
    if (listSize_ < values_.length)
    {
        listSize_ ++;
    }
}

/**
 * @return last value added to array.
 */
public double getNewestValue()
{
    if (newestValueIndex_ < 0)
    {
        return Double.NaN;
    }
    else
    {
        return values_[newestValueIndex_];
    }
}

public double getOldestValue()
{
    return values_[oldestValueIndex_];
}

public int getListSize()
{
    return listSize_;
```

```
    }
}
```

Configuring custom MD field

Add field to MDF fields metadata resource:

Directory/ElTrader/TMS/Config/MarketData/MarketQuoteFieldsMetaData.xml

```
<?xml version="1.0"?>
<MarketMetadata defaultFieldsMetaDataName="Marketdata MetaData">
 <FieldList
  defaultFieldMetaDataClassName="com.inforeach.recorddata.RDFieldMetaData"
  name="Marketdata MetaData"
 >
  ...
  <Field
   className="recorddata.HFRecentAvgCustomMDField"
   metaDataClassName="recorddata.HFRecentAvgCustomMDFieldMetaData"
   name="HFRecentAvgCustomMDField_RecentPxAvg"
   identifier="2002"
                    type="double"
   fieldToAvg="LastPx"
   numOfRecentValuesToAvg="3"
  />
 </FieldList>
</Metadata>
```

After TMS restart the field should appear in MDF records.

Using Quote Montage Data Server (QMDS)

QMDS configuration

Quote Montage Data Server configuration

- Configuration example for inward adapter for ELTMarketDataFacilities component in <TMSystem> section in processes.xml:

\Directory\ElTrader\TMS\Config\processes.xml

```
<ELTMarketDataFacilities name="ELTMarketDataFacilities">
        <TMSystem
                ...
                <QuoteMontageDataServer
                        name = "QuoteMontageDataServer"
                        recordFacilityName = "QuoteMontageFacility"
                        blockingTimeout = "200"
                        unsubsribeTimeout = "300000"
                >
                        <InwardAdapterList>
                                <InwardAdapter kind = "RMI" name = "QuoteMontageDataServer" />
                        </InwardAdapterList>
                </QuoteMontageDataServer>
```

blockingTimeout attr is timeout for blocking QMDS `get*()` call when there is still no data.

unsubsribeTimeout attr is timeout for unsubscribe from QM RDF after some time of inactivity.

- Configuration example for outward adapter for ELTPortfolioSatellite (TMS clients will most likely need to use ELTClientApp or ELTHiFreq App instead of ELTPortfolioSatellite) component in <TMSystem> section in processes.xml:

\Directory\ElTrader\TMS\Config\processes.xml

```
<ELTPortfolioSatellite name="ELTPortfolioSatellite">
    <TMSystem
        ...
        <QuoteMontageDataServer>
            <OutwardAdapter kind="RMI" name = "QuoteMontageDataServer" />
        </QuoteMontageDataServer>
```

- We have to have sendChildRecords=true and autoPopulateParentRecords=false for "QuoteMontageFacility":
  - for TMS 7.0:

\Directory\ElTrader\TMS\Config\processes.xml

```
<ELTMarketDataFacilities name="ELTMarketDataFacilities">
    <TMSystem
        ...
        <RecordFacility
            name="QuoteMontageFacility"
            autoPopulateParentRecords="false"
            sendChildRecords="true"
            ...
        >
```

  - for TMS 7.1:

\Directory\ElTrader\TMS\Config\MarketData\RecordFacilities.xml

```
<QuoteMontageFacility>
...
 <Section name="Server">
 ...
  <RecordFacility
   name="QuoteMontageFacility"
   autoPopulateParentRecords="false"
   sendChildRecords="true"
   ...
  >
```

Quote Montage Data Server scaled configuration

- For scaling QuoteMontageDataServer usage start ELTMarketDataSatellite.pl with following scaled server number (like "1", "2" etc.) or uncomment needed in Start_All.pl
- Configuration example for scaled outward adapters for two servers for ELTPortfolioSatellite component in <TMSystem> section in proces ses.xml:

\Directory\ElTrader\TMS\Config\processes.xml

```
      <ELTPortfolioSatellite name="ELTPortfolioSatellite">
            <TMSystem
                  ...
  <QuoteMontageDataServer>
   <ServerAssigner

implementationClass="com.inforeach.eltrader.tms.components.qmdataserver.assigners.TMSQuoteMontageDataServerIndex
MembershipBasedAssigner"
    groupingType="S&amp;amp;P"
    groupingValue="S&amp;amp;P 100"
    symbolRange="'A'-'Z'"
   />
   <OutwardAdapterList>
    <OutwardAdapter
     kind="RMI" name="QuoteMontageDataServer1"
    />
    <OutwardAdapter
     kind="RMI" name="QuoteMontageDataServer2"
    />
   </OutwardAdapterList>
  </QuoteMontageDataServer>
```

Here ServerAssigner section defines specific server selection strategy: `TMSQuoteMontageDataServerDefaultAssigner`, `TMSQuoteMontageDataServerAlphabetBasedAssigner` or `TMSQuoteMontageDataServerIndexMembershipBasedAssigner`. If section absent default assigner will be used.

- Default assigner implements simple server selection algorithm bases on instrument's hash code.
  Configuration example for `TMSQuoteMontageDataServerDefaultAssigner`

```
 <QuoteMontageDataServer>
  <ServerAssigner

implementationClass="com.inforeach.eltrader.tms.components.qmdataserver.assigners.TMSQuoteMontageDataServerDefaultAssign
er"
  />
       ...
 </QuoteMontageDataServer>
```

- The alphabet based assigner divides symbol range defined in configuration on equal parts. Each part corresponds one server. The alphabet based assigner identifies for what part belongs first instrument symbol and, thus, which server will be use for this instrument.
  Configuration example for `TMSQuoteMontageDataServerAlphabetBasedAssigner`:

```
 <QuoteMontageDataServer>
  <ServerAssigner

implementationClass="com.inforeach.eltrader.tms.components.qmdataserver.assigners.TMSQuoteMontageDataServerAlphabetBase
dAssigner"
   symbolRange="'A'-'Z'"
  />
       ...
 </QuoteMontageDataServer>
```

- The index membership based assigner evenly assigns server from available servers list among "known" instruments. Known instruments set defines via grouping type and value from configuration and obtains from static data. If instrument not belongs to known instruments set it gets server via alphabet based assigner.
  Configuration example for `TMSQuoteMontageDataServerIndexMembershipBasedAssigner`

```
<QuoteMontageDataServer>
 <ServerAssigner

implementationClass="com.inforeach.eltrader.tms.components.qmdataserver.assigners.TMSQuoteMontageDataServerIndexMembers
hipBasedAssigner"
  groupingType="S&amp;P"
  groupingValue="S&amp;P 100"
  symbolRange="'A'-'Z'"
 />
      ...
</QuoteMontageDataServer>
```

Getting QMDS instance

Use `getQuoteMontageDataServer()` method of `TMSystem`.

Pre-subscribing for instrument

It is possible to pre-subscribe QMDS for specified instrument so that it can have data at hand when it needs it.
For this use `startListeningForQuotes(String)` method of `ITMSQuoteMontageDataServer`.

Obtaining L1 data

Use methods of `ITMSQuoteMontageDataServer` like shown in the sample below:

```
TMSQuoteMontageTopEntries qmData =
   qmds.getLevelOneEntries("IBM");
System.out.println("L1 entries for IBM:");
System.out.println("\tasks:");
for (TMSQuoteMontageEntry askEntry : qmData.getAskEntries())
{
   System.out.println("\t\t" + askEntry.getSource() + ": " + askEntry.getSize() + " shares @ "
         + askEntry.getPrice());
}
System.out.println("\tbids:");
for (TMSQuoteMontageEntry bidEntry : qmData.getBidEntries())
{
   System.out.println("\t\t" + bidEntry.getSource() + ": " + bidEntry.getSize() + " shares @ "
         + bidEntry.getPrice());
}
```

Obtaining L2 data

Use methods of `ITMSQuoteMontageDataServer` like shown in the sample below:

```
qmData = qmds.getLevelTwoEntries(
        "IBM",
        ITMSQuoteMontageDataServer.ALL_SIDE,
        false,
        3,
        20,
        ITMSConstants.PRICE_OFFSET_UNIT_TICK,
        null/*all sources*/);
System.out.println("\nL2 entries for IBM:");
System.out.println("\tasks:");
for (TMSQuoteMontageEntry askEntry : qmData.getAskEntries())
{
    System.out.println("\t\t" + askEntry.getSource() + ": " + askEntry.getSize() + " shares @ "
            + askEntry.getPrice());
}
System.out.println("\tbids:");
for (TMSQuoteMontageEntry bidEntry : qmData.getBidEntries())
{
    System.out.println("\t\t" + bidEntry.getSource() + ": " + bidEntry.getSize() + " shares @ "
            + bidEntry.getPrice());
}
```

Security Master

The basics of accessing Security Master

⚠ In INFOREACH API "Security Master Database" is referred to as "Static Data".

Obtain `ITMSStaticDataAccessor` instance by calling `HFSystem.getStaticData()`. Then use its methods to get the necessary Security Master data.

```
basicStaticDataFieldsSample("IBM", staticDataAccessor);
basicStaticDataFieldsSample("MSFT", staticDataAccessor);
```

```
/**
 * Shows obtaining values of basic static data fields of instrument info.
 */
private static void basicStaticDataFieldsSample(
        String instrument,
        ITMSStaticDataClient staticDataAccessor) throws TMSStaticDataException
{
    ITMSInstrumentInfo instrumentInfo = staticDataAccessor.getInstrumentInfo(instrument);
    System.out.println("### (static data test) instrument info for '" + instrument + "': "
            + instrumentInfo);
    if (instrumentInfo == null)
    {
        return;
    }
    System.out.println("### (static data test) primary exchange for '" + instrument
            + "': " + instrumentInfo.getExchange());
    System.out.println("### (static data test) tick size for '" + instrument + "': "
            + instrumentInfo.getTickSize());
    System.out.println("### (static data test) currency for '" + instrument + "': "
            + instrumentInfo.getCurrency());
}
```

Accessing custom Security Master data fields

Use methods of `TMSStaticDataCustomFieldsContext` and `ITMSInstrumentInfo`:

```
customStaticDataFieldsSample("IBM",  staticDataAccessor);
customStaticDataFieldsSample("MSFT", staticDataAccessor);
```

```
/**
   * Shows obtaining values of custom static data fields of instrument info.
   */
  private static void customStaticDataFieldsSample(
    String instrument,
          ITMSStaticDataClient staticDataAccessor) throws TMSStaticDataException
  {
   ITMSInstrumentInfo instrumentInfo = staticDataAccessor.getInstrumentInfo(instrument);
   if (instrumentInfo == null)
     {
       System.err.println("### (static data test) no instrument info for '"+ instrument +"'.");
       return;
     }

   // TMSStaticDataCustomFieldsContext instance is used
   // to obtain info about available custom fields
     ObjectToIntHashMap<String> customFieldIndices = staticDataAccessor.getInstrumentInfoCustomFieldIndices();

     // obtain ClosePx
int closePxFieldIndex = customFieldIndices.get("ClosePx");
   double closePx = instrumentInfo.getNumericCustomFieldValueAt(closePxFieldIndex);
   System.out.println("### (static data test) ClosePx for '"+ instrument +"': " + closePx);

   // obtain value for arbitrary custom field by its name
   String nameOfCustomField = "DailyVolatility";
   int customFieldIndex = customFieldIndices.get(nameOfCustomField);
   if (customFieldIndex < 0)
   {
    System.err.println("### (static data test) there is no such custom field: '"+ nameOfCustomField +"'.");
    return;
   }
     double valueOfCustomField = instrumentInfo.getNumericCustomFieldValueAt(customFieldIndex);
     System.out.println("### (static data test) value of '"
           + nameOfCustomField + "' numeric custom field for '"+ instrument +"': "
           + valueOfCustomField);
  }
```

Obtaining alternate instrument IDs (Bloomberg ID, SEDOL, etc.)

Use methods of `ITMSStaticDataAccessor` and `ITMSInstrumentInfo`:

```
String idSource_Bloomberg = ITMSStaticDataFacilityImplementor.ALTERNATE_ID_SOURCE_BLOOMBERG;
String idSource_SEDOL = ITMSStaticDataFacilityImplementor.ALTERNATE_ID_SOURCE_SEDOL;
String idSource_Custom = "<some custom ID source>";
alternateInstrumentIDSample("IBM", idSource_Bloomberg, staticDataAccessor);
alternateInstrumentIDSample("MSFT", idSource_SEDOL, staticDataAccessor);
alternateInstrumentIDSample("GOOG", idSource_Custom, staticDataAccessor);
```

```
/**
 * Shows obtaining alternate instrument ID, e.g. Bloomberg ID, SEDOL, etc.
 */
private static void alternateInstrumentIDSample(
    String instrument,
    String idSource,
    ITMSStaticDataClient staticDataAccessor)
        throws TMSStaticDataException
{
    String[] alternateIdSources = staticDataAccessor
            .getInstrumentAlternateIdSources();
    if (UtilityFunctions.indexOfInArray(idSource, alternateIdSources) < 0)
    {
        System.err.println("### (static data test) '" + idSource
                + "' alternate ID source is N/A.");
        return;
    }

    ITMSInstrumentInfo instrumentInfo = staticDataAccessor.getInstrumentInfo(instrument);
    if (instrumentInfo == null)
    {
        System.err.println("### (static data test) no instrument info for '" + instrument
                + "'.");
        return;
    }
    String alternateID = instrumentInfo.getAlternateInstrumentId(idSource);
    System.out.println("### (static data test) instrument '" + instrument + "' has ID='"
            + alternateID + "' in '" + idSource + "' alternate ID source.");
}
```

Obtaining exchange open/close times

Obtain relevant `ITMSExchangeInfo` instance using method `ITMSStaticDataAccessor.getExchangeInfo(String)`. Then use its methods:

```
exchangeInfoSample("NYSE", staticDataAccessor);
exchangeInfoSample("NASDAQ", staticDataAccessor);
exchangeInfoSample("<some exchange>", staticDataAccessor);
```

```
/**
 * Shows obtaining exchange info.
 */
private static void exchangeInfoSample(
        String exchange,
        ITMSStaticDataClient staticDataAccessor)
        throws TMSStaticDataException
{
    ITMSExchangeInfo exchInfo = staticDataAccessor.getExchangeInfo(exchange);
    System.out.println("### (static data test) exchange info for '" + exchange + "': "
            + exchInfo);
    if (exchInfo == null)
    {
        String[] knownExchanges = staticDataAccessor.getExchangeIds();
        System.err.println("### (static data test) there is no info on '" + exchange
                + "' exchange in static data. Please try these: "
                + Arrays.toString(knownExchanges) + ".");
        return;
    }

    if (!exchInfo.areOpenCloseTimesDefined())
    {
        System.err.println("### (static data test) no open / close times are defined"
                + " for '" + exchange + "'.");
        return;
    }

    Date openDateTimeHolder = new Date();
    Date closeDateTimeHolder = new Date();
    Date currentDateTime = new Date();
    TimeZone timeZone = GlobalSystem.getLocalTimeZone();
    Calendar calendar = Calendar.getInstance();
    exchInfo.getOpenCloseTimesForCurrentDate(openDateTimeHolder, closeDateTimeHolder,
            currentDateTime, timeZone, calendar);
    System.out.println("### (static data test) today open date/time for '"
            + exchange + "': " + openDateTimeHolder + ".");
    System.out.println("### (static data test) today close date/time for '"
            + exchange + "': " + closeDateTimeHolder + ".");
}
```

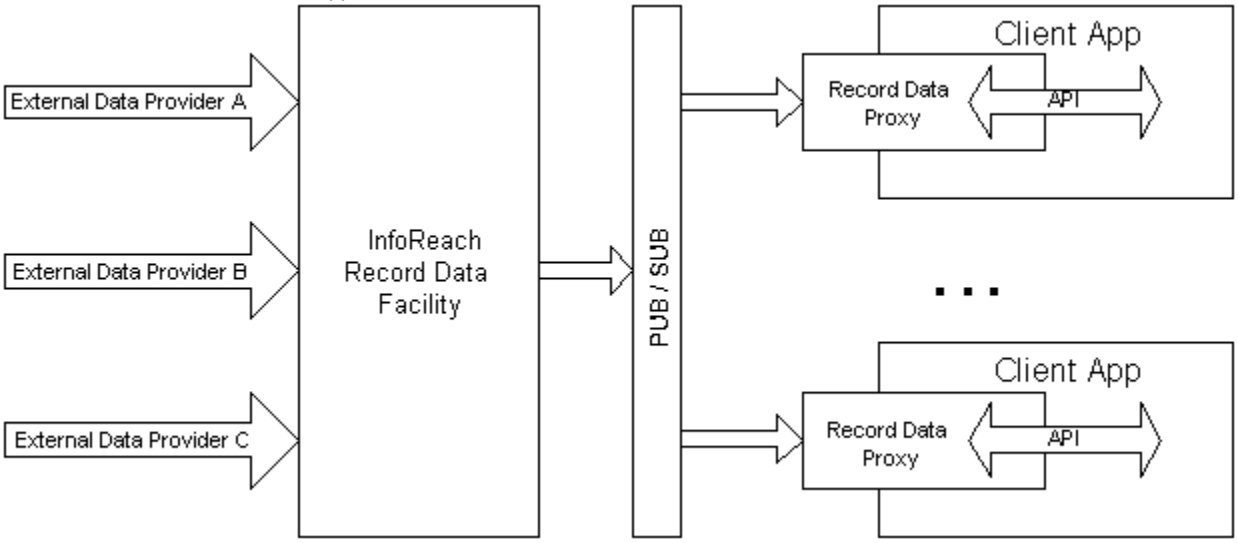
Custom Record Data

Record Data Facility overview

TMS Record Data Facility (RDF) accepts data flow from one or several third party providers and then publishes the data in the form of homogeneous records to all TMS components. TMS components internally use RDF API to subscribe for info from the RDF. This same API can be used by custom applications to subscribe for the record flow from the TMS RDF. By using this API to receive record data info from the TMS RDF users can avoid coding to each third party provider and achieve faster time to market for their external-data-dependent apps.

As shown in the figure below the custom application has to be configured to contain Record Data Facility Proxy (RDF Proxy) in order to receive the records from RDF. All communication between the client app and the RDF is done through the RDF Proxy. Thus, the client app is completely

oblivious the details of TMS deployment. The RDF can be running in any process on any machine. As long as the proxy is configured correctly to communicate with RDF the client app can receive the record data flow.



Implementing RDF provider overview

To implement custom provider a class extending `com.inforeach.recorddata.RDProvider` has to be written and registered with TMS Record Data Facility. Methods that must be implemented are (for detailed description refer to `RDProvider` javadoc);

- `subscribe(IRDRecord record)` - subscribes for record updates.
- `unsubscribe(String name)` - unsubscribes from record updates.
- `onRecordUpdate(IRDRecord record, Object param)` - here the data received from the external data source in the form of an arbitrary object is placed in RDF record's fields.
- `reconnectImpl()` - reconnect implementation.

RDF provider code sample

Please, refer to `backend\samples\java\ELTHiFreqApp\src\recorddata\ *.java`.
`com.inforeach.recorddata.RDProvider` is extended by `HFEmployeeInfoRDFProviderSample.java`.

RDF provider guidelines and tips

- Make sure not to subscribe second time to provider if an item is already subscribed.
- Make sure that provider works correctly with sequence of actions "subscribe","unsubscribe","subscribe". Check it with "staySubscribed="false" flag for Data Facility.
    - By default RDF never unsubscribes from once subscribed items. With staySubscribed="false" it does.
- Provider has to be implemented in such a way, so that multiple instances can be instantiated in the same process in the same or different RecordDataFacilities.
- "`d lt`" command could be used to track number of threads.
- Make sure to synchronize on `RDProvider#getLockObjForRecord` result when modifying RDF records.

RDF provider configuration

RDF MetaData

Create file like shown below:

Directory/ElTrader/TMS/Config/CustomRecordDataFieldsMetaData.xml

```
<?xml version="1.0"?>
<Metadata defaultFieldsMetaDataName="CustomRecordData MetaData">
 <FieldList
  defaultFieldMetaDataClassName="com.inforeach.recorddata.RDFieldMetaData"
  name="CustomRecordData MetaData"
 >
  <Field
   identifier="0"
   name="EmployeeID"
   type="String"
   />
  <Field
   identifier="1"
   name="FirstName"
   type="String"
   />
  <Field
   identifier="2"
   name="LastName"
   type="String"
   />
  <Field
   identifier="3"
   name="Position"
   type="String"
   />
  <Field
   identifier="4"
   name="Salary"
   type="double"
  />
 </FieldList>
</Metadata>
```

RDF providers resource

Create file like shown below:

Directory/ElTrader/TMS/Config/CustomRDFDataProviders.xml

```
<?xml version="1.0"?>
<RecordFacilityProviderList
 cacheResourceFile="directory://ElTrader/TMS/Config/CustomRDFDataCache.cfg"
 delayedUpdating="0"
 staySubscribed="false"
>
 <RecordFacilityProvider
  name = "HFEmployeeInfoRDFProviderSample"
  implementationClass="recorddata.HFEmployeeInfoRDFProviderSample"
 />
</RecordFacilityProviderList>
```

Server RDF

Adjust `processes.xml` like shown below:

⚠ In sample it is assumed that server and client RDFs run in separate processes.

⚠ It is advised to use `ELTHiFreqDataSatellite` process for hosting server RDF. To configure server RDF in another process some additional steps are required.

Directory/ElTrader/TMS/Config/processes.xml

```
<?xml version="1.0"?>
...
   <ELTHiFreqDataSatellite>
      <GlobalSystem
  ...
   <RecordFacility
    name="CustomRecordData"
    fieldsMetadataResource="/ElTrader/TMS/Config/CustomRecordDataFieldsMetaData.xml"
    checkNameAccessibility="false"
    subscriptionCleanupInterval="10"
    proxyImplementationClass="com.inforeach.eltrader.tms.recorddata.TMSRecordFacilityInprocessProxy"
    contextAssignerClassName="com.inforeach.recorddata.DefaultContextAssigner"
    logRecords="false"
    playbackInterval="3000"
    providersResource="/ElTrader/TMS/Config/CustomRDFDataProviders.xml"
    replayMode="false"
   >
    <!--ProcessList>
     <Process
      name="ELTHiFreqDataSatelliteBackup"
      recordFacilityName="CustomRecordDataDataBackup"
      />
    </ProcessList-->
    <InwardAdapterList>
     <InwardAdapter kind="RMI"/>
    </InwardAdapterList>
   </RecordFacility>

...
```

Client RDF proxy

Adjust `processes.xml` like shown below:

⚠ The sample below is given for applications running as ELTHiFreqApp.
For applications running as ELTClientApp `<ELTClientApp>...</ELTClientApp>` xml section should be adjusted instead.
To find out whether some application is ELTClientApp or ELTHiFreqApp check its run script.
E.g. `samples/ELTHiFreqCustomRDFSampleApp.pl` has

```
_ARGUMENTS() => [
      "-type"       => "ELTHiFreqApp",
      "-name"       => "timerSample",
      "-bootstrapDataFile"  => "../cfg/BackendBootstrap.xml"
   ]
```

which tell us it is ELTHiFreqApp (see second line).

⚠ In sample it is assumed that server and client RDFs run in separate processed.

Directory/ElTrader/TMS/Config/processes.xml

```
<?xml version="1.0"?>
...
   <ELTHiFreqApp name="ELTHiFreqApp">
      <GlobalSystem
  ...
   <RecordFacility
    name="CustomRecordData"
    fieldsMetadataResource="/ElTrader/TMS/Config/CustomRecordDataFieldsMetaData.xml"
    checkNameAccessibility="false"
    subscriptionCleanupInterval="10"
    proxyImplementationClass="com.inforeach.eltrader.tms.recorddata.TMSRecordFacilityProxy"
    cacheResourceFile="/ElTrader/TMS/Config/CustomRDFDataCache.cfg"
    propagateWholeRecord="true"
   >
    <AdjacentRecordFacility name="CustomRecordData">
     <OutwardAdapter kind="RMI"/>
    </AdjacentRecordFacility>
    <!--AdjacentRecordFacility name="CustomRecordDataBackup">
     <OutwardAdapter kind="RMI"/>
    </AdjacentRecordFacility-->
   </RecordFacility>
...
```

Using custom RDF provider

1. Run process hosting server RDF (e.g. `ELTHiFreqDataSatelliteApp.pl`).
2. Ensure "rdf gfn" command output contains "CustomRecordData".
3. Run process hosting client RDF (e.g. `backend/bin/samples/ELTHiFreqCustomRDFSampleApp.pl`).

Creating record post-processor for RDF provider

Purpose of record post-processor

Custom Record Data Facility (RDF) provider's post-processor allows to examine/adjust records after an update is received from provider, but before the update is fired to subscribers (e.g. to client RDF proxies).

Record post-processor code sample

A component that examines/augments records published from RDF must implement `com.inforeach.recorddata.IRDProviderPostprocessor` interface:

```
/**
 * Sample RD provider postprocessor implementation. Prints out updates and
 * changes salary to 123 ({@link #SALES_MANAGER_SALARY}) in case new employee position
 * is Sales Manager ({@link HFEmployeeInfoRDFProviderSample#POSITION_SALES_MANAGER}).
 */
public class HFRDProviderPostprocessorSample implements IRDProviderPostprocessor
{
    private static final int SALES_MANAGER_SALARY = 123;

    public HFRDProviderPostprocessorSample()
    {
        log("created using default constructor");
    }

    public HFRDProviderPostprocessorSample(String configStr)
    {
```

```
    log("created using constructor taking single java.lang.String parameter;"
            + " parameter value: '" + configStr + "'");


}

@Override
public void initialize(IRDRecordFacilityForProvider facility)
{
}

@Override
public void process(
        IRDRecord record,
        IRDRecordFieldChecker fieldChecker)
{
    IIntIterator iter = fieldChecker.getFieldIterator();

    String changedFieldsInfo = ""; // (use StringBuilder to increase performance)
    boolean shouldChangeSalary = false;
    while (iter.hasNext())
    {
        int tag = iter.next();
        if (!fieldChecker.hasFieldChanged(tag))
            continue;

        // Else

        // append info on changed fields to buffer
        String fieldName = record.getContext().getFieldName(tag);
        changedFieldsInfo += "\n\t" + fieldName + " -> "
                + (record.getContext().isFieldNumeric(tag)
                        ? record.getDoubleFieldValue(tag)
                        : record.getStringFieldValue(tag));


        // in case employee position is Sales Manager make sure his salary is proper
        if (tag == HFEmployeeInfoRDFProviderSample.RDFIELD_EMPLOYEE_POSITION)
        {
            String newPosition = record.getStringFieldValue(tag);

            if (HFEmployeeInfoRDFProviderSample.POSITION_SALES_MANAGER.equals(newPosition))
                continue; // we are not interested in this update

            shouldChangeSalary = record.getDoubleFieldValue(
                    HFEmployeeInfoRDFProviderSample.RDFIELD_EMPLOYEE_SALARY)
                    != SALES_MANAGER_SALARY;
        }
    }

    if (changedFieldsInfo.length() != 0)
    {
        log("fields changed in record " + record + ": " + changedFieldsInfo);
    }
    if (shouldChangeSalary)
    {
        record.setIntFieldValue(
                HFEmployeeInfoRDFProviderSample.RDFIELD_EMPLOYEE_SALARY,
                SALES_MANAGER_SALARY);
        log("!!! changed salary to " + SALES_MANAGER_SALARY + " for record " + record);
    }
}

private void log(String message)
{
    System.out.println(
            "### (" + getClass().getSimpleName() + ") RD provider postprocessor: " + message + ".");
```

```
  }
}
```

Record post-processor configuration

In order to plug in the custom post-processor into the Record Data Facility one has to add `postprocessorImplementationClass` attribute to RDF providers' configuration file:

```
<?xml version="1.0"?>
<RecordFacilityProviderList
 cacheResourceFile="directory://ElTrader/TMS/Config/CustomRDFDataCache.cfg"
 delayedUpdating="0"
 staySubscribed="false"
 postprocessorImplementationClass = "recorddata.HFRDProviderPostprocessorSample"
 postprocessorConfig = "===TEST==="

>
 <RecordFacilityProvider
  name = "HFEmployeeInfoRDFProviderSample"
  implementationClass="recorddata.HFEmployeeInfoRDFProviderSample"
 />
</RecordFacilityProviderList>
```

If `postprocessorConfig` attribute is specified in addition to `postprocessorImplementationClass` and post-processorimplementation class has constructor taking single `java.lang.String` parameter then post-processor is instantiated by this constructor, value of `postprocessorConfig` is passed as parameter. Otherwise default constructor is used.

Creating RD field for RDF provider

Same mechanisms can be utilized as for creating custom MD field.

Adding custom RDF providers field into GUI report

> ⚠ Steps below are not applicable for adding fields to TMS system reports, i.e. those which reside in `ElTrader/TMS/Config/SystemReports`.

Adjust tms.xml

Add support of custom RDF provider's fields to process(es) that host report(s) where the fields are to be added. For this in `tms.xml`, for relevant process section(s) (sample below is for `ELTHiFreqUserReportsSatellite`) add:

```
<ELTHiFreqUserReportsSatellite
 ...
>
 ...
 <TargetingEventSourceList>
  ...
  <TargetingEventSource
   coalesceRecords="true"
   specImpl="com.inforeach.recorddata.report.RDTargetingEventSourceSpec"
   subscriptionIdBuilderImpl="com.inforeach.recorddata.report.RDDefaultSubscriptionIdBuilder"
   threadPriority="3"
  >
   <RecordDataSource recordFacilityName="<Custom RDF name here>"/>
  </TargetingEventSource>
 </TargetingEventSourceList>
```

For HiFreq use case most probable processes to be adjusted are: `ELTHiFreqUserReportsSatellite`, `ELTHiFreqApp` and `ELTHiFreqDataSatellite`.

Find report's metadata and levels resources

1. Find out report's metadata name from report spec - it is stored in `metaDataName`/`tableMetaDataName`/`treeMetaDataName` attribute.
2. In `ElTrader\TMS\Config\MetaDataList.xml` find relevant `<MetaData>` section (by metadata name).
    - metadata resource - see value of `configurationResource` in `<Fields.../>`.
    - (for tree reports) levels resource - see value of `configurationResource` in `<Levels.../>`.

Adjust report's metadata

Add section like shown below to relevant report's metadata:

- (for table report)

```
<RPTFields>
 ...
 <FieldList ...>
  ...
  <Field
   className="com.inforeach.report.table.targetedtable.RPTTargetedTableField"
   metaDataClassName="com.inforeach.report.table.targetedtable.RPTTargetedTableFieldMetaData"
   identifier="<Arbitrary field name (for GUI report) here>"
   recordFieldId="<Name of custom RDF provider's field to be added to report>"
   targetingSourcesDependencies="<Custom RDF name here>,<Subscription identifier field name>"
   type="<field data type>"
  />
```

- (for tree report)

```
<RPTFields>
 ...
 <FieldList ...>
  ...
  <Field
   className="com.inforeach.report.tree.targetedtree.RPTTreeTargetedField"
   metaDataClassName="com.inforeach.report.tree.targetedtree.RPTTreeTargetedFieldMetaData"
   identifier="<Arbitrary field name (for GUI report) here>"
   recordFieldId="<Name of custom RDF provider's field to be added to report>"
   targetingSourcesDependencies="<Custom RDF name here>,<Subscription identifier field name>"
   type="<field data type>"
  />
```

Adjust tree report's levels

For tree reports: in corresponding levels resource add new field to desired levels.

Add fields to report specification

Just add `<Field id="<field name>"/>` to fields list in report specification.
For tree report make sure to add new field to proper level.

Subscription identifier, RDF field name and GUI report field name

Subscription identifier is key on which equivalence/mapping is established between RDF and report records.
On GUI report's side it is obtained from GUI report field (configured in `targetingSourcesDependencies`). On RDF's side it is obtained by calling `com.inforeach.recorddata.IRDRecord#getName()`.

Example
Let's assume we have custom RDF named `LastPxRDF` providing `LastPxFromCustomRDF` for subscribed instruments.
Custom RDF metadata will have only 1 field:

```
<!-- this is our only data field -->
<Field
 identifier="15000"
 name="LastPxFromCustomRDF"
 type="double"
>
```

Let's assume we need to show this `LastPxFromCustomRDF` in "Market Targets" report as field named `LastPxGUI`:

- `identifier="LastPxGUI"`;
- `recordFieldId="LastPxFromCustomRDF"`;
- `targetingSourcesDependencies="LastPxRDF,Instrument"`;
- for our records `com.inforeach.recorddata.IRDRecord#getName()` should return relevant instrument name;
- `Instrument` is already present in "Market Targets" report.

Add custom field to GUI report

After performed adjustments new field should appear in GUI report editor. Further steps are pretty trivial (to be performed in GUI report editor):

1. add field to "Selected fields" (in "Structure");
2. add field to "Visible field" (in "Field Order").

Adding Custom GUI Actions

Purpose

The document describes how invoke custom client's code from InfoReach GUI.

Overview of custom GUI actions

- InfoReach GUI uses actions to build menus, context-sensitive menus and toolbar buttons.
- Actions are "instances" of action types. InfoReach custom actions relate to action types as Java objects relate to Java classes.
- There are three ways to create new action capable of invoking custom client's code from InfoReach GUI:
  - Create custom GUI actions using `External Action` type.
  - Implement new action type. This also assumes implementing action-specific action logic, description, spec, spec editor, spec converter.
  - Create GUI actions invoking custom code in backend.
    - Simple implementation allows passing configurator parameters and selected records to backend code.
    - Advanced implementation additionally allows passing any custom parameters to backend code and creating an editor for these.

Creating custom GUI actions using External Action type

Overview External Action type custom GUI actions

Adding custom GUI actions using `External Action` type is the most simple way to invoke custom client's code from InfoReach GUI.
It assumes 2 steps:

1. Implementing `IRPTCustomContextActionHandler`.
2. Creating new action of `External Action` type.

Implementing IRPTCustomContextActionHandler

Main idea is to implement a method to be invoked from InfoReach GUI.
Note it accepts currently selected records and configured params.

Sample `IRPTCustomContextActionHandler` implementation:

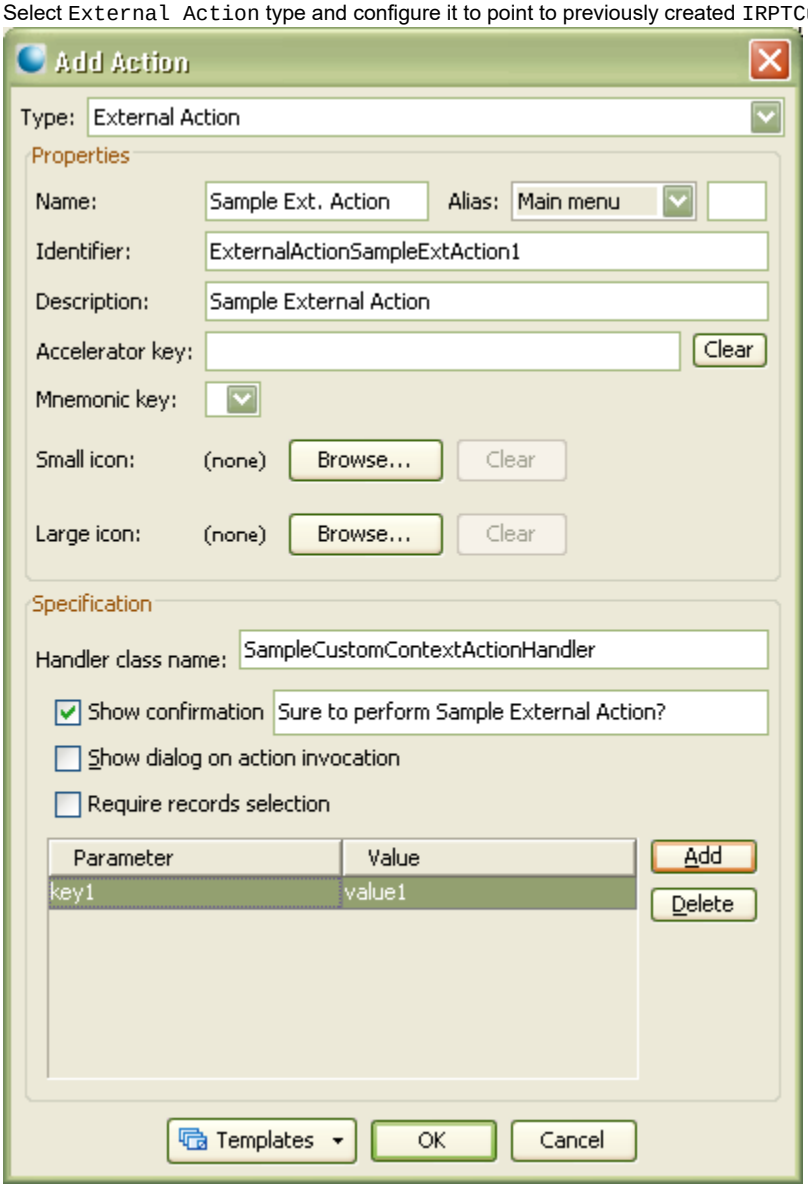
```
public class HFSampleCustomContextActionHandler implements IRPTCustomContextActionHandler
{
   @Override
   public void handleAction(IRecord[] selectedRecords, Configurator params)
   {
      String message = "";

      // List selected records
      message += "Selected records: ";
      if (selectedRecords.length == 0)
      {
         message += " (none)";
      }
      else
      {
         message += "\n";
         final int clOrdIdFieldIndex = selectedRecords[0].getFieldIndex("ClientOrdId");
         final int managerIdFieldIndex = selectedRecords[0].getFieldIndex("ManagerId");

         for (int i = 0; i < selectedRecords.length; i++)
         {
            if (i != 0)
            {
               message += "\n";
            }
            message += selectedRecords[i].getIdentifier();

            // append ClientOrdId and ManagerId (if present)
            boolean recordHasClientOrdId = clOrdIdFieldIndex >= 0;
            boolean recordHasManagerId = managerIdFieldIndex >= 0;
```

```
        if (recordHasClientOrdId || recordHasManagerId)
        {
            message += " (";
            if (recordHasClientOrdId)
            {
                message += "ClientOrdId="
                        + selectedRecords[i].getStringFieldValueAt(clOrdIdFieldIndex);
            }
            if (recordHasManagerId)
            {
                if (recordHasClientOrdId)
                {
                    message += ", ";
                }
                message += "ManagerId="
                        + (int) selectedRecords[i]
                                .getNumericFieldValueAt(managerIdFieldIndex);
            }
            message += ")";
        }
    }
}

message += "\n\n";

// List Configurator params
message += "Action parameters:";
if (params.size() == 0)
{
    message += " (none)";
}
else
{
    message += "\n";
    Set<Map.Entry> entrySet = params.entrySet();
    for (Map.Entry e : entrySet)
    {
        message += "key: '" +  e.getKey() + "', value: '" +  e.getValue() + "'\n";
    }
}

JOptionPane.showMessageDialog(null, message);
```

```
  }
}
```

Creating new action of External Action type

1. Go to `Tools` -> `Actions...`.
2. Select `Custom Actions` tab.
3. Press `Add...`.
4. Select `External Action` type and configure it to point to previously created `IRPTCustomContextActionHandler` implementation:



Implementing new action type

Overview of implementing new action type

If creating of new action of `External Action` type is not flexible enough, then implementing new action type should be considered. Implementing new action type involves:

1. Implementing `IActionSpec` - to hold action-specific configuration.
2. Implementing `IConfiguredAction` - to hold action's logic.
3. Implement `IActionSpecConverter` - to facilitate saving/loading of action spec.

4. Implement `IActionSpecEditor` - to facilitate editing action spec.
5. Register action type - to make new custom action visible to InfoReach system.
6. Register action spec converter - to make new custom action visible to InfoReach system.

> Note
> None of aforementioned steps can be skipped. E.g. if action assumes no spec, then dummy spec, spec converter and spec editor should be implemented and registered properly.

Implementing IActionSpec

The recommended way is to extend `AbstractActionSpec`.
Below is sample of extending `AbstractActionSpec` that can hold one String property - `message`:

```
public class HFSampleCustomActionSpec extends AbstractActionSpec
{
   private final String message_;

   public HFSampleCustomActionSpec(String message)
   {
      message_ = message;
   }

   public String getMessage()
   {
      return message_;
   }

   @Override
   public boolean equalsTo(IActionSpec anotherSpec)
   {
      if (!super.equalsTo(anotherSpec))
      {
         return false;
      }
      else if (!(anotherSpec instanceof HFSampleCustomActionSpec))
      {
         return false;
      }
      else
      {
         HFSampleCustomActionSpec anotherSampleCustomActionSpec =
               (HFSampleCustomActionSpec) anotherSpec;
         String anotherMessage = anotherSampleCustomActionSpec.message_;
         return message_ == anotherMessage
            || (message_ != null && message_.equals(anotherMessage));
      }
   }

   public int hashCode()
   {
      final int prime = 31;
      int result = 1;
      result = prime * result + ((message_ == null)
         ? 0
         : message_.hashCode());
      return result;
   }
}
```

Implementing IConfiguredAction

The recommended way is to extend `AbstractConfiguredAction`.
Below is sample of extending `AbstractConfiguredAction` that when fired shows message dialog with message taken from spec:

```
public class HFSampleCustomAction extends AbstractConfiguredAction<HFSampleCustomActionSpec>
{
   public HFSampleCustomAction(IActionProperties properties)
   {
      super(properties);
   }

   @Override
   public Class<HFSampleCustomActionSpec> getSpecClass()
   {
      return HFSampleCustomActionSpec.class;
   }

   @Override
   public void actionPerformed(ActionEvent e)
   {
      HFSampleCustomActionSpec spec = getSpec();
      JOptionPane.showMessageDialog(null, spec.getMessage());
   }
}
```

Implement IActionSpecConverter

Sample `IActionSpecConverter` implementation:

```
public class HFSampleCustomActionSpecConverter implements IActionSpecConverter<HFSampleCustomActionSpec>
{
   private static final String ATTR_SMESSAGE = "message";

   public Class<HFSampleCustomActionSpec> getSpecClass()
   {
      return HFSampleCustomActionSpec.class;
   }

   @Override
   public String specToString(HFSampleCustomActionSpec spec) // should return XML string
   {
      if (!(spec instanceof HFSampleCustomActionSpec))
      {
         return null;
      }

      XMLData xmlData = new XMLData();
      HFSampleCustomActionSpec actionSpec = (HFSampleCustomActionSpec)spec;

      xmlData.setAttributeValue(ATTR_SMESSAGE, actionSpec.getMessage());

      return xmlData.toString();
   }

   @Override
   public HFSampleCustomActionSpec stringToSpec(String str) // str is an XML string
   {
      XMLData xmlData = XMLData.valueOf(str);
      String message = xmlData.getAttributeStringValue(ATTR_SMESSAGE, null);

      return new HFSampleCustomActionSpec(message);
   }
}
```

Implement IActionSpecEditor

Sample `SampleCustomActionSpecEditor` implementation:

```
public class HFSampleCustomActionSpecEditor extends JPanel implements IActionSpecEditor<HFSampleCustomActionSpec>
{
    /**
     *
     */
    private static final long serialVersionUID = -7246319394584327567L;
    private JTextField messageEditor_ = new JTextField(10);

    public HFSampleCustomActionSpecEditor()
    {
        super();
        add(new JLabel("Message:"));
        add(messageEditor_);
    }

    @Override
    public Component getEditorComponent()
    {
        return this;
    }

    @Override
    public Class<HFSampleCustomActionSpec> getActionSpecClass()
    {
        return HFSampleCustomActionSpec.class;
    }

    @Override
    public HFSampleCustomActionSpec getActionSpec()
    {
        return new HFSampleCustomActionSpec(messageEditor_.getText());
    }

    @Override
    public void checkEditorData() throws ActionConfigurationException
    {
        // nothing here
    }

    @Override
    public void setActionSpec(HFSampleCustomActionSpec spec)
    {
        HFSampleCustomActionSpec actionSpec = (HFSampleCustomActionSpec) spec;
        messageEditor_.setText(actionSpec.getMessage());
    }
}
```

Registering action type

Add to `directory://ElTrader/TMS/Config/Actions/ActionTypes.xml` an entry like:

```
<ActionType
 class="action.HFSampleCustomAction"
 description="Sample custom actions showing dialog with configured message"
 specEditorClass="action.HFSampleCustomActionSpecEditor"
 type="Sample Custom Action"
/>
```

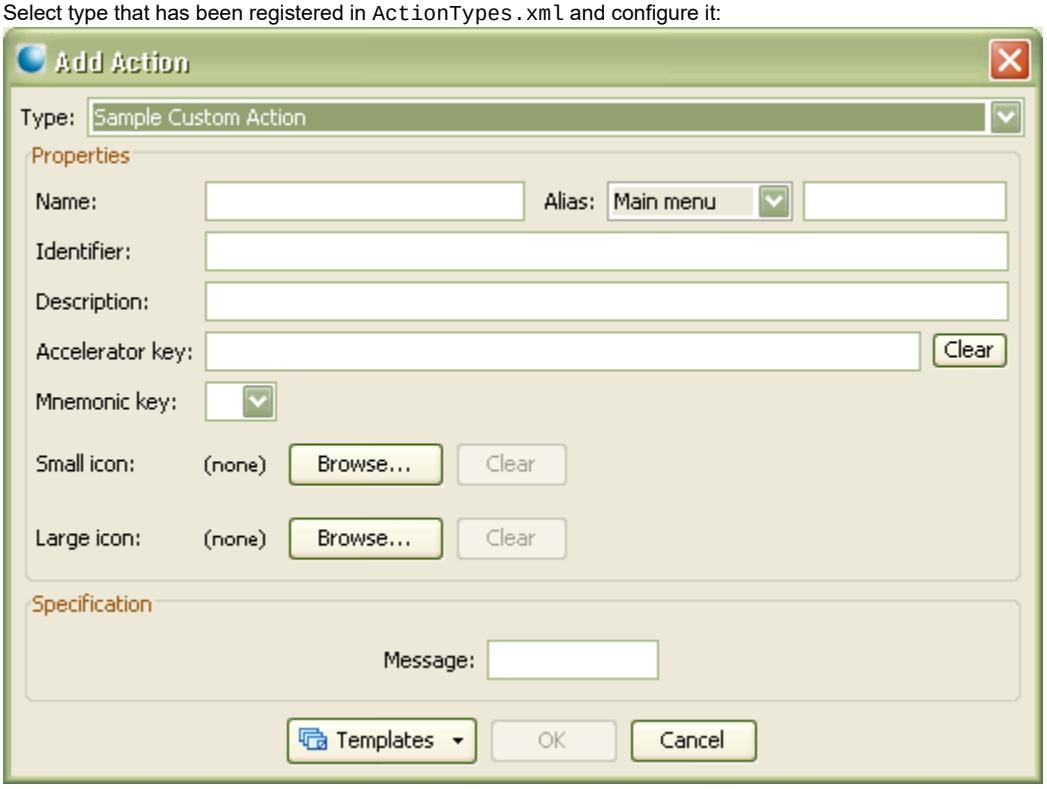
Registering action spec converter

Add to `directory:///ElTrader/TMS/Config/Actions/ActionSpecConverters.xml` an entry like:

```
<converter className="action.HFSampleCustomActionSpecConverter"/>
```

Creating new action of new type

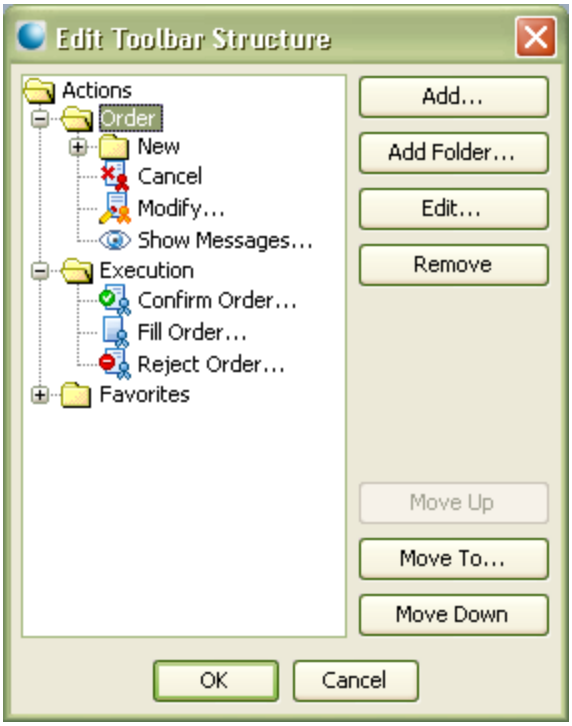
1. Go to `Tools` -> `Actions...`
2. Select `Custom Actions` tab
3. Press `Add...`
4. Select type that has been registered in `ActionTypes.xml` and configure it:



Adding actions to GUI

GUI actions can be added to a variety of places in InfoReach GUI: window toobars, report toolbars etc.
Here is an example of adding GUI action to window toobar:

1. Go to `View` -> `Toolbars` -> `Customize...`
2. Select a toolbar to contain new action button and press `Add...`



3. Select `Custom Actions` tab
4. Select previously created action
5. New button should appear on the toolbar

Executing custom code on backend from GUI actions

Simple remote actions

Overview of creating simple remote actions

These steps are required:

1. Configure a remote action manager in GUI app.
2. Instantiate corresponding action manager in backend app.
3. Wrap custom code to be executed on backend into `IRemoteAction` implementation.
4. In backend app register created remote action within action manager.
5. Create GUI action of Execute Remote Action type.

Configure a remote action manager in GUI app

Add `<Manager name="<arbitrary name here>"/>` to `<RemoteActionManagers>` section in relevant GUI process' section, e.g.:

directory://ElTrader/TMS/Config/processes.xml

```
<ELTHiFreqView name="ELTHiFreqView">
 ...
 <ViewSystem
  ...
 >
  <RemoteActionManagers>
   <!-- the name is to be used in code on next step -->
   <Manager name="HiFreqActionManager"/>
  </RemoteActionManagers>
```

Instantiate corresponding action manager in backend app

Have this code executed in backend app:

> ⚠ It's a good idea to place the code in a method overriding `HFAbstractBaseApp.initializeAfterLogin()`.

```
// manager name should be exactly the same as in GUI app configuration
RemoteActionManager remoteActionManager = new RemoteActionManager("HiFreqActionManager");
```

Wrap custom code to be executed on backend into IRemoteAction implementation

Sample `IRemoteAction` implementation:

```
public class HFSampleRemoteAction implements IRemoteAction<IRemoteActionSpec, Void>
{
   @Override
   public Void perform(IRemoteActionSpec actionSpec) throws Exception
   {
      System.out.println("### remote action " + toString() + " was invoked: \n"
            + "\t action class: " +  getClass().getName() + "\n"
            + "\t action spec class: " + actionSpec.getClass().getName() + "\n"
            + "\t action spec: " + actionSpec);

      if (!(actionSpec instanceof IRecordsRemoteActionSpec))
      {
         throw new Exception("Misconfiguration: action " + getClass()
               + " doesn't support custom action spec types, but "
               + actionSpec.getClass().getName()
               + " was passed to it.");
      }

      IRecordsRemoteActionSpec recordsSpec = (IRecordsRemoteActionSpec) actionSpec;

      doSomething(
            recordsSpec.getParameters(),
            recordsSpec.getSelectedRecords(),
            recordsSpec.getSelectedViewName(),
            recordsSpec.getSelectedLevelName());
      return null;
   }

   private void doSomething(
         Configurator parameters,
         IRecord[] selectedRecords,
         String selectedViewName,
         String selectedLevelName)
   {
      String dataStr = "";

      // List selected records
      dataStr += "\t selected records: ";
      if (selectedRecords == null || selectedRecords.length == 0)
      {
         dataStr += " (none)";
      }
      else
      {
         dataStr += "\n";
         // accessing record's fields
         final int clOrdIdFieldIndex = selectedRecords[0].getFieldIndex("ClientOrdId");
         final int managerIdFieldIndex = selectedRecords[0].getFieldIndex("ManagerId");

         for (int i = 0; i < selectedRecords.length; i++)
```

```
    {
        if (i != 0)
        {
            dataStr += "\n";
        }
        dataStr += "\t\t";
        dataStr += selectedRecords[i].getIdentifier();

        // append ClientOrdId and ManagerId (if present)
        boolean recordHasClientOrdId = clOrdIdFieldIndex >= 0;
        boolean recordHasManagerId = managerIdFieldIndex >= 0;
        if (recordHasClientOrdId || recordHasManagerId)
        {
            dataStr += " (";
            if (recordHasClientOrdId)
            {
                // accessing record's ClientOrdId string field
                dataStr += "ClientOrdId="
                    + selectedRecords[i].getStringFieldValueAt(clOrdIdFieldIndex);
            }
            if (recordHasManagerId)
            {
                if (recordHasClientOrdId)
                {
                    dataStr += ", ";
                }
                // accessing record's ManagerId numeric field
                dataStr += "ManagerId="
                    + (int)selectedRecords[i].getNumericFieldValueAt(managerIdFieldIndex);
            }
            dataStr += ")";
        }
    }
}

dataStr += "\n";

// List Configurator params
dataStr += "\t action parameters:";
if (parameters.size() == 0)
{
    dataStr += " (none)";
}
else
{
    dataStr += "\n\t\t";
    Set<Map.Entry> entrySet = parameters.entrySet();
    int entryIndex = 1;
    for (Map.Entry e : entrySet)
    {
        dataStr += "key" + entryIndex + ": '" +  e.getKey()
            + "', value" + entryIndex + ": '" +  e.getValue() + "'\n";
        entryIndex++;
    }
}

System.out.println("### remote action " + toString()
        + " received the following data from GUI:\n" + dataStr);
```

```
    }
}
```

In backend app register created remote action within action manager

Have this code executed in backend app:

> ⚠ It's a good idea to place the code in a method overriding `HFAbstractBaseApp.initializeAfterLogin()`.

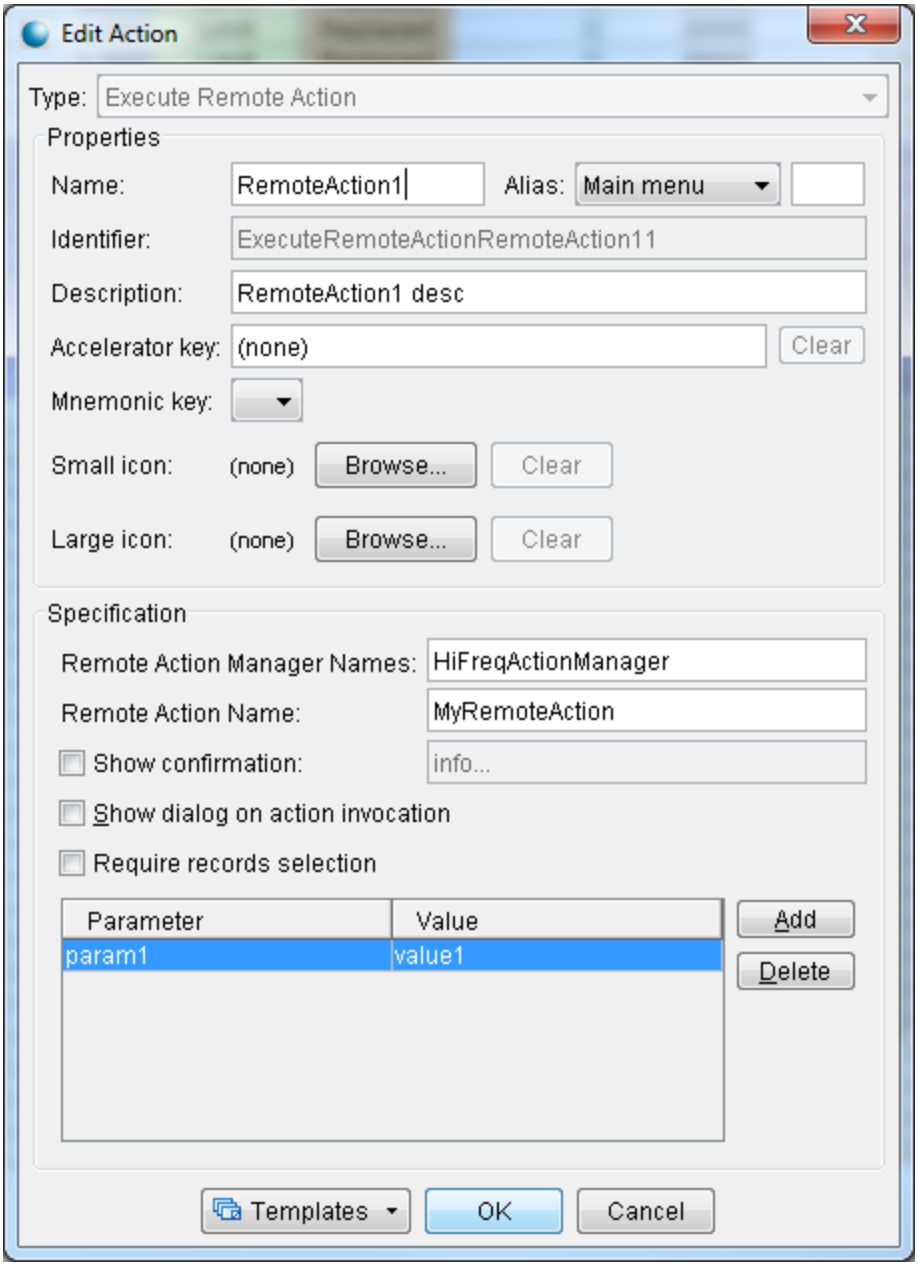
```
remoteActionManager.registerAction(
        "MyRemoteAction",
        new HFSampleRemoteAction());
remoteActionManager.registerAction(
        "MyRemoteActionAdvanced",
        new HFSampleRemoteActionAdvanced());
```

Create GUI action of Execute Remote Action type

1. Go to `Tools` -> `Actions...`.
2. Select `Custom Actions` tab.
3. Press `Add...`.
4. Select `Execute Remote Action` type and configure it to point to previously created remote action manager and registered `IRemoteAction` implementation:

> ⓘ Selected Records
> If records need to be passed to remote action handler in a spec, "Require records selection" must be checked.

> ⓘ Scalability note
> Because for scalability purposes more than one action manager can be registered (e.g. one per process). If you want to call the same action on multiple managers from GUI, you should specify their names, not just one name. Separate names with comma



Advanced remote actions

Overview of creating advanced remote actions

Compared to simple remote actions one has to:

1. Create a `Serializable` spec implementing `IRemoteActionSpec` to hold action's data to be passed from GUI to backend.
2. Create an editor for the spec by implementing `IRemoteActionSpecEditor`.
3. Wrap your backend code using custom data into `IRemoteAction` implementation.
4. Create GUI action of `Execute Remote Action Advanced` type (instead of `Execute Remote Action`)

Implementing IRemoteActionSpec to hold action's data

Sample `IRemoteActionSpec` implementation:

```
public class HFSampleRemoteActionAdvancedSpec implements IRemoteActionSpec
{
   private static final long serialVersionUID = 7633451173370488953L;

   private final String data_;
 private final IRecord[] selectedRecords_;

   public HFSampleRemoteActionAdvancedSpec(String data, IRecord[] selectedRecords)
   {
      data_ = data;
      selectedRecords_ = selectedRecords;
   }

   public String getStringData()
   {
      return data_;
   }

   public  IRecord[] getSelectedRecords()
   {
    return selectedRecords_;
   }

   @Override
   public String toString()
   {
      return getClass().getSimpleName()
       + "[string data: '" + data_
       + "'; selected records: "
        + (selectedRecords_ == null ? "(none)" : Arrays.toString(selectedRecords_)) + "]";
   }
}
```

Implementing IRemoteActionSpecEditor to edit action's data

> Selected Records
> Selected records (if any) are always passed to the editor.

Sample `IRemoteActionSpecEditor` implementation:

```
public class HFSampleRemoteActionSpecEditor implements IRemoteActionSpecEditor
{
   @Override
   public IRemoteActionSpec createSpec(
         IActionSpecEditorDataProvider actionSpecEditorDataProvider,
         Configurator parameters)
   {
      Frame parentFrame = actionSpecEditorDataProvider.getParentFrame();
      String input = JOptionPane.showInputDialog(
            parentFrame,
            "Please input data for remote action");

      IRecord[] selectedRecords = actionSpecEditorDataProvider.getSelectedRecords();

      return new HFSampleRemoteActionAdvancedSpec(input,
        selectedRecords); // pass currently selected records
   }
}
```

Wrap your backend code using custom data into IRemoteAction implementation

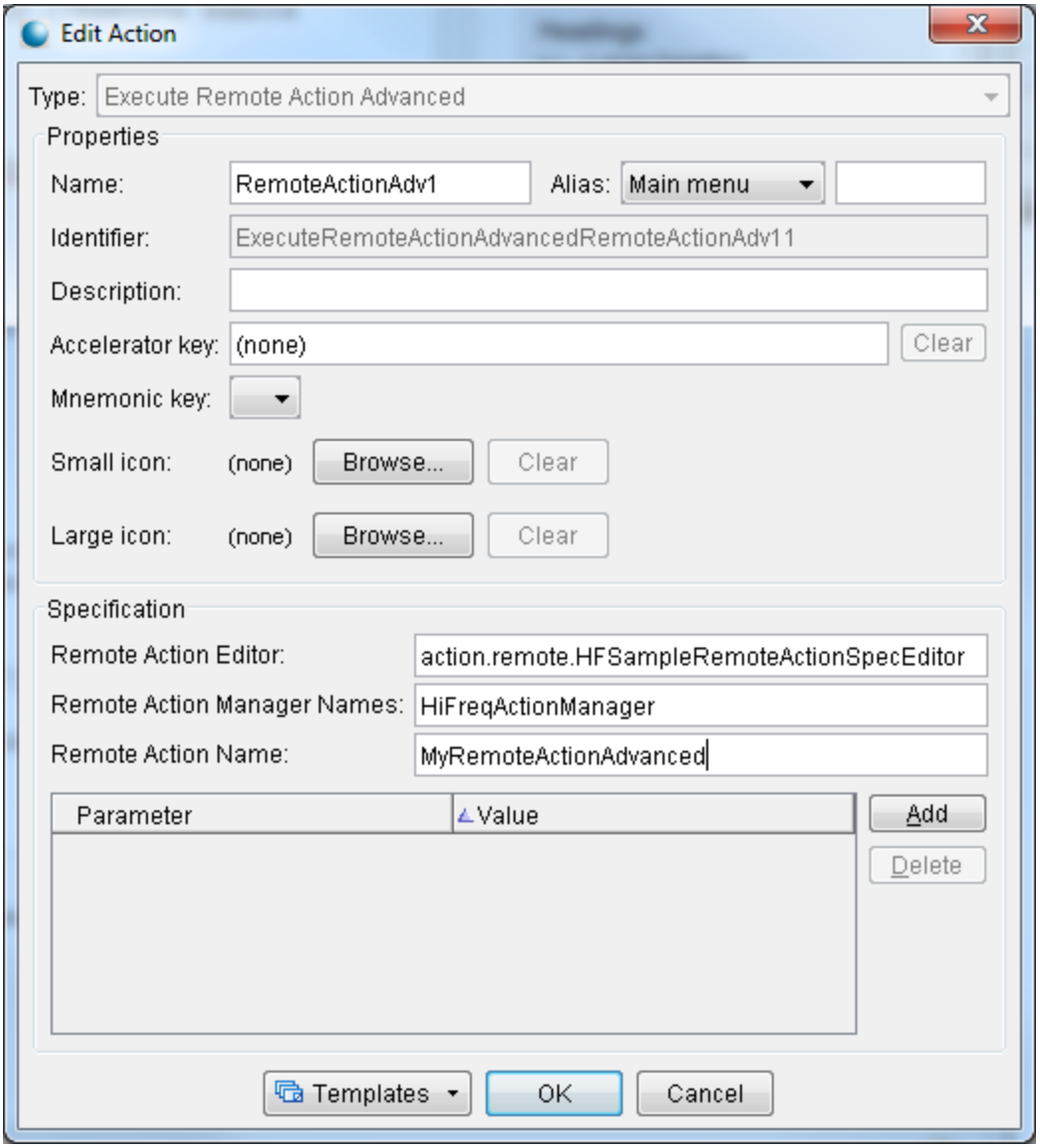
Sample `IRemoteAction` implementation for advanced remote action:

```
public class HFSampleRemoteActionAdvanced implements IRemoteAction<IRemoteActionSpec, Void>
{
   @Override
   public Void perform(IRemoteActionSpec actionSpec) throws Exception
   {
      System.out.println(
            "### remote action " + toString() + " was invoked: \n"
            + "\t action class: " +  getClass().getName() + "\n"
            + "\t action spec class: " + actionSpec.getClass().getName()
            + "\n"
            + "\t action spec: " + actionSpec);

      if (!(actionSpec instanceof HFSampleRemoteActionAdvancedSpec))
      {
         throw new Exception("Misconfiguration: action " + getClass()
               + " only supports "
               + HFSampleRemoteActionAdvancedSpec.class.getName()
               + " action spec type, but "
               + actionSpec.getClass().getName()
               + " was passed to it.");
      }

      HFSampleRemoteActionAdvancedSpec castedSpec =
         (HFSampleRemoteActionAdvancedSpec) actionSpec;

      doSomething(castedSpec.getStringData(), castedSpec.getSelectedRecords());

      return null;
   }

   private void doSomething(String data, IRecord[] selectedRecords)
   {
      System.out.println("### remote action " + toString()
            + " received from GUI:\n"
            + "\t string: '" + data + "'\n"
            + "\t selected records: "
             + (selectedRecords == null ? "(none)" : Arrays.toString(selectedRecords)));
   }
}
```

Create GUI action of Execute Remote Action Advanced type

1. Go to `Tools` -> `Actions...`.
2. Select `Custom Actions` tab.
3. Press `Add...`.
4. Select `Execute Remote Action Advanced` type and configure it to point to previously created remote action manager, editor and registered `IRemoteAction` implementation:

> Scalability note
> Because for scalability purposes more than one action manager can be registered (e.g. one per process). If you want to call the same action on multiple managers from GUI, you should specify their names, not just one name.



Report subscription API

Report subscription API overview

Use methods of `RPTSystemUtils`:

- `subscribeForTableReportData(UserTicket, String, String, String, IRPTReconnectableEventListener)`
- `subscribeForTableReportData(UserTicket, String, String, String, Object[], IRPTReconnectableEventListener)`
- `subscribeForTreeReportData(UserTicket, String, String, String, String, IRPTReconnectableEventListener)`
- `subscribeForTreeReportData(UserTicket, String, String, String, String, Object[], IRPTReconnectableEventListener)`

`IRPTReconnectableEventListener` implementations are recommended to extend `RPTAbstractRecordEventListener`.

Report Subscription API code sample

- Subscribing to report:

```
// defaultManagerName attribute value from RPTManagerList.xml
String reportManagerName = "HiFreq report manager" ;
String tableReportName = "System Usage Report"; // available in any report manager
String filter = "Action = 'Registered'";

// subscribe for table report data
ReportDataListener tableReportDataListener = new ReportDataListener(
      "Table report listener: "+ reportManagerName+":"+tableReportName+":"+filter,
      true);

System.out.println("### Subcribing for '" + tableReportName + "' table report"
      + " on '" + reportManagerName + "' report manager"
      + " with filter='" + filter + "'...");
RPTSystemUtils.subscribeForTableReportData(
      GlobalSystem.getUserTicket(),
      reportManagerName,
      tableReportName,
      filter,
      tableReportDataListener);
```

- Extending `RPTAbstractRecordEventListener`:

```
private static class ReportDataListener extends RPTAbstractRecordEventListener
{
   private String name_;
   private boolean debugLevelMax_;

   public ReportDataListener(String name, boolean debugLevelMax)
   {
      name_ = name;
      debugLevelMax_ = debugLevelMax;
   }

   public void onReportDataFeedDisconnected()
   {
      System.out.println("### (" + name_ + ") report data feed disconnected" + "\n");
   }

   public void onReportDataFeedReconnected()
   {
      System.out.println("### (" + name_ + ") report data feed reconnected" + "\n");
   }

   public void onError(RPTException ex)
   {
      System.out.println("### (" + name_ + ") report data listener received an error");
      ex.printStackTrace();
   }

   protected void processRecord(IRPTRecord record, int action)
   {
      StringBuilder buf = new StringBuilder();
      buf.append("### (").append(name_).append(") processing record ");
      switch (action)
      {
         case RPTRecordEvent.EVENT_RECORD_CHANGED:
            buf.append("changed");
            break;
         case RPTRecordEvent.EVENT_RECORD_INSERTED:
            buf.append("inserted");
            break;
         case RPTRecordEvent.EVENT_RECORD_REMOVED:
            buf.append("removed");
            break;
```

```
    }
    buf.append(" ID=").append(record.getIdentifier());

    // Extract information from the record
    if (debugLevelMax_)
        buf.append(", ").append(recordToString((IRecord) record));
    buf.append("\n");
    System.out.println(buf);
}

protected void processRecordUpdate(IRPTRecordUpdate recordUpdate)
{
    StringBuilder buf = new StringBuilder();
    buf.append("### (").append(name_).append(")");
    buf.append(" processing recordUpdate ID=" + recordUpdate.getIdentifier());
    if (debugLevelMax_)
        buf.append(", ").append(recordUpdateToString((IRecordUpdate) recordUpdate));
    buf.append("\n");
    System.out.println(buf);
}

protected void onInitialStateReceived()
{
    System.out.println("### (" + name_ + ") initial state received" + "\n");
```

```
    }
}
```

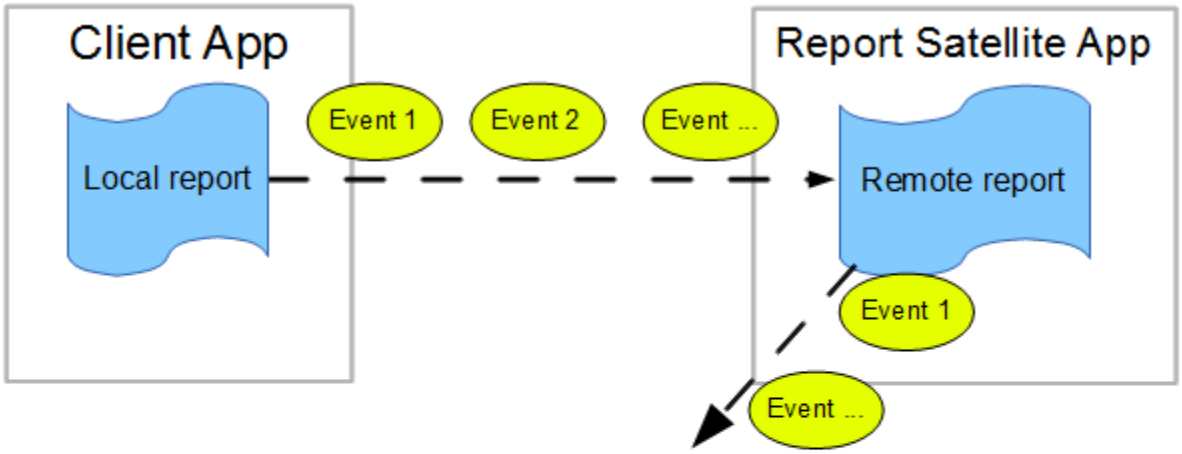
Push report data API

Out-of-process

Out-of-process - use case

- Client's model generates some data that has to be transformed and presented via Reporting Framework.
- We want to avoid all complexity of having custom report implementation or anything like it.
- Client's application only generates data to be presented in report, but does not host it.
    - Client's application can be shut down at any moment - but the report should be always available.
    - Client's application is able to connect to (externally hosted) report and somehow update data that report holds (i.e. report data should not be reset on connection to it).

Out-of-process - general architecture



There are two reports: "Local" (data provider) and "Distributed"(get data from provider and stores it). "Local" is created on start-up in client's application implemented as ELTClientApp TMS process type. "Distributed" report is loaded on start-up in ELTUserReportsSatellite TMS process. When client app creates report adapter with ITMSReportAdapter,"Distributed" report is automatically connected as listener to "Local" report. "Distributed" report content could be optionally cleared here. Then all data additions/updates/removals are automatically propagated to "Distributed" report. After finishing pushing needed data the client app disconnects reports (so "Distributed" report doesn't listen for "Local" one any more) via ITMSReportAdapter API call and can exit.

Out-of-process - caveats

- A problem can occur on client app exit. Some events could be stored to internal TMS EventService queues but not sent yet when the client app exists. There is maxWaitToEmptyQueuesOnExit parameter in ELTClientAppEventBroker (in EventService.xml) that could be used as protection. If this parameter is set then EventService will wait for configured timeout or until EventService queues are empty (whatever happens first). uninitializeTimeout attribute of GlobalSystem element in process.xml should be potentially adjusted accordingly.
- The same XML specification can't be used for both reports as it holds report's name, but reports obviously need to have different names.

- Both reports should have exactly the same report structures.

Out-of-process - configuration

- prepare custom metadata for reports if needed and register in MetaDataList.xml
- prepare local and distributed report specifications

FAQSamples/src/PushDataToOutOfProcessReport.SampleReportSpec.LocalReport.xml

```
<Report>
 <ReportSpecification name="PushToOutOfProcessReportSample.LocalReport" type="Chain Report" >
  <ReportElement tableMetaDataName="Portfolio table report fields" type="Copying Table" >
   <Field id="Instrument"/>
   <Field id="TgtQty"/>
   <Field id="Px"/>
  </ReportElement>
  <ReportElement eventCount="500" timeout="1000" type="Buffered Records" />
 </ReportSpecification>
</Report>
```

FAQSamples/src/PushDataToOutOfProcessReport.SampleReportSpec.RemoteReport.xml

```
<Report>
 <!--
 clearStateOnDisconnect="false" is required here
 to avoid flushing this report's data on counterpart
 report disconnection
 -->
 <ReportSpecification name="PushToOutOfProcessReportSample.RemoteReport" type="Chain Report" >
  <ReportElement tableMetaDataName="Portfolio table report fields" type="Copying Table" >
   <Field id="Instrument"/>
   <Field id="TgtQty"/>
   <Field id="Px"/>
  </ReportElement>
  <ReportElement eventCount="500" timeout="1000" type="Buffered Records" />
 </ReportSpecification>
</Report>
```

- in tms.xml configure reports to be loaded on startup
  - local report in ELTClientApp process

```
<ReportsToLoadOnStartupList>
    <ReportReference
        managerNames="Portfolio report manager%processName%"

resourceName="../java/FAQSamples/src/PushDataToOutOfProcessReport.SampleReportSpec.LocalReport.xml"
    />
</ReportsToLoadOnStartupList>
```

  - distributed report in ELTUserReportSatellite

```
<ReportsToLoadOnStartupList>
...
   <ReportReference
      managerNames="Portfolio report manager%processName%"

resourceName="../java/FAQSamples/src/PushDataToOutOfProcessReport.SampleReportSpec.RemoteReport.xml"
   />
</ReportsToLoadOnStartupList>
```

- Adjust RPTManagerList.xml.

  ⚠ The sample below is given for applications running as ELTClientApp.
  For applications running as ELTHiFreqApp `<ELTHiFreqApp>...</ELTHiFreqApp>` xml section should be adjusted instead.
  To find out whether some application is ELTClientApp or ELTHiFreqApp check its run script.
  E.g. `samples\ELTClientApp.pl` has

  ```
  _ARGUMENTS() => [
        "-type"       => "ELTClientApp",
        "-name"       => $ARGV[0],
        "-bootstrapDataFile"  => "../cfg/BackendBootstrap.xml"
     ]
  ```

  which tell us it is ELTClientApp (see second line).

  In order to be able host local report in ELTClientApp in RPTManagerList.xml change
  from

  ```
  <ELTClientApp>
   <ReportManagerList defaultManagerName="Portfolio report manager">
      ...
   <ReportManager
    name="Portfolio report manager"
    type="domainManagerProxy"
   />
  ```

  to

  ```
  <ELTClientApp>
   <ReportManagerList defaultManagerName="Portfolio report manager">
      ...
   <ReportManager
    name="Portfolio report manager"
    type="domainManagerAssistant"
    cannotCreateReports="true"
   />
  ```

Out-of-process - Java

Out-of-process - Java API

The central part of Push report data API is com.inforeach.eltrader.tms.domain.portfolio.api.ITMSReportAdapter.

```
/**
 * Facade interface for pushing data to any reporting
 */
public interface ITMSReportAdapter
{
    /**
     * Connects report adapter to remote distributed report
     * @param clearExistingData if true that data in distributed report
     *         will be cleard after connect
     * @throws RPTException
     */
    public void connect(boolean clearExistingData) throws RPTException;

    /**
     * Disconnects report adapter from remote distributed report
     * @throws RPTException
     */
    public void disconnect() throws RPTException;

    /**
     * Adds record to report.
     * @see TMSUtils.createRecord()
     * @param record
     */
    public void addRecord(IRecord record);

    /**
     * Updates record in report.
     * @see TMSUtils.createRecordUpdate()
     * @param recordUpdate
     */
    public void updateRecord(IRecordUpdate recordUpdate);

    /**
     * Removes record with specified ID from report
     * @param recordId
     */
    public void removeRecord(Object recordId);
}
```

To get instance of adapter use method of ITMSRemoteClient:

```
/**
* Returns report adapter for pushing data to report. Both local and remote (AKA distributed)
* reports should be pre-created on startup in corresponding processes.
*
* @param localReportName  name of report pre-created in client process (i.e. in-process)
* @param remoteReportName name of report pre-created on corresponding satellite
* @throws RPTException
*/
ITMSReportAdapter getReport(String localReportName, String remoteReportName) throws RPTException;
```

Utility methods for data structures creation are available in com.inforeach.eltrader.tms.api.TMSUtils

```
/**
 * Creates record context for given field names and types.
 * For types value see <code>com.inforeach.util.value.Value.type* </code> constatns
 * @param fieldNames array of field names in context
 * @param types array of field types in context
 * @return create context
 */
public static IRPTRecordContext createRecordContext(String[] fieldNames, int[] types)

/**
 * Creates record with specified field values.
 * @param ctx record context, should be created via <code>createRecordContext</code>, and then reused
 * @param strFieldNames array of string field names
 * @param strFieldValues array of string field values
 * @param numFieldNames array of numeric field names
 * @param numFieldValues array of numeric field values
 * @return created record
 */
public static IRecord createRecord(IRPTRecordContext ctx, String[] strFieldNames, String[] strFieldValues,
    String[] numFieldNames, double[] numFieldValues)

/**
 * Create update that could be applied to record
 * @param ctx record context, should be created via <code>createRecordContext</code>, and then reused
 * @param id identifiers of record to update
 * @param strFieldNames array of string field names
 * @param strFieldValues array of string field values
 * @param numFieldNames array of numeric field names
 * @param numFieldValues array of numeric field values
 * @return created update
 */
public static IRecordUpdate createRecordUpdate(IRPTRecordContext ctx, Object id,
    String[] strFieldNames, String[] strFieldValues,
    String[] numFieldNames, double[] numFieldValues)
```

Out-of-process - Java API usage sample

FAQSamples/src/PushDataToOutOfProcessReport.java

```
IRPTRecordContext ctx = TMSUtils.createRecordContext(
        new String[]{ // field names from report's metadata
                "Instrument",
                "TgtQty",
                "Px"},
        new int[]{ // field types from report's metadata
                Value.typeAlphaNumeric,
                Value.typeInteger,
                Value.typeDouble});
    ITMSReportAdapter reportAdapter = facade.getReport(// <-- the adapter is created
            "PushToOutOfProcessReportSample.LocalReport",
            "PushToOutOfProcessReportSample.RemoteReport");
    reportAdapter.connect(true /*clearExistingData=true*/);
    String[] strFields = new String[]{"Instrument"};
    String[] numFields = new String[]{"TgtQty", "Px"};

    IRecord[] records = new IRecord[] // to be pushed to report
    {
        TMSUtils.createRecord(ctx,
                strFields, new String[]{"IBM"},
                numFields, new double[]{1000,65.44}),
        TMSUtils.createRecord(ctx,
                strFields, new String[]{"MSFT"},
                numFields, new double[]{2000, 11.43}),
        TMSUtils.createRecord(ctx,
                strFields, new String[]{"ORCL"},
                numFields, new double[]{300,33.12}),
        TMSUtils.createRecord(ctx,
                strFields, new String[]{"YHOO"},
                numFields, new double[]{4500, 11.43})
    };

    // add records to report
    System.out.println("### Adding records to report...");
    for (int i = 0; i < records.length; i++)
    {
        System.err.println(recordToString(records[i]));
        reportAdapter.addRecord(records[i]);
    }

    Thread.sleep(3000); // to allow user notice changes on e.g. GUI

    // update records in report
    System.out.println("### Updating records in report...");
    for (int i = 0; i < records.length; i++)
    {
        IRecordUpdate update = TMSUtils.createRecordUpdate(
                ctx,
                records[i].getIdentifier(),
                EmptyArrays.STRING_ARRAY,
                EmptyArrays.STRING_ARRAY,
                numFields,
                new double[] {(i+1)*1000, (i+1)*10});
        reportAdapter.updateRecord(update);
    }

    Thread.sleep(3000); // to allow user notice changes on e.g. GUI

    // remove a record from report
    Object recordIDToRemove = records[3].getIdentifier();
    System.out.println("### Removing record " + recordIDToRemove + " from report...");
    reportAdapter.removeRecord(recordIDToRemove);

    Thread.sleep(3000); // to allow user notice changes on e.g. GUI

    System.out.println("### Disconnecting from report...");
    reportAdapter.disconnect ();
```

Out-of-process - C++

Out-of-process - C++ API

The Push report data API is in portfolio/ITMSRemoteClient.h.

```
class TMS_API ITMSRemoteClient : public virtual ITMSClient
{
...
    // ITMSReportAdapter
    /**
    * Returns report adapter for pushing data to report. Both local and remote (AKA distributed)
    * reports should be pre-created on startup in corresponding processes.
    *
    * @param localReportName  name of report pre-created in client process (i.e. in-process)
    * @param remoteReportName name of report pre-created on corresponding satellite
    * @throws RPTException
    */
    virtual UniLong getReport(const String &localReportName, const String &remoteReportName) throw(RPTException) = 0;

    /**
     * Connects report adapter to remote distributed report
     * @param clearExistingData if true that data in distributed report
     *        will be cleard after connect
     * @throws RPTException
     */
    virtual void connect(UniLong reportAdapterHandler, bool clearExistingData) throw(RPTException) = 0;

    /**
     * Disconnects report adapter from remote distributed report
     * @throws RPTException
     */
    virtual void disconnect(UniLong reportAdapterHandler) throw(RPTException) = 0;

    /**
     * Adds record to report.
     * @see TMSUtils.createRecord()
     * @param record
     */
    virtual void addRecord(UniLong reportAdapterHandler, IRecordPtr record) = 0;

    /**
     * Updates record in report.
     * @see TMSUtils.createRecordUpdate()
     * @param recordUpdate
     */
    virtual void updateRecord(UniLong reportAdapterHandler, IRecordUpdatePtr recordUpdate) = 0;

    /**
     * Removes record with specified ID from report
     * @param recordId
     */
    virtual void removeRecord(UniLong reportAdapterHandler, UniLong recordId) = 0;
}
```

Utility methods for data structures creation are available in the corresponding classes (IRecord and IRecordUpdate)

```
class TMS_API IRecord
{
...
   /**
   * Creates record with specified field values.
   * @param strFields array of string field names
   * @param strings array of string field values
   * @param numFields array of numeric field names
   * @param numbers array of numeric field values
   * @return created record
   */
   static void createRecord(IRecordPtr &irecord, UniLong id,
      StringVector strFields, StringVector strings,
      StringVector numFields, DoubleVector numbers);
}

class TMS_API IRecordUpdate
{
...
   /**
   * Create update that could be applied to record
   * @param id identifiers of record to update
   * @param context record context
   * @param stringUpdateFields array of string field names to update
   * @param stringUpdateValues array of string field values to update
   * @param numericUpdateFields array of numeric field names to update
   * @param numbericUpdateValues array of numeric field values to update
   * @return created update
   */
   static void createRecordUpdate(IRecordUpdatePtr &irecordUpdate,
      UniLong id, IRPTRecordContextPtr context,
      StringVector stringUpdateFields, StringVector stringUpdateValues,
      StringVector numericUpdateFields, DoubleVector numbericUpdateValues);
}
```

Out-of-process - C++ API usage sample

```
UniLong reportHandle = remoteClient->getReport("Custom Data (local)", "Custom Data");

remoteClient->connect(reportHandle, true);

StringVector strFields = arrayToStringVector(1, "Instrument");
StringVector numFields = arrayToStringVector(2, "Qty", "Px");

StringVector names = arrayToStringVector(3, "Instrument", "Qty", "Px");
IntVector types = arrayToIntVector(3, (int)Value::typeAlphaNumeric, (int)Value::typeDouble, (int)Value::typeDouble);
IRPTRecordContextPtr context(new RPTRecordContext(names, types));


const int recordsCount = 2;
IRecordPtr records[recordsCount];

StringVector instruments = arrayToStringVector(recordsCount, "IBM", "MSFT");

int i = 0;
for (i = 0; i < recordsCount; i++)
{
   const char *instrument = instruments[i].c_str();

   IRecordPtr record;
   IRecord::createRecord(record, i+1,
      strFields, arrayToStringVector(1, instrument),
      numFields, arrayToDoubleVector(2, (double)((i+1)*1200), (double)((i+1)*62.05)) );

   records[i] = record;
   remoteClient->addRecord(reportHandle, records[i]);
}

StringVector emptyArray;
for (i = 0; i < recordsCount; i++)
{
   IRecordUpdatePtr recordUpdate;
   IRecordUpdate::createRecordUpdate(recordUpdate, records[i]->getIdentifier(), context,
      emptyArray, emptyArray, numFields, arrayToDoubleVector(2, (double)((i+1)*1500), (double)((i+1)*77)) );

   remoteClient->updateRecord(reportHandle, recordUpdate);
}

remoteClient->removeRecord(reportHandle, records[0]->getIdentifier());

remoteClient->disconnect(reportHandle);
```

In-process

In-process - use case

ITMSReportAdapter provides a handy API to create report and push/update/remove data to/in/from it.

In-process - configuration

⚠ The sample below is given for applications running as ELTClientApp.
For applications running as ELTHiFreqApp `<ELTHiFreqApp>...</ELTHiFreqApp>` xml section should be adjusted instead.
To find out whether some application is ELTClientApp or ELTHiFreqApp check its run script.
E.g. `samples\ELTHiFreqTimerSampleApp.pl` has

```
_ARGUMENTS() => [
        "-type"       => "ELTHiFreqApp",
        "-name"       => "timerSample",
        "-bootstrapDataFile"  => "../cfg/BackendBootstrap.xml"
    ]
```

which tell us it is ELTHiFreqApp (see second line).

In order to be able host local report in ELTClientApp in RPTManagerList.xml change from

```
<ELTClientApp>
 <ReportManagerList defaultManagerName="Portfolio report manager">
    ...
 <ReportManager
  name="Portfolio report manager"
  type="domainManagerProxy"
 />
```

to

```
<ELTClientApp>
 <ReportManagerList defaultManagerName="Portfolio report manager">
    ...
 <ReportManager
  name="Portfolio report manager"
  type="domainManagerAssistant"
  cannotCreateReports="true"
 />
```

In-process - Java API usage sample

FAQSamples/src/PushDataToInProcessReport.java

```
final String pathToReport = "PushDataToInProcessReport.SampleReportSpec.xml";
// create report adapter
ITMSReportAdapter reportAdapter = TMSReportAdapter.createReportAdapter(
        pathToReport,
        "Portfolio report manager"); // defaultManagerName attribute value from RPTManagerList.xml

// create/prepare data structures
IRPTRecordContext ctx = TMSUtils.createRecordContext(new String[]{"Instrument", "TgtQty", "Px"},
    new int[]{Value.typeAlphaNumeric, Value.typeInteger, Value.typeDouble});
reportAdapter.connect(true);
String[] strFields = new String[]{"Instrument"};
String[] numFields = new String[]{"TgtQty", "Px"};

IRecord[] records = new IRecord[]
{
    TMSUtils.createRecord(ctx, strFields, new String[]{"IBM"}, numFields, new double[]{1000,65.44}),
    TMSUtils.createRecord(ctx, strFields, new String[]{"MSFT"}, numFields, new double[]{2000, 11.43}),
    TMSUtils.createRecord(ctx, strFields, new String[]{"ORCL"}, numFields, new double[]{300,33.12}),
    TMSUtils.createRecord(ctx, strFields, new String[]{"MSFT"}, numFields, new double[]{4500, 11.43})
};

// add records to report
for (int i = 0; i < records.length; i++)
{
    reportAdapter.addRecord(records[i]);
}

// update records in report
for (int i = 0; i < records.length; i++)
{
    IRecordUpdate update = TMSUtils.createRecordUpdate(
            ctx,
            records[i].getIdentifier(),
            EmptyArrays.STRING_ARRAY,
            EmptyArrays.STRING_ARRAY,
            numFields,
            new double[] {(i+1)*1000, (i+1)*10});
    reportAdapter.updateRecord(update);
}

// remove a record from report
reportAdapter.removeRecord(records[3].getIdentifier());
```

Referenced report spec:

FAQSamples/src/PushDataToInProcessReport.SampleReportSpec.xml

```
<Report>
 <ReportSpecification name="TestReport" type="Chain Report" >
  <ReportElement tableMetaDataName="Portfolio table report fields" type="Copying Table" >
   <Field id="Instrument"/>
   <Field id="TgtQty"/>
   <Field id="Px"/>
  </ReportElement>
  <ReportElement eventCount="500" timeout="1000" type="Buffered Records" />
 </ReportSpecification>
</Report>
```

Router components

One To Many Target Routing Component

com.inforeach.eltrader.tms.hifreq.localapi.component.HFRTOneToManyTargetRouterComponent implements one-to-many routing use case.
It handles inbound trade request FIX messages, converts them into targets, or updates existing targets.
Application using this component decides how and when to slice targets into outbound orders.
The component handles FIX report messages incoming from an outbound connection - it routes them back to a respective inbound connection.

Supported services

- Maintains relationships between inbound order, target, outbound orders
- Encapsulated synchronization model
- Notifies client application at points of interest
- Recovers and routes whatever was unprocessed

Synchronization

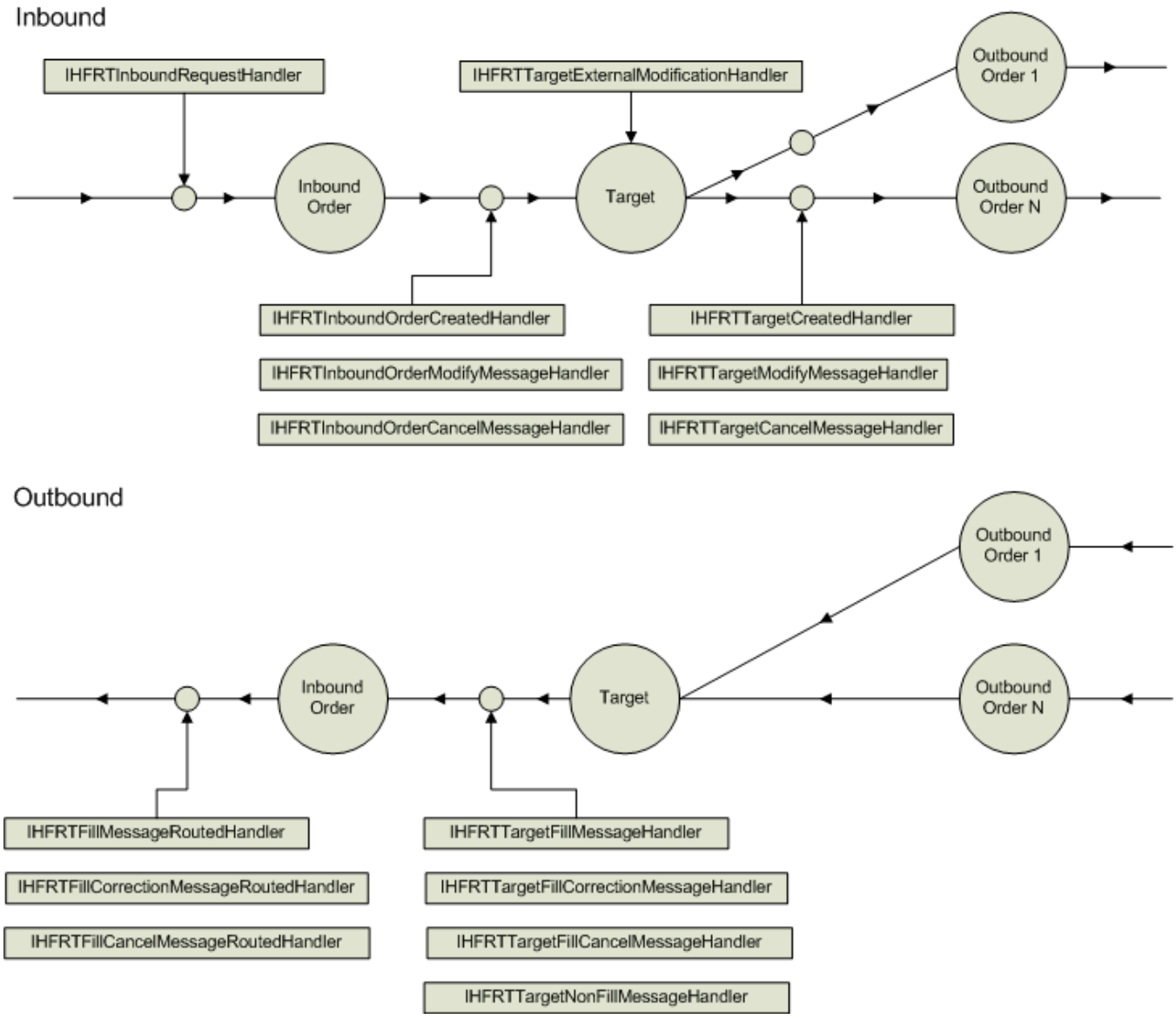
Component takes care about synchronization. It keeps related inbound order, target and outbound orders synchronized by the same object.
Hence, all notification on*() methods are called when lock is already obtained, and it is safe to perform any actions relevant to the parameters passed into on*() methods.

In case the application initiates an action (e.g. it needs to modify a target upon some event) outside of the scope of on*() methods, it must obtain a proper lock using getLockObjectFor*() API.

```
synchronized(routerComponent_.getLockObjectForTarget(target))
{
    IHFLocalClient localClient = routerComponent_.getLocalClient();
    HFLocalTargetFieldsContainerById targetChanges = new HFLocalTargetFieldsContainerById();
    targetChanges.addText("New target text");
    localClient.modifyTarget(target, targetChanges);
}
```

Notification API

See com.inforeach.eltrader.tms.hifreq.localapi.component.router.handler for all supported handlers.
Implement desired handlers and pass imlementation(s) into routerComponent.setHandler(...) method.
Note that when notification method is called, respective inbound order, target and outbound orders are locked, see Synchronization



Sample

See routing.HFTargetRouterComponentApp in the project \backend\samples\java\ELTHiFreqAppLocal\routing\
It receives targets and slices them into outbound orders.
Samples describes how to use/run samples.

DMA Routing Component

com.inforeach.eltrader.tms.hifreq.localapi.component.router.dma.HFRTDMALocalRouterComponent implements one-to-one routing use case. Inbound trade request FIX messages are received and an application specific implementation of IHFRTDMALocalInboundProcessor decides where the request should be routed.
The component handles incoming outbound FIX report messages - routes them back to inbound connection.

Supported services

- Maintains relationships between inbound and outbound orders
- Encapsulated synchronization model
- Notifies client application at points of interest
- Recovers and routes whatever was unprocessed

Synchronization

Component takes care of synchronization. It keeps inbound order and outbound orders synchronized using the inbound order.
All notification on*() methods are called when lock is obtained. So it is safe to perform any actions related to notification method parameters.

If the application wants to use a different lock object it must extend HFRTDMALocalRouterComponent and implement the createLockProvider() the method.
The default lock provider is implemented as follows:

```
protected static class LockProvider implements IHFLocalClientLockProvider
{
   @Override
   public Object getLockObject(final IHFLocalOrder order)
   {
      if (order.isInbound())
         return order;
      //inbound order is an extenstion object in outbound order
      final Object lockObject = order.getExtensionObject();
      // inbound order is purged, in this case we use outbound order as lock object.
      return (null == lockObject) ? order : lockObject;
   }

   @Override
   public Object getLockObject(final IHFLocalTarget target)
   { return target; }

   @Override
   public Object getLockObject(final IHFLocalTargetList targetList)
   { return targetList; }

   @Override
   public Object getLockObject(final IHFLocalStrategyData strategyData)
   { return strategyData; }
}
```

Notification API

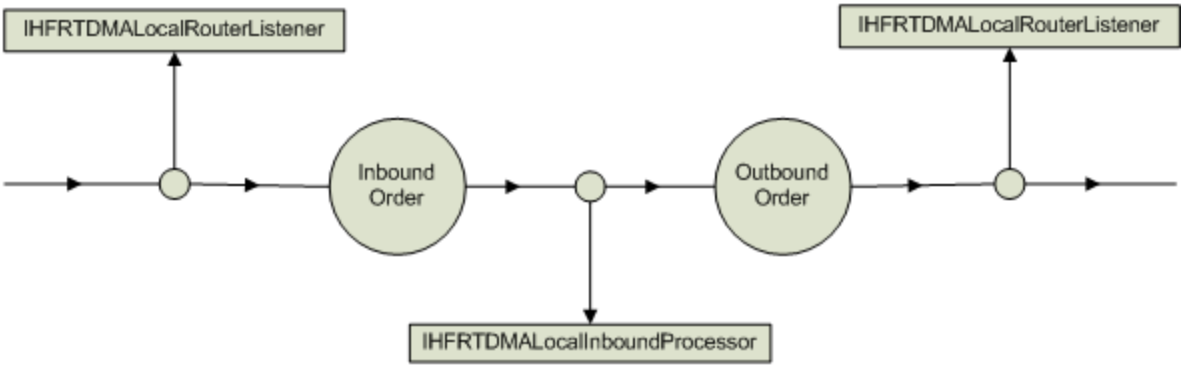
There are three notification handlers that may be implemented by the application

- com.inforeach.eltrader.tms.hifreq.localapi.component.router.dma.IHFRTDMALocalInboundProcessor
- com.inforeach.eltrader.tms.hifreq.localapi.component.router.dma.IHFRTDMALocalOutboundProcessor
- com.inforeach.eltrader.tms.hifreq.localapi.component.router.dma.IHFRTDMALocalRouterListener

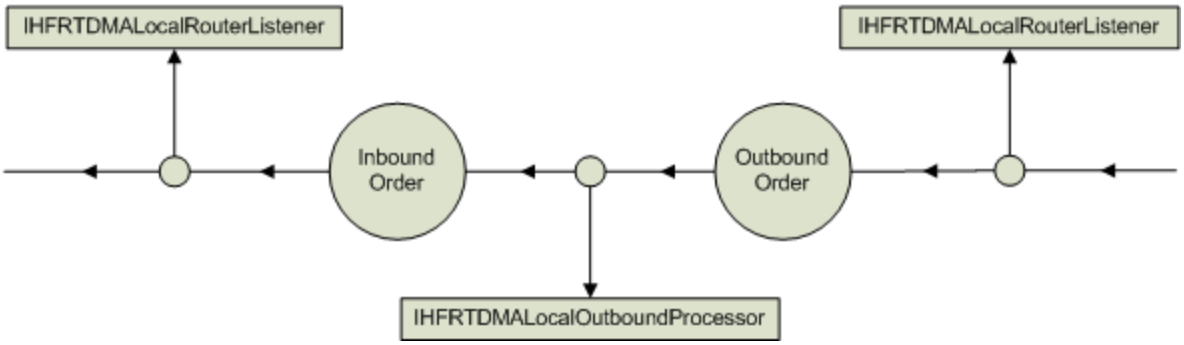
Only the IHFRTDMALocalInboundProcessor is mandatory and cannot be null.

Implement desired handlers and pass imlementation(s) into the appropriate HFRTDMALocalRouterComponent constructor.

Inbound



Outbound



Sample

See routing.HFDMARouterComponentApp in the project \backend\samples\java\ELTHiFreqAppLocal\routing\
It receives inbound order and routes them into outbound orders.
Samples describes how to use/run samples.

Miscellaneous tasks

Posting an alert to users' GUIs

It is possible for an external application to post an alert into users' GUIs.
Use methods of `com.inforeach.eltrader.tms.util.TMSUtilityFunctions`:

```
// - alert description a string that will appear at the alert window;
// - alert details is a string that will represent alert details available
//   through "React..." button at the alert window.
```

```
TMSUtilityFunctions.postAlertMessageToAllUserViews(
        ITMSConstants.ALERT_TYPE_WARNING,
        "alert description",
        "alert sent to views of all users",
        false);

TMSUtilityFunctions.postAlertMessageToUsersViews(
        new String[] {"demo"},
        ITMSConstants.ALERT_TYPE_WARNING,
        "alert description",
        "alert sent to views of 'demo' user",
        false);

TMSUtilityFunctions.postAlertMessageToUserGroupsViews(
        new String[] {"monitor","managers"},
        ITMSConstants.ALERT_TYPE_WARNING,
        "alert description",
        "alert sent to views of members of 'monitor' and/or 'managers' user groups",
        false);
```

General API notes

Default time zone

Applications using InfoReach API should not set default time zone.
I.e. `java.util.TimeZone.setDefault(TimeZone)` should not be invoked.
The default time zone should always be GMT (set internally).

Samples

Description

The samples are located at backend\samples\java\ELTHiFreqAppLocal
and organized into these folders:

- simple - simple strategies, e.g. order modification by timer or by market data updates
- targets - demonstrates usage of targets and target lists
- forex - forex trading samples
- routing - routing component samples
- server - default server application for remote API access.

Compilation

Sources are pre-built into backend\samples\java\ELTHiFreqAppLocal\ELTHiFreqAppLocal.jar
The samples can be launched as is, without a need for compilation.
If source code modifications need to be compiled, Eclipse or Ant can be used.

Eclipse

Eclipse can be used as a convenient and free IDE.
Importing and debugging samples in Eclipse is described in backend\samples\java\eclipse_readme.txt.
It is also recommended to read backend\samples\java\readme.txt.
In short, use File/Import.../General/Existing Projects into Workspace - and select backend\samples\java\ELTHiFreqAppLocal\ folder. It'll find sample projects.
Each project is configured to compile sources to backend\samples\java\ELTHiFreqAppLocal\<project>\classes folder.
This "classes" folder is included into sample application classpath before ELTHiFreqAppLocal.jar, so classes compiled with Eclipse take precedence over ELTHiFreqAppLocal.jar classes.

Ant

Ant build tool can be used as a building alternative.
Download Ant from http://ant.apache.org/
Set JAVA_HOME environment variable to path to JDK, run "ant" command in backend\samples\java\ELTHiFreqAppLocal folder.
It'll compile sources and update ELTHiFreqAppLocal.jar

Running

All logs are located in the logs folder.
The log file is named <prefix>_ELTHiFreqApp.log

Non-routing samples

These scripts to be used:

- backend\bin\Start_All_HiFreq.pl - starts all required processes.
  - Edit the file to have one of the local API samples uncommented. Or run a sample directly from backend\samples\java\ELTHiFreqAppLocal\<Chosen sample>\bin folder.
- backend\bin\admin\ELTAdmin_KillAllProcessesQuick.pl - kills all processes
- backend\bin\admin\ELTAdmin_DropDBTables.pl - drops all database tables
- frontend\bin\ELTHiFreqView.pl - runs HiFreq GUI
  - Login as "demohifreq"
  - Select a worskpace corresponding to a running sample.
- For development purposes the HiFreq sample application may need to be restarted frequently. In this case:
  - modify Start_AllHiFreq.pl to exclude a sample app - comment it with # at the start of the respective line.
  - make sure the sample's .pl file has _OPT_REDIRECT_OUTPUT() => "false",
  - run the sample from backend\samples\java\ELTHiFreqAppLocal\<Chosen sample>\bin
    Console will have the output.
  - use "exit" console command to exit from the running sample.

Routing sample

- It's run same way as all other samples, but also a client simulator process can be used to send inbound requests to the router.
  - See that Start_All_HiFreq.pl runs ELTHiFreqClientSimulatorApp.pl
  - Either in a console of that client simulator or with help of AppAdmin tool run a command "sto th 100" to send 100 test inbound orders.
  - If needed the client simulator sample code can be adjusted to satisfy custom requirements.

- frontend\bin\ELTHiFreqView.pl - runs HiFreq GUI
  - Login as "demohifreq"
  - Select a "Routing" workspace.

Development

The recommended way of development is to copy one of most relevant existing projects and customize it according to the desired use cases.
Note that there is a requirement: high frequency application class must extend HFAbstractBaseApp.

Debugging

Each sample is run with a debug port configured by default, see sample *pl file
_OPT_DEBUG_PORT_OFFSET() => 230, # real port by default is 9230
In Eclipse add Run/Debug Configurations/Remote Java Application/ and choose application host and port (9230)

Broker Simulator

In all the samples orders are sent to broker simulator.
Broker simulator understands the following instructions in Text field of execution request messages:

Command	
0	Request will be completely ignored by simulator. In this case subsequent requests most likely will be rejected due to unknown OrigClOrdID (since simulator just skipped previous request)
NOCONFIRM	D - leave order unacknowledged. F/G - cancel/replace order w/o confirming F/G
NOFILL	N/A, doesn't fill w/o explicit instruction anyway

REJECT	D - reject order, no further requests accepted. F/G - reject this cxl/cxr, further requests accepted, i.e. order stays alive
CANCEL	D - order is confirmed and then unsolicited cancel right away. G - request ignored, unsolicited cancel for D or previous G
FILL_N	if N<=10 then N is the number of fill reports to be sent back, in this case whole order quantity is evenly split between fills. Otherwise N is the number of shares to be filled in each fill report sent back. In both cases the order will be filled completely. Command can be submitted later in G.
PARTIAL_N_M	N, M - number of fill reports and number of shares to be filled with each execution report. If both specified - then smaller param is number of fills, another - number of shares. If only N specified (i.e. PARTIAL_N) - then single partial fill for N shares.

For example for order not to be filled use
newOrderMessage.setText("NOFILL");

Out-of-process API - Remote client

Out-of-process strategies use Remote client API to perform trading and control HiFREQ core remotely, as well as to subscribe to order/target changes, market data, custom data, position data, etc.

Base Terminology and Naming Convention

Remote Client

Provides access to the remote HiFreq application and is notified about application data events including order/target events, market data events, custom data events, position events, application status events.

Remote Client Listeners

Optional listeners for the notification about changes at the remote HiFreq application.

FIX engine connection

Physical FIX-based connection to or from a counter party.
Only one message at a time can be sent by a HiFreq application. Concurrency can be achieved by adding connections to the counter party. Same is true for receiving messages.

FIX message

The message which is sent through the FIX connection. Remote Client provides API for constructing/sending/processing received FIX messages.

Transaction destination

Transaction destination is an abstraction configured on top of a specific FIX connection.
A destination has a set of rules associated with it (configured in a destination-specific XML file) that get applied automatically by the HiFreq framework upon releasing FIX messages to a FIX connection. Multiple sets of validation or defaulting rules can be specified and can be nested. Boolean expression and all instructions have access to all FIX tags of the message and can read or modify the values.

More than one destination can be configured on top of a FIX connection, i.e. many-to-one relationship.

All API calls for sending FIX messages take destination as a parameter.

Outbound destination

Transaction destination for sending trade requests (orders) out and processing incoming trade reports (executions).

Inbound destination

Transaction destination for processing incoming trade requests (orders) and sending trade reports (executions) back.

Order manager

Order manager is a unit of the framework, residing on top of a transaction destination. Manager calculates and keeps order state.
Implements API for processing FIX messages received from the destination, or to be sent to the destination.
Manager works with a single transaction destination, more than one manager can be used for a destination. I.e. n..1 relationship

Outbound order manager

Order manager for sending trade requests out and processing incoming trade reports.

Inbound order manager

Order manager for processing incoming trade requests and sending trade reports back.

Order

Order record is a representation of a trading transaction. The HiFreq framework processes FIX messages and updates the order record state accordingly.

Target

Target record is a representation of a trading goal (parent order) for a given symbol. Each target is executed by sending one or more orders.
A target record is linked to its order records, and order records are linked to their targets.

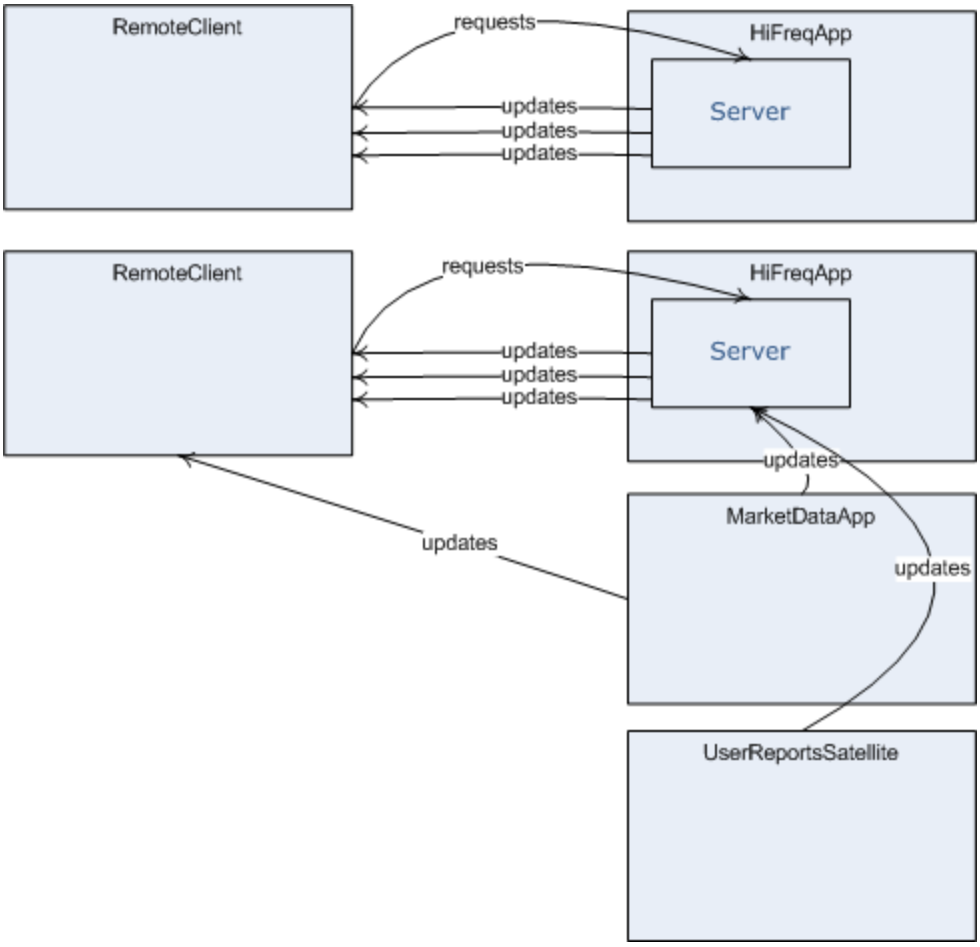
Target List

Represents a set of targets. It is used to unite targets into the logical units like baskets/portfolios, pairs, etc.

Strategy Data

Keeps data relevant to an application strategy. This data is displayed and is modifiable from GUI. Using relevant notifications the application strategy code can react to GUI user actions, like changing strategy parameters or starting/pausing a strategy.

Data and Thread Model



Data Model

Remote API is a thin facade layer on top of the remote HiFreq application.
This facade does not store data objects. It only passes data between a client application and the remote HiFreq application.
Because of that it is up to the client application how to organize and keep data.
It can utilize standard or custom collections, organize links of any kind and so on.

Remote client connects to a server and sends action requests.
The server notifies trading client about data changes with events.

- Remote server sees all data available in the deployment.
- N-1 relationship between clients and server. More than 1 client can connect to a single server.

Data exchange:

- Calls to the remote client.
  Whenever remote API accepts some data objects it does not modify them and releases them once call is completed.
  Client application is safe to reuse the object if needed.
  For example, the message object in _remoteClient.sendOrder(..., FieldsContainerByTag message) can be reused for multiple sending operations.
- Calls from the remote client.
  Whenever an object is received from the remote API it becomes the property of the client application. The remote API layer will not attempt to modify it.

Records

Most of the data objects are records. Orders, targets, target lists, strategy datas - all extend IRecord interface.
Updates to the data are delivered as IRecordUpdate's
The utility com.inforeach.util.PublicUtilities.applyUpdateToRecord(IRecord, IRecordUpdate) should be used to apply updates to the record.

Thread Model

HiFreq Remote client notifies about framework events on more than one thread.
Different type of events go through different channels and each channel uses its own thread.
Market data events can be configured to arrive directly from MarketDataApp.

Notifications

The remote client allows subscribing multiple listeners to different types of events.
Each listener interface has a corresponding adapter class with empty methods. It's recommended to use it when not all notifications are needed.

The entity records obtained with on...Added(...) methods can be stored if required.
In order to keep them up-to-date the on...Updated(...) updates should be applied to the stored records. See "Keeping stored records updated" example below.

Order, Target, Target List, Strategy Data listeners

See IHFRemoteClient.subscribeTo...(...) methods for the full list of possible listeners.
Corresponding _IHFRemoteClient.unsubscribeFrom...(...) are to be used for unsubscribing.
Some subscription methods take additional parameters to limit what data to subscribe for.

Examples

Order subscription:

```
private static class RemoteOrderListener extends HFRemoteOrderListenerAdapter
{
   @Override
   public void onOrderAdded(IHFRemoteOrder order)
   {
      LOGGER.debug("Order added, order = " + order);
   }
}

IHFRemoteOrderListener remoteOrderListener_ = new RemoteOrderListener();

private void subscribeToOrderData() throws HFException
{
   //Subscribe to "Buy" outbound orders only, for 2 fields only.
   remoteClient_.subscribeToOutboundOrderData(USER_ID, remoteOrderListener_, "Side = 'Buy'",
                                 new String[] {"Instrument", "TgtQty"});

   //Subscribe to "Sell" inbound orders only, for 2 fields only.
   remoteClient_.subscribeToInboundOrderData(USER_ID, remoteOrderListener_, "Side = 'Sell'",
                                 new String[] {"Instrument", "TgtQty"});

   //Subscribe to all outbound orders, all fields.
   remoteClient_.subscribeToOutboundOrderData(USER_ID, remoteOrderListener_);
}

private void unsubscribeFromOrderData() throws HFException
{
   remoteClient_.unsubscribeFromInboundOrderData(USER_ID, remoteOrderListener_);
   remoteClient_.unsubscribeFromOutboundOrderData(USER_ID, remoteOrderListener_);
}
```

Target subscription:

```
private static class RemoteTargetListener extends HFRemoteTargetListenerAdapter
{

   @Override
   public void onTargetAdded(IHFRemoteTarget target)
   {
      LOGGER.debug("Target added, target = " + target);
   }

   @Override
   public void onTargetPaused(long targetId)
   {
      LOGGER.debug("Target paused, targetId = " + targetId);
   }

   @Override
   public void onTargetResumed(long targetId)
   {
      LOGGER.debug("Target resumed, targetId = " + targetId);
   }

}

IHFRemoteTargetListener remoteTargetListener_ = new RemoteTargetListener();

private void subscribeToTargetData() throws HFException
{
   //Subscribe to "Buy" targets only, for 2 fields only.
   remoteClient_.subscribeToTargetData(USER_ID, remoteTargetListener_, "Side = 'Buy'",
                                       new String[] {"Instrument", "TgtQty"});

   //Subscribe to all targets, all fields.
   remoteClient_.subscribeToTargetData(USER_ID, remoteTargetListener_);
}

private void unsubscribeFromTargetData() throws HFException
{
   remoteClient_.unsubscribeFromTargetData(USER_ID, remoteTargetListener_);
}
```

Target list subscription:

```
private static class RemoteTargetListListener extends HFRemoteTargetListListenerAdapter
{
   @Override
   public void onTargetListAdded(IHFRemoteTargetList targetList)
   {
      LOGGER.debug("Target list added, targetList = " + targetList);
   }

   @Override
   public void onTargetListRemoved(String targetListName)
   {
      LOGGER.debug("Target list removed, targetListName = " + targetListName);
   }
}

IHFRemoteTargetListListener remoteTargetListListener_ = new RemoteTargetListListener();

private void subscribeToTargetListData() throws HFException
{
   //Subscribe to "MyList" target list, all fields.
   remoteClient_.subscribeToTargetListData(USER_ID, remoteTargetListListener_, true, "Name = 'MyList'", null);

   //Subscribe to all target lists, all fields.
   remoteClient_.subscribeToTargetListData(USER_ID, remoteTargetListListener_);
}

private void unsubscribeFromTargetListData() throws HFException
{
   remoteClient_.unsubscribeFromTargetListData(USER_ID, remoteTargetListListener_);
}
```

Strategy data subscription:

```
private static class RemoteStrategyDataListener extends HFRemoteStrategyDataListenerAdapter
{
   @Override
   public void onStrategyDataAdded(IHFRemoteStrategyData strategyData)
   {
      LOGGER.debug("Strategy data added, strategyData = " + strategyData);
   }

   @Override
   public void onStrategyDataUpdated(String strategyName, IRecordUpdate strategyDataUpdate)
   {
      LOGGER.debug("Strategy data updated, strategyDataName = " + strategyName + " recordUpdate = " + strategyDataUpdate);
   }
}

IHFRemoteStrategyDataListener remoteStrategyDataListener_ = new RemoteStrategyDataListener();

private void subscribeToStrategyData() throws HFException
{
   remoteClient_.subscribeToStrategyData(USER_ID, remoteStrategyDataListener_);
}

private void unsubscribeFromStrategyData() throws HFException
{
   remoteClient_.unsubscribeFromStrategyData(USER_ID, remoteStrategyDataListener_);
}
```

Keeping stored records updated:
TODO: link to snippet

Remote client status listener

This listener is used to notify the application about remote client's connectivity status as well as about statuses of the requests made through the remote client API.
See IHFRemoteClient.setStatusListener(IHFRemoteClientStatusListener). Note that IHFRemoteClientStatusListener extends IActionEventListener for the action(request) status notifications.
The corresponding adapter class is HFRemoteClientStatusListenerAdapter.
It is possible to have only a single status listener. It can be set to "null" in order to stop listening to any status events.

```
private static class RemoteClientStatusListener extends HFRemoteClientStatusListenerAdapter
{

   @Override
   public void onConnected()
   {
      LOGGER.debug("Remote client connected");
   }

   @Override
   public void onDisconnected()
   {
      LOGGER.debug("Remote client disconnected");
   }

   @Override
   public void onProgressReport(long actionId,
                                String actionDescription,
                                String statusMessage,
                                int currentStep,
                                int totalSteps,
                                Object extraInfo)
   {
      LOGGER.debug("Action progress report, actionId = " + actionId +                " statusMessage = " + statusMessage + " currentStep = " + currentStep + " totalSteps = " + totalSteps);
   }

   @Override
   public void onErrorOccurred(long actionId, String actionDescription, Throwable exception, Object extraInfo)
   {
      LOGGER.error("Action error occurred, actionId = " + actionId +
          " actionDescription = " + actionDescription, exception);
   }

   @Override
   public void onFinished(long actionId, String actionDescription, Object extraInfo)
   {
      LOGGER.debug("Action completed, actionId = " + actionId);
   }
}


private void setRemoteClientStatusListener()
{
   remoteClient_.setStatusListener(new RemoteClientStatusListener());
}

private void cleanRemoteClientStatusListener()
{
   remoteClient_.setStatusListener(null);
}
```

Order Operations

Order request operations

A remote application works with order-related API for sending/modifying/canceling orders.
Each method has a multi-entity version. E.g. sendOrders(...) does a job of multiple sendOrder(...) calls.
It is recommended to use a multi-entity method version whenever multiple entities are known ahead of time, as it is more efficient comparing to multiple remote calls of a single-entity method version.

Sending a new order

sendOrder methods take FieldsContainerByTag parameter that contains all tag values that will be populated in the New Order Single ("D") FIX message.
Alternative sendAndGetOrder(...) can be used in case a reference to the order transaction record is needed right away.

```
static void newOrder()
{
   try
   {
      String managerName = "Simulator-HiFreq";
      FieldsContainerByTag orderSingleRequest = new FieldsContainerByTag();
      populateNewOrderSingle(orderSingleRequest);
      remoteClient_.sendOrder(USER_ID, managerName, orderSingleRequest);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when sending order", e);
   }
}

static void populateNewOrderSingle(FieldsContainerByTag orderSingleRequest)
{
   orderSingleRequest.add(ITMSConstants.MSGFLDTAG_SYMBOL, "IBM");
   orderSingleRequest.add(ITMSConstants.MSGFLDTAG_INSTRUMENT, "IBM");
   orderSingleRequest.add(ITMSConstants.MSGFLDTAG_ORD_TYPE,
      ITMSConstants.ORDTYPE_LIMIT);
   orderSingleRequest.add(ITMSConstants.MSGFLDTAG_PRICE, 70);
   orderSingleRequest.add(ITMSConstants.MSGFLDTAG_HANDL_INST,
      ITMSConstants.HANDLINST_AUTOMATED_EXECUTION_ORDER_PRIVATE_NO_BROKER_INTERVENTION);
   orderSingleRequest.add(ITMSConstants.MSGFLDTAG_SIDE, ITMSConstants.SIDE_BUY);
   orderSingleRequest.add(ITMSConstants.MSGFLDTAG_TIME_IN_FORCE,
      ITMSConstants.TIMEINFORCE_DAY);
   orderSingleRequest.add(ITMSConstants.MSGFLDTAG_ORDER_QTY, 100);
   orderSingleRequest.add(ITMSConstants.MSGFLDTAG_CURRENCY, "USD");
   orderSingleRequest.add(ITMSConstants.MSGFLDTAG_TIMESTAMP,
      System.currentTimeMillis());
}
```

Modifying an order

modifyOrder methods take FieldsContainerByTag parameter that contains all tag values that will be populated in the Order Replace ("G") FIX message.

```
static void modifyOrder(long orderIdToModify)
{
   try
   {
      IRecord order = remoteClient_.getOrder(USER_ID, orderIdToModify);

      double price = order.getNumericFieldValue(IHFManagersConstants.ORDER_PRICE_FIELD_NAME);
      double newPrice = price + 10; /* Increasing price */

      FieldsContainerByTag modifyMessage = new FieldsContainerByTag();
      modifyMessage.add(ITMSConstants.MSGFLDTAG_PRICE, newPrice);
      remoteClient_.modifyOrder(USER_ID, orderIdToModify, modifyMessage);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when modifying order " + orderIdToModify, e);
   }
}
```

Canceling an order

There are two ways of canceling an order.

Cancel an order without any additional fields in the Cancel Request ("F") FIX message:

```
static void cancelOrder1(long orderIdToCancel)
{
   try
   {
      remoteClient_.cancelOrder(USER_ID, orderIdToCancel, null);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when canceling order " + orderIdToCancel, e);
   }
}
```

Specify additional fields in the Cancel Request ("F") FIX message:

```
static void cancelOrder2(long orderIdToCancel)
{
   try
   {
      FieldsContainerByTag orderCancelRequest = new FieldsContainerByTag();
      orderCancelRequest.add(ITMSConstants.MSGFLDTAG_TEXT, "Cancel me");
      remoteClient_.cancelOrder(USER_ID, orderIdToCancel, orderCancelRequest);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when canceling order " + orderIdToCancel, e);
   }
}
```

Canceling multiple orders

The cancelOrders(...) method takes an optional filter and a fields container. Orders that satisfy the filter criteria are canceled. The fields from the fields container are populated in the Cancel Request message.
If the filter is null, all orders are canceled.
If the fields container is null, no additional fields are specified int the Cancel Request messages.

```
static void cancelOrders()
{
   try
   {
      /* Cancel Buy orders. */
      remoteClient_.cancelOrders(USER_ID, "Side = 'Buy'", null);

      /* Cancel Sell orders, with a Text field in a cancel message. */
      FieldsContainerByTag extraFields = new FieldsContainerByTag();
      extraFields.add(ITMSConstants.MSGFLDTAG_TEXT, "Cancel Me");
      remoteClient_.cancelOrders(USER_ID, "Side = 'Sell'", extraFields);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when canceling multiple orders", e);
   }
}
```

Trade report operations

Confirming a request

```
static void confirmOrder(long orderId)
{
   try
   {
      remoteClient_.confirmOrder(USER_ID, orderId, null);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when sending execution report.", e);
   }
}
```

Rejecting an order request

```
static void rejectOrder(long orderId)
{
   try
   {
      FieldsContainerByTag orderRejectReport = new FieldsContainerByTag();
      orderRejectReport.add(ITMSConstants.MSGFLDTAG_TEXT, "Too late");
      remoteClient_.rejectOrder(USER_ID, orderId, orderRejectReport);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when sending execution report.", e);
   }
}
```

Rejecting an order cancel request or an order replace request

```
static void rejectOrderCancel(long orderId, IELTTradeRequestMessage requestMessage)
{
   try
   {
      FieldsContainerByTag orderCancelRejectMessage = new FieldsContainerByTag();
      orderCancelRejectMessage.add(ITMSConstants.MSGFLDTAG_TEXT, "Too Late to cancel");
      remoteClient_.rejectOrderCancel(USER_ID, orderId, orderCancelRejectMessage);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when sending order cancel reject.", e);
   }
}
```

Accepting a Replace Request for an order

```
static void replaceOrder(long orderId, IELTTradeRequestMessage replaceRequestMessage)
{
   try
   {
      FieldsContainerByTag orderReplaceReport = new FieldsContainerByTag();
      remoteClient_.replaceOrder(USER_ID, orderId, orderReplaceReport);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when sending execution report.", e);
   }
}
```

Filling an order

```
static void fillOrder(long orderId, IELTTradeRequestMessage requestMessage)
{
   try
   {

      IRecord order = remoteClient_.getOrder(USER_ID, orderId);

      FieldsContainerByTag executionReport = new FieldsContainerByTag();
      double fillQty = order.getNumericFieldValue(IHFManagersConstants.ORD_QTY_FIELD_NAME);
      double fillPrice = order.getNumericFieldValue(IHFManagersConstants.ORDER_PRICE_FIELD_NAME);
      executionReport.add(ITMSConstants.MSGFLDTAG_LAST_QTY, fillQty);
      executionReport.add(ITMSConstants.MSGFLDTAG_LAST_PX, fillPrice);
      remoteClient_.fillOrder(USER_ID, orderId, executionReport);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when sending execution report.", e);
   }
}
```

Outing an order (accepting a Cancel Request)

```
static void outOnOrder(long orderId, IELTTradeRequestMessage requestMessage)
{
   try
   {
      FieldsContainerByTag executionReport = new FieldsContainerByTag();
      remoteClient_.outOnOrder(USER_ID, orderId, executionReport);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when sending execution report.", e);
   }
}
```

Target Operations

Adding targets

All Targets must belong to a Target List. The following sample adds a target list first (if it does not exist yet) and then adds a target to the list.

```
static  void addTarget()
{
   try
   {
      String targetListName = "My Portfolio";

      IHFRemoteTargetList targetList = remoteClient_.getTargetList(USER_ID, targetListName);
      if (targetList == null) /* if initial running, no portfolios & targets yet recovered */
      {
         targetList = remoteClient_.addAndGetTargetList(USER_ID, targetListName, null);
      }

      HFRemoteTargetFieldsContainerById targetFieldsContainer = new HFRemoteTargetFieldsContainerById();
      targetFieldsContainer.addSymbol("IBM");
      targetFieldsContainer.addInstrument("IBM");
      targetFieldsContainer.addSecurityType(ITMSConstants.SECURITYTYPE_COMMON_STOCK);
      targetFieldsContainer.addTargetQuantity(1000);
      targetFieldsContainer.addWaveSize(200);
      targetFieldsContainer.addSide(ITMSConstants.SIDE_BUY);
      targetFieldsContainer.addType(ITMSConstants.ORDTYPE_LIMIT);
      targetFieldsContainer.addPrice(11);

      IRecord target = remoteClient_.addAndGetTarget(USER_ID, targetListName, targetFieldsContainer);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when adding a target.", e);
   }
}
```

Sending target orders

When sending an order for a target, sendTargetOrder(...) method should be used instead of sendOrder(...), so that the framework properly links the order with its parent target.

```
static void newTargetOrder(long targetId)
{
   try
   {
      final String managerName = "Simulator-RemoteHiFreq";
      FieldsContainerByTag newOrderMessage = new FieldsContainerByTag();
      remoteClient_.sendTargetOrder(USER_ID, targetId, managerName, newOrderMessage);
   }
   catch (HFException e)
   {
      LOGGER.error("Exception when sending an order of a target " + targetId, e);
   }
}
```

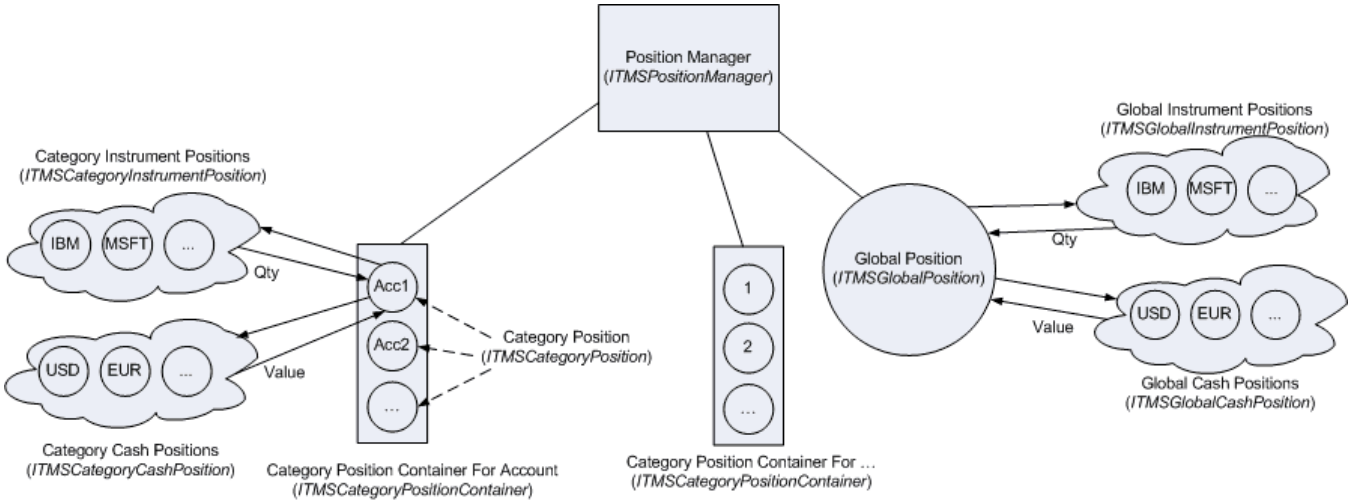
Positions Management

General Positions Management

Positions overview

General positions overview

Here is a basic positions structure and relationship diagram:



Key points:

- Positions types are: global, instrument, cash and category.
- Examples of category position type are: Account, Strategy, etc.
- Qtys are propagated from instrument positions to global position.
- Values are propagated from cash positions to global position.

Positions terms

- Category type + name pair - some criterion to classify positions category. Typically category type is name of some field in order messages or order records, and category name is value of given field. However categories might be compound and include few fields, in this case category position will be maintained for each unique combination of these fields. Category type cannot be free-form but should be one of registered types.
  Example of category type + name pair: "`Account`" (category type) + some account name (category name).
- Normalization currency - is a currency in which all category and global position values are expressed. Instrument and cash position values might be in different currencies, they are normalized to given currency and accounted in category/global position totals.
- Position info types:
  - Loaded - these values aren't accumulated in manager, new values will override previously loaded values;
  - Traded - accumulated values traded intraday;

- Open - open values that aren't traded yet, i.e. expected delta for current position values.

> ⚠ Positions manager doesn't use open positions at all:
>
> - Open positions are not used for calculating position totals (see below).
> - Hence open positions are not exported.
> - I.e. they are only reflected in position data structures.
> - It's up to application only whether to maintain open positions or not.

- For each type of maintained values (loaded/traded/open) position manager maintains:
  - Long quantity/value - increase when something is bought (i.e. obtained or expected to be obtained);
  - Short quantity/value - increase when something is sold (i.e. removed or expected to be removed);
  - For each type (loaded/traded/open) always: `Total = Long - Short`;
  - For cash positions:
    - long value increases when getting some cash (as result of some FX trade or when selling some non-FX instrument);
    - short value increases when spending money (as result of some FX trade or when buying some non-FX instrument).
  - For category position (including global):
    - long/short values only reflect instrument position values;
      - i.e. these fields are regular sums of corresponding fields for all underlying instrument positions;
    - fields `LoadedCashVal`, `TradedCashVal` and `OpenCashVal` represent money position for each type loaded/open/traded;
      - i.e. these fields include FX and regular trades, and also loaded cash position values;
    - this way category/global position has pure instrument position related information (how many shares are sold/bought and how much money spent/obtained from this), and also total net cash values.
  - Both long and short quantities/values are non-negative numbers.
- Position totals are calculated for API user convenience.
  Loaded and traded values are summed up in totals per position. E.g. for category/instrument/cash positions:
  - `Qty = LongQty - ShortQty`
    - `LongQty = LoadedLongQty + TradedLongQty`
    - `ShortQty = LoadedShortQty + TradedShortQty`
  - `Value = LongValue - ShortValue`
    - `LongValue = LoadedLongValue + TradedLongValue`
    - `ShortValue = LoadedShortValue + TradedShortValue`
  - There are no reporting fields to keep these values, but there are API methods to access these values. I.e. position itself in this case takes care of calculating totals.

Custom category types and names

Application updating some position values in given manager should provide implementation of `ITMSPositionCategoryNameProvider` if at least one category type is registered in manager.

- Position manager will figure out category name (to update proper category position) for each registered category type.
- Typically category type is just name of some field in order records, in messages, etc. So typically implementation of such provider would just take value of given field from record/message/etc;
- It is very desirable that numeric fields are not used as category types since it would require conversion of numeric values to strings (since category name must be expressed as string);

Normalization currency

Category position total values (expressed in normalization currency) are calculated based on cash position values only (see diagram above).

- I.e. instrument position values are expressed in primary currency of given instrument. They are not normalized.
- Cash position values are in given currency, they are normalized and accounted in category position totals.
  - i.e. this way category position total quantities are calculated based on underlying instrument positions, and total values - based on normalized cash position values(see diagram above).
- Normalization currency is required for each position manager.
  - Even if all values will be expressed in the same currency, e.g. USD, then USD should be specified as normalization currency.
  - In this case normalization will not be really performed while currency for all positions is USD. Forex rates data feed will not be used.

Positions export/import

> ⓘ Example of positions export output

```
Instrument Currency CategoryType CategoryName LongQty LongValue ShortQty ShortValue CustomNum1 CustomNum2
<cash> USD Account test1  4000  8000 1 2
IBMEU EUR Account test2 1000 50000 500 100000 3 4
MSFT USD Account test2 200 4000 400 8000 5 6
<cash> EUR Account test2  50000  100000 7 8
IBM USD Account test3 0 0 0 0 9 10
<cash> USD Account test3  10000  0 11 12
<cash> JPY Account test3  0  100000 13 14
<cash> EUR Account test3  0  0 15 16
```

Exporting/importing of instrument and cash positions is supported:

- Export/import conditions.
  - Import happens either on start up (in case `externalDataResource` is set in server position manager XML configuration) or on API call.
  - Export only happens on API call.
- Instrument and cash positions are together exported to and imported from the same resource.
- For cash positions `Instrument` column is populated with "`<cash>`" and `Currency` column is populated with cash position currency.
- Notes on columns:
  - `Instrument, Currency, LongQty, LongVal, ShortQty, ShortVal` are required columns in external position data resource.
  - More columns with data that helps to identify instruments can be present (i.e. custom fields, alternate instrument ID columns, etc).
    - Obviously for cash positions these columns will stay empty.
  - Obviously for cash positions `*Qty` columns will stay empty.
  - Extra columns with externally generated data to be stored in positions can be present.
- Data columns are delimited by TABs.
- Global or category positions export is exclusive:
  - If only global instrument/cash positions are maintained - then global positions are exported/imported (`CategoryType` is either empty or `CategoryType="Global"` and `CategoryName` is empty).
  - Otherwise, i.e. if there is at least one category type registered - then only category positions are be exported/imported.
- Positions export result contains only totals of loaded and traded values: `LongQty, LongVal, ShortQty, ShortVal`.

> ⚠ Open positions
> Open positions are not used for calculating these totals, i.e. ignored during export.

- Formatted numbers are also supported in imported data (since TMS_7.1.2009.02.09.r14013):
  - current locale's number format is assumed;
  - numbers in parentheses are considered negative;
  - Excel's export number format (`" 5,844.72 "`) is understood (since TMS_7.1.2009.02.09.r14015).
- Import granularity is "per position":
  - importing position data does not mean currently accumulated list of positions should be cleared and then replaced with new data;
  - in case data being imported has position already present in system, existing position is replaced with imported position.

Position manager distribution (optional)

- Overview
  Position management can be distributed in case it needs to be shared between processes and be centralized.
  A client position manager serves as a proxy representative of a server manager, which resides in the remote process.
  A server position manager is the actual position manager.
- Technical details
  - Position system report maintenance is only required if position manager is distributed.
  - Updating of positions is asynchronous. This includes programmatic loading of start-of-day positions as well.
    - Importing of start-of-day position happens in client position manager. I.e. file is loaded on client side and then loaded values are passed to server.
    - Exporting always happens on the backend. If invoked on client site, then exported data is taken from server and then can be saved to local resource.
  - Position structures on client side are replicated based on position system reports:
    - Client manager is always subscribed for category position system report. Hence per category position totals are replicated right away.
    - Underlying instrument and cash positions contributing to each category position are downloaded and maintained on demand, i.e. on first usage.
    - Client manager may optionally maintain only cash, only instrument or both positions for specified category types;
    - Client manager may pre-subscribed for instrument and cash positions for some category type (including Global) if needed.
  - Once category position container (for given category type) or category position (for given type+name), proxy will subscribe for

needed data and will keep maintaining these positions.
    - I.e. application may get some [TS:category/instrument/cash] position and keep the reference, and this instance is kept updated all the time with latest received info.
- Local positions
    - With distributed position manager multiple clients may contribute to shared data on server, and client is optionally being updated asynchronously with server-side data
    - Even if there is only one client process updating given category (entire category or only some category positions within given category) - normally data is still flowing thru the server
    - This creates certain delay in how soon up-to-date totals are becoming available in the client, which might be a concern in certain very sensitive cases when trading limits, for instance, are very strict
    - In cases like this some position categories can be marked as local, and it means that client position manager will not be getting totals for given category from the server, but only use own totals
    - Updates to local categories are still aggregated on server side, so server would have all totals, the only difference is that category totals on client side are not picking up other changes, only updates from the same process.
    - This way when category is declared as local - position totals are immediately available in the local client process.
        - For instance multiple HiFreq order managers might be updating same local category position, and totals across all managers in this process would be immediately available for trading limit checks, for example.
- Caveats
    - Data is not available on client, though updates from client are processed on server.
        - Explanation:
          Client position manager uses reporting and hence event service for data replication. Hence there should be an even service connection (configured in `EventService.xml`) between processes hosting position manager server and clients. Thus if both client and server managers are hosted in separate processes of the the type, data will not be available on client, though updates from client will be processed on server.
        - Workarounds:
            - Host server manager in some separate process, and in both custom apps have client managers.
            - Route events through client hub, but that's not desirable due to performance considerations.
    - If local position categories are in use - LoadedXXX field values for such categories are also local.
        - Meaning that positions cannot be loaded in one central location (on server) and then just distributed across all clients.
        - Instead each process should load own portion of start-of-day positions so it's properly reflected in local data structures and also reported to server.

Positions recovery

Position manager does not facilitate its own recovery.
Technical details:

- Position manager does not provide any persistence for provided data.
    - Each application component that contributes some position updates must have recovery and provide same position values again on startup.
    - Though distributed position manager has some internal recovery to restore position data if some position manager client or server is restarted.
    - If position manager server process is restarted, it will gather current state from all connected position manager clients.

Other notes on position management

- Global position and instrument positions for global and each category position are always maintained.
- Cash positions are optional now. I.e. can be turned off. In this case only trades in currency same as normalizing one can be processed/accounted in position manager.
- Same interface `ITMSPositionManager` represents both server and client side of position manager, so usage by application should be absolutely transparent.
- `ITMSPositionDataAccessor` represents read-only part of position manager, i.e. it has only data accessing methods and doesn't allow to update data in the manager.

Positions configuration

Positions XML configuration

The easiest way to configure a position manager is through an XML resource.
Folder `backend\Directory\ElTrader\TMS\Config\Positions` contains samples of xml configuration.

- Server position manager XML configuration
  `HiFreqPositionManager.xml` contains sample of a server position manager configuration:

```
<!--
Server position manager configuration.

<PositionManager> attributes:
 - name - the name of the server position manager.
```

Values: free-form case-sensitive string.
Validation: not validated.
Required: yes.

- mode - must have "server" value

- distributed - indicates whether position manager should be distributed.
Values: true/false.
Required: no.
Default Value: true.

- normalizingCurrency - indicates the normalization currency. Values will be normalized to specified currency.
Values: USD/EUR/GBP/...
Required: yes.

- expectingOpenPositionUpdates - indicates whether position manager is expecting to receive position manage updates.
I.e. this determines if open positions will be updated by users of this manager.
Allows to control if open positions should be tracked for both long and short, or separately.
Values: true/false/longOnly/shortOnly
Required: no.
Default Value: true.

- maintainingCashPositions - indicates whether position manager will maintain cash positions.
If cash positions aren't maintained, then it will allow only positions for currency same as normalizing one.
Values: true/false.
Required: no.
Default Value: true.

- maintainingIntradayTradedPositions - indicates whether position manager will expect traded positions to be intraday.
Not supported in HiFreq yet, i.e. traded positions are all positions currently present in HiFreq.
In portfolio domain if traded positions should be intraday, then orders system report should have intraday related fields.
Values: true/false.
Required: no.
Default Value: false.

- maintainingCategoryTotalsForFXInstrumentPositions - indicates whether position manager
will maintain include (roll up) FX instrument positions in category position totals.
Values: true/false.
Required: no.
Default Value: false.

- reportManagerName - report manager name where position reports will be kept.
Required: yes if distributed = "true"

- forexRatesDataFeedName - name of FX rates data feed.
Required: no.
Default Value: "Forex"

- metaDataName - name of fields metadata to be used for system reports.
Required: no.
Default Value: "Position table report fields"

- externalDataResource - name of resource with start-of-day positions. Will try to load this file on startup.
Required: no.

- externalDataAlternateInstrIdSources - comma-delimited list of alternate instrument IDs to be exported (or accounted when import)
in position data resource.
Values: RIC, CUSIP, ISIN, etc - will make data columns RIC.ID, CUSIP.ID,ISIN.ID, etc be exported/imported.
Required: no.

- externalDataExtraFieldsToResolveInstrId - comma-delimited list of custom fields that can be used during import to identify security/instrument. E.g. can be Symbol, SymbolSfx, SecurityExchange, Country, etc.
Required: no.

- externalDataCustomFields - comma-delimited list of custom fields to be imported/exported for instrument and cash positions

```
  to/from position data resource.
        Required: no.

- categoryPositionsReportName - name of category position system report.
        Required: no.
        Default Value: generated as "managerName - Category Position Report"

- instrumentPositionsReportName - name of instrument position system report. If not specified, then generated as
  "managerName - Instrument Position Report".
        Required: no.
        Default Value: generated as "managerName - Instrument Position Report"

- cashPositionsReportName - name of cash position system report.
        Required: no.
        Default Value: generated as "managerName - Cash Position Report"

 <Categories> - list of category types to be maintained by manager. By default only Global positions maintained.

 <Category> attributes:
  - type - name of the type of position category.
                E.g. "Account", "Portfolio", ...
  - adjustCategoryPositionByCoef - rather exotic parameter that allows to adjust/scale
   instrument and category position quantities by certain coefficient that should
   be specified as custom instrument field (PositionCoef.CUSTOM) in security master.
   Values: true/false.
   Required: no.
   Default Value: false.
  - local - indicates if given category should be locally updated. Updates to this category
   are still reported to server, however totals are not taken from server, which means
   that such category doesn't have updating delay, totals in given category
   are immediately available in updating process.
   Values: true/false.
   Required: no.
   Default Value: false.
  - expectingOpenPositionUpdates - same optional true/false/longOnly/shortOnly parameter as for
   entire position manager, but allows to control to have open positions maintained
   only for some categories, but not others.
  - borrowTrackingCategory - allows to indicate that special handling of updates, that is
   suitable for "borrow tracking" should be enabled for this whole category,
   or only some specific position categories for this type.
   "borrow tracking" position means - traded long (buy cover portion) qty cannot exceed
   traded short qty at any given moment, i.e. cannot cover more than shorted.
   Values: true/false/<regexp>.
   Required: no.
   Default Value: false.

 <ReportBuffer> - spec of report buffering element to be used with position system reports.
   Default is 3 sec/500 events. Buffer is fired after timeout or if event count is reached.

 <ReportElement> attributes
        - type - must have "Buffered Records" value.
        - timeout - buffering timeout in milliseconds.
        - eventCount - event count -->
<PositionManager
 name="HiFreq Positions"
 mode="server"
 distributed="true"
 normalizingCurrency="USD"
 forexRatesDataFeedName="Forex"
 metaDataName="Position table report fields"
 reportManagerName="HiFreq report manager"
 expectingOpenPositionUpdates="true"
 maintainingCashPositions="true"
>
 <Categories>
  <Category type="Strategy"/>
 </Categories>
```

```
<ReportBuffer>
 <ReportElement type="Buffered Records" timeout="100" eventCount="25" />
```

```
 </ReportBuffer>
</PositionManager>
```

- Client position manager XML configuration
  `HiFreqClientPositionManager.xml` contains sample of a client position manager configuration:

```
<!--
Client position manager configuration.

<PositionManager> attributes:
 - name - the name of the client position manager. Must be the same as a server position manager name.
  Values: free-form case-sensitive string.
  Validation: not validated.
  Required: yes.

 - mode - must be "client".

 - maintainInstrumentPositionsForCategoryTypes - comma-delimited list of category types (including 'Global' if needed)
   to maintain instrument positions for. Or * to maintain them for all types.
        Required: no.
        Default Value: "Global"

 - maintainCashPositionsForCategoryTypes - comma-delimited list of category types (including 'Global' if needed)
   to maintain cash positions for. Or * to maintain them for all types.
        Required: no.
        Default Value: "Global"

 - presubscribedCategoryTypes - comma-delimited list of category types (including 'Global' if needed) to pre-subscribe
   and maintain instrument/cash positions for. Or * to pre-subscribe for all types.
   By default doesn't pre-subscribe.
   Makes sense only if instrument/cash positions should be maintained at least for some category types or "Global".
       Required: no.
       Default Value: empty, so it does not presubscribe.

    - resetPositionsOnReconnect - true/false attribute indicating whether cached position data
                should be reset/removed when connection to server is lost and client position
                manager is restoring connection.
        Required: no.
        Default Value: "false"

 - waitForInitialPositionDataTimeoutInSec - timeout in seconds indicating for how long client
   position manager is allowed to wait for initial data to come and the time of
   initialization (when subscribing for category positions and optionally for
   instrument/cash positions for specified "presubscribe" category types).
   0 or negative values mean that position manager client doesn't wait for data
   to come, so it will be coming in asynchronously some time later after initialization.
        Required: no.
        Default Value: "10"-->
<PositionManager name="HiFreq Positions" mode="client" />
```

Access to positions data via API

Most important API methods

- `ITMSPositionDataAccessor` (note `ITMSPositionManager` extends `ITMSPositionDataAccessor`):
  - `ITMSGlobalPosition getGlobalPosition()` - returns container of global position info.
  - `ITMSCategoryPosition getCategoryPosition(String categoryType, String categoryName)` - returns category position for the specified category type and name.
  - `ITMSCategoryPositionsContainer getCategoryPositionsContainer(String categoryType)` - category positions container for specified category type.
  - `ITMSPositionManagerConfigProvider getConfigProvider()` - returns container with all configuration data for this position manager (e.g. available category types, whether open positions be updated etc).

- `ITMSCategoryPositionsContainer`
  - `ITMSCategoryPosition getCategoryPosition(String categoryName)` - returns category position for the specified category name.
  - `ITMSCategoryPosition ensureCategoryPositionExists(String categoryName)` - returns category position for the specified category name.
- `ITMSCategoryPosition`
  - `ITMSCategoryCashPosition getCashPosition(String currency)` - returns cash position in this category for specified currency.
  - `ITMSCategoryInstrumentPosition getInstrumentPosition(String instrumentId)` - returns instrument position in this category for specified instrument.
- `ITMSGlobalPosition`
  - `ITMSGlobalInstrumentPosition getInstrumentPosition(String instrumentId)` - returns global instrument position for specified instrument.
  - `ITMSGlobalCashPosition getCashPosition(String currency)` - returns global cash position for specified currency.
- `ITMSPositionManager`
  - `void importPositions(String resourceName)` - allows to import start-of-day positions from specified external resource.

Position synchronization notice

- Position manager implementation is thread safe.
- The only case when external synchronization might be required is synchronization by position object (category/instrument/cash position) when calling multiple `getXXX()` methods (to avoid data changes between `getXXX()` calls).
- For position listeners the updated position is locked while `on*()` calls.
- Access to another positions should not be done in this call because of that lock. It will help to prevent deadlock with another thread, accessing positions in a different order.
  - If access to other positions is needed in this `on*()` method, thread boundary should be introduced and another thread should access positions.

Positions GUI

Position reports specified in XML or by config provider are available from GUI.

HiFreq Remote Positions Management

Configuration

Positions are configured in the remote HiFreq application. See the regular non-remote API Programmer's Guide for the configuration info.

Subscription to position data updates

In order to subscribe/unsubscribe to/from positions data, use IHFRemoteClient.subscribeTo*PositionData(...) and corresponding unsubscribeFrom*PositionData(...) methods.

```
private static class RemotePositionListener extends TMSInstrumentPositionEventRemoteListenerAdapter
{
   @Override
   public void onInstrumentPositionAdded(String instrument, IRecord instrumentPositionRecord)
   {
      LOGGER.debug("Instrument position added = " + instrumentPositionRecord);
   }

   @Override
   public void onInstrumentPositionUpdate(String instrument, IRecordUpdate instrumentPositionRecordUpdate)
   {
      LOGGER.debug("Instrument position updated, instrument = " + instrument + " update = " + instrumentPositionRecordUpdate);
   }
}

ITMSInstrumentPositionEventRemoteListener remotePositionListener_ = new RemotePositionListener();

private void subscribeToPositions() throws HFException
{
   remoteClient_.subscribeToGlobalInstrumentPositionData(USER_ID, remotePositionListener_);
}

private void unsubscribeFromPositions() throws HFException
{
   remoteClient_.unsubscribeFromGlobalInstrumentPositionData(USER_ID, remotePositionListener_);
}
```

Strategy Data Management

Strategy data is a record that is visible and editable from GUI. Application can use that data to parametrize its strategy. It can react to changes of strategy data record fields and change its behavior accordingly.

See simple.HFTimerSample for a working example of strategy data API utilization.

Initializing strategy data

This example creates a strategy data record when running on a clean DB. Sequential runs will have strategy data recovered.

```
private static final String STRATEGY_NAME = "Timer Strategy";
private static final String PX_INCREMENT_FIELD = "PxIncrement";
private static final String PERIOD_FIELD = "Period";

private static void addStrategyData() throws Exception
{
   if (remoteClient_.getStrategyData(USER_ID, STRATEGY_NAME) == null)
   {
      //No strategy data recovered from DB, so need to create a new one.
      HFRemoteStrategyDataFieldsContainerById strategyFields = new HFRemoteStrategyDataFieldsContainerById();
      strategyFields.addName(STRATEGY_NAME);
      strategyFields.addStatus(IHFRemoteStrategyData.STATUS_ACTIVE);
      strategyFields.add(PX_INCREMENT_FIELD, 250);
      strategyFields.add(PERIOD_FIELD, 4000);
      remoteClient_.addStrategyData(USER_ID, strategyFields);
   }
}
```

Using strategy data

This example gets strategy data record and uses its values to schedule a timer task.

```
private static void useStrategyData()
{
   IRecord strategyData;
   try
   {
      strategyData = remoteClient_.getStrategyData(USER_ID, STRATEGY_NAME);
      long period = (long) strategyData.getNumericFieldValue(PERIOD_FIELD);
      long status = (long) strategyData.getNumericFieldValue(IHFManagersConstants.STATUS_FIELD_NAME);
      if (status == IHFRemoteStrategyData.STATUS_ACTIVE)
      {
         // schedule task with specified period
      }
   }
   catch (HFException e)
   {
      e.printStackTrace();
   }
}
```

Listening to strategy data updates

This part demonstrates how to be notified about GUI-initiated changes to a strategy data record. The listener should be added to the remote client in order to get notified.

```
private static final class StrategyDataListener extends HFRemoteStrategyDataListenerAdapter
{
   @Override
   public void onStrategyDataUpdated(String strategyName, IRecordUpdate strategyDataUpdate)
   {
      boolean restartTask = false;
      if (strategyDataUpdate.getUpdatedFieldIndex(PERIOD_FIELD) >= 0)
      {
         restartTask = true;
      }

      int statusFieldIndex = strategyDataUpdate.getUpdatedFieldIndex("Status");
      if ( statusFieldIndex >= 0)
      {
         int newStatus = (int) strategyDataUpdate.getUpdatedNumericFieldValueAt(statusFieldIndex);
         if (newStatus == IHFRemoteStrategyData.STATUS_ACTIVE)
         {
            restartTask = true;
         }
         else
         {
            //stop timer task here
         }
      }

      if (restartTask)
      {
         //restart timer task here
      }
   }
}
```

Audit Utility

Audit Utility is used to report necessary information to the "Audit" GUI report view.
The following sample sends a message through the utility.

```
private static void sendAuditInformation()
{
   remoteClient_.getAuditUtility().sendAuditRecord(IHFRemoteClientAuditUtility.AUDIT_SEVERITY_INFORMATION, "Every HiFreq
Sample has started up successfully");
}
```

Different severity levels can be used. The values of "Severity" field can be seen/modified at
\backend\Directory\ElTrader\TMS\Config\ElementSettings\TableElement\AutoTraderAuditTableFieldsMetaData.xml
Default values are AUDIT_SEVERITY_ERROR, AUDIT_SEVERITY_WARNING, AUDIT_SEVERITY_INFORMATION.

Market Data and Custom Record Data

Subscription

For market data subscription see IHFRemoteClient.subscribeToMarketData(...) and IHFRemoteClient.unsubscribeFromMarketData(...) methods.
The object implementing ITMSMarketDataEventListener interface will be passed to the subscribe call and it will receive events for the subscribed instruments.
The same listener object can be passed to multiple subscribe calls for different instruments.

Example

This example shows subscription/unsubsription for the "IBM" instrument.

```
private static class MarketDataListener extends TMSMarketDataEventListenerAdapter
{
   @Override
   public void onMarketDataUpdate(String instrument, IRDRecord record)
   {
      final double bidPx = record.getDoubleFieldValue(MDDefs.RDFIELD_BID_PRICE);
      final double askPx = record.getDoubleFieldValue(MDDefs.RDFIELD_ASK_PRICE);

      LOGGER.debug("Market data is received for instrument = " + instrument + " bidPx = " + bidPx + " askPx = " + askPx);
   }
}

private ITMSMarketDataEventListener marketDataListener_ = new MarketDataListener();

private void subscribe() throws HFException
{
   remoteClient_.subscribeToMarketData(USER_ID, "IBM", marketDataListener_);
}

private void unsubscribe() throws HFException
{
   remoteClient_.unsubscribeFromMarketData(USER_ID, "IBM", marketDataListener_);
}
```

Report subscription API

The IHFRemoteClient.getReportClient() gives access to the remote reporting framework.
See IRPTClient javadoc for information about available report access and subscription options.

Samples

Description

The samples are located at backend\samples\java\ELTHiFreqAppRemote
and organized into these folders:

- simple - simple strategies, e.g. order modification by timer or by market data updates
- targets - demonstrates usage of targets and target lists

Compilation

Sources are pre-built into backend\samples\java\ELTHiFreqAppRemote\ELTHiFreqAppRemote.jar
The samples can be launched as is, without a need for compilation.
If source code modifications need to be compiled, Eclipse or Ant can be used.

Eclipse

Eclipse can be used as a convenient and free IDE.
Importing and debugging samples in Eclipse is described in backend\samples\java\eclipse_readme.txt.
It is also recommended to read backend\samples\java\readme.txt.
In short, use File/Import.../General/Existing Projects into Workspace - and select backend\samples\java\ELTHiFreqAppRemote\ folder. It'll find sample projects.
Each project is configured to compile sources to backend\samples\java\ELTHiFreqAppRemote\<project>\classes folder.
This "classes" folder is included into sample application classpath before ELTHiFreqAppRemote.jar, so classes compiled with Eclipse take precedence over ELTHiFreqAppRemote.jar classes.

Ant

Ant build tool can be used as a building alternative.
Download Ant from http://ant.apache.org/
Set JAVA_HOME environment variable to path to JDK, run "ant" command in backend\samples\java\ELTHiFreqAppRemote folder.
It'll compile sources and update ELTHiFreqAppRemote.jar

Running

The logs of HiFreq processes are located in the logs folder.
Remote client application does not produce logs, it's up to the application layer how to organize logging.

Samples

These scripts are to be used:

- backend\bin\Start_All_HiFreq.pl starts all required backend processes
  - Comment default sample there and uncomment ELTHiFreqDefaultServerApp.pl
- run one of remote app samples at backend\bin\samples\hifreq\remote
  - ELTHiFreqSimpleSampleApp.pl
  - ELTHiFreqTargetSampleApp.pl
- backend\bin\admin\ELTAdmin_KillAllProcessesQuick.pl - kills all processes (besides remote sample).
- backend\bin\admin\ELTAdmin_DropDBTables.pl - drops all database tables
- frontend\bin\ELTHiFreqView.pl - runs HiFreq GUI
  - Login as "demohifreq"
  - Select workspace corresponding to the selected remote sample. Either "Orders" or "Targets"

Development

The recommended way of development is to copy one of most relevant existing projects and customize it according to the desired use cases.

Debugging

Each sample is run with a debug port configured by default, see sample *pl file
_OPT_DEBUG_PORT_OFFSET() => 235, # real port by default is 9235
In Eclipse add Run/Debug Configurations/Remote Java Application/ and choose application host and port (9235)

Broker Simulator

In all the samples orders are sent to broker simulator.
Broker simulator understands the following instructions in Text field of execution request messages:

Command	
0	Request will be completely ignored by simulator. In this case subsequent requests most likely will be rejected due to unknown OrigClOrdID (since simulator just skipped previous request)
NOCONFIRM	D - leave order unacknowledged. F/G - cancel/replace order w/o confirming F/G
NOFILL	N/A, doesn't fill w/o explicit instruction anyway
REJECT	D - reject order, no further requests accepted. F/G - reject this cxl/cxr, further requests accepted, i.e. order stays alive
CANCEL	D - order is confirmed and then unsolicited cancel right away. G - request ignored, unsolicited cancel for D or previous G
FILL_N	if N<=10 then N is the number of fill reports to be sent back, in this case whole order quantity is evenly split between fills. Otherwise N is the number of shares to be filled in each fill report sent back. In both cases the order will be filled completely. Command can be submitted later in G.
PARTIAL_N_M	N, M - number of fill reports and number of shares to be filled with each execution report. If both specified - then smaller param is number of fills, another - number of shares. If only N specified (i.e. PARTIAL_N) - then single partial fill for N shares.

For example for order not to be filled use
newOrderMessage.setText("NOFILL");