MARX Client Programming Manual

Version 1.7.0 05/04/16

Table of Contents

MARX Overview

MARX stands for Market Routing And eXecution. It is an enterprise class Java JavaEE based application which provides multi-asset-class order management, routing, execution management, market data access, and compliance reporting functions. In addition a Java Rich Client Platform (RCP) desktop trading client application provides access to the system. APIs exist to allow external programmatic access to the server side (back end) components of the system via SOAP over HTTP, REST/JSON and FIX4.4. Live market data may be accessed via a custom binary protocol, web sockets, or FIX4.x

MARX has been developed entirely with open source software and established industry standard protocols. This allows maximum compatibility with other software and minimizes the expense of development and deployment of the system.

Market Access

Access to sell-side markets is provided via gateways. A gateway is responsible for translating orders into a format specific to the sell side system, transmitting the orders, and receiving messages. Most sell side systems use the FIX protocol. Other order routing protocols can be supported via additional protocol support modules. Depending on the specifics of how FIX is utilized some custom code, data dictionary definition, and configuration will be required in order to inter-operate with FIX based sell side systems.

Market Data

MARX provides access to market data feeds via feed handlers. Each particular vendor feed protocol (IE Bridge, CME, Nexa) requires a protocol support module. A feed handler can load any combination of protocol support modules and multiple handlers may be coordinated in order to support any desired combination of feeds. Feed handlers also support loadable client side protocol handlers, allowing a feed handler to supply data to other applications in a variety of formats. MARX is supplied with a standard client protocol module which is utilized by the system for market data access. A FIX protocol module is also supplied for rapid integration with client trading applications or other systems which utilize FIX, and a websockets based module for integrating with browser-based market data displays.

Client

MARX is provided with a Java RCP client. This client is based on the Eclipse Rich Client Platform. RCP provides numerous enterprise class capabilities including remote client update, built in help system, and the ability to support additional plugins for enhanced functionality, white labeling, and internationalization.

3rd party client software can integrate with the MARX OMS and feed handler via simple straightforward means detailed in the Client Side Programming section.

MARX Concepts

This section explains Marx functionality in terms of the types of data presented by the APIs, its uses, structure, and semantics. Additional concepts include the security model and order processing model.

OMS Data

This subsection covers all the details of the various data structures information is broken down into three main pieces, Market Data, Order Data, and ETS Data.

ETS Data

This category covers data which represents users, departments, companies, and the various roles and permissions they are assigned within the system. Certain other data closely associated with users is also stored here, including preferences, watchlists, etc.

User: Users are the most fundamental item of data in the Marx system. Each user object represents one set of logon credentials and permissions for the Marx system. Every user belongs to a department and a company. Each user has a user object and a profile object, which contains extended information (IE contacts, etc). In addition to a username and password, each User object also holds a unique ‵handle`. The handle is used to identify the user for display purposes, allowing usernames to remain private.

Profile: These are simply used to hold some additional fields for each user which are rarely needed by the system.

Address: This is an element used to hold postal addresses, they are attached to profiles.

Application: These hold requests to create user accounts.

Company: This holds information defining a company. Each user belongs to a company, and each CompanyDivision (department) also belongs to a company.

CompanyDivision: These are also known as departments. They represent logical groupings of users. Every user falls within some department, and each department belongs to a single company. departments may also contain other departments.

Preference: These objects belong to users and represent settings which can be applied by client applications or other portions of the OMS as needed. Each Preference has a name and a value, and belongs to a specific user.

WatchList: A watchlist is a group of instruments with a name and ownership. It is usually used by client applications to organize the display of quote.

Role: A role represents some sort of role a user plays in the OMS system, or a more specific permission which users can have. Some of the objects in the other data categories can have specific permissions reflected in this table. Each role has a RoleName and a RoleCategory. The RoleCategory indicates what the purpose of the role is, while the RoleName uniquely identifies the role. RoleNames may also be used by application logic to indicate specific things. The exact semantics of each role depend on its category.

Message: These are used to hold messages which a user's client application should display. They are simple text objects.

Order Data

This category of data holds all order-related data elements.

Account: These hold orders. Every order belongs to an account, and each account belongs to a company. Accounts are used to manage margin and balance. Note that accounts don't 'belong' to specific users. Each account is associated with a Role in the category UserAccount, where the RoleName is identical to the account's id.

Position: A position represents some quantity of some instrument held in an account. It can be either long (positive quantity) or short (negative quantity). Each position also has a price, which represents the amount paid for the position up to the last execution divided by the quantity. Each instrument has a single position in each account, the OMS does not separately track each order or have any concept of a 'deal'.

Destination: A destination represents a logical endpoint to which orders may be routed. This could represent a liquidity provider, an ECN, a Clearing Firm, etc depending on the business model and the individual destination. Every order is routed to a specific destination. The destination object itself is associated with a gatewayname, which identifies the OMS gateway (communications session) used to route orders to the destination.

DestinationInfo: These objects represent all the information describing the capabilities of a destination. This consists of a list of all the Market objects that are valid for this destination, and the id of the destination itself.

Market: A market object holds information describing all the valid options when routing an order to a given destination and market. This includes the market id, the destination id, a description of the specific route, and a series of lists of options. The options include the following:

- instrumentTypes - The types of instruments which this market will accept orders for.
- accountIds - The ids of all the accounts which are able to trade with this market.
- sides - The valid side values for this market.
- tifs - The valid TimeInForce values for this market.
- ordertypes - The valid OrderType values for this market.

There is also a flag indicating whether or not an AON (AllOrNothing) order is valid for this market.

Ticket: A ticket represents a single action requested by a trader from the system. This could be a single order or it could represent several orders tied together by some sort of order handling strategy. It might also represent a series of orders, replaces, and cancels. Every order is associated with a single ticket.

Order: An order represents the placement of a single buy or sell of some sort at a trading venue. It has a status, a quantity, an instrument, a destination, and a destination market. Each order is owned by a specific user and belongs to a specific account, and a specific ticket. Some orders may be ‘complex’, either they belong to a strategy associated with their ticket, or they are OCO (one cancels other). Such orders usually have other orders associated with them as ‘legs’. An order may also be associated with a related order as either a replace or a cancel.

Execution: These represent individual order state transitions. Each time some significant change in state occurs on a given order an execution object is generated to describe it. Each execution is associated with a single order. The

ExecutionType describes the nature of the transition and the InitialState and State fields describe respectively what the order's state was before the execution and the state after it.

Adjustment: An adjustment represents a non-order change to a position in an account. These are used to compensate for changes to positions in an account that are out of the scope of the OMS (IE manual transactions or out-of-band corrections). Each adjustment has an account, instrument, price, and quantity, analogous to the same fields in order and execution objects.

Market Data

The market data category contains all data related to instruments and sources of quote. It also contains data which allows for marking external prices up. Note that orders rely on some of this data, the Order Data and Market Data are not entirely independent.

Instrument: An instrument represents one tradeable entity. It has an InstrumentType (stock, foreign exchange, option, future, etc), a symbol, underlying id, basesymbol, market, and a collection of other arbitrary attributes. The symbol is used for display purposes, but also should be unique within a given market, meaning a given symbol/market pair is unique. Underlying ids are used by contracts (futures and options) to indicate the instrument id of the underlying instrument the contract is on. Basesymbol represents the 'stem' of the instrument's symbol, the portion which is invariant for all contracts on the same commodity or underlying instrument. (IE a CME e-mini contract might be E6M5 and the basesymbol might be 'E6' with M5 representing a specific maturity). Market indicates the id of the listing market for the instrument, noting that this may not be the same as the market it is being traded or quoted on! Some instruments may exist on several markets, each such listing has a unique market id and instrument object in Marx. Other attributes are instrument and usage specific, they all consist of name/value pairs.

Market: Also frequently called 'exchange' a market is a uniquely identified trading venue which generates its own independent quotes. Each market has its own id and a description.

Carrier: A carrier is a unique identifier for an individual distributor of market data. Note that this is different from source of data. Each carrier may have unique carrier symbols for each market. These simply identify instruments in the symbology used by the carrier as opposed to that of the OMS, which is represented by the symbol element of the instrument class.

Supplier: Unique identifier for the organization supplying a given quote. This may be the operator of a given market, or some other entity.

Route: A market data route describes where a specific quote is coming from. It contains a number of fields. Some fields may not be supplied, which produces a partial route, the unspecified fields being wildcards. A route consists of:

- marketid - The market id to acquire data from
- symbol - Instrument to acquire data for
- supplier - Supplier to use
- carrier - Carrier to use
- protocol - Market data protocol to employ
- connection string - Protocol specific connection information
- level - The type of data to request

Level: Level indicates a type of data, Level1 is stock last price and BBO, Level2 is stock depth of market, FutLevel1 is last price/BBO for futures, etc.

InstrumentType: Indicates the type of an instrument, Stock, Future, Currency Pair, etc.

Markup: A markup describes, for a given supplier, target market, and instrument certain values which are used to modify quotes supplied on matching routes. These values are used by various feed handler modules and also by the OMS to adjust pricing, both for quotes originating from the route, and or orders directed to the market.
NOTE: one limitation in the current generation OMS is that it doesn’t capture the supplier of a given price, meaning that if a single instrument on a given market has more than one markup, the OMS cannot determine which markup should be utilized. It is best to limit any given instrument to one markup per market, regardless of whether data is available from more than one supplier or not. This primarily affects F/X data and requires each pair to be defined as a separate instrument per LP.

The Market Data system is structured so that it can be queried to request legal routes given a subset of routing information, and for other similar purposes. Suppliers and carriers are associated, creating pairs of supplier/carrier, and further associated with markets. Suppliers are also associated with levels. Finally protocols, and connection strings are linked to each unique combination of carrier, supplier, level, and market. This is linked to the market id fields in the instrument objects to project the fully specified collection of legal routes. Any subscription sent via a protocol using a connection string with the specified elements should produce data.

Some other associations also exist. One is to carrier symbols. Carrier symbols act as translators between the Marx OMS symbology and data provider symbologies. Each carrier symbol association consists of an instrument id, a carrier, and a carrier symbol.

Another association is destination symbols. This is analogous to carrier symbols but applies to destinations. It provides symbol translation used by the OMS when routing orders as opposed to carrier symbols, which are used by the feed subsystem when requesting quotes.

MARX APIs

Client applications can access the system via several mechanisms. These include a set of SOAP based web services, a FIX order routing facility, REST/JSON, and either FIX, websockets, or binary feed handler protocols.

REST

REST web services APIs allow simple applications written in Javascript or other lightweight tools to interact with the Marx OMS. The REST API closely mirrors the SOAP API and they expose approximately the same functions, including most of the Marx OMS functionality.

Three basic sets of service calls are supported:

- Data Management - Functions relating to user management and permissions.
- Market Data - Functions related to market data access and the symbol master.
- Order Management - Functions related to order placement and related

functions.

Client side REST wrappers

Each of the 3 APIs above has a corresponding Java client library which can be used to implement client applications. REST is a little lighter weight than the SOAP interface. The underlying Resteasy library also pre-generates javascript libraries which can be used to perform AJAX calls.

SOAP

These are provided by the JBoss WS web services stack. The client submits SOAP requests over HTTP and receives data in the response. This mechanism is entirely synchronous. A client requiring order status updates to populate a blotter for instance will need to poll the web service periodically for updates.

There are 3 sets of service calls which are supported.

- Data Management - Functions relating to user management and permissions. See Ets Data Management War.
- Market Data - Functions relating to market data access. See Ets Market data War.
- Order Management - Functions relating to order flow. See Ets OMS Services War.

Client Side SOAP Wrappers

Each of the web services APIs above has a companion client side Java wrapper which is intended to simplify interaction and allow some decoupling between client application logic and the web services stack. Client software written in Java can utilize these pre-generated wrappers in order to simplify access.

These modules are:

- Ets Data Management Client - Wraps the data management functions
- Ets Marketdata Client - Wraps the market data access functions
- Ets Order Services Client - Wraps the order management functions

Non-Java client code will need to utilize native WSI compliant SOAP client tools to build a similar interface. We provide an SDK which includes sample integration code using SOAP and FIX4.4 in C# which can be acquired from Whit's End technical support.

Authentication and State Management

Web services rely on session level authentication in order to identify the trading system user making a request. When the initial request is made by the client it will be redirected to an authentication URL. After supplying client credentials to this facility the client's session state will be retained. A JSESSIONID cookie will be returned to the client which identifies this session in future requests. Authorization is accomplished automatically by the server side container binding credentials into a JAAS context. Each web service component deployable (war) provides the requisite mappings in its web.xml deployment descriptor. State is shared between all three web service APIs, authentication with any of them will be carried over to the other two. These sessions will time out after a defined period of inactivity (currently 30 minutes).

The following code example from the Java client illustrates authentication handling. In this case we just make an HTTP call to the base URL for one of the services we want to access. An HTTP redirect will be returned with a cookie header containing the JSESSIONID cookie of the client's session. Next the initial URL with /j_security_check added to the end of the URL is called. The previously returned cookie is passed back along with url-encoded request body in a POST. The parameters j_username and j_password, with the client's username and password are contained in the request body. Successful authentication will result in a 302 response intended to redirect the client back to the originally requested URL. At this point any request from this client containing the session id cookie will be authorized according the settings in the ETS user database (or an alternate JAAS configured credential store).

```
/**
 * Perform the actual authentication to the auth form and capture the resulting
 * cookie for this session.
 *
 * @param username String username to use.
 * @param password String password to use.
 * @return boolean true if login was successful.
 * @throws MalformedURLException if the domainUrl/application is not a valid http URL.
 * @throws IOException if the request fails.
 */
public boolean login(String username, String password)
throws MalformedURLException, IOException {
 URL url = new URL(this.domainUrl + application);
 URLConnection uconn = url.openConnection();
 if( uconn instanceof HttpURLConnection ){
 HttpURLConnection http = (HttpURLConnection)uconn;
 HttpURLConnection.setFollowRedirects(false);
 http.setRequestMethod("POST");
 http.connect();
 http.getContent();
 this.cookies = http.getHeaderField( "Set-Cookie" );
 this.cookies = this.cookies.substring(0, this.cookies.indexOf(';'));
 //System.out.println( this.cookies );
 //this.displayHeader(http);

 url = new URL(this.domainUrl + application + securityCheckRoot);
 http = (HttpURLConnection)url.openConnection();
 http.setDoOutput( true );
 http.setRequestMethod("POST");
 http.setRequestProperty("Cookie", this.cookies);
 http.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
 String body = String.format("j_username=%s&j_password=%s", username, password);
 //System.out.println(body);
 http.setRequestProperty("Content-Length", Integer.toString(body.length()));
 //uconn.connect();
 OutputStream stream = http.getOutputStream();
 OutputStreamWriter osw = new OutputStreamWriter( stream );
 osw.write( body );
 osw.flush();
 osw.close();

 isLoggedIn = verifyLocation(http);
 //displayResponse(http);
 //displayHeader(http);

 url = new URL(this.domainUrl + application);
 http = (HttpURLConnection)url.openConnection();
 http.setRequestProperty("Cookie", this.cookies);
 http.getInputStream();
 }
 return isLoggedIn;
}

private boolean verifyLocation( HttpURLConnection connection ) throws IOException{
 connection.getInputStream();
```

```
   int rs = connection.getResponseCode();
   return ( rs == 302 );

  }
```

The same basic logic in C# looks like:

```
  public static void ConnectWebService(String authUsername, String authPassword, String
serviceName) {
      string serviceURL = "http:/69.20.70.158:8080/" + serviceName;
      string serviceAuthURL = "http://69.20.70.158:8080/"+serviceName+"/j_security_check";

      // Open auth window and get cookie
      HttpWebRequest request = (HttpWebRequest)WebRequest.Create(serviceURL);
      request.CookieContainer = new CookieContainer();
      HttpWebResponse response = (HttpWebResponse)request.GetResponse();
      response.Close();

      string myUsername = authUsername;
      string myPassword = authPassword;

      System.Text.ASCIIEncoding encoding = new System.Text.ASCIIEncoding();
      string postData = "j_username=" + myUsername;
      postData += ("&j_password=" + myPassword);
      byte[] data = encoding.GetBytes(postData);

      // Prepare web request...
      HttpWebRequest postRequest = (HttpWebRequest)WebRequest.Create(serviceAuthURL);
      postRequest.CookieContainer = request.CookieContainer;
      postRequest.Method = "POST";
      postRequest.ContentType = "application/x-www-form-urlencoded";
      postRequest.ContentLength = data.Length;
      System.IO.Stream newStream = postRequest.GetRequestStream();
      // Send the data.
      newStream.Write(data, 0, data.Length);
      newStream.Close();
      response = (HttpWebResponse)postRequest.GetResponse();
      response.Close();

      Console.WriteLine("Going in!");
```

```
    orderService = new OrderService.OMSOrderManagementService();
    orderService.CookieContainer = request.CookieContainer;

    orderServiceConnected = true;
  }
```

SOAP Data Management Service

The Data Management Service provides an API for managing general information, permissions, company and department data, user preferences, etc.

---

getWatchListIds

public java.lang.String[] **getWatchListIds**() throws DataManagementFault

Get an array of WatchList ids for the current user.

Returns:
String[] watchlist ids.

Throws:
DataManagementFault - on error.

---

getWatchList

public com.tradedesksoftware.etsdata.stock.WatchList **getWatchList**(int watchlistid) throws DataManagementFault

Get a WatchList with the given id.

Parameters:
watchlistid - int id of the watch list.

Returns:
WatchList the list with the given id.

Throws:

DataManagementFault - on error.

---

sendMessage

public int **sendMessage**(com.tradedesksoftware.etsdata.messages.Message message) throws DataManagementFault

Send a user message to the OMS.

Parameters:
message - Message to send.

Returns:
int id of new message object.

Throws:
DataManagementFault - on error.

---

getMessages

public com.tradedesksoftware.etsdata.messages.Message[] **getMessages**() throws DataManagementFault

Get user messages from OMS. This will return any existing undelivered messages.

Returns:
Message[] all outstanding undelivered user messages.

Throws:
DataManagementFault - on error.

---

getUserPreferences

public com.tradedesksoftware.etsdata.users.Preferences **getUserPreferences**() throws DataManagementFault

Get the preferences object associated with the current user.

Returns:

Preferences the current user's preferences.

Throws:

DataManagementFault - on error.

---

saveUserPreferences

public int **saveUserPreferences**(com.tradedesksoftware.etsdata.users.Preferences prefs) throws DataManagementFault

Save the current user's preferences.

Parameters:

prefs - Preferences the user's updated preferences.

Returns:

int id of user's Preferences object.

Throws:

DataManagementFault - on error.

---

getCompany

public com.tradedesksoftware.etsdata.users.group.Company **getCompany**(int id) throws DataManagementFault

Get information on a given company.

Parameters:

id - int id of the company to get data for.

Returns:

Company company data.

Throws:

DataManagementFault - on error.

---

saveCompany

public int **saveCompany**(com.tradedesksoftware.etsdata.users.group.Company ci) throws DataManagementFault

Add or update information for a company. If the company object has an id of 0 a new company will be added to the ets database. If the id corresponds to an existing company the record will be updated.

Parameters:

ci - Company the data to add or update.

Returns:

int id of the company object created/updated.

Throws:

DataManagementFault - on error.

---

getDepartment

public com.tradedesksoftware.etsdata.users.group.CompanyDivision **getDepartment**(int id) throws DataManagementFault

Get information for a department.

Parameters:

id - int id of the department.

Returns:

CompanyDivision data for the given department.

Throws:

DataManagementFault - on error.

---

saveDepartment

public int **saveDepartment**(com.tradedesksoftware.etsdata.users.group.CompanyDivision ci) throws DataManagementFault

Add or update a department. If the groupid of the department is 0 a new department will be added, otherwise the given department will be updated.

Parameters:

ci - CompanyDivision the data to be added/updated.

Returns:

int id of the CompanyDivision object.

Throws:

DataManagementFault - on error.

---

getUserProfile

public com.tradedesksoftware.etsdata.users.Profile **getUserProfile**(int userId) throws DataManagementFault

Get the profile data for a given user. Profile data consists of supplementary information for the given user (contact info, etc). This data is useful for business purposes but is not generally needed for trading.

Parameters:

userId - int id of the user to get profile data for.

Returns:

Profile the profile data for the given user.

Throws:

DataManagementFault - on error.

---

saveUserProfile

public int **saveUserProfile**(com.tradedesksoftware.etsdata.users.Profile up) throws DataManagementFault

Update the profile for a given user. All users have a profile and all profiles are linked to a user. The user id for the owner of the profile must exist.

Parameters:

up - Profile the profile data to update.

Returns:

int id of the Profile object.

Throws:

DataManagementFault - on error.

---

getUserPermissions

public com.tradedesksoftware.etsdata.users.Role[] **getUserPermissions**(int id) throws DataManagementFault

Get all role objects associated with the given user.

Parameters:

id - int id of the user.

Returns:

Role[] all roles the given user has.

Throws:

DataManagementFault - on error.

---

setUserPermissions

public void **setUserPermissions**(int id,java.lang.Integer[] roleids) throws DataManagementFault

Associate a group of role ids with the given user id.

Parameters:

id - int id of user to associate roles with.

roleids - Integer[] ids of the roles to associate.

Throws:

DataManagementFault - on error.

---

saveFullUser

```
public int saveFullUser(com.tradedesksoftware.etsdata.users.User ui,
                        com.tradedesksoftware.etsdata.users.Profile up,
                        java.lang.Integer[] roleids)
                 throws DataManagementFault
```

Update or create a user. The user object, profile, and roles the user should be associated with will be added to the ets database or updated.

Parameters:

ui - User user object for the new/updated user.
up - Profile extended user information to update/add.
roleids - Integer[] list of roles the user should be associated with.

Returns:

int id of the User.

Throws:

DataManagementFault - on error.

---

getUser

public com.tradedesksoftware.etsdata.users.User **getUser**(int id) throws DataManagementFault

Get a user with the given id.

Parameters:

id - long id of user

Returns:

UserInfo info for the user

Throws:

DataManagementFault - if access is forbidden

---

saveUser

public int **saveUser**(com.tradedesksoftware.etsdata.users.User ui) throws DataManagementFault

Create or update a user. Note that this shouldn't be used to create new users since no profile information or roles will be associated with the user. It is more efficient than saveFullUser() if profile and role information is not being changed.

Parameters:

ui - User user to create or update.

Returns:

long id of user.

Throws:

DataManagementFault - on error.

---

getCurrentUser

public com.tradedesksoftware.etsdata.users.User **getCurrentUser**() throws DataManagementFault

Get user info for the currently authenticated user.

Returns:

User authenticated user

Throws:

DataManagementFault

---

getCurrentDepartment

public com.tradedesksoftware.etsdata.users.group.CompanyDivision **getCurrentDepartment**() throws DataManagementFault

Get the department of the currently authenticated user.

Returns:

CompanyDivision current user's department.

Throws:

DataManagementFault

---

getTopLevelUsers

public com.tradedesksoftware.etsdata.users.User[] **getTopLevelUsers**(int id, boolean showDeleted) throws DataManagementFault

Get the top level (Company) users for a given company. If showDeleted is true then records market deleted will be returned, otherwise they will not.

Parameters:

id - long id of company.

showDeleted - boolean if true deleted records will be returned.

Returns:

User[] of user descriptions.

Throws:

DataManagementFault - if access is forbidden

---

getPrimaryRole

public java.lang.String **getPrimaryRole**() throws DataManagementFault

Get the name of the most permissive trading role the current user has. This is useful when a client application wants to present role-specific options.

Returns:

String name of most permissive role for this user.

Throws:

DataManagementFault

---

getAllRoles

public com.tradedesksoftware.etsdata.users.Role[] **getAllRoles**() throws DataManagementFault

Get all roles defined on this system. This will return an array of Role objects contained in the ets database.

Returns:

Role[] an array of all available roles.

Throws:

DataManagementFault - on error.

---

getAllCompanies

public com.tradedesksoftware.etsdata.users.group.Company[] **getAllCompanies**(boolean showDeleted) throws DataManagementFault

Get all companies visible to the current user.

Parameters:
showDeleted - boolean if true show deleted records.

Returns:
CompanyInfo[] all visible companies

Throws:
DataManagementFault

---

getCompanyDepartments

public com.tradedesksoftware.etsdata.users.group.CompanyDivision[] **getCompanyDepartments**(int id,boolean showDeleted) throws DataManagementFault

Get all departments visible to the current user in the given company.

Parameters:
id - long id of company
showDeleted - boolean if true show deleted records

Returns:
DepartmentInfo[] all visible departments

Throws:
DataManagementFault - if user is denied access to the given company entirely

---

getDepartmentUsers

public com.tradedesksoftware.etsdata.users.User[] **getDepartmentUsers**(int id,
boolean showDeleted) throws DataManagementFault

Get all users visible to the current user in the given department.

Parameters:

id - long id of department

showDeleted - boolean if true show deleted records

Returns:

UserInfo[] all visible users

Throws:

DataManagementFault

---

getSubDepartments

public com.tradedesksoftware.etsdata.users.group.CompanyDivision[] **getSubDepartments**(int id,boolean showDeleted) throws DataManagementFault

Get all subdepartments of the given department visible to the user

Parameters:

id - long id of parent department

showDeleted - boolean if true show deleted records

Returns:

DepartmentInfo[] all visible departments

Throws:

DataManagementFault

SOAP Market Data Service

The market data service is mainly used to access information on instruments (symbol master), feeds, and markets. It is also possible to request market data updates and poll for current market data. This last feature is not exceptionally efficient but can be useful in situations where the client simply needs to display a static snapshot or low update frequency data view. Currently requesting market data updates is deprecated, clients should utilize the websockets or native binary interfaces of the Feed Handler instead.

getRoutes

public Route[] **getRoutes**(String symbol) throws SymbolMasterException, RemoteException

Return a set of Route objects describing all the possible sources of market data available for the given symbol.

Parameters:

symbol - String ticker symbol to search for routes to.

Returns:

Route[] all available routes for the given data.

Throws:

SymbolMasterException

RemoteException

---

getExchanges

public Exchange[] **getExchanges**() throws RemoteException

Get all exchanges known to the trading system.

Returns:

Exchange[] all known exchanges.

Throws:

RemoteException

---

updateExchange

public int **updateExchange**(Exchange exchange) throws SymbolMasterException, RemoteException

Update or add an exchange to the known exchanges. Returns the id assigned to the exchange. If this is an existing exchange the id will be the same as before, otherwise it will be a newly assigned id for the new exchange.

Parameters:

exchange - Exchange the data for the exchange.

Returns:

int id of the exchange.

Throws:

SymbolMasterException

RemoteException

---

updateSymbolInfo

public int **updateSymbolInfo**(Instrument info) throws SymbolMasterException, RemoteException

Update the symbol master data for an instrument, or add a new instrument to the symbolmaster. Returns the unique internal id of the symbol.

Parameters:

info - Instrument the data to add/update.

Returns:

int id of the instrument.

Throws:

SymbolMasterException

RemoteException

---

getInstruments

public Instrument[] **getInstruments**(int[] ids) throws RemoteException

Get a group of instruments in one call. This is highly efficient for instance when fetching all the instrument definitions related to a watch list.

Parameters:

ids - int[] ids of instruments to get data for.

Returns:

Instrument[] array containing all the instruments.

Throws:

java.rmi.RemoteException - on faiure.

---

findSymbolInfo

public Instrument[] **findSymbolInfo**(SymbolFilter filter) throws RemoteException, SymbolMasterException

Get symbol info for all instruments which match the given filter. This allows for searching the symbol master on various criteria. The filter allows setting of various search parameters.

Parameters:

filter - SymbolFilter the search criteria.

Returns:

Instrument[] all instruments matching the search.

Throws:

RemoteException

SymbolMasterException

---

subscribe (deprecated)

public void **subscribe**(SubscriptionDTO subscription) throws MarketDataException

Create a subscription to one instrument for this client.

Parameters:

subscription - SubscriptionDTO containing the parameters for the subscription

Throws:

MarketDataException - if the symbol cannot be subscribed due to invalidity, unavailability or lack of client permission.

---

unsubscribe (deprecated)

public void **unsubscribe**(SubscriptionDTO subscription) throws MarketDataException, NoSuchSubscriptionException

Destroy a subscription to one instrument for this client.

Parameters:

subscription - SubscriptionDTO containing the parameters for the subscription to remove

Throws:

MarketDataException - if an error occurs in the market data service

NoSuchSubscriptionException - if the client is not currently subscribed to the symbol

---

subscribe (deprecated)

public void **subscribe**(SubscriptionDTO[] subs) throws MarketDataException, NoSuchSubscriptionException

Given an array of subscribe/unsubscribe requests process all of them. Note that currently an exception is thrown if ANY one of the subscriptions fails. This can create a problem knowing which request in the group caused the problem, and which ones were processed successfully. This behaviour will need to be cleaned up at some point.

Parameters:

subs - SubscriptionDTO[] array of subscribe/unsubscribe requests

Throws:

MarketDataException - if a subscribe or unsubscribe fails for any of a variety of reasons

NoSuchSubscriptionException - if an unsubscribe references a symbol/level which has not been previously subscribed by this client.

---

getLevel1Updates (deprecated)

public SymbolData[] **getLevel1Updates**() throws MarketDataException

Poll for level 1 updates and return an array of SymbolData objects. Each entry in the array contains updated information for any symbol which has been updated since the same client, identified via http session, last called this method.

Returns:

SymbolData[] array of symbol data containing all symbols which have been updated.

Throws:

MarketDataException

getLevel2Updates (deprecated)

public OrderData[] **getLevel2Updates**() throws MarketDataException

Poll for level 2 updates and return an array of OrderData objects. Each entry in the array contains updated information for a particular level 2 row. Deleted rows will be sent with size=0, indicating the client should discard the corresponding row in its level 2 display.

Returns:

OrderData[] array of order data containing all updated rows for all subscribed symbols.

Throws:

MarketDataException

doLevel1 (deprecated)

public SymbolData[] **doLevel1**(SubscriptionDTO[] subs) throws MarketDataException, NoSuchSubscriptionException

Combines the functionality of batch subscribe and getLevel1Updates. This is faster and more efficient and avoids potential problems with clients issuing multiple overlapping SOAP requests for subscription and update service.

Due to the asyncronous nature of market data updates vs subscriptions it is quite likely the client will not receive an immediate update for symbols which have been subscribed in the same request. There is no particular way to determine beforhand how long the delay between subscription and data reception may be, some feeds provide updates immediately, others may only provide an initial snapshot after some delay.

Parameters:

subs - SubscriptionDTO[] Array of subscribe/unsubscribe messages. May be null.

Returns:

Symbol[] Array of symbols which have been updated.

Throws:

MarketDataException - if an error occures in the market data service
NoSuchSubscriptionException - if a request is made to unsubscribe from a symbol not subscribed to

---

doLevel2 (deprecated)

public OrderData[] **doLevel2**(SubscriptionDTO[] subs) throws MarketDataException, NoSuchSubscriptionException

This provides data and subscription functions for level 2 data. It is essentially the same as the doLevel1() method, except it polls for level 2 data.

Parameters:

subs - SubscriptionDTO[] subscriptions, may be null.

Returns:

OrderData[] all updated level 2 data.

Throws:

MarketDataException
NoSuchSubscriptionException

SOAP Order Management Service

The order management service provides functions for placing, replacing, and canceling orders. In addition it exposes functions which manage accounts, destinations, positions, recover status of orders, etc.

getAccountPositions

public Position[] **getAccountPositions**(int acctid) throws RemoteException

Get all positions associated with a given account.

Parameters:

acctid - MARX id of the account.

Returns:

Position[] all open positions associated with this account

Throws:

RemoteException - on failure to process request

---

getCompanyAccounts

public Account[] **getCompanyAccounts**(int companyid) throws OMSOrderServiceFault, RemoteException

Get all accounts owned by the given company.

Parameters:

companyid - MARX id of the company

Returns:

Account[] all accounts owned by the company.

Throws:

OMSOrderServiceFault - if the request cannot be filled. Usually a permission error

RemoteException - on failure to process request

---

getCompanyDestinationInfo

public DestinationInfo[] **getCompanyDestinationInfo**(int companyid) throws RemoteException

Get Destination information on all destinations which the given company has permission to route orders to.

Parameters:

companyid - MARX id of the company

Returns:

DestinationInfo[] info on each destination

Throws:

RemoteException - on failure to process request

---

getAllDestinationInfo

public DestinationInfo[] **getAllDestinationInfo**() throws RemoteException

Get destination information for all destinations configured for this system.

Returns:

DestinationInfo[] info on each destination

Throws:

RemoteException - on failure to process request

---

getDestinationInfo

public DestinationInfo **getDestinationInfo**(int destinationid) throws OMSOrderServiceFault, RemoteException

Get information on a specific destination.

Parameters:

destinationid - int id of the destination.

Returns:

DestinationInfo information on the given destination.

Throws:

OMSOrderServiceFault - business level error, usually permissions.
RemoteException - on failure to process request.

---

getUserDestinationInfo

public DestinationInfo[] **getUserDestinationInfo**() throws OMSOrderServiceFault, RemoteException

Get information on all destinations the current user is permitted to route orders to.

Returns:

DestinationInfo[] info on each destination

Throws:

OMSOrderServiceFault - business level error, usually permissions.
RemoteException - on failure to process request.

---

placeOrder

public LogEntry **placeOrder**(Order order) throws OMSOrderServiceFault, RemoteException

Place a new order.

Parameters:
order - Order the order to place.
Returns:
LogEntry currently this is always null.
Throws:
OMSOrderServiceFault - on business level error.
RemoteException - on failure.

---

replaceOrder

public LogEntry **replaceOrder**(Order replacement) throws OMSOrderServiceFault, RemoteException

Replace an existing order.

Parameters:
replacement - Order the replacement order.
Returns:
LogEntry currently always null.
Throws:
OMSOrderServiceFault - on business level error.
RemoteException - on failure.

cancelOrder

public LogEntry **cancelOrder**(Order order) throws OMSOrderServiceFault,
RemoteException

Cancel an existing order.

Parameters:
order - Order the order to be canceled.

Returns:
LogEntry currently always null.

Throws:
OMSOrderServiceFault - on business level error.
RemoteException - on failure.

---

dockOrder

public LogEntry **dockOrder**(Order order) throws OMSOrderServiceFault,
RemoteException

Dock an order. Docked orders are entered into the database but are not routed.

Parameters:
order - Order the new order.

Returns:
LogEntry currently always null.

Throws:
OMSOrderServiceFault - on business level error.
RemoteException - on failure.

---

undockOrder

public LogEntry **undockOrder**(Order order) throws OMSOrderServiceFault,

RemoteException

Undock an order. A previously docked order is released.

Parameters:

order - Order the new order.

Returns:

LogEntry currently always null.

Throws:

OMSOrderServiceFault - on business level error.

RemoteException - on failure.

---

getUserActiveOrders

public Order[] **getUserActiveOrders**() throws RemoteException

Get all 'active' orders owned by the current user. These are orders which are not currently filled, canceled, or rejected etc and should be active.

Returns:

Order[] active orders.

Throws:

RemoteException - on failure.

---

getOrderExecutions

public Execution[] **getOrderExecutions**(int orderid) throws RemoteException

Get all execution for a given order id.

Parameters:

orderid - int internal order id of order to get executions for.

Returns:

Execution[] executions for the given order.

Throws:

RemoteException - on failure.

setOrderEventSource

public void **setOrderEventSource**(String etype) throws OMSOrderServiceFault, RemoteException

Indicate to the OMS subscription manager what scope of order status/execution data the client is interested in. There are 3 possible scopes. "COMPANY" returns executions and order status for all active orders owned by the user's company. "DEPARTMENT" returns executions and order status for all orders placed within the user's department. "USER" returns executions and order status for active orders owned by the current user. Once this method is called the subscription manager will forward order status for all active orders in the scope and all related executions to the client's queue.

Parameters:

etype - String, one of COMPANY, DEPARTMENT, or USER.

Throws:

OMSOrderServiceFault
RemoteException

getOrderEvents

public ETSEventCollection **getOrderEvents**() throws OMSOrderServiceFault, RemoteException

Get all events queued for this client since the last call. setOrderEventSource(String) should be called to define the scope of events requested and announce the client to the OMS Subscription Manager.

Throws:

OMSOrderServiceFault
RemoteException

getOrder

public Order **getOrder**(int id) throws RemoteException

Get the current state of an order the given id.

Throws:

RemoteException

getDestination

public Destination **getDestination**(int id) throws RemoteException

Get basic information on a given destination.

Throws:

RemoteException

getAccount

public Account **getAccount**(int id) throws RemoteException

Get basic information on a given account.

Throws:

RemoteException

getUserDestinations

public Destination[] **getUserDestinations**() throws RemoteException

Get all destinations the current user is permitted to route order to.

Throws:

RemoteException

---

findAccounts

public Account[] **findAccounts**(AccountFilter filter) throws RemoteException

Look up accounts which match a given set of search criteria.

Throws:

RemoteException

---

getUserAccounts

public Account[] **getUserAccounts**() throws RemoteException

Get all accounts which the current user is permitted to view.

Throws:

RemoteException

---

findOrders

public Order[] **findOrders**(OrderFilter filter) throws RemoteException

Get orders which match a set of search parameters.

Throws:

RemoteException

getCandidateExchanges

public int[] **getCandidateExchanges**(String symbol) throws RemoteException

Get a list of all exchanges which list a given symbol.

Throws:
RemoteException

SOAP Schema

MARX utilizes a number of complex types in SOAP which the client will need to deal with. These types also mostly correspond directly to server-side Java classes. The following types and their fields are exposed.

Order

An order instance holds all the information required to define an order which can be routed to a destination and holds basic state information for the order. The following fields are present in the order type:

Name	Type	Notes
accountId	xs:int	MARX internal id of the account this order is trading against.
avgprice	xs:decimal	Average price of all executions on this order to date.
capacity	tns:capacities	Capacity for this order. Note: this is largely obsolete/only relevant to certain equity markets.
companyId	xs:int	Id of the company to which the trader placing the order belongs. Note that this field generally will be filled in automatically and can typically be left

		undefined.
cpOrderId	xs:string	The order id assigned by the sell-side system. This is typically not assigned by the client and is useful for display or debugging purposes (IE correlating to orders on the sell-side).
creationTime	Xs:long	Unix epoch time value at which the order was created with millisecond precision. The client should set this to the time at which they created the order.
cumqty	xs:int	Cumulative executed quantity for this order.
destinationid	xs:int	MARX destination id of the destination this order is being routed to.
dispQty	xs:int	Display quantity for this order. A non-zero value indicates what quantity of the order will be displayed on the book by the sell-side when they support this feature. Should be a value less than the quantity.
expireTime	xs:long	The time of expiration for orders with timesInForce value of GTD. The value is expressed in Unix epoch time with millisecond precision.
id	xs:int	MARX internal id for this order. This will be set by the OMS.
instrument	tns:instrument	The instrument being ordered. Note that the client only needs to supply either the marketid and symbol fields OR the instrumentid value, other fields can be left undefined.
lastState	tns:orderStatuses	The status of this order prior to the last state change. For new orders this is undefined.
leavesqty	xs:int	Remaining quantity of the order on the sell-sides book.

market	xs:int	MARX market/exchange id of the market this order is being routed to.
notes	xs:string	This field is persisted with the order by the OMS and may be passed through to sell-side systems which provide this feature. The OMS ignores this field and simply passes it back to the client for its own use.
offset	xs:decimal	Pegging offset. The exact semantics of this feature depend on the type of order, execution instructions, and sell-side logic. An order containing an offset will generally be assumed to be a pegged order (IE if it is a LIMIT, STOP, or STOP_LIMIT type).
orderId	tns:externalID	Id assigned to the order by the OMS which is published to the sell-side as ClOrdID or equivalent. Client should leave this value undefined on order submission.
price	xs:decimal	Price for this order. This is generally the LIMIT price. Complex order types may have different semantics.
quantity	xs:int	Order quantity. The type of units represented depend on the type of instrument being traded. In the case of FOREX this is a currency amount in the currency of the account (typically USD).
relatedOrderId	xs:int	MARX internal order id of a related order. This should be set to the id of the order being canceled or replaced in the case of a cancel/replace operation. Complex orders may also use this field to indicate an order is a leg.
side	tns:sides	The side for this order.
state	tns:Orderstatuses	Current status of this order.

stopPrice	xs:decimal	Stop price for a STOP or STOP_LIMIT order. Some complex order types may use this in various ways.
ticketId	xs:int	MARX internal ticket id of the customer ticket this order is associated with. Typically this value is left undefined by the client. The OMS will construct a dummy ticket. Currently there is no facility for clients to construct tickets.
tif	tns:timesInForce	Time in force value for this order. If undefined the order is assumed to be a DAY order in most cases.
type	tns:orderTypes	The type of order.
ownerId	tns:int	The id of the trader who placed this order.
groupId	tns:int	The id of the department of the trader who placed this order.
custOrderId	tns:string	Customer's id for this order. The customer/client application can supply it's own unique identifier for the order in this field if desired. This value will be persisted but is not used internally by the OMS.
execInst	tns:ExecutionInstructions	An execution instruction for this order. Some markets allow or require a value in order to access certain vendor-specific features. The exact meaning of any given value may depend on the vendor and the type of order.
relatedClOrdId	xs:string	The value of the orderId field for a related order. This can be used in lieu of the relatedOrderId to identify a related order. Otherwise it facilitates tracking of order chains with the sell-side.
quoteId	xs:string	Id of a vendor quote this order is in response to. This may need to be supplied in some trading models. Note that MARX

		currently does not implement direct support for this model.

Execution

An execution represents a state transition of an order. Usually initiated in response to messages sent to the OMS order gateway by the sell-side. In some cases the OMS may initiate internal executions in response to business logic.

Name	**Type**	**Notes**
accountNumber	xs:string	The sell-side account number of the account this execution is booked against.
avgPrice	xs:decimal	Average price for all executions on this order up to and including this one.
broken	xs:broken	True if this execution is a broken trade. In this case another execution will be related to this one which provides a restatement.
companyId	xs:int	Id of the company the order for this execution belongs to.
cpOrderid	xs:string	The id of the order for this execution on the sell-side.
cumQuantity	xs:int	Cumulative quantity of the order executed up to and including this execution.
destinationid	xs:int	The id of the destination this execution was received from.
execid	xs:string	The id of this execution on the sell-side.
finalState	tns:orderStatuses	The state of the order after this execution.
groupId	xs:int	The id of the department of the trader placing this order.
historical	xs:boolean	This execution is part of a set of historical executions. Normally when a client connects to the system they will be sent executions for existing orders with this flag set. These can be used for display purposes or simply discarded.

id	xs:int	MARX internal id for this execution. These ids are always globally unique within the trading facility.
initialState	tns:orderStatuses	The state of the order before this execution was applied.
instrument	tns:instrument	The instrument of this execution's order.
lastPrice	xs:decimal	The price of the execution.
lastQuantity	xs:int	The quantity of the execution.
leavesQuantity	xs:int	The remaining unfilled quantity of the order. Note that executions which don't represent a fill generally do not define this value.
message	xs:string	Any text returned by the sell-side associated with this execution. This field generally holds error messages when an execution is rejecting an order. Internal rejects will supply a relevant explanatory text.
orderid	xs:int	The MARX internal id of the order this execution relates to.
origOrderId	tns:externalID	The orderId of the original order this execution relates to. Generally only set for cancel/replace operations.
ourOrderId	tns:externalID	The orderId of the order this execution relates to.
ownerId	xs:int	The id of the trader who placed the order.
possdup	xs:boolean	An execution which may have been previously transmitted to the client. The client should compare the id of the execution to others which have already been processed in order to determine whether or not this is actually a duplicate.
relatedExecId	xs:string	Sell-side execution id of an execution which this execution is restating etc.
relatedOrderId	xs:int	MARX internal id of any order related to the

		order this execution is for.
relatedid	xs:int	MARX internal id of an execution which this execution is restating etc.
Side	Tns:sides	Side of the order.
Time	Xs:long	Unix epoch time when this execution took place with millisecond resolution.
type	tns:executionTypes	Type of execution. This indicates what activity this execution is reporting.

Instrument

The instrument type represents a currency pair or other unique tradeable instrument known to the system.

Name	**Type**	**Notes**
type	tns:instrumentTypes	Type of instrument.
id	xs:int	MARX instrument id of this instrument.
symbol	xs:string	Symbol for this instrument. Note that symbols by themselves may not be unique, they are only unique in combination with a market id.
baseSymbol	xs:string	Portion of the symbol which identities the root instrument, such as the contract type on a future (IE '6E' is the root of '6EM8').
exchangeid	xs:int	MARX market/exchange id of the exchange this instrument is listed on. (In the case of FOREX this is the same as the id of the market being traded on).
underlyingId	xs:int	The id of an instrument which underlies this instrument. This could be a stock underlying an option. A FOREX instrument with an underlying id is a reverse pair of a common pair (IE USDEUR, the reverse of EURUSD). The OMS usually normalizes reverse pairs and they may not be configured on a given system at all.

entry	xs:complextype	0 or more additional attributes applying to this instrument.
key	xs:string	Each entry has a unique key identifying the type of attribute.
value	xs:string	Value of the corresponding key.

ExternalID

ExternalIDs represent order and execution ids which the OMS will supply to sell-side systems. An ExternalID is an xs:complextype containing a single field, stringValue of type xs:string.

Account

An account type represents a trading or customer account.

Name	**Type**	**Notes**
balance	xs:decimal	The current balance of the account in the currency type of the account.
companyId	xs:int	The id of the company which owns this account.
currency	xs:string	The ISO currency code of the currency this account is denominated in.
id	xs:int	The id of this account.
lastUpdate	xs:long	Unix epoch time with millisecond precision when this account was last updated.
limit	xs:decimal	
margin	xs:decimal	
maxordersize	xs:int	Fat finger limit for max single order size for this account.
maxposlimit	xs:int	Sanity check limit for the total size of all open positions for this account.
name	xs:string	Human readable name for this account.
number	xs:string	Account number by which this account is known to the outside world.
pandl	xs:decimal	Total realized P&L for this account since the last

		update.
reuseFund	xs:decimal	
type	tns:accountTypes	Type of account.

Destination

A destination represents an endpoint to which orders can be directed. The destination type provides some basic identifying information for each destination.

Name	Type	Notes
canCreate	xs:boolean	When true this indicates a destination which can originate orders.
enabled	xs:boolean	True if this destination is enabled for trading.
gateway	xs:string	Name of the MARX gateway handling connectivity for this destination.
id	xs:int	Destination id.
name	xs:string	Human readable description of the destination.

DestinationInfo

This is a DTO which is used to pass a list of market type instances to the client. Each contained market instance describes one market/exchange which the given destination supports.

Name	Type	Notes
destinationid	xs:int	Id of the destination.
markets	tns:market	Collection of markets which are accessible via this destination.

Market

Represents an exchange, LP, ECN etc at which orders may be placed.

Name	Type	Notes
accountids	xs:int	Ids of all accounts which can trade on the given

		market.
aon	xs:boolean	True if the market supports AllOrNone.
description	xs:string	Human readable description of this market/destination route.
destinationid	xs:int	Id of the destination used to access this market. Note that a given market could show up at different destinations, but it will only show up once in the context of any one destination.
instrumentTypes	xs:int	Instrument type values for each valid type of instrument which can be routed to this market.
marketid	xs:int	Id of the market.
orderTypes	xs:int	Order type values for all valid order types supported by this market.
sides	xs:int	Side type values for all valid sides supported by this market.
tifs	xs:int	TIF values which are supported by this market.

Position

Represents an open position on an account.

Name	Type	Notes
accountid	xs:int	Id of the account this position pertains to.
instrument	tns:instrument	Instrument this position is in.
price	xs:decimal	Average entry price for this position.
quantity	xs:int	Size of the position. Short positions will have a negative size value.

ETSLogEvent

Log events are returned by getOrderEvents() in case of a basic business level rejection of an order. In this case the order is never instantiated or allocated an order id by the OMS Order Handler. A logEntry describing the error and a copy of the order are contained in the logEntry and order elements respectively.

LogEntry

A log entry object provides information related to an order rejected by the OMS Order Handler or a similar event.

Enumerations

The following enumerations (restrictions) are common to many of the above objects.

Enumeration Name	**Enumeration Value**	**String value**
capacities	0	AGENCY
	1	PROPRIETARY
	2	INDIVIDUAL
	3	PRINCIPAL
	4	RISKLESS_PRINCIPAL
	5	AGENT_FOR_OTHER_MEMBER
instrumentTypes	0	STOCK
	1	STOCKOPTION
	2	FUTURE
	3	FOREIGNEXCHANGE
	4	OPTION
orderStatuses	0	NEW
	1	CANCELLED
	2	DOCKED
	3	DONEFORDAY
	4	EXPIRED
	5	FILLED
	6	OPEN
	7	PARTIALLYFILLED
	8	PENDING
	9	PENDINGCANCEL

	10	PENDINGREPLACE
	11	REJECTED
	12	REPLACED
	13	SENDING
	14	HALTED
sides	0	BUY
	1	SELL
	2	BUYMINUS
	3	SELLPLUS
	4	SELLSHORT
	5	SELLSHORTEXEMPT
	6	UNDISCLOSED
	7	CROSS
	8	CROSSSHORT
	9	CROSSSHORTEXEMPT
	10	ASDEFINED
	11	OPPOSITE
	12	SUBSCRIBE
	13	REDEEM
	14	LEND
	15	BORROW
timesInForce	0	DAY
	1	GOOD_TILL_CANCEL
	2	AT_THE_OPENING
	3	IMMEDIATE_OR_CANCEL
	4	FILL_OR_KILL
	5	GOOD_TILL_CROSSING
	6	GOOD_TILL_DATE

	7	AT_THE_CLOSE
orderTypes	0	MARKET
	1	LIMIT
	2	STOP
	3	STOP_LIMIT
	4	MARKET_ON_CLOSE
	5	WITH_OR_WITHOUT
	6	LIMIT_OR_BETTER
	7	LIMIT_WITH_OR_WITHOUT
	8	ON_BASIS
	9	ON_CLOSE
	10	LIMIT_ON_CLOSE
	11	FOREX_MARKET
	12	PREVIOUSLY_QUOTED
	13	PREVIOUSLY_INDICATED
	14	FOREX_LIMIT
	15	FOREX_SWAP
	16	FOREX_PREVIOUSLY_QUOTED
	17	FUNARI
	18	MARKET_IF_TOUCHED
	19	MARKET_WITH_LEFTOVER_AS_LIMIT
	20	PREVIOUS_FUND_VALUATION_POINT
	21	NEXT_FUND_VALUATION_POINT
	22	PEGGED
	23	ONE_CANCELS_OTHER
	24	IF_DONE
	25	IF_DONE_ONE_CANCELS_OTHER
exectionInstructions	0	NOT_HELD

1	WORK
2	GO_ALONG
3	OVER_THE_DAY
4	HELD
5	PARTICIPATE_DONT_INITIATE
6	STRICT_SCALE
7	TRY_TO_SCALE
8	STAY_ON_BIDSIDE
9	STAY_ON_OFFERSIDE
10	NO_CROSS
11	OK_TO_CROSS
12	CALL_FIRST
13	PERCENT_OF_VOLUME
14	DO_NOT_INCREASE
15	DO_NOT_REDUCE
16	ALL_OR_NONE
17	REINSTATE_ON_SYSTEM_FAILURE
18	INSTITUTIONS_ONLY
19	REINSTATE_ON_TRADING_HALT
20	LAST_PEG
21	MID_PRICE
22	NON_NEGOTIABLE
23	OPENING_PEG
24	MARKET_PEG
25	CANCEL_ON_SYSTEM_FAILURE
26	PRIMARY_PEG
27	SUSPEND
28	FIXED_PEG_TO_BBO

	29	CUSTOMER_DISPLAY_INSTRUCTION
	30	NETTING
	31	PEG_TO_VWAP
	32	TRADE_ALONG
	33	TRY_TO_STOP
	34	CANCEL_IF_NOT_BEST
	35	TRAILING_STOP_PEG
	36	STRICT_LIMIT
	37	IGNORE_PRICE_VALIDITY_CHECKS
	38	PEG_TO_LIMIT_PRICE
	39	WORK_TO_TARGET_STRATEGY
accountTypes	0	RETAIL
	1	WHOLESALE
	2	PROPRIETARY
	3	EMPLOYEE
	4	COMBINED
exceptionTypes	0	COMMUNICATIONERROR
	1	GATEWAYERROR
	2	NOORDER
	3	ALREADYPROCESSED
	4	UNSUPPORTED
	5	OUTOFRANGE
	6	INVALIDPARAMETER
	7	ACCESSDENIED
	8	ILLEGALORDERSTATE
	9	OMSERROR
	10	NOTAVAILABLE
executionTypes	0	NEW

1	DONE_FOR_DAY
2	CANCELED
3	REPLACED
4	PENDING_CANCEL
5	REJECTED
6	PENDING_NEW
7	TRADE
8	TRADE_CORRECT
9	TRADE_CANCEL
10	ORDER_STATUS
11	STOPPED
12	SUSPENDED
13	RESTATED
14	CALCULATED

REST Interface

The REST interface is largely similar to the SOAP version. Many of the functions require input parameters. In the case of functions with no input parameters, or a single simple scalar input which are idempotent (do not change the state of the OMS in any way) a simple GET request with any parameter encoded into the URL is used. In cases where more complex input is required the function will accept a POST with the data JSON encoded as described in the following schema section. Simple return values will be likewise returned as basic text in the body of the response, complex data types will be JSON encoded in the response.

Security is provided in exactly the same fashion as documented in the SOAP section, when utilizing a web browser no special considerations need to be made, the browser will display the login form when the client connects for the first time, and then redirect the request to the proper function if acceptable credentials are provided. Pure Javascript clients or other types of clients using REST will need to authenticate in the same fashion as documented for SOAP clients.

The REST interface is broken down into the same three overall sections as the SOAP interface, Market Data, Data Management, and Order Services. Each section's functions are described below. Each section's functions begin with a unique base path.

General API Information

The MARX REST API is a JSON-based API built on the RESTEasy (http://www.jboss.org/resteasy) implementation of the JAX-RS specification. There are three main API entry points on a running MARX server. Presuming “127.0.0.1” as the server address, they are located as follows:

- OMSOrderManagementRESTServices: http://127.0.0.1:8080/omsrestservices/
    - Provides order-related services such as order placement and execution retrieval.
- DataManagementRESTServices: http://127.0.0.1:8080/etsdatamanagement/
    - Provides user-related services such as retrieving user permissions and managing accounts.
- MarketDataRESTServices:http://127.0.0.1:8080/etsmarketdata/
    - Provides market-related services such as retrieving symbol lists or markups for a given market.

HTML5/Javascript

The raw javascript API scripts generated by RESTEasy can be accessed by appending “rest-js” to any of these services.

For convenience and performance reasons, the raw scripts have been cached in the */javascript* folder of a running server, and a utility script - */javascript/marx.core.js* - has been created to ease access to the APIs via simple javascript calls:

```
useUserService().methodToCall();
useOrderService().methodToCall();
useMarketDataService().methodToCall();
```

These convenience methods simply reset a global javascript value - REST.apiURL - to point to the appropriate REST endpoint, as RESTEasy does not automatically do this in a multi-endpoint environment.

Other Languages

The REST endpoints can be used with various REST-based libraries or simply by GET and POST requests directly to the endpoints from any HTML client. All API calls should be prefixed with “rest”, i.e. http://127.0.0.1:8080/etsdatamanagement/rest/watchlist/ids to access the “/watchlist/ids” path.

Important: as the REST endpoints are located behind the JBoss application server, authentication and security must be handled as detailed in "Authentication and State Management" in the MARX Programming Guide. Specifically, if attempting to access the REST services from outside the JBoss server, you will need to obtain a JSESSIONID from a cookie header returned by the JBoss server on the initial call, perform user authentication, and provide the JSESSIONID in a header with every following request to the API. If your application is contained inside a .war or .war folder under the "/deploy/" directory in the JBoss installation location, then these steps will be unnecessary.

Getting Started - HTML5/Javascript

In order to use the API in a typical web page, you'll need to do the following:

Add the following to the <head> section of your page:

```
<script type="text/javascript" src="/javascript/rest/omsservices.js"></script>
<script type="text/javascript" src="/javascript/rest/etsdatamanagement.js"></script>
<script type="text/javascript" src="/javascript/rest/etsmarketdata.js"></script>
```

It is strongly recommended that you also include the following helper scripts, all of which require a current jQuery library to be included prior:

```
<script type="text/javascript" src="/javascript/pubsub.js"></script>
<script type="text/javascript" src="/javascript/marx.core.js"></script>
<script type="text/javascript" src="/javascript/marx.marketdata.core.js"></script>
<script type="text/javascript" src="/javascript/marx.orders.core.js"></script>
```

With these included, you can then make API calls in normal <script> blocks or included javascript as follows:

```
useUserService().getCurrentUser();
useOrderService().placeOrder({$entity: orderToSubmit});
useMarketDataService().getInstruments({$entity: instrumentIDArray});
```

For API calls which take objects (like orders or search filters) as arguments, RESTEasy requires that they be passed to the API script as a "$entity" field on a new javascript object, as in the examples above. For API calls which take simple primitive arguments, typically you will use the following format instead.:

```
useOrderService().getCompanyAccounts({'id':thisCompany.id});
```

Convenience Scripts - HTML5/Javascript

There are several scripts which provide a number of functions that will likely need to be created in any HTML/javascript project relying on the API.

pubsub.js

This is an open-source jQuery pub/sub plugin by Peter Higgins (dante@dojotoolkit.org) which allows you to publish to arbitrary "channels" and create callback methods to subscribe to events on those channels. Used by the WebSocket marketdata client in marx.marketdata.feeder.js and the REST order event poller in marx.orders.core.js.

marx.core.js

This script provides a number of helper functions in addition to the "use*Service()" methods detailed previously:

- **createUUID()** - returns a string containing a RFC4122-compatible UUID which can be used for various purposes, such as guaranteeing a unique client-generated order ID for a new order.
- **addCommas(nStr,n)** - for an input number "nStr", returns a string truncated to "n" decimals and formated with commas for pretty presentation.
- **loadUserInfo()** - populates a global "**userInfo**" variable with user information for the signed-in user.
- **Instrument(id, type, symbol, exchangeid)** - creates an Instrument object which conforms to the MARX definition of an instrument and can be used for requesting data as part of a subscription to the feed handler or for saving/loading instruments on "useMarketDataService()..." calls.

marx.orders.core.js

This script provides useful methods related to placing orders. Note that it makes use of both the UserService and OrderService endpoints, as many order-related functions require some knowledge of the logged-in user and information about his or her company and permitted accounts to function.

Important functions and objects are:

- **ordTIFs, ordTypes, ordSides, ordInstTypes, accountTypes** - these objects contain a list of order times-in-force, types, and sides; valid instrument types; and valid account types that can be applied to orders or used to interpret values reported in an execution or other object returned from the OMS. Note that not every destination supports every possible value - the actual values supported depend on the particular LP or other endpoint to which the order relates. Additionally, these values can be requested from the OrderService "meta" GET endpoints.

- **setEventSource()** - configures useOrderService().getOrderEvents() calls to return events based on the user's highest level of permissions; e.g., an administrator will see all events for the company, whereas a trader will only be subscribed to user-level events.
- **getAllDestinations()** - populates and returns an "**allDestinations**" global object with information on all order destinations to which the user has access
- **getAllUsers()** - populates and returns an "**allUsers**" global variable with information on all users that the logged-in user is permitted to see. Also populates an "**allCompanies**" object containing all companies and departments which the user is permitted to see.
- **getAllCompanies()** - returns "**allCompanies**" object as described above.
- **getOrderSet()** - returns an object containing all non-retired orders that the current user can access.
- **getAllAccounts()** - returns an object containing information on all accounts to which the logged-in user has access.
- **statusPoller** - this object polls for new system status events and publishes them via the pubsub plugin to a "systemstatus" channel. statusPoller.start() and statusPoller.stop() can be used to enable or disable the polling.
- **positionPoller** - this object polls for user positions and publishes them on a "/accounts/positions/[accountID]" channel. positionPoller.start() and positionPoller.stop() can be used to enable or disable polling. A typical use of the poller is show below: in this examples, self.options.account is the integer account ID, and a callback function is passed as the second parameter, which forwards the "positionList" returned by an update to a KnockoutJS model for further processing.

```
// Start the position poller
                self.subkey = positionPoller.subscribe(self.options.account,
                    function(positionList) {
                        //callback code, should rebuild the whole thing
          PositionWidget.positionWidgetModel.updatePositionList(position
List);
                    }
                );
```

marx.marketdata.core.js

This script contains one useful method - **getAllRoutes** - which returns an object containing all permitted routes to marketdata for the user, along with all instruments available on each route. The method will also populate a global "**allExchanges**" object which contains metadata on each exchange ID, such as its description and any FIX or MIDs related to it.

User Management API

Base URL: http://{IP or address}:8080/etsdatamanagement/rest

- **GET** /watchlist/ids
  - Get an array of watchlist IDs accessible to the current user.
- **GET** /watchlist/{id}
  - Return the watchlist identified by an integer {id}. Note that the API returns helper fields which should NOT be submitted on an update request - specifically, groupId, map, symbols, and instrumentIds
- **POST** /watchlist/add
  - Add or update a watchlist.

```
Expected object structure:


{
 name: "Watchlist", // String name of watchlist
 id: id, // Integer ID of watchlist; 0 = new, otherwise updating
 ownerId, // Integer user ID of creator
 companyId, // Integer company ID of creator
 list // String list of instruments to add to watchlist, separated by "^" (caret).
}


Each entry in the watchlist should be formatted with fields separated by
the "|" (pipe) symbol, and containing the following:


instrumentID|instrumentSymbol|instrumentRoute


Above, "instrumentRoute" is a JSON-stringified route object string as
returned by the MarketData service; for example, an entire entry in the
watchlist for a symbol "E5U3" on the ESIGNAL data feed would be stored
as follows:
312|E5U3|
{"protocol":"com.tradedesksoftware.feedhandler.feed.tcp.MarxFeed","desc
ription":null,"level":1,"symbol":null,"carrier":"ESIGNAL","marketid":1005,"su
btopic":"feedhandler:8787","supplier":"ESIGNAL"}
```

- **POST** /watchlist/delete
  - Delete a watchlist. Expected object structure is the same as "/watchlist/add" above.
- **GET** /companies/{id}
  - Return company identified by integer {id}.

- **POST** /companies
    - Save a company identified by the “id” property on the company object. An ID of “0” will add a new company, otherwise the company identified by the given ID will be updated.

```
Expected object structure:
{
 bdid: “XYZABC”, //
 description: “Company Description”,
 id: 9999, // 0 for new company
 name: “Company Name”,
 state: “DISABLED”  // DISABLED or ENABLED
}


Note: “bdid” is “Broker Dealer ID” - keep in mind that you must have a
matching sequence created in order to place orders for this company.
```

- **GET** /companies/divisions/{id}
    - Get company division identified by integer {id}.
- **POST** /companies/divisions
    - Save a division identified by the “id” property on the division object. An ID of “0” will add a new company division, otherwise the company division identified by the given ID will be updated.

```
Expected object structure:
{
 access: “ENABLED”, // Deprecated - must submit identical to state
 companyId: 999,
 description: “Division Description”,
 id: 9999, // 0 for new division
 name: “Division Name”,
 parentDepartment: 9999, // 0 for top-level
 state: “ENABLED” // DISABLED or ENABLED
}
```

- **GET** /users/{id}/profile
    - Get the profile data for a given user identified by integer {id}. Profile data consists of supplementary information for the given user (contact info, etc). This data is useful for business purposes but is not generally needed for trading.
- **GET** /users/{id}/permissions
    - Get an array of Role objects associated with the integer {id} of the user.
- **POST** /users/userinfo
    - Add or update a user fully, including profile and role information.

```
Expected object structure:
{
 profile: {
   dayPhone: “”, // can be null
   fax: “”, // can be null
   firstName: "First",
   handle: "username",
   homePage: “”, // can be null
   id: 9999, // 0 for new user
   imageName: “”, // can be null
   lastName: "Last",
   mobilePhone: “”, // can be null
   nightPhone: “”, // can be null
   ownerId: 999, // id of user creating record
   privateEmail: "email@email.com’’,
   publicEmail: “email@email.com”, // can be null
 },
 roles: [ // array of role objects or null
   {
     "roleCategory":"RoleCategory",
     "roleName":"RoleName",
     "id":9999
   }
 ],
 user: {
   companyId: 9999, // id of company to which user will be added
   groupId: 9999, // id of division to which user will be added - can be
omitted
   id: 9999, // 0 for new user
   ownerId, // id of user creating record
   recoveryConfirmation: null, // leave null -used for email confirmation
   state: “ENABLED”, // ENABLED or DISABLED
   userName: “username”
 }
}
```

For “roles”, see **GET** /roles/all for possible values.

- **GET** /users/{id}
  - Return information about the user identified by integer {id}.
- **GET** /users/current

    - Return information about the authenticated user.
- **GET** /users/current/companies/divisions
    - Return information about the authenticated user's company division.
- **GET** /users/current/companies/all
    - Return all companies the authenticated user can see, including disabled/deleted companies.
- **GET** /users/current/companies/live
    - Return all active companies the authenticated user can see.
- **GET** /users/current/permissions/primary
    - Get the name of the most permissive trading role the current user has. This is useful when a client application wants to present role-specific options.
- **GET** /users/all
    - Get all users visible to the authenticated user.
- **GET** /roles/all
    - List all roles defined in the system.

Market Data Management API

Base URL: http://{IP or address}:8080/etsmarketdata/rest

- **GET** /routes/{symbol}/all
    - Return all known routes for the given {symbol}
- **GET** /exchanges/all
    - List all permitted exchanges for the authenticated user
- **GET** /routes/permitted/all
    - List all permitted routes for the authenticated user
- **GET** /routes/{symbol}/permitted
    - Show all routes for the given {symbol} permitted to the authenticated user.
- **POST** /exchanges/clone
    - Clones an exchange, including markups and attributes, for the posted integer "exchangeid"

    ```
    Expected object structure:
    9999 // ID of exchange to clone
    ```
- **POST** /exchanges/update
    - Add or update an exchange - use id 0 for a new exchange.

    ```
    Expected object structure:
    {
     description: "XYZ Market", // name/description of market
     fix42Code: "XYZ", // FIX 4.2 code (not used in FX)
     id: 9999,
     micCode: "XYZ" // market ID code (not used in FX)
    ```

```
}
```

- **GET** /exchanges/{symbol}
  - Return all exchanges containing instruments whose symbol matches the {symbol} string.
- **POST** /symbols/delete
  - Delete an instrument for the posted integer instrument ID

```
Expected object structure:
9999 / ID of instrument to delete
```

- **POST** /symbols/update
  - Update instrument information - use id 0 for a new instrument

```
Expected object structure:
{
 attributes: {NOSPREAD: 2}, // map of key:value pairs
 baseSymbol: "AAABBB",
 exchangeid: 999, // ID of exchange to which to add instrument
 id: 999, // 0 for new instrument, otherwise update
 symbol: "AAABBB",
 type: "FOREIGNEXCHANGE",
 underlyingid: 6
}


Currently meaningful attributes:
```

- PRICEDIGITS - Set the number of decimal places for a given symbol
- RNDSPREAD - Set to 1 to have min/max spread refer to limits for a random spread added to bid price. Unset to disable.
- TSTRAILINGPIPS -The number of pips to trail on a broker trailing stop for this symbol
- TSOFFSETPIPS - The number of pips to offset stop prices from the current market price on a broker trailing stop for this symbol
- MATYEARMON - Maturity date for futures
- CONTRACTSIZE - Some LPs require this per instrument
- NOSPREAD - Set to 1 to force the ask price equal to the bid price

```
Valid instrument types: STOCK, STOCKOPTION, FUTURE,
FOREIGNEXCHANGE, OPTION, CFD
```

- **POST** /symbols/all
    - Return instrument information

    ```
    Expected object structure:
    [0,1] // array of instrument IDs for which to return instruments
    ```

- **POST** /symbols/delete
    - Delete instrument by ID

    ```
    Expected object structure:
    9999 // ID of instrument to delete
    ```

- **GET** /symbols/exchange/{exchangeid}
    - Return all instruments for exchange identified by integer {exchangeid}
- **GET** /symbols/exchange/{exchangeid}/{carrier}
    - Return map of instrumentID:carriersymbol for the given integer {exchangeID} and {carrier} string.
- **POST** /symbols/carrier/saveMultiple
    - Update carrier symbols

    ```
    Expected object structure:
    {
     ‘“CARRIER”: { // carrier name
       999 : “CARRIERSYMBOL” // instrumentID: carrier symbol
     }
    }


    You may save multiple carrier symbols and/or symbols for multiple carriers
    at once by adding extra key:value pairs.
    ```

- **GET** /symbols/destinations/{destid}
    - Return map of instrumentID:destination symbol for the given destination ID
- **POST** /symbols/destination/saveMultiple
    - Update destination symbols (sent to destination when order is placed)

    ```
    Expected object structure:
    {
     destinationID: { // ID of destination for which to save destination symbols
       999 : “DESTINATIONSYMBOL” // instrumentID: destination symbol
     }
    }
    You may save multiple carrier symbols and/or symbols for multiple carriers
    at once by adding extra key:value pairs.
    ```

- **POST** /markups/update
    - Update all markup-related information for an instrument.

```
Expected object structure:
{
 marketId: 999, // exchange ID for which to update markups
 markups: {AAABBB:[0.0006, 0.0003, 0.0005, 0.0002, 0.0003, 0.001,
0.0033,  0.0005, 0.0008, 0.001]}, // map of symbol:[value array] pairs
 supplier: "SUPPLIER" // Supplier for which these markups apply
}
```

Note: the value array expected for each symbol *must* contain one decimal entry for each of the following (ten in total):

```
markupbid - markup to apply to bid side
markupask - markup to apply to ask side
priceband - price band (0.00000 if not applicable)
minspread - minimum spread to enforce (offset from buy side)
minmarkup - minimum markup to enforce
minsellslip - minimum ask slip amount
maxsellslip - maximum ask slip amount
minbuyslip - minimum buy slip amount
maxbuyslip -maximum buy slip amount
maxspread - maximum allowable spread after markups and
slippage
```

- **GET** /markups/{marketid}
    - Get markups for given integer exchange

Order Management API

Base URL: http://{IP or address}:8080/omsrestservices/rest

- **GET** /systemevents/all
    - Return list of all system events
- **POST** /accounts/new
    - Add **or update** an account.

```
Expected object structure:
{
 balance: null, // null or amount
 companyId: 999, // id of company to which account belongs
 currency: "USD", // account currency
 id: 999, // 0 for a new account, otherwise id of account to update
 lastUpdate: 1371546676000, // Java timestamp (include milliseconds)
 limit: 0, //
 margin: null, // margin for account
 maxordersize: 0, // order size limit - 0 for no limit
 maxposlimit: 0, // order position limit - 0 for no limit
 name: "Account Name", // name of account
 number: "TST-0001", // account number (typically sent in Account field for
FIX)
 reuseFund: null, //
 type: "PROPRIETARY"
}


Valid account types:
```

- RETAIL
- WHOLESALE
- PROPRIETARY // default
- EMPLOYEE
- COMBINED

- **POST** /accounts/{id}/delete
  - Delete account by integer {id}

```
Expected object structure:
9999 // ID of account to delete


NOTE: accounts may not be deleted while they are linked to destinations
under destination configuration.
```

- **POST** /accounts/{id}/retireTickets
  - Retire all tickets and orders for integer {id}

```
Expected object structure:
9999 // ID of account to retire tickets
```

- **POST** /accounts/{id}/deleteadjustments
  - Delete position adjustments for integer {id}

```
Expected object structure:
9999 // ID of instrument to delete
```

- **POST** /accounts/adjustments/new
  - Update account position

```
Expected object structure:
```

- **GET** /accounts{id}
  - Return information for account identified by integer {id}

- **GET** /accounts/{id}/positions
    - Get all positions for account by integer {id} of account
- **GET** /companies/{id}/accounts
    - Get all accounts for company identified by integer {id}
- **GET** /destinations/company/{id}
    - Get all destinations linked to company identified by integer {id}
- **GET** /destinations/all
    - Get all destinations visible to authenticated user - returns Destination objects with basic destination info such as name.
- **GET** /destinations
    - Get all destination info for destinations visible to authenticated user - returns DestinationInfo objects with full destination configuration including markets, accounts, sides, etc.
- **GET** /destinations/{id}
    - Get all destination info for destination identified by integer {id}
- **GET** /accounts/{id}/markets
    - Get all exchanges on destinations which are linked to account identified by {id}
- **GET** /destinations/user
    - Get all DestinationInfo for destinations which authenticated user is actually permitted to route orders to.
- **POST** /destinations/save
    - Save destination

```
Expected object structure:
{
 canCreate: false, // treat unknown incoming execution as new order
command
 enabled: true, // boolean
 gateway: "destgateway", // gateway to which destination applies
 id: 999, // 0 for new destination
 name: "Destination Name"
}
```

- **POST** /destinations/info/save
    - Save destination configuration information

```
Expected object structure:
{
 destinationid: 999,
 markets: [
   {
     marketid: 999, // id of exchange to enable on destination
```

```
    accountIds: [998, 999], // array of account ids supported for market/dest
    instrumentTypes: [3,4], // enum value of supported instrument type[s]
    ordertypes: [0,1], // enum value of supported order type[s]
    sides: [0,1], // enum value of supported order side[s]
    tifs: [0,1] // enum value of supported times in force
  }, { … }
 ]
}
```

- **GET** /roles/primary
    - Return primary (highest, most-permissive) role available to the authenticated user
- **GET** /userids
    - Returns array [userID, groupID, companyID] for authenticated user.
- **POST** /orders/place
    - Delete position adjustments for integer {id}

```
Expected object structure:
{
 accountId: 999, // account ID to book order to
 aon: false, // all or none execution
 avgprice: null, // leave null (average executed price)
 capacity: "AGENCY", // AGENCY [default], PROPRIETARY, INDIVIDUAL,
PRINCIPLE, RISKLESS_PRINCIPAL, AGENT_FOR_OTHER_MEMBER
 companyId: 999, // company ID of user placing trade
 cpOrderId: null, // null (counterparty order ID)
 creationTime: 1396592147102, // Java timestamp with milliseconds
 cumqty: 0, // leave null (cumulative executed quantity)
 custOrderId: "CustomerOrderID", // must be unique! UUID/GUID or similar
 destinationId: 999, // ID of destination to which order should be routed
 dispQty: 0, // order quantity to display, if supported by destination
 execInst: null, // execution instructions - GET meta/executions/instructions
 expireTime: 0, // Java timestamp for order to expire if time-in-force requires
 groupId: 0, //  group ID of user placing trade
 id: null, // null for new order, otherwise integer ID of order generated by OMS
 instrument: { // same as marketdata "POST symbols/update"
   attributes:{PRICEDIGITS:5},
   baseSymbol:AAABBB,
   exchangeid: 999,
  id:999,
  symbol:AAABBB,
  type: “FOREIGNEXCHANGE”,
  underlyingid: 0  },
 lastState: null, // previous state for this order, if not new
 leavesqty: 0, // quantity remaining to be executed, if not new
 market: 999, // ID of market to which
 notes: null, // String of notes to apply to order
 orderId: null, // OMS ID generated for order if not new
 ownerId: 999 // ID of user placing order
 price: "1.5", // If order type requires price, decimal price
 quantity: 10000, // Integer quantity for order
 quoteId: null, // ID of quote, if applicable
 relatedClOrdId: null, // If order type CANCEL/REPLACE, set to CustOrderId
of related order generated by client
 relatedOrderId: 0, // if order type CANCEL/REPLACE, set to integer “id” of
order generated by OMS
 side: "SELL", // see GET meta/orders/sides
 state: null, // see GET meta/orders/states
 stopPrice: null, // stop price if order type requires
 ticketId: null, // ticket ID generated by OMS if not new
 tif: "DAY", // see GET meta/orders/tifs
 type: "LIMIT", // see GET meta/orders/types}
```

- **POST** /orders/{id}/replace
    - Replace order with integer {id}

```
Expected object structure:
Same as POST /orders/place


Replace orders should have relatedClOrdId andn/or relatedOrderID set to the
ID[s] of the order being replaced, and a unique custOrderId. The order
otherwise should reflect the desired end state of the replaced order with
regards to quantity/price.
```

- **POST** /orders/{id}/cancel
    - Cancel order with integer {id}

```
Expected object structure:
Same as POST /orders/place


Cancel orders should have relatedClOrdId andn/or relatedOrderID set to the
ID[s] of the order being replaced.  The order requires its own unique
custOrderId.
```

- **POST** /orders/dock
  - Same as /orders/place, but does not route the order until an UNDOCK request is received.

```
Expected object structure:
Same as POST /orders/place
```

- **POST** /orders/{id}/undock
  - Undock order with integer {id} - will route the order to its destination.

```
Expected object structure:
Same as POST /orders/place
```

- **GET** /orders/active
  - Return all orders not in a final state (e.g., not filled/cancelled/expired/rejected.)
- **GET** /orders/{id}/executions
  - Return all executions for order with OMS-generated integer {id}
- **GET** /orders/eventsource/{type}
  - Set the event source to the designated {type} string (COMPANY, DEPARTMENT, USER). Controls scope of information returned when polling **GET** /orders/events.
- **GET** /orders/eventsource/{type}/history
  - Same as **GET** /orders/eventsource/{type} but will return all applicable events in system memory since last OMS startup on first poll to **GET** /orders/events
- **GET** /orders/events
  - Get all events queued for this client since last call.
- **GET** /orders/{id}/calculations
  - Returns average price and other database-intensive computed information for the order identified by OMS-generated integer {id}.
- **GET** /orders{id}
  - Return order as currently stored in database.
- **GET** /meta/orders/types
  - Return all order types supported by OMS (not necessarily by a specific destination!)
- **GET** /meta/orders/states
  - Return all order states supported by OMS (not necessarily by a specific destination!)
- **GET** /meta/orders/tifs
  - Return all order times-in-force supported by OMS (not necessarily by a specific destination!)

- **GET** /meta/orders/sides
    - Return all order sides supported by OMS (not necessarily by a specific destination!)
- **GET** /meta/executions/types
    - Return all execution types supported by OMS (not necessarily by a specific destination!)
- **GET** /meta/executions/instructions
    - Return all order execution instructions supported by OMS (not necessarily by a specific destination!)
- **GET** /meta/accounts/types
    - Return all account types supported by OMS (not necessarily by a specific destination!)
- **GET** /meta/tickets/states
    - Return all ticket statuses supported by OMS (not necessarily by a specific destination!)
- **GET** /meta/instruments/types
    - Return all instrument types supported by OMS (not necessarily by a specific destination!)

Historical Data Acquisition Facility

MARX provides access to historical market data via the Historical Data Facility of the ETSMarketData service. This data can be accessed as follows:

In order to access historical data, authenticate with the server exactly as if you are authenticating to the OMS, but to the following URL:

http://<marxServerURL>:8080/etsmarketdata

Once authenticated, you should submit a GET request to http://<markServerURL>:8080/etsmarketdata/History with the following arguments:

Parameter	**Description**
daysrequested	int number of days, going back in time, including today
symbol	string subject of history request, using ETS symbols
symex	int symbol exchange ID
carrier	string symbol carrier name
datatype	string, currently unused, defaults to "Forex" internally

For example:
http://<markServerURL>:8080/etsmarketdata/Historydaysrequested=2&symbol=USDCHF&symex=501&carrier=HOTSPOT

You will receive an XML document with the following structure:

```
<?xml version="1.0"?>
<HistoryResultFiles>
<HistoryFile>http://69.20.70.158:8080/history/USDCHF.HOTSPOT.501.2010_11_16</HistoryFile>
<HistoryFile>http://69.20.70.158:8080/history/USDCHF.HOTSPOT.501.2010_11_16.GAPFILL</HistoryFile>
</HistoryResultFiles>
```

Each HistoryFile contains all the history parsed for the date in the filename. The client will need to fetch each file from the list. There should always be a single GAPFILL file which contains data from the last parsing time of today's date up until the instant the request reaches the history server, to fill any gaps in the data.

All history files (both regular and gapfill) contain binary records (big-endian/network order) with the following structure:

Type	**Description**
Long	Date in Unix Epoch ms.
Double	Open price
Double	High price
Double	Low price
Double	Close price
Int	Volume
Int	Enum 0 = BID, 1 = ASK, 2 = TRADE

Each record is a 30-second OHLCV record. The client will be responsible for consolidating these into coarser time increments as necessary.

Note re: caching
The history service automatically deletes old GAPFILL files once all the raw history

for that day is consolidated into a day file, so if you are caching, you should always re-fetch days where you have a gapfill file in the cache, as it will indicate that your day file for that day is incomplete.

FIX Order Routing Facility

The MARX OMS can accept orders and provide execution reports via FIX. Certain other functions can also be performed via FIX, such as requesting order status and position information. The MARX OMS FIX client handler uses FIX 4.4. Any OMS user with the FIXUser role can connect via FIX. The FCH can be configured in various ways, but the standard configuration is as follows:

SETTING	**VALUE**
SenderCompID	Same as user's MARX login
TargetCompID	MARXOMS
Port	8999
Address	Will be provided, contact TSI CS.

Note: The FIX client handler is currently configured to reset sequence numbers when a client logs out. Each login should have sequence number set to 1. FCH FIX attributes can be set up via the MARX Client/Gateway Sessions panel on the web UI. Individual sessions can be constructed with whatever session level setup is desired (IE day or week long sessions, reset on logon, session open and close times, etc).

User Request Message

MARX has the ability to route execution reports to different clients. In order to request execution reports the client application needs to send a FIX UserRequest message. The following application tags and values are used:

Tag	**Value**	**Require d**	**Notes**
553 Username	jsmith	Y	Login id of the subscribing user account
554 Password	secret	Y	Password of subscribing user account
923	20110920-	Y	Unique id for this request

UserRequestID	001		
924 UserRequestType	1	Y	Required by FIX spec
96 RawData	01/01	Y	2 integers delimited by a '/'. The first value indicates the type of subscription, 0 = DELETE, 1 = USER, 2 = DEPARTMENT, 3 = COMPANY. The second value indicates whether the client desires status reports for existing active orders 0 = NO, 1 = YES.

User Response Message

MARX will respond to a UserRequest message with a UserResponse. The following fields/values will be returned.

Tag	Value	Required	Notes
923 UserRequestID	20110920-001	Y	Value provided by the client. Associates this response to the corresponding request.
553 Username	jsmith	Y	Value provided by the client in UserRequest.
927 UserStatusText	01/01/01	N	MARX userid/departmentid/companyid of the requesting user. This will only be present if the subscription was accepted.
926 UserStatus	1	Y	Either 1 (LoggedIn) or 2 (NotLoggedIn). 1 indicates the subscription was accepted, 2 indicates the subscription was not accepted.

If the subscription was accepted UserStatus will be set to 1, otherwise it will be set to 2. The UserStatusText value will contain the MARX ids of the user, the user's department, and the user's company if the subscription was accepted. Tags

553 and 923 simply reflect the values sent by the client.

NewOrderSingle Message

Orders are sent to the MARX OMS using the NewOrderSingle message. The format of the message is as follows:

Tag	Value	Required	Notes
11 ClOrdID	20110920-001	Y	Client's id for this order, must be unique.
453 NoPartyIDs	1	Y	Should always be 1
+448 PartyID	101	Y	MARX user id of the user this order is submitted for.
40 OrdType	1	Y	Type of order, the supported types depend on the destination and market the order is being routed to.
44 Price	1.01	N*	Required for limit type orders, etc.
99 StopPx	1.02	N*	Required for stop type orders, etc.
1 AccountID	1	Y	MARX account id of the account this trade is being executed on.
54 Side	1	Y	Normally 1 = BUY, or 2 = SELL. Some destinations may support other values.
55 Symbol	EURUSD	N	Symbol for instrument being traded (IE currency pair).
207 SecurityExchange	2	N	Must be set if symbol is set to establish scope of the symbol, this will normally be the id of the market the order is being sent to.
48 SecurityID	202	N	May be provided instead of tags 55 and 207. MARX instrument id of the instrument being traded.
100 ExDestination	1:2	Y	Destination id and market id to which the order is being routed.

			Values will be established by OMS configuration.
38 OrderQty	1000000	Y	Quantity of order. Note for F/X this is in whole currency units of the primary currency.
18 ExecInst	9	N	Allowed values depend on destination.
58 Text	Abc	N	Optional note that will be attached to the order. The OMS doesn't use this data but will store it and return it in order status responses.
59 TimeInForce	0	N	Different TIF values are supported by different destinations.
126 ExpireTime	20110920-12:00:00	N*	Required for TIFs needing an expiration time/date.

Note that several fields can take a range of values which depend on the configuration of the destination the order is being routed to. Each destination generally supports some specific set of options.

ExecutionReport Message

The MARX OMS will respond to each NewOrderSingle with an ExecutionReport.

Tag	Value	Required	Notes
37 OrderID	NNNNNN0390000001	Y	Unique id assigned by the OMS to this order.
60 TransactTime	20110920-12:00:01	Y	Date/time of the transaction.
55 Symbol	EURUSD	Y	Symbol for the instrument.
167 SecurityType	FOR	Y	Type of instrument.
11 ClOrdID	20110920-002	N	Client order id. Note that orders submitted by some algorithms and non-FIX clients may not supply this

			value.
198 SecondaryOrderID	NNNNNA0391234567	N	Additional order id. This is usually an id supplied by the executing market. May also be an internal MARX id allocated by a routing algorithm or may not be present in some cases.
41 OrigClOrdID	20110920-001	N*	Client order id of a related order, usually an order which was replaced or canceled. Not present if there is no related order.
17 ExecID	NNNNQQ0391234566	Y	Unique identifier for this execution. Note that this may not be available in some cases.
150 ExecType	0	Y	Execution type.
39 OrdStatus	0	Y	Order Status
54 Side	0	Y	Side
38 OrderQty	1000000	Y	Quantity ordered
151 LeavesQty	500000	Y	Remaining unexecuted quantity for this order
14 CumQty	500000	Y	Currently executed quantity
6 AvgPx	1.01	Y	Average price for executions on this order
31 LastPx	1.01	N	Price for this execution if it is a FILL
32 LastQty	500000	N	Size of this execution if this is a FILL
18 ExecInst	9	N	Execution instructions for this order, if any

58 Text	Message	N	May contain human-readable error message text on a reject
103 OrdRejReason	1	N	Present if the order is a reject

OrderCancelReplace Message

Tag	Value	Required	Notes
11 ClOrdID	20110920-002	Y	Client's id for this order, must be unique.
37 OrderID	AAA-BBB-CCC	N	OMS ID of order being canceled, not required.
41 OrgClOrdID	20110920-001	Y	ClOrdID of the order being canceled.
453 NoPartyIDs	1	Y	Should always be 1
+448 PartyID	101	Y	MARX user id of the user this order is submitted for.
40 OrdType	1	Y	Type of order, the supported types depend on the destination and market the order is being routed to.
44 Price	1.01	N*	Required for limit type orders, etc.
99 StopPx	1.02	N*	Required for stop type orders, etc.
1 AccountID	1	Y	MARX account id of the account this

			trade is being executed on.
54 Side	1	Y	Normally 1 = BUY, or 2 = SELL. Some destinations may support other values.
55 Symbol	EURUSD	N	Symbol for instrument being traded (IE currency pair).
207 SecurityExchange	2	N	Must be set if symbol is set to establish scope of the symbol, this will normally be the id of the market the order is being sent to.
48 SecurityID	202	N	May be provided instead of tags 55 and 207. MARX instrument id of the instrument being traded.
100 ExDestination	1:2	Y	Destination id and market id to which the order is being routed. Values will be established by OMS configuration. NOTE: this must be the same value as provided with the original order.
38 OrderQty	1000000	Y	Quantity of order.

			Note for F/X this is in whole currency units of the primary currency.
18 ExecInst	9	N	Allowed values depend on destination.
58 Text	Abc	N	Optional note that will be attached to the order. The OMS doesn't use this data but will store it and return it in order status responses.
59 TimeInForce	0	N	Different TIF values are supported by different destinations.
126 ExpireTime	20110920-12:00:00	N*	Required for TIFs needing an expiration time/date.

Note that most destinations will allow only certain values to be modified with respect to the original order. Usually size, price, and possibly execution instructions are alterable. TIF may also be alterable.

OrderCancelRequest Message

Cancel the given order. Note that the instrument's Symbol, SecurityExchange, SecurityID, Side, Account, etc MUST all be the same values that were present in the order being canceled.

Tag	Value	Required	Notes
11 ClOrdID	20110920-002	Y	Client's id for this order, must be

			unique.
37 OrderID	AAA-BBB-CCC	N	OMS ID of order being canceled, not required.
41 OrgClOrdID	20110920-001	Y	ClOrdID of the order being canceled.
453 NoPartyIDs	1	Y	Should always be 1
+448 PartyID	101	Y	MARX user id of the user this order is submitted for.
40 OrdType	1	Y	Type of order, the supported types depend on the destination and market the order is being routed to.
1 AccountID	1	Y	MARX account id of the account this trade is being executed on.
54 Side	1	Y	Normally 1 = BUY, or 2 = SELL. Some destinations may support other values.
55 Symbol	EURUSD	N	Symbol for instrument being traded (IE currency pair).
207 SecurityExchange	2	N	Must be set if symbol is set to establish scope of the symbol, this will normally be

			the id of the market the order is being sent to.
48 SecurityID	202	N	May be provided instead of tags 55 and 207. MARX instrument id of the instrument being traded.
100 ExDestination	1:2	Y	Destination id and market id to which the order is being routed. Values will be established by OMS configuration. NOTE: this must be the same value as provided with the original order.
58 Text	Abc	N	Optional note that will be attached to the cancel. The OMS doesn't use this data but will store it and return it in order status responses.

OrderCancelReject Message

This message is returned when the OMS rejects a cancel or replace operation. In most cases the OMS will respond with an ExecutionReport, but in some cases this may not be feasible and OrderCancelReject will be returned instead.

Tag	Value	Required	Notes
11 ClOrdID	20110920-002	Y	Client's id for the for rejected cancel.
37 OrderID	AAA-BBB-CCC	N	OMS ID of order

			being canceled, not required.
SecondaryOrderId	XXX-YYY-ZZZ	N	Id assigned to the cancel by a remote counterparty, if any.
OrdStatus		Y	Status of the cancel request.
CxRejResponseTo	1	Y	Was this a cancel or a replace we are rejecting. Note that OMS almost always rejects replaces with an Execution Report.
CxlRejReason	0 – too late to cancel 1 – unknown order	Y	Reason request was rejected.
TransactTime	YYYY-MM-DD HH:MM:SS	Y	Time of rejection.

OrderMassStatusRequest Message

These can be issued by the client to return execution reports describing the status of various orders. The MassStatusReqType tag value will determine the scope as follows:

- STATUS_FOR_ALL_ORDERS (7) – All orders the MARX user has permission to access will be returned. Note: This could generate a LARGE number of responses if the user account has company or department permissions.
- STATUS_FOR_ORDERS_FOR_A_PARTYID (8) – Return status for all orders placed by the user id in the partyids block (there may only be one such value provided). Note that department or company admin rights are required to access other user's orders. For trader accounts this will be equivalent to STATUS_FOR_ALL_ORDERS.
- STATUS_FOR_ORDERS_FOR_A_SECURITY (1) – Return all orders for the given security. Again, the user's rights will determine whether or not this will be for ALL orders for the given security, or only for those owned by the

given user.

- STATUS_FOR_ORDERS_FOR_AN_ACCOUNT (9) – Return all orders for the given OMS trading account. Again, only orders the user is permitted for will be returned.

Tag	Value	Required	Notes
MassStatusReqID	20110920-001	Y	A unique id for this request.
MassStatusReqType	See above	Y	Determines the scope of the execution reports returned.
NoPartyIDs	1	N*	Required if MassStatusReqType = 8
PartyID	123	N*	Required if MassStatusReqType = 8, MARX userid to request order status for.
55 Symbol	EURUSD	N*	Required if MassStatusReqType = 1
Account	123	N*	Required if MassStatusReqType = 9. MARX trading account id.

RequestForPositions Message

Requests positions for a given OMS trading account. The OMS will respond with either a PositionRequestAck message, indicating that no open positions exist for the given account. If positions do exist then a series of PositionReport messages will be returned instead.

Note: The OMS maintains separate positions for each symbol (currency pair, etc) AND each market/exchange id (LP/Prime Broker in the case of FX). It is thus possible for the OMS to return more than one position for a given pair if an account contains trades going to different destinations or with different markups, etc.

Tag	Value	Required	Notes
PosReqID	20110920-002	Y	Client's id for position request.
PosReqType	0	Y	POSITIONS (0) is the only value currently supported.
Account	134	Y	Account to request positions for.

PositionReport Message

As described above, a group of 1 or more of these messages will be issued, one per open position. TotalNumPosReports will indicate the total number to be expected in the response to the given PosReqID. This allows the receiving side to know when it has received all expected PositionReport messages.

Tag	Value	Required	Notes
PosMaintRptID	AABZ1110101	Y	Unique id for this report.
PosReqID	20110920-002	Y	Client's id for position request which this report responds to.
TotalNumPosReports	5	Y	Total number of reports being generated in response to the PosReqID this report responds to.
PosReqResult	0	Y	Always 'VALID_REQUEST' as other scenarios are handled via PositionReportRequestAck
ClearingBusinessDat	19790101	Y	Value is a placeholder

e			currently.
55 Symbol	EURUSD	Y	Symbol/Pair of this position.
SecurityExchange	1	Y	MARX market id of the position.
SecurityID	1	Y	MARX instrument id corresponding to the exchange and symbol.
NoPartyIDs	1	Y	Always 1
PartyID	2	Y	Id of the OMS account this position is on.
Account	2	Y	Account this position is on, this is the same as the PartyID, provided for FIX compatibility.
AccountType	2	Y	Always set to 2 (ACCOUNT_IS_CARRIED_ON_NON_CUSTOMER_SIDE_OF_BOOKS)
SettlePrice	1.2345	Y	Actual average price paid for this position.
SettlPriceType	2	Y	Always set to 2 (THEORETICAL)
PriorSettlePrice	1.2345	Y	Same as SettlePrice
NoPositions	1	Y	Always 1
PosType	ASF	Y	Always AS_OF_TRADE_QTY
LongQty	10000	N	Quantity which is long
ShortQty	10000	N	Quantity which is short, one of LongQty and ShortQty are always present.
NoPosAmt	1	Y	Always 1

PosAmtType	CASH	Y	Always 'CASH' (CASH_AMOUNT)
PosAmt	101046.2345	Y	Total cash amount paid for position.

PositionReportAck Message

This will be generated in response to a RequestForPositions message only when no positions are available to return. It serves to indicate this null response to the client.

Tag	Value	Required	Notes
PosMaintRptID	AABZ1110101	Y	Unique id for this report.
PosReqID	20110920-002	Y	Client's id for position request which this report responds to.
TotalNumPosReports	0	Y	Always 0.
PosReqResult	2	Y	Always 2 (NO_POSITIONS_FOUND_THAT_MATCH_CRITERIA)
PosReqStatus	0	Y	Always 0 (COMPLETED)
NoPartyIDs	1	Y	Always 1
PartyID	2	Y	User the report was requested for.
Account	1	Y	Account positions requested for
AccountType	1	Y	Always 1 (ACCCOUNT_IS_CARRIED_ON_CUSTOMER_SIDE_OF_BOOKS)

Feed Handler Access

The MARX feed handler may be accessed via either FIX or a custom binary TCP feed protocol. The feed handler connects to data providers, normalizes all data into a standard format, and forwards it to clients via a subscription based mechanism. Multiple data sources are supported, so for instance a feed handler can supply FOREX quote for a given currency pair simultaneously on more than one market or from more than one quote source. Multiple levels of data are supported. Level 1 data provides trades and best bid/offer, volume, VWAP, and other general instrument specific and market specific top level data for a given instrument. Level 2 data provides consolidated book and depth of market data in the form of individual quotes for a given instrument on a given market. Some instruments can also provide other specialized data, for example an equity instrument may be able to provide OPTLIST data listing available related options contracts.

MARX Binary Feed Protocol

The MARX Feed Handler supports a proprietary high efficiency network protocol for distributing market data. This protocol utilizes TCP. The client establishes a TCP connection to the feed handler, sends subscribe messages to request streams of data for individual instruments and unsubscribe messages to stop receiving data. The feed handler responds with market data messages. Each message is structured as detailed below. All data is binary and big-endian. Strings are all null terminated. There are no heartbeats. TCP keepalive can be utilized in situations where connections might be lost. If the connection drops the client's subscriptions will be unmapped automatically, it is not necessary to manually unsubscribe from each symbol.

Message Types

There are 2 basic message types, subscription and market data. Both of them are similar in overall format. Each message begins with a header containing the following fields:

Field Name	Length	Notes
Frame Indicator	2 bytes	Always has the value 0xFFFF. This simply indicates the start of a message. Note that 0xFFFF could appear in other fields. It is mainly useful during development and

		troubleshooting
Sequence Number	4 bytes (Integer)	Monotonically increasing sequence number of messages sent. Each side increments it's outgoing sequence number for each message sent. Allows detection of framing or other errors.
Body Length	2 bytes (short integer)	Number of bytes contained in the remainder of the message.
Message Type	1 byte	Type of message 0 = Subscription, 1 = Market Data

Subscription Message

Subscription messages start with a standard header. The remainder of the message is structured as follows:

Field Name	Length	Notes
Subscribe Flag	1 byte	Indicates subscribing or unsubscribing, 1 = subscribe, 0 = unsubscribe
Level	1 byte	Level of data being requested (see below for values)
Market Id	4 bytes (Integer)	MARX Market id of the market for the desired instrument
Symbol	String	MARX Symbol of instrument being subscribed to
Carrier	String	MARX name of the carrier data is requested from
Supplier	String	MARX name of the supplier for this data

Market Data Message

Field Name	Length	Notes
Action	1 byte	Message Action (see below for values)
Type	1 byte	Data type (see below for values)
Data	Variable	Sequence of data fields

action values are as follows:

Name	Valu	Definition

	e	
ADD	0	new data which should replace existing data
UPDATE	1	data which should be merged with existing data
DELETE	2	this object should be deleted (level 2 row)
INVALIDATE	3	data for this instrument is invalid/unavailable
AVAILABLE	4	data for this instrument is now available

type values are as follows:

Name	Value	Definition
LEVEL1	1	Level 1 equity data
LEVEL2	2	Level 2 equity data
OPTLEVEL1	3	Level 1 option data
OPTLEVEL2	4	Level 2 option data
OPTLIST	5	List of option contracts
TICKDATA	6	Tick data
INDEX	7	Level 1 index data
LEVEL3	8	Level 3 equity data (full book)
OPTLEVEL3	9	Level 3 option data (full book)
FUTLEVEL1	10	Level 1 future data
FUTLEVEL2	11	Level 2 future data
FOREXLEVEL1	12	Level 1 forex data
FOREXLEVEL2	13	Level 2 forex data

Data Fields

Market data messages contain a sequence of standardized data fields. These fields can appear in any order. Specific fields may or may not be present depending on the type of data. Fields which are not present in a given message either have an undefined value or should be assumed to retain their current value depending on the value of the Action field. Each data field consists of a field identifier byte followed by the value of the field. The following field types are defined.

Field	Identifier Value	Length	Notes
LASTUPDATE	0	8 bytes (Long)	UTC Unix time of last received update
CARRIER	1	String	Identifies data carrier
SUPPLIER	2	String	Identifies data supplier
ACCVOLUME	3	4 bytes (Int)	Accumulated volume
PREVCLOSEPRICE	4	8 bytes (Double)	Previous closing price
OPENPRICE	5	8 bytes (Double)	Last opening price
LOWPRICE	6	8 bytes (Double)	Daily low price
HIGHPRICE	7	8 bytes (Double)	Daily high price
LASTTRADEPRICE	8	8 bytes (Double)	Last trade price
LASTTRADESIZE	9	4 bytes (Int)	Last trade size
LASTTRADEEXCHANGE	10	2 bytes (Short)	Market id of exchange of last trade
LASTTRADEDATE	11	8 bytes (Long)	UTC Unix time of last trade
BIDPRICE	12	8 bytes (Double)	Bid Price
BIDEXCHANGE	13	2 bytes (Short)	Market id of exchange for last bid
BIDSIZE	14	4 bytes (Int)	Size of bid
ASKPRICE	15	8 bytes (Double)	Bid Price

ASKSIZE	16	4 bytes (Int)	Size of offer
ASKEXCHANGE	17	2 bytes (Short)	Market id of exchange for last offer
LASTQUOTEDATE	18	8 bytes (Long)	UTC Unix time of last quote
CURRENCY	19	String	ISO Currency code of prices
TICK	20	1 byte	U = up, D = down
EXCHANGE	21	2 bytes (Short)	Market id of quote exchange
COMPANY	22	String	Issuing company
PREVCLOSEDATE	23	8 bytes (Long)	UTC Unix time of previous close
YEARHIGHPRICE	24	8 bytes (Double)	Yearly high price
YEARHIGHDATE	25	8 bytes (Long)	UTC Unix time of year high
YEARLOWPRICE	26	8 bytes (Double)	Yearly low price
YEARLOWDATE	27	8 bytes (Long)	UTC Unix time of year low
TOTALSHARES	28	4 bytes (Int)	Total size of issue/outstanding shares
AVGVOLUME	29	4 bytes (Int)	Average daily trading volume
EARNINGS	30	8 bytes (Double)	Last earnings/dividend/etc
EARNINGSDATE	31	8 bytes (Double)	UTC Unix time of last earnings
CUSIP	32	String	CUSIP
ISIN	33	String	ISIN
DOLLARS	34	8 bytes (Double)	
MMID	35	String	Market Maker ID
PRICE	36	8 bytes	Price of bid or offer

		(Double)	
SIDE	37	1 byte	Side, 0 = Bid, 1 = Offer
SIZE	38	4 bytes (Int)	Size of bid or offer
ORDEREXCHANGE	39	2 bytes (Short)	Market id of originating bid or offer exchange
ORDERDATE	40	8 bytes (Long)	UTC Unix time of bid or offer
STATUS	41	1 byte	Trading status
BASESYMBOL	42	String	Base symbol for series (futures and options)
MATURITYDATE	43	String	Maturity date of this instrument (futures and options)

FIX Feed Protocol

The feed handler uses standard FIX4.4 market data messages when using this protocol. Currently there is no authentication on feed connections. Valid sender and target ids should be negotiated with the operator of the feed handler. When setting up feed handlers these are normally defined in a QuickFixJ configuration file supplied with the feed handler application. Other details such as sequence number reset policies and session timing are also defined here and will need to be determined on a case-by-case basis.

All Messages (header fields)

The following fields are significant. Other required header fields are used normally. Other optional header fields are not supported.

Tag	Name	Required	Required (FIX4.4)	Notes
49	SenderCompID	Y	Y	Value will be supplied by provider
56	TargetCompID	Y	Y	Value will be supplied by provider
50	SenderSubID	N*	N	Provider may require a specific

				value in this field.
57	TargetSubID	N*	N	Provider may require a specific value in this field.

Protocol Level Messages

All fields in these messages are used in a standard fashion. The following fields values may need special consideration.

Tag	**Name**	**Message Type**	**Required**	**Notes**
108	HeartBeatInt	Logon	Y	Required value should be negotiated with provider.
141	ResetSeqNumFlag	Logon	N	May be set true to initiate sequence number reset. Use of this flag should be negotiated with the provider.

Market Data Messages

Only 2 message types are utilized, MarketDataRequest and MarketDataFullRefresh.

Market Data Request

Tag	**Name**	**Req'd**	**Required (FIX4.4)**	**Notes**
262	MDReqID	Y	Y	Must be unique, or the ID of previous Market Data Request to disable if SubscriptionRequestType = Disable previous Snapshot + Updates Request
263	SubscriptionRequestType	Y	Y	0 = not supported 1 = Snapshot/Update 2 = Unsubscribe

264	MarketDepth	Y	Y	1 (level 2 is not currently supported via FIX)
265	MDUpdateType	N	N	Provisionally required when SubscriptionRequestType = 1. Value should always be 0 (full updates).
267	NoMDEntryTypes	Y	Y	Number of following MDEntryType values.
269	MDEntryType	Y	Y	Standard FIX 4.4 values. Note that the FIX interface currently ignores this information and returns all relevant entry types.
146	NoRelatedSym	Y	Y	Number of related symbols.
55	Symbol	Y	N	Symbol being subscribed to.
207	SecurityExchange	Y	N	MARX market id of market requesting quote for.

Market Data Full Refresh

Tag	**Name**	**Req'd**	**Required (FIX4.4)**	**Notes**
262	MDReqID	Y	Y	Currently a dummy value is returned for this tag.
55	Symbol	Y	N	Symbol data is for.
207	SecurityExchange	Y	N	MARX market id of market quote is for.
167	SecurityType	N	Y	'FOR' (foreign exchange contract).
460	Product	N	Y	4 (currency).
268	NoMDEntries	Y	Y	Number of entries following.
269	MDEntryType	Y	Y	Type of entry. Other fields in the

	MDFullGrp will be present as indicated. Only fields with a reportable value will be present.

The Marx Databases

MARX stores most of its operating information in four MySQL database schemas (sometimes simply called databases in MySQL terminology). Applications can access the various tables, views, and procedures in these schemas in order to access data on accounts, trades, users, market data configuration, etc.
Note: that generally speaking changes to the database won’t be automatically visible to the OMS or Feed Handler. It is usually better to do routine management functions via the web service APIs or web-based UI. However it may be more convenient to do bulk ETL type tasks using 3rd party tools. Just understand that if for instance you add an exchange to the marketdata.exchanges table or a symbol to the marketdata.symbolmaster table the OMS and Feed Handler won’t automatically know about the change. It may be necessary to restart some services or the entire application.

The ETS Schema

The ETS schema holds information related to MARX system users as well as their permissions. The primary data consists of Company and CompanyDivision tables which contain information about companies (brokers) and divisions (departments or groups of traders). Note that a CompanyDivision can contain lower level CompanyDivisions, but this hierarchy is currently only effective to one sub-level.

The User table holds basic data on ETS Users, their login credentials, user id, and some other basic information. Each User generally also has an entry in the Profile table with extended information, and may be linked to one or more Address records holding contact information. The Role table contains information on all the different security roles and permissions which the system can grant. users are linked many-to-many to the Role table via the user_roles table.

Some other ancillary tables are also described below which serve to hold information used by client applications, provisioning tools, etc.

Address	

id	int NOT NULL AUTOINCREMENT	pk for address record
visible	bit NOT NULL	if 1 then it is ok to display this at the client
region	varchar(255)	state or equivalent
postalCode	varchar(255)	ZIP or equivalent
country	varchar(255)	country, usually 2 digit ISO
city	varchar(255)	city portion of address
address	varchar(255)	street address
address2	varchar(255)	any additional, c/o etc
type	int	address type, home, business, etc
owner_id	int	user profile this is associated with
Indexes		
pk_address primary key	ON id	
FK1ED033D4E98193F3	ON owner_id	
Foreign Keys		
FK1ED033D4E98193F3	(owner_id) ref Profile (id)	

Each address is associated with a Profile (and thus indirectly with a User record). Any number of addresses can be linked to a given user. Note that other entities such as companies don’t have addresses. A specific contact individual should be created and an address attached to them.

Application	
id	int NOT NULL AUTOINCREMENT

region	varchar(64)
publicEmail	varchar(128)
privateEmail	varchar(128) NOT NULL
postalCode	varchar(12)
lastName	varchar(128) NOT NULL
homePage	varchar(128)
firstName	varchar(128) NOT NULL
country	varchar(2)
city	varchar(64)
address	varchar(128)
userName	varchar(180) NOT NULL
passWord	varchar(16) NOT NULL
handle	varchar(40) NOT NULL
address2	varchar(128)
created	datetime
type	int
addressType	int
nightPhone	varchar(18)
mobilePhone	varchar(18)
fax	varchar(18)

dayPhone	varchar(18)	
processed	bit NOT NULL DEFO 0	true indicates application has been processed
optin	bit NOT NULL DEFO 0	true indicates the applicant has opted into receiving more information
Indexes		
pk_application primary key	ON id	
userName unique	ON userName	
handle unique	ON handle	

Application records can be used as a way to capture input user information before committing to creating an actual user record. This table isn't directly used by the OMS although a procedure does exist to construct User, Profile, and Address data from it. External untrusted applications can safely be given write-only access to this table to input data (say from a sign-up form).

Company		
id	int NOT NULL AUTOINCREMENT	pk of company record
state	smallint NOT NULL	0 for deleted, 1 for disabled, 2 for active
bdid	varchar(12)	broker dealer id (MPID) of company
name	varchar(80)	human readable name
description	varchar(255)	long description
Indexes		
pk_company primary key	ON id	

bdid unique	ON bdid
name unique	ON name

The Company table holds records identifying brokers who use the OMS. Each company has a unique BDID and name. For regulated entities this will normally consist of the entities MPID. Some large firms may have multiple MPIDs and this is not currently supported (multiple company records can be utilized instead if a work-around is required).

CompanyDivision		
id	int NOT NULL AUTOINCREMENT	pk of department record
state	smallint NOT NULL	0 deleted, 1 disabled, 2 active
parentDepartment	int	null for top level, parent pk if subdepartment
companyId	int	id of owning company
description	varchar(255)	long description of department
name	varchar(64) NOT NULL	human readable name
Indexes		
pk_companydivision primary key	ON id	
FKEF3FBC2A8D896049	ON companyId	
FKEF3FBC2A23E0553A	ON parentDepartment	
Foreign Keys		
FKEF3FBC2A8D896049	(companyId) ref Company (id)	
FKEF3FBC2A23E0553	(parentDepartmen	

A	t) ref CompanyDivision(id)

CompanyDivision holds department/group/division records. Each CompanyDivision holds Users and subdivisions. Note that only one level of subdivisions are supported.

Message	
id	int NOT NULL AUTOINCREMENT
statusOrdinal	int NOT NULL
subject	varchar(255)
sentOn	datetime
body	text
deleted	bit NOT NULL
messageType	varchar(255)
sender_id	int
recipient_id	int
Indexes	
pk_message primary key	ON id
FK9C2397E78698C789	ON recipient_id
FK9C2397E73D25D9ED	ON sender_id
Foreign Keys	
FK9C2397E73D25D9ED	(sender_id) ref User (id)

FK9C2397E78698C789	(recipient_id) ref User (id)

The Message table can be used to post messages from one OMS user to another. These are strictly internal messages and delivery is entirely dependent on whatever client the user is accessing the MARX platform with. The web based trading client and user interface will display messages, other clients may or may not have a facility to do this.

Preferences		
id	int NOT NULL AUTOINCREMENT	pk of preference
preferences	text	xml serialized preference data
name	varchar(255)	name of preferences set
user_id	int	id of owning user
Indexes		
pk_preferences primary key	ON id	
FKDA0486D832759897	ON user_id	
Foreign Keys		
FKDA0486D832759897	(user_id) ref User (id)	

The Preferences table is used to hold user preference information for use by client applications. Each record holds a named set of data associated with a user. The actual data is serialized XML name/value
pairs. Semantics for individual items and data type conventions are handled by the OMS preference handling functions.

Profile	
id	int NOT NULL

	AUTOINCREMENT	
publicEmail	varchar(128)	
privateEmail	varchar(128) NOT NULL	
lastName	varchar(128) NOT NULL	
homePage	varchar(128)	
handle	varchar(40) NOT NULL	display name for user
firstName	varchar(128) NOT NULL	
imageName	varchar(80)	url for an image
nightPhone	varchar(18)	
mobilePhone	varchar(18)	
fax	varchar(18)	
dayPhone	varchar(18)	
user_id	int NOT NULL	id of matching user record
Indexes		
pk_profile primary key	ON id	
handle unique	ON handle	
FK50C7218932759897 unique	ON user_id	
Foreign Keys		
FK50C7218932759897	(user_id) ref User (id)	

Each Profile record relates one-to-one with one User record and holds additional information on a user. This mainly exists to speed up database operations because User records are very frequently accessed and the additional fields held in Profile are required far less often. While it is possible for a User to exist with no

Profile the OMS UI won’t normally allow such a thing to exist.

Role		
id	int NOT NULL AUTOINCREMENT	pk of role
roleCategory	varchar(128) NOT NULL	defines use for this role
roleName	varchar(128) NOT NULL	name of role for container mapping
Indexes		
pk_role primary key	ON id	
roleCategory unique	ON roleCategory, roleName	

The Role table holds all permissioning information for the OMS. This includes both JAAS roles used by the OMS EJB container for basic authorization as well as other arbitrary roles used programmatically or by other components as needed. Each Role has a category which defines its semantics (for example roleCategory=’Role’ indicates a JAAS role available to the container for declarative authorization usage. The meaning of the roleName is application dependent. JAAS uses this as the name of the role, but application code is free to interpret them as desired. Generally 3rd party applications should create their own unique roleCategories if it is desired to share authorization information with the OMS.

SpreadSheet	
id	int NOT NULL AUTOINCREMENT
name	varchar(40) NOT NULL
data	text
owner_id	int
company_id	int
Indexes	

pk_spreadsheet primary key	ON id
FKE33B138C9E5C48AF	ON owner_id
FKE33B138CAF38CA0E	ON company_id
Foreign Keys	
FKE33B138CAF38CA0E	(company_id) ref Company (id)
FKE33B138C9E5C48AF	(owner_id) ref User (id)

This table was intended to hold data defining an embedded spreadsheet in client software which is currently not supported. Marx 1.4 and up do not currently support this.

User		
id	int NOT NULL AUTOINCREMENT	pk of user record
companyId	int NOT NULL	id of company user belongs to
departmentId	int	id of department, null if top level
userName	varchar(180) NOT NULL	user name for login
passWord	varchar(40) NOT NULL	SHA1 hash of password for login
recoveryConfirmation	varchar(64)	used to confirm password recovery
state	smallint NOT NULL	0 deleted, 1 disabled, 2 active

Indexes	
pk_user primary key	ON id
userName unique	ON userName
FK285FEB88FD19AB	ON departmentId
FK285FEB8D896049	ON companyId

Foreign Keys	
FK285FEB8D896049	(companyId) ref Company (id)
FK285FEB88FD19AB	(departmentId) ref companyDivision(id)

The User table provides primary authentication data for the OMS. A special JAAS auth module recovers the username and password hash and supplies them to the JBoss authz layer. Each user belongs to a company and a department (CompanyDivision), and also has a recovery confirmation nonce which is set when a user or administrator invokes the OMS password reset mechanism. When this value is non-null login is disabled until the account owner has completed the recovery process, at which time the confirmation value is nulled out. Note that there is a trigger on the this table which applies a SHA1 hash to the value of the password field when that column is updated. There is a mysql session variable which can turn this behavior off for situations like restoring or importing user data.

Third party applications should only be granted access to the User table cautiously. It is better to construct a stored procedure to perform whatever operations are needed or a view which excludes sensitive columns vs giving direct access to sensitive security related information.

Table WatchList		
id	int NOT NULL AUTOINCREMENT	PK of watch list
name	varchar(40) NOT NULL	Name of watch list
list	text	XML data

owner_id	int	id of owner 0 if global/company
company_id	int	id of owning company 0 if global
Indexes		
pk_watchlist primary key	ON id	
FK66B409ED9E5C48AF	ON owner_id	
FK66B409EDAF38CA0E	ON company_id	
Foreign Keys		
FK66B409EDAF38CA0E	(company_id) ref Company (id)	
FK66B409ED9E5C48AF	(owner_id) ref User (id)	

The WatchList table holds the data for client's instrument watch lists. Each record contains one watch list encoded as XML. Each watchlist also has a name, an id, an owner id, and an owning company.

Table companydefaults		
companyid	int NOT NULL	id of the company this default pertains to
roleid	int NOT NULL	id of the default role
Indexes		
companydefaults_uk unique	ON companyid, roleid	
companydefaults_companyid	ON companyid	
companydefaults_roleid	ON roleid	
Foreign Keys		

companydefaults_companyid	(companyid) ref Company (id)
companydefaults_roleid	(roleid) ref Role (id)

The companydefaults table holds default roleids for roles which should be assigned to any new user created in that company. Note that OMS 1.5 doesn't yet fully support this feature.

Table departmentdefaults		
deptid	int NOT NULL	id of the department this default applies to
roleid	int NOT NULL	id of the default role
Indexes		
deptdefaults_uk unique	ON deptid, roleid	
deptdefaults_deptid	ON deptid	
deptdefaults_roleid	ON roleid	

The departmentdefaults table serves the same role as the companydefaults table, except obviously it relates to departments and not companies.

Table user_roles		
userid	int NOT NULL	pk of user
roleid	int NOT NULL	pk of role
Indexes		
userid	ON (userid, roleid)	each role assignment is unique per user
FK73429949965DCAE	ON userid	
FK734299494108744	ON roleid	
Foreign Keys		
FK734299494108744	(roleid) ref Role (id)	
FK73429949965DCAE	(userid) ref User (id)	

The user_roles table provides the many-many link between the Users and Roles. An entry here assigns a JAAS Principle to the corresponding user.

Table usertemplate_roles		
templateid	int NOT NULL	pk of user template
roleid	int NOT NULL	links a role to this template
Indexes		
utr_temp_role unique	ON templateid, roleid	
utr_templateid	ON templateid	
utr_roleid	ON roleid	
Foreign Keys		
usertemplates_roles	(roleid) ref Role (id)	
usertemplates_ids	(templateid) ref usertemplates (template_id)	

The usertemplate_roles table holds references to the roles which belong to each user template.

Table usertemplates		
template_id	int NOT NULL AUTOINCREMENT	pk of user template
template_company_id	int	company id - NULL for default
template_division_id	int	division id - NULL for default
template_role_name	varchar(64)	role this template describes

template_description	varchar(255)	extended description
Indexes		
pk_usertemplates primary key	ON template_id	

The usertemplates table holds the top level information for user templates, which are intended to allow system administrators to set up ‘templates’ which will be used to build new user’s information. Each template has a scope defined by the template_company_id and template_division_id fields. The other two columns simply establish a name and description for administering the template.

View UserInfo

```
select `u`.`id` AS `id`, `u`.`companyId` AS `companyId`,
`u`.`departmentId` AS `departmentId`, `u`.`userName` AS `userName`,
`u`.`passWord` AS `passWord`, `u`.`recoveryConfirmation` AS
`recoveryConfirmation`, `u`.`state` AS `state`, `p`.`id` AS `profileid`,
 `p`.`publicEmail` AS `publicEmail`, `p`.`privateEmail` AS `privateEmail`,
 `p`.`lastName` AS `lastName`, `p`.`homePage` AS `homePage`,
`p`.`handle` AS `handle`, `p`.`firstName` AS `firstName`,
`p`.`imageName` AS `imageName`, `p`.`nightPhone` AS `nightPhone`,
`p`.`mobilePhone` AS `mobilePhone`, `p`.`fax` AS `fax`, `p`.`dayPhone`
 AS `dayPhone`, `p`.`user_id` AS `user_id`,
cast(group_concat(`ur`.`roleid` separator ',') AS char charset utf8) AS
`CAST(GROUP_CONCAT(ur.roleid) AS CHAR)` FROM ((`ets`.`User` `u` join
`ets`.`Profile` `p` on((`u`.`id` = `p`.`user_id`))) left join
`ets`.`user_roles` `ur` on((`u`.`id` = `ur`.`userid`))) group by `u`.`id`
```

The UserInfo view is used to capture all the core user-related information in one select. OMS internal logic uses this view to support user editing.

View permissiondefaults

```
select `ets`.`Role`.`id` AS `id`,`ets`.`Role`.`roleCategory` AS
`roleCategory`, `ets`.`Role`.`roleName` AS `roleName`,
`ets`.`departmentdefaults`.`deptid` AS  `deptid`,
`ets`.`CompanyDivision`.`companyId` AS `companyid` from
((`ets`.`departmentdefaults` left join `ets`.`Role` on
((`ets`.`departmentdefaults`.`roleid` = `ets`.`Role`.`id`))) left join
`ets`.`CompanyDivision`on((`ets`.`CompanyDivision`.`id` =
`ets`.`departmentdefaults`.`deptid`)) union select `ets`.`Role`.`id` AS
`id`,`ets`.`Role`.`roleCategory` AS `roleCategory`,
`ets`.`Role`.`roleName` AS `roleName`,NULL AS `deptid`,
`ets`.`companydefaults`.`companyid` AS `companyid` from
(`ets`.`companydefaults` left join `ets`.`Role` on
((`ets`.`companydefaults`.`roleid` = `ets`.`Role`.`id`)))
```

The permissiondefaults view provides a joined view of the department and company default roles. The GUI uses this.

Programming Examples

How to write MARX client applications. MARX provides a wide variety of functions which client applications can use to accomplish various tasks. These tasks fall into several broad categories:

1. Placing and Managing Orders
2. Managing Market Configurations
3. Managing User Data and Permissions
4. Acquiring Streaming Market Data
5. Acquiring Historical Market Data

Placing and Managing Orders

Orders are placed using the OMSOrderServices SOAP service, REST service, or via the equivalent FIX client interface. When using the SOAP service API the data elements required are described in the SOAP section, REST and FIX interfaces are likewise described in their own sections above. The SOAP and REST APIs are very similar and present largely the same functions and data. FIX is a higher level protocol, but many of the same concepts apply.

In order to place an order several basic pieces of information are required. First of all we need to know which trading account the order will be placed on. Trading accounts are identified by a unique integer account id. These ids can be discovered using one of several functions depending on the client's requirements. For example a client application can determine what accounts are available for their use by calling the getUserAccounts() function. This function will return an array of Account objects containing all valid accounts for the currently authenticated user. For example:

```
//Example Java code, note omsService object is our SOAP endpoint (built by Axis).
```

```
Account[] accounts = omsService.getUserAccounts();
for(Account account : accounts) {
 System.out.println("Account ID: "+account.getId()+" Account Number: "+account.getNumber());
}
```

Once we know the id of the account we can use it to place an order. However we will still need several more pieces of information. We will need to know where the order is going to be routed, what exchange it should be executed on, and what instrument is being traded. We also need to know if the trading account we are using is valid for trading at a particular location. Trading locations are represented by Destinations, each of which also has a unique integer id, the destination id. The MARX OMS and Gateway will use the destination id to determine what endpoint to route an order to. This may represent a specific LP, an intermediary, a direct link to a market, or a connection to a clearing firm. Orders also specify an exchange id (sometimes referred to as a market id) which is unique for every exchange. This value is significant when a given destination might forward orders to multiple markets or differentiate between significantly different classes of execution. Note that each Instrument also has a market id. These allow the unique identification of instruments which have the same text symbol but list at different markets.

A client can use certain functions in the order services SOAP/REST API to determine which combinations of destination, market, and accounts are valid. For example:

```
DestinationInfo[] diarray = omsService.getUserDestinationInfo();
Market selectedMarket = null;
for(DestinationInfo di : diaarray) {
if(di.getDestinationId() == 1) {
for(Market market : di.getMarkets()) {
if(market.getMarketId() == 101) {
selectedMarket = market;
}
}
}
}
if(selectedMarket != null) {
for(int acctid : selectedMarket.getAccountIds()) {
if(acctid == 42) {
Order order = new Order();
order.setAccountId(acctid);
order.setExchangeId(selectedMarket.getMarketId());
order.setDestinationId(1);
// ...
}
}
}
```

illustrates going through destination info returned by getUserDestinationInfo(). For each destination a set of Market instances indicate which markets are available on that destination. Each Market object holds the ids of valid accounts for the given combination of marketid and destinationid. The Market instance will also contain information detailing what order types, time in force values, etc are available there. For the sake of simplicity we will refer to a combination of destination id and market id as an order route.

Finally we need to know what instrument is being traded. MARX can identify instruments in 2 ways, via an instrument id or by a combination of symbol and listing market id. Note that some classes of instruments will generally only exist with a single market id. For example a stock “MSFT” might have a market id of 19 (NASDAQ). This is the listing market for this instrument. Other classes of instruments such as FOREX don't have a concept of a listing market and will appear multiple times in the symbol master, once for each LP or other market they can be traded on. In the later case the market id in an instrument will usually be the same as the market id in the order route (this might not be true in some configurations). The MarketDataService SOAP API allows access to symbol master data. Functions include getInstruments(int[] ids) and findSymbolInfo(SymbolFilter filter). In either case an array of Instrument objects is returned which match the input criteria.

Actually placing an order via the SOAP API is accomplished by calling the placeOrder(Order order) function. Several other functions provide additional ordering functionality. Calling replaceOrder(Order replacement) will replace an existing order with the new order. The id of order being replaced will need to be specified via order.setRelatedId(int replacedid). Calling cancelOrder(Order order) will simply cancel the given order (only the order id field is required in this case, the OMS will fill in other parameters automatically). It is also possible to use dockOrder(Order order) to place an order with the OMS which is not released immediately for trading. A later call to undockOrder(Order order) will release the order (again only the order id is required in this case).

The OMSOrderService SOAP API can also be used to receive execution reports. This is accomplished using the setOrderEventSource(String scope,boolean wanthistory) and the getOrderEvents() methods. The setOrderEventSource() function registers the client with the OMS SubscriptionManager component and tells it the desired scope of messages to be received. The getOrderEvents() method will return all events queued since the last call. The scope parameter to setOrderEventSource() determines which messages are queued. There are 4 possible values, 'USER', 'DEPARTMENT', 'COMPANY', and 'DELETE'. A USER scope subscription will return only events related to orders owned by the authenticated user. A DEPARTMENT scope subscription will return all messages relating to orders placed by users in the same department as the authenticated user. A COMPANY scope subscription will return all messages related to orders placed by anyone in the same company as the authenticated user. The DELETE scope simply unregisters the client, no more messages will be queued. Note that only one subscription may exist at a given time for a given user.

When getOrderEvents() is called, an array of ETSEvent objects are returned. Each one represents a single event related to one order. Once getOrderEvents() is called any given event in the queue is deleted and will not be received again. Events are returned in the order in which they occurred. There are several types of events which may be returned. An ETSRouteEvent is used to describe an existing order or a new order which has been placed. An ETSExecutionEvent represents a change in state to an order (usually corresponding to something like an incoming FIX execution report). In addition ETSLogEvent messages can be received, these describe order routing failures within the OMS. Each ETSRouteEvent message will contain an associated Order object. ETSExecutionEvent messages contain an Execution object which describes the reason for the order state change and related details. ETSLogEvent messages may contain either or both an Order object and/or an Execution object. It will always contain a LogEntry object which describes the error.

Clients can simply call getOrderEvents() on a periodic basis once they have created a subscription with setOrderEventSource() in order to monitor activity. Depending on the client's requirements any desired polling interval can be used. As long as the client's HTTP session remains valid messages will be queued.

Historical orders and executions may also be acquired automatically using getOrderEvents() if desired. Setting the boolean wanthistory argument to a true value will signal the SubscriptionManager to return ETSRouteEvent and ETSExecutionEvent messages for all currently active orders within the designated scope. The Execution objects returned will have the historical flag set to true indicating they are the result of a history request. Once all historical messages have been sent new messages will be queued as normal. Clients may also use the findOrders(OrderFilter) function, which provides greater selectivity and may be called at any time to acquire or re-acquire order status information.

Market Data

There are two aspects to market data, quote, and meta data. Quote is handled via feed handlers. Meta data is acquired via the MarketDataService SOAP API. First we will discuss meta data because this information will be required in order to effectively use the feed handler.

Meta Data and the Symbol Master

MARX holds information related to instruments in a symbol master database. Client applications access this information via the MarketDataService SOAP API. There are several useful functions here. The getInstruments(int[] ids) function has already been mentioned with respect to order management. This function allows the client to recover detailed information about a given instrument using its instrument id. The findSymbolInfo(SymbolFilter filter) function is useful for finding instrument data using a variety of search criteria. A SymbolFilter has a number of fields which correspond to attributes of instruments in the symbol master. Any of these fields which are set to a specific value will select for instruments with that particular attribute value. So for instance all instruments on a given market can be returned by setting just the exchangeid field of the filter and all instruments with a given symbol can be found by setting the symbol field. This is highly useful when you want to search for specific data but don't have an instrument id (for example when a client UI supplies just the symbol).

There are also a couple of other functions in the SOAP API which are useful. The getExchanges() function returns data related to all available markets known to the system. This will return Exchange objects which identify a market id and a corresponding human-readable description of that market. The MIC and Reuter's codes for these markets are also returned (though in many cases there are no standard values for many markets, these ids are mostly useful to the OMS gateways).

The getRoutes(String symbol) and getMarketDataRoutes() functions are quite useful for discovering the parameters required to subscribe to the feed handler for a given quote. We will touch on these again after discussing a market data subscription and how market data is identified.

The Feed Handler

Feed handlers provide market data to the client. This can be supplied in any protocol supported by the feed handler and the client. Currently 2 main protocols are supported, FIX 4.4, and MARX binary protocol. The same general information is available via either protocol but some of the details of handling subscriptions vary from one to the other. In general the binary protocol is more capable and more efficient and should be used in preference to FIX. The FIX protocol is however compatible with most client software out of the box.

In order to access quotes the feed handler needs to know exactly what information is being requested. Any given system may be acquiring data from multiple data sources. Some of that data may overlap from one source to another. For this reason it is necessary to specify several parameters which identify the precise instrument and source. Five parameters are involved: Level identifies the type of data being requested, Symbol identifies the instrument's symbol, Marketid identifies the trading venue the data is to be supplied from, Supplier identifies the source of the data, and Carrier distinguishes between different feed vendors. The combination of all of these parameters is referred to as a market data route.

When using the FIX client protocol the feed handler uses an internal mapping to identify the carrier and supplier parameters internally based on symbol, level, and market id. When using the binary protocol all of these parameters must be determined by the subscribing client. Some examples of market data routes and typical use cases follow.

Suppose a system is set up for stock trading. Market data might be available in several ways. For example the system could be acquiring NASDAQ NMS and NYSE data from Comstock, NMS from a clearing firm, and BATS quote directly from BATS. Since BATS is an ECN it doesn't list any instruments of its own, but it will have a market id. Suppose we want to subscribe for a quote to MSFT. We can find this in the symbol master using findSymbolInfo() and we discover that it has a listing market id of 19 (NASDAQ). However we still need to determine where to get the quote from. Looking at our system we find that we have quotes for MSFT from both BATS and NASDAQ NMS. Furthermore we have 2 sources of NMS quote, the clearing firm and Comstock. Let us assume we have the following mappings in our symbol master

Supplier NASDAQ Carriers COMSTOCK, CLEARING

Supplier NYSE Carrier COMSTOCK

Supplier BATS Carrier BATS

This means we have several choices if we want to look at MSFT quote. First of all you have 2 markets to look at, NASDAQ and BATS, lets say BATS has market id

10, and NASDAQ is 19. Now, if we want to look at the MSFT BATS quote we only have one choice of supplier (BATS), and carrier (again BATS). We could thus subscribe to MSFT,10,BATS,BATS,LEVEL1 (level 1 is basic stock ticker with BBO). If we wanted to look at the NASDAQ NMS quote for MSFT we would be using market id 19, but we would have 2 choices of carrier, COMSTOCK and CLEARING. Thus MSFT,19,NASDAQ,CLEARING,LEVEL1 would get us quotes carried by the clearing firm and using COMSTOCK for carrier would give us the same instrument's quotes using the COMSTOCK connection. Normally quotes for nationally marketed securities like stocks will be the same on all quotes, but there may be business reasons for a trader to favor one over the other or quality differences. For products such as FOREX which have no nationally quoted single market each feed could be quite different and traders will almost certainly need to differentiate.

Putting this together a client could subscribe to MSFT something like this. First using getExchanges() we can determine the market ids for BATS and NASDAQ. The client could then decide to ask for NASDAQ quotes. Calling getMarketDataRoutes(LEVEL1,”NASDAQ”,null,”MSFT”,19); will return 2 Route objects, one with carrier COMSTOCK and one with carrier CLEARING. Selecting one of these the client might then create a Subscription object with supplier NASDAQ, market id 19, Symbol MSFT, carrier COMSTOCK, and level LEVEL1. The Route will also contain a 'subtopic' string which contains the connection parameters for the proper feed handler instance, for example “fh1.tradedesksoftware.com:8787” might be the subtopic value. Connecting to the feedhandler at this address/port and sending the above subscription message should result in the NASDAQ quote from COMSTOCK for MSFT level 1 updates being sent back to the client.