**Download OptionsCity Freeway .jar files:**

<http://utilities.optionscity.com/files/freeway/>

**Launch DEVELOPMENT Freeway server:**

<http://64.74.102.102:10500/optionscity/optionscity/>

**Launch PRODUCTION Freeway server:**

<http://209.249.193.106:8088/optionscity/optionscity/>

**Launching Freeway servers - the LONG way:**

If you are having issues with the Freeway app launching directly from the web link, you can download and save the “onramp.jar” file then execute it manually from your local command line. Download the .jar file with the following link:

<http://209.249.193.106:8088/optionscity/optionscity/onramp/onramp.jar>

THEN, from a command line, change to the directory where you saved onramp.jar and use the following:

**java -jar onramp.jar 209.249.193.106 9041 false**

Freeway AlgoStore and DOCUMENTATION:

<http://algostore.optionscity.com/storefront/>

JDK 7 downloads:

<http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html>

Various Java downloads:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

NetBeans downloads:

<https://netbeans.org/downloads/>

Eclipse downloads:

<https://www.eclipse.org/downloads/>

javax JAR package files

<http://search.maven.org/>

# **Here are some general development notes:**

## 

## 

## com.optionscity.freeway.api.helpers package:

The Freeway API has a package labeled *com.optionscity.freeway.api.helpers*. This package contains data structures or utility classes that might be common to several trading algorithms and can save both time and coding efforts for developers. Each of the helper classes are broken down in brief detail below. It is not necessary to use any of the helper classes to code trading strategies. They exist merely as a convenience.

## Best Edge Queue

The **BestEdge** class provides a queue-like structure to track various instruments and their related edge values. It is essentially a pre-fab priority queue for instruments and their respective edge values. The code excerpt below demonstrates a class using the **BestEdge** to track the edge of several instruments, and then quickly select the best overall edge when ready to make a trade. The **BestEdge.Entry** class contains the following:

|  |  |
| --- | --- |
| double | [**edge**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/helpers/BestEdge.Entry.html#edge) |
| java.lang.String | [**instrumentId**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/helpers/BestEdge.Entry.html#instrumentId) |

## ValueTracker

The **ValueTracker** provides user with an easy way to track weighted values. Developers can continue to add value or weight, and get the adjusted ratio with a simple 'getter' call. Reset, clear, and smoothing operations are also provided. The example below demonstrates using the **ValueTracker** to automatically track ratios as events happen and then simply get the value immediately when needed. This class can be useful for tracking values based upon market events, and then accessing the value in an **onTimer()** block to update probes or log values.

## Pricing

The Pricing class contains helpful functions for price manipulation. Here are some examples:

|  |  |
| --- | --- |
| static double | [**findClosestPrice**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/helpers/Pricing.html#findClosestPrice(java.lang.String,%20double))(java.lang.String instrumentId, double price)  find closest valid price, if the price is already valid it is returned |
| static double | [**findClosestPriceDown**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/helpers/Pricing.html#findClosestPriceDown(java.lang.String,%20double))(java.lang.String instrumentId, double price)  find closest valid price rounding down, if the price is already valid it is returned |
| static double | [**findClosestPriceUp**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/helpers/Pricing.html#findClosestPriceUp(java.lang.String,%20double))(java.lang.String instrumentId, double price)  find closest valid price rounding up, if the price is already valid it is returned |
| static double | [**findRoundAwayPrice**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/helpers/Pricing.html#findRoundAwayPrice(java.lang.String,%20Side,%20double))(java.lang.String instrumentId, Side side, double price)  find closest valid price, rounding up if selling, down if buying, if the price is already valid it is returned |
| static double | [**getMarketMakingAskEdge**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/helpers/Pricing.html#getMarketMakingAskEdge(com.optionscity.freeway.api.Prices))([**Prices**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/Prices.html) prices) |
| static double | [**getMarketMakingBidEdge**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/helpers/Pricing.html#getMarketMakingBidEdge(com.optionscity.freeway.api.Prices))([**Prices**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/Prices.html) prices) |
| static double | [**getMarketTakingAskEdge**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/helpers/Pricing.html#getMarketTakingAskEdge(com.optionscity.freeway.api.Prices))([**Prices**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/Prices.html) prices) |
| static double | [**getMarketTakingBidEdge**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/helpers/Pricing.html#getMarketTakingBidEdge(com.optionscity.freeway.api.Prices))([**Prices**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/Prices.html) prices) |
| static double | [**roundNTicks**](http://utilities.optionscity.com/files/freeway/4.2/freeway_javadoc/com/optionscity/freeway/api/helpers/Pricing.html#roundNTicks(java.lang.String,%20double,%20int))(java.lang.String instrumentId, double price, int ticks)  round the provided price N ticks up/down to a valid price. |

## Parsing

The Parsing class provides an easy way to convert a YYYYMMDD date into a java Date:

#### **parseDate**

* public static java.util.Date parseDate(java.lang.String date)
* parse a String date into a java Date
* **Parameters:**
* date - the date, which is either YYYYMMDD, or YYYYMMDDHHMMSS
* **Returns:**
* the java date

# **Communication Between Jobs With Signals**

Freeway provides jobs the ability to communicate with other jobs via message publishing. The usage is similar to a pub/sub pattern and thus is event-driven like the other message types. One thing to note is that the existing Signal Object is a provided construct that Freeway uses internally for its UI-generated signals. It is provides the minimal amount of information that internal system needs to relay UI driven events and is not necessarily meant to be the every-case Object that developers use for their message passing. Developers can either utilize the existing class if it suffices, or extend it to create their own 'Signal' objects.

## Subscribing and Handling Signals

As shown in the [Event Handling](http://wiki.optionscity.com/display/FWPUB/Inter-Job+Communication#) section, a job subscribes to signals in the same way that it subscribes to other event messages.

|  |
| --- |
| public void begin(IContainer container) {  super.begin(container);    container.subscribeToSignals();  } |

The handling of Signals is a little different than the other events, however. Currently, signal are broadcast events not matched to a particular job. Any job that has subscribed to Signal events will receive every signal that is published by another job. It is up to the receiving job to determine which signals it wants to handle.

## Customizing Job Signals

The code excerpt below demonstrates a custom signal class that has extended the base Signal class. This custom signal can then provide additional details specific to its context. In this case, the message provides an instrumentId and price. You can imagine the receiving job then using this information to complete an action such as trading on the specified instrument id when the specified price is reached. This design pattern not only allows one 'controller' job to communicate with several receiver jobs and change multiple jobs dynamically, but also promotes the abstraction of logic into task-specific jobs.

|  |
| --- |
| public class CustomJobSignal extends Signal {    public final String instrumentId;  public final double price;    public CustomJobSignal(String instrumentId, double price) {  super(null, "Price Target Changed");  this.price = price;  this.instrumentId = instrumentId;  }  } |

## Sender / Receiver Example

The code excerpts below provide a typical use-case for inter-job communcation. The sender job is monitoring events for specific conditions and then signals one or more receiver jobs to begin acting within some type of context when that condition is met. In this case, the sender is just waiting for 'X' amount of bid/ask events and then notifying the receiver that the condition has been met.

Sender example:

|  |
| --- |
| public class SenderJob extends AbstractJob {    public void install(IJobSetup setup) {  setup.setDefaultDescription("simple sender job");  setup.addVariable("desired", "desired quantity, or 0 for all", "int", "0");  setup.addVariable("instruments", "list of instruments", "instruments", "");  setup.addVariable("aggressiveness", "how aggressive?", "choice:0=timid;1=normal;2=raging", "1");  }    int desired;  int total=0;    public void begin(IContainer container) {  super.begin(container);    desired = getIntVar("desired");  total = 0;    container.subscribeToMarketBidAskMessages();  container.filterMarketMessages(getStringVar("instruments"));  }    public void onMarketBidAsk(MarketBidAskMessage msg) {  if(++total==desired){  container.signal(new Signal("sender", "I saw " + total + " interesting instruments"));  total=0;  }  }  } |

Receiver example:

|  |
| --- |
| public class ReceiverJob extends AbstractJob {    public void install(IJobSetup setup) {  setup.setDefaultDescription("simple receiver job");  }    public void begin(IContainer container) {  super.begin(container);    container.subscribeToSignals();  }    public void onSignal(Signal signal) {  log(signal.sender+" says \""+ signal.message+"\"");  }  } |

Overloading the onSignal Callback

To improve performance, we recommend overloading the onSignal callback to specify which types of Signal objects your onSignal handler is listening to. When the system receives a signal, it is then able to selectively pass the signal only to the onSignal methods that have the proper method signature.

In the following example, there are 4 different onSignal methods that will be called depending on the type of signal that is passed.

|  |
| --- |
| public void onSignal(UserSignal signal) {  log("I received a UserSignal");  }    public void onSignal(GridChangeSignal signal) {  log("I received a GridChangeSignal");  }    public void onSignal(ExternalAPIMessage signal) {  log("I received an ExternalAPIMessage");  }    public void onSignal(NoActiveUsers signal) {  log("I received an NoActiveUsers signal");  } |

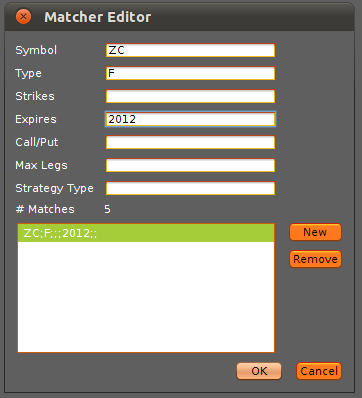
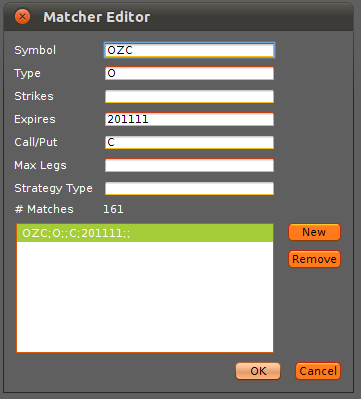
The instrument matcher is a utility provided to help users more easily select their filtering expressions. Freeway job developers are encouraged to utilize the API's filtering methods to narrow the event scope to only those that it cares about. The freeway **Container** class provides methods for adding these filters as touched upon briefly in the [InstrumentService](http://wiki.optionscity.com/display/FWPUB/Instrument+Matcher#) section. The signature for the filter methods can be seen in the code excerpt below:

|  |
| --- |
| /\*\*  \* filter market events to specific instruments  \*  \* @param instruments a comma delimited String of specific instruments to allow or a filtering expression. All other instrument notifications will be ignored.  \*/  void filterMarketMessages(String instruments); |

The method takes a String as a parameter. This String can either be a specific Instrument id that is given to the user via events messages, or it can be an **Expression** that matches multiple instruments. To aid users in the creation of these expressions, freeway has provided the Instrument Matcher to help formulate the expression String objects.

## Declaring an Instrument Matcher

The instrument matcher is declared as a **Variable** type during the job installation. When a developers declares an 'instrument' type variable, that variable will be displayed in the UI console's configuration menu similar to any other variable. Instead of having a simple textfield where users can specify an explicit variable, however, a widget will be displayed when a user double clicks to modify the variable's value. The displayed widget can be seen in the image below:

A user can create a new matching expression by following the steps below:

1. Bring up the instrument matcher by double-clicking the 'Value' cell of the relevant variable
2. Click the 'New' button
3. Enter a Symbol: ZC, OZC, NG, G, etc.
4. Enter a Type: F - Future, O - Option, S - Strategy, E - Equity, I - Index
5. Enter a Strike if relevant: 10, 10.5, 10.25, 20, etc.
6. Enter an Expiration if relevant: 2011, 201110, 20111009. **See 'Expiration Date Formats' below for more information on expiration dates in the matcher.**
7. Specify call or put if relevant: C or P
8. Enter Max Legs if relevant: 1, 2, etc.
9. Enter a Strategy Type if relevant: VRTCL, 3WAY, etc. The naming convention of strategy types is the same format as seen in OptionsCity, Metro.
10. Click 'New' to repeat this process and add an additional filter, or 'Ok' to save the settings.

It should be noted that only the symbol field is required. The other fields provide the opportunity for additional filtering if desired or relevant. Please see the [Training Guide](http://wiki.optionscity.com/display/FWPUB/Instrument+Matcher#) for additional details on this process. Once the expressions are saved, the variable can be accessed within the code similar to any other type of variable. The String returned will be in a format specific to freeway's internal matching logic. It is encouraged the developers utilize the Intrument Matcher to decouple their code from maintanence of expression formats. If the matcher widget is utilized, a developer can be assured that the widget will always return the proper String format and that is will in turn be handled properly within the expression matching logic. The code excerpt below deomnstrates a job 'getting' the expression variable and utilizing it to filter market messages.

|  |
| --- |
| public class HelloWorld extends AbstractJob {    public void install(IJobSetup setup){  setup.addVariable("instruments","instruments to filter on","instruments","");  }    public void begin(IContainer container){  super.begin(container);    container.subscribeToMarketBidAskMessages();    container.filterMarketMessages(getStringVar("instruments"));  }    // Due to filtering, only events for matching instruments will cause the invocation of this method  public void onMarketBidAsk(MarketBidAskMessage m) {  log("I have received a Bid/Ask event for " + m.instrumentId);  }    } |

### Expiration Date Formats

Many exchanges, including the CME, actually send 2 different expirations for each instrument: the **display expiration** and the **contract expiration.**

The display expiration is a string that looks like the following, using the standard 3 letter month codes: JAN13, FEB13, MAR14, etc. This indicates the standard notion of the month of the option.

The contract expiration is a number string that encodes the date the contract will cease to exist. These look like: 20130920, 20140319, etc. They will always be 8 digits, in YYYYMMdd format indicating the exact year, month and date that the contract ceases to be listed.

For most options, the expiration of the option falls inside of the option month. ES options, for example, expire on the third Friday of the listed option month – SEP13 ES options expire on September 20, 2013 (20130920). For some products however, such as treasuries, the expiration of the option is actually before the option month listing. So, for example, the SEP13 OZB options expire August 23, 2013 (20130823), per the convention set by the CME.

Strategies add even more complications – the **contract expiration** for a strategy may have nothing to do with the expiration of the legs. On the CME, for example, when a new strategy is first created via an RFQ, the strategy receives an expiration date of two weeks from the day it is created. The legs themselves may not expire for 6 months or more from the date that the strategy is created, but the exchange will stop listing that strategy after 2 weeks if there is no activity on it. The **display expiration,** however, reflects the expiration of the shortest-dated leg. When the shortest dated leg expires, then the strategy no longer exists as a strategy. For example, if today is July 10, 2013, a newly created DEC14-MAR14 call spread will have a **contract expiration** of July 24th (20130724) and a **display expiration** of DEC14.

|  |  |
| --- | --- |
|  | The instrument matcher supports both types of expirations depending on how the user enters the date. If the user enters JAN13-APR13, then all options and strategies that have a **display expiration** from JAN13 to APR13 (inclusive) will pass the filter. On the other hand, if the user enters 201301-201304, all options and strategies that have a **contract expiration** between 201301 and 201304 (inclusive) will pass the filter.  To ensure that you're matching the correct expiration, we typically recommending using the **display expiration** format, since it will always match what the exchange considers the "month" of the contract. For more information about contract expirations and display months, please see the contract specifications for the exchange you're trading on. |

# **Shared Libraries in JAR Files**

Freeway supports shared libraries in the form of JAR uploads. To use this functionality, create a JAR file with the complied shared library class files (all in the default package) and upload it as you would upload a job. The JAR file itself does not need to be 'started' or otherwise loaded – as soon as you upload, the classes in the JAR will be available to all of your jobs. To update the JAR, simply re-upload another JAR with the same filename – the new JAR will replace the old one on the server.

To avoid class conflicts with your libraries, please check the following as you build out your environment:

* Do not duplicate class names in any of your shared libraries. This includes across different JARs. Since all JARs are in the default package, duplicate class names in your uploaded JARs will conflict
* Do not package your Freeway job in your JAR file. Job files are intended to be uploaded separately so that they can be managed in the main interface
* Do not package the Freeway API file (freeway.jar) in your JAR file
* If you upload JAR files that have duplicate class names, be sure to remove all but the most current file with your libraries. You can see all JARs currently loaded (and remove unused ones) in the 'Add' screen.
* Be sure that all jobs are stopped before changing or removing any JAR files. Because jobs share library files, this is the only way to ensure jobs and JARs are reloaded correctly on library file changes

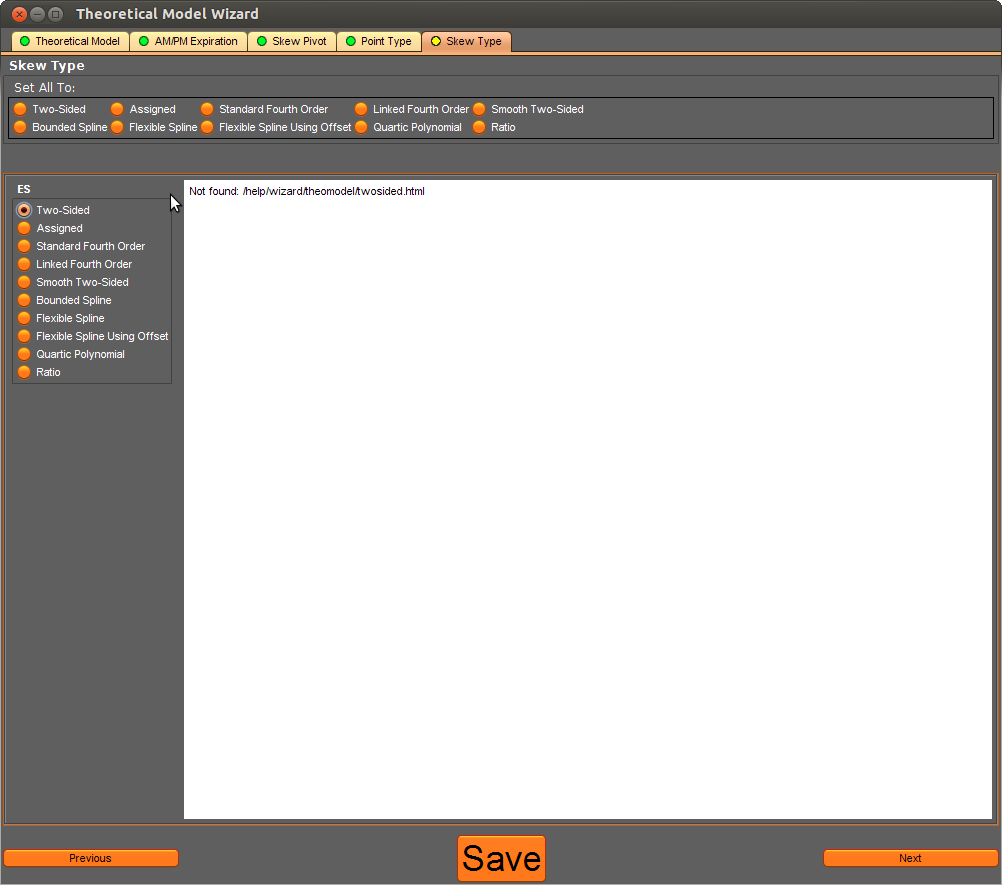
Using shared libraries also presents the usual issues that affect concurrent systems. We recommend:

* All shared libraries should be thread safe. Multiple jobs can access the same library at the same time, so it's important to use synchronized data structures
* Signals should be immutable. All variables inside a signal class should be final.

Freeway provides getter and setter methods to retrieve, modify and set the skew curves and vol paths used for determining theos on CityServer. Changes made to curves in Freeway take effect immediately, and will be appear in the Model Settings view in Metro.

# **Skew Curves**

To manipulate your skew curve, first determine the type of curve you are using. If you have the Metro front end, this information can be found in the Theoretical Model Wizard in the last tab:

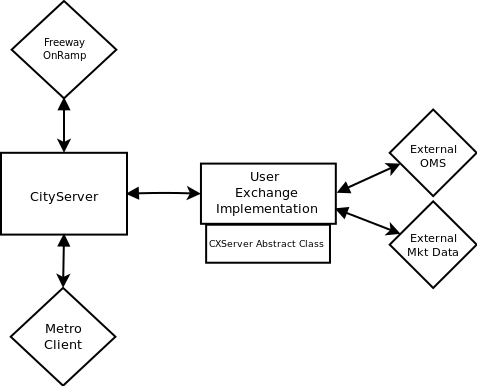


From here, you can now retrieve the curve in Freeway and configure the settings appropriate to that curve. The code below shows the appropriate getter method and the fields that can be changed for the Two Sided curve (selected). Other curves have different fields that determine their shape – for more information, see the [Volatility Curve reference](http://utilities.optionscity.com/freeway/javadoc/4.1.3/com/optionscity/freeway/api/VolatilityCurve.html) section in the Javadoc.

# **CX Exchange - Introduction**

The CX Exchange is an API designed to allow you to easily create your own exchange implementation to bring in third-party market data and order management systems. The CX Exchange API provides callbacks for exchange operations (such as order submissions, cancels, market data updates) and allows you to write your logic to handle those callbacks. Integrating third party data and order systems involves simply passing the appropriate information between the CX API and the third party APIs.

To use the CX Exchange, we suggest writing two classes – one to override some or all of the callback methods in the abstract CXServer class, and one to serve as the standalone class to launch your server. Once you've written these classes, you can simply start your exchange implementation class on your server, and point your clients (Metro and/or Freeway) to the correct hostname and port where the exchange implementation is running.



# **CX Exchange - Getting Started**

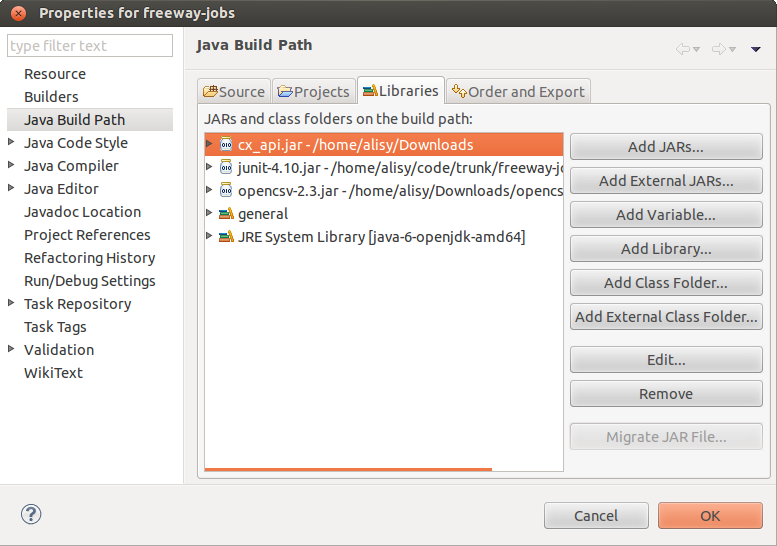
## CXServer API Download

To get started, download the following JARs and place them in your build path.

* [messaging.jar](http://wiki.optionscity.com/download/attachments/3900725/messaging.jar?version=1&modificationDate=1387471209555)
* [cx.jar](http://wiki.optionscity.com/download/attachments/3900725/cx.jar?version=1&modificationDate=1387471226976)
* [jna.jar](http://wiki.optionscity.com/download/attachments/3900725/jna.jar?version=1&modificationDate=1387471252527)
* [log4j-1.2.17.jar](http://wiki.optionscity.com/download/attachments/3900725/log4j-1.2.17.jar?version=1&modificationDate=1387471270429)

## Eclipse Configuration

Once you've downloaded the CX API jars, add it to your class/build path in Eclipse. To do this, right click on your package, select 'Build Path' -> 'Configure Build Path'. In the 'Libraries' tab, click 'Add External JARs' and locate the files downloaded from the links above. Once you've done this, Eclipse will be able to autocomplete methods and members in the CXServer class and dynamically check your code.



## Creating the Outer Class

To get started, first create your own standalone class to wrap the CXServer abstract class. This class will instantiate the server, provide a main(), and start listening for connections. For now, leave this class mostly blank, as we need to first extend CXServer before we can instantiate it.

# 

# 

# **Extending CXServer**

Inside of your outer standalone class, create a new class that extends the CXServer abstract class. In this inner class, choose which market event listeners that you would like to handle, and write custom logic to override and respond to them appropriately. For the list of events and the corresponding responses, [see the Javadoc.](http://utilities.optionscity.com/cx/javadoc/index.html)

|  |
| --- |
| //Sample implementation of CXServer    class MyCXServer extends CXServer {  public MyCXServer(String name, int port) {  super(name, port);  }  @Override  protected void onStartSymbol(String symbol) {  }  @Override  protected void onNewOrder(NewOrder request, String session) {  OrderStatus os=new OrderStatus();  OrderRequestReject reject = new OrderRequestReject();  reject.orderId=request.orderId;  reject.reason="ordering not supported";  send(session,reject);  }  @Override  protected void onModifyOrder(ModifyOrder request, String session) {  OrderRequestReject reject = new OrderRequestReject();  reject.orderId=request.orderId;  reject.reason="ordering not supported";  send(session,reject);  }  @Override  protected void onCancelOrder(CancelOrder request, String session) {  OrderRequestReject reject = new OrderRequestReject();  reject.orderId=request.orderId;  reject.reason="ordering not supported";  send(session,reject);  }    ///  // ... more overrides ...  ///  } |

## 

## 

## Responding to Requests

For each type of event, your event handler function should send a specific type of response, depending on the action triggered by the event. For example, here are [Javadoc](http://utilities.optionscity.com/cx/javadoc/index.html) listings for several event listeners:

|  |  |
| --- | --- |
| protected abstract void | **onCancelOrder**(CancelOrder request, java.lang.String session)  if cancel is accepted, send an OrderStatus, else send a OrderRequestReject |
| protected abstract void | **onCreateStrategy**(CreateStrategy request, java.lang.String session)  always send an CreateStrategyAck in response |
| protected abstract void | **onMassQuote**(MassQuote request, java.lang.String session)  always send an MassQuoteAck in response |
| protected abstract void | **onMassQuoteCancel**(MassQuoteCancel request, java.lang.String session)  always send an MassQuoteAck in response |
| protected abstract void | **onModifyOrder**(ModifyOrder request, java.lang.String session)  if modify is accepted, send an OrderStatus, else send a OrderRequestReject |
| protected abstract void | **onNewOrder**(NewOrder request, java.lang.String session)  if order is accepted, send an OrderStatus, else send a OrderRequestReject |
| protected abstract void | **onOrderStatusRequest**(OrderStatusRequest request, java.lang.String session)  if status is accepted, send an OrderStatus, else send a OrderRequestReject |
| protected abstract void | **onRequestForQuote**(RequestForQuote request, java.lang.String session)  always send a RequestForQuoteAck in response, successful requests should also cause a QuoteRequest to be sent |
| protected void | **onSessionClose**(java.lang.String session)  called when an active session closes/disconnects |
| protected void | **onSessionOpen**(java.lang.String session)  called when a session if first opened |
| protected abstract void | **onStartSymbol**(java.lang.String symbol)  this method is called for the first subscription to a symbol by a client. |

These responses are sent via the send() method in the CXServer class. For detailed descriptions of the proper responses for each method as well as the method signatures for the send() function, please see the CXServer section of the Javadoc.

# **Server Configuration**

The Freeway server will connect to your implementation of the CXServer, but to do so it will need some configuration options. These options need to be configured on the server before the full communication between the Freeway server and your implementation of the CX server can work properly. **For most users, OptionsCity support will handle the necessary on-server configuration.**

|  |  |  |
| --- | --- | --- |
| Key | Value | Description |
| UseIdSymbolAsSecurityId | false | Bypass keeping track of your own mapping between OC securityId and your own internal securityId. It is recommended this be false until users are more familiar with the OC idSymbol format |
| InstrumentDownloadEnabled | true | Enable this CX Exchange to send instrument definitions explicitly |
| MarketDataEnabled | true | Enable this CX Exchange to send market data |
| host | "" | Host of the remote CXServer. This is the hostname that the Freeway Server will connect to |
| port | "" | Port of the remote CXServer. This is the port that the Freeway Server will connect to |
| user | "" | User for this session. The specificed user is passed to the remote CXServer for authentication |
| password | "" | Password for this session. The specified password is passed to the remote CXServer for authentication |

# 

# 

# **Starting the Exchange**

Now that you have a shell standalone outer class and an inner class that extends CXServer, you can instantiate and start your exchange. Start by creating a new server, then add a start() method where you set up users and instruments.

In the CX Exchange, users are unique and identify the instance of the Metro or Freeway server that is connecting to the exchange. In all callbacks, the session identifier will be passed to the callback function as a string called 'session'. The CX exchange itself does not preserve any state, so the users must be loaded each time the exchange process is started.

Once users are added, the next step is to add the instruments to the server. Again, the CXServer abstract class does not preserve any state of its own, so your standalone class must re-initialize all instruments each time it starts up. In many applications, it may make sense to store instrument definitions in a database and read the definitions in at startup.

|  |
| --- |
| public class TestCXExchange {  final CXServer server;  public TestCXExchange(String name, int port) {  server = new MyCXServer(name,port);  }  public void start() throws IOException {  // in a real implementation, set the valid users here...  // they can also be added real-time after the server starts but this is usually not  // required  server.addUser("test","test");  // for demonstration purposes, define and send a equity and an option  // these can also be done in real-time after the server starts, i.e. in  // response to exchange instrument definitions  server.send(createEquity());  server.send(createCall());    // now start the server listening for connections  server.start();    // the following is for demonstration purposes, and just generates random  // market data every second  new Thread(new SendMarketData(server,1000,1000)).start();  }  // ...  // Create demo equities instrument and options instrument  private Instrument createEquity() {  Instrument equity = new Instrument();  equity.symbol="IBM";  equity.securityId ="1";  equity.type= Instrument.Type.EQUITY;  equity.minPriceIncrement=.01;  return equity;  }  private Instrument createCall() {  Instrument call = new Instrument();  call.symbol="IBM";  call.strike=100;  call.securityId ="2";  call.type= Instrument.Type.CALL;  call.underlyingSecurityId="1";  call.expiration=nextMonth();  call.minPriceIncrement=.25;  return call;  }  } |

## Sending Market Data

To populate your exchange implementation with market data, you send that data via the send() method in your server instance. In the [Javadoc](http://utilities.optionscity.com/cx/javadoc/index.html), you'll find several method signatures for sending market data to your exchange:

|  |  |
| --- | --- |
| void | **send**(BookDepth book)  send a book depth to all interested sessions |
| void | **send**(Bulletin bulletin)  send an exchange bulletin to interested sessions |
| void | **send**(Instrument instrument)  send an instrument to all interested sessions |
| void | **send**(Market market)  send a market message to all interested sessions |
| void | **send**(QuoteRequest rfq) |

To send this information, create the appropriate objects (using definitions in the Javadoc) and send them to your server:

|  |
| --- |
| BookDepth book = new BookDepth();  book.securityId ="1";  book.bids=new BookDepth.Level[1];  book.bids[0]=new BookDepth.Level();  book.bids[0].orders=1;  book.bids[0].price=price-.05;  book.bids[0].quantity=10;  book.asks=new BookDepth.Level[1];  book.asks[0]=new BookDepth.Level();  book.asks[0].orders=1;  book.asks[0].price=price+.05;  book.asks[0].quantity=10;  server.send(book);  Market market = new Market();  market.securityId ="1";  market.type= Market.Type.LAST;  market.price=price;  market.quantity=1;  server.send(market); |

|  |  |
| --- | --- |
|  | In many cases, to interface with a third-party market data provider, you will likely want to create a listener thread that will wait for incoming information, repackage it, and pass it along to your exchange implementation. |

# **Full TestCXExchange Implementation Sample Code**

|  |
| --- |
| import java.io.IOException;  import java.util.Calendar;  import java.util.Date;  import java.util.Random;  import com.optionscity.cx.api.BookDepth;  import com.optionscity.cx.api.CancelOrder;  import com.optionscity.cx.api.CreateStrategy;  import com.optionscity.cx.api.CreateStrategyAck;  import com.optionscity.cx.api.Instrument;  import com.optionscity.cx.api.Market;  import com.optionscity.cx.api.MassQuote;  import com.optionscity.cx.api.MassQuoteAck;  import com.optionscity.cx.api.MassQuoteCancel;  import com.optionscity.cx.api.ModifyOrder;  import com.optionscity.cx.api.NewOrder;  import com.optionscity.cx.api.OrderRequestReject;  import com.optionscity.cx.api.OrderStatus;  import com.optionscity.cx.api.OrderStatusRequest;  import com.optionscity.cx.api.RequestForQuote;  import com.optionscity.cx.api.RequestForQuoteAck;  import com.optionscity.cx.api.exchange.CXServer;  /\*\*  \* a simple CX implementation for testing, creates a few instruments and sends random market data  \*/  public class TestCXExchange {  final CXServer server;  public TestCXExchange(String name, int port) {  server = new MyCXServer(name,port);  }  public void start() throws IOException {  // in a real implementation, set the valid users here...  // they can also be added real-time after the server starts but this is usually not  // required  server.addUser("test","test");  // for demonstration purposes, define and send a equity and an option  // these can also be done in real-time after the server starts, i.e. in  // response to exchange instrument definitions  server.send(createEquity());  server.send(createCall());    // now start the server listening for connections  server.start();    // the following is for demonstration purposes, and just generates random  // market data every second  new Thread(new SendMarketData(server,1000,1000)).start();  }  public static class SendMarketData implements Runnable{  CXServer server;  int iterations, sleep;  public SendMarketData(CXServer server, int iterations, int sleep){  this.server=server;  this.iterations=iterations;  this.sleep=sleep;  }  public void run(){  for (int i = 0; i<iterations; i++){  double price = 100;  Random r = new Random();  boolean direction = r.nextBoolean();  if(direction)  price+=.01;  else  price-=.01;  BookDepth book = new BookDepth();  book.securityId ="1";  book.bids=new BookDepth.Level[1];  book.bids[0]=new BookDepth.Level();  book.bids[0].orders=1;  book.bids[0].price=price-.05;  book.bids[0].quantity=10;  book.asks=new BookDepth.Level[1];  book.asks[0]=new BookDepth.Level();  book.asks[0].orders=1;  book.asks[0].price=price+.05;  book.asks[0].quantity=10;  server.send(book);  Market market = new Market();  market.securityId ="1";  market.type= Market.Type.LAST;  market.price=price;  market.quantity=1;  server.send(market);  try {  Thread.sleep(sleep);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  }  public static void main(String[] args) throws IOException {  if(args.length!=2){  System.err.println("usage: TestCXExchange name port");  System.exit(1);  }  TestCXExchange exchange = new TestCXExchange(args[0],Integer.parseInt(args[1]));  exchange.start();  }  class MyCXServer extends CXServer {  public MyCXServer(String name, int port) {  super(name, port);  }  @Override  protected void onStartSymbol(String symbol) {  }  @Override  protected void onNewOrder(NewOrder request, String session) {  OrderStatus os=new OrderStatus();  OrderRequestReject reject = new OrderRequestReject();  reject.orderId=request.orderId;  reject.reason="ordering not supported";  send(session,reject);  }  @Override  protected void onModifyOrder(ModifyOrder request, String session) {  OrderRequestReject reject = new OrderRequestReject();  reject.orderId=request.orderId;  reject.reason="ordering not supported";  send(session,reject);  }  @Override  protected void onCancelOrder(CancelOrder request, String session) {  OrderRequestReject reject = new OrderRequestReject();  reject.orderId=request.orderId;  reject.reason="ordering not supported";  send(session,reject);  }  @Override  protected void onOrderStatusRequest(OrderStatusRequest request, String session) {  OrderRequestReject reject = new OrderRequestReject();  reject.orderId=request.orderId;  reject.reason="ordering not supported";  send(session,reject);  }  @Override  protected void onMassQuote(MassQuote request, String session) {  MassQuoteAck ack = new MassQuoteAck();  ack.state= MassQuoteAck.State.REJECTED;  ack.reason="mass quotes not supported";  send(session,ack);  }  @Override  protected void onMassQuoteCancel(MassQuoteCancel request, String session) {  MassQuoteAck ack = new MassQuoteAck();  ack.state= MassQuoteAck.State.REJECTED;  ack.reason="mass quotes not supported";  send(session,ack);  }  @Override  protected void onCreateStrategy(CreateStrategy request, String session) {  CreateStrategyAck ack = new CreateStrategyAck();  ack.state= CreateStrategyAck.State.REJECTED;  ack.reason="create strategy not supported";  send(session,ack);  }  @Override  protected void onRequestForQuote(RequestForQuote request, String session) {  RequestForQuoteAck ack = new RequestForQuoteAck();  ack.requestId=request.requestId;  ack.state= RequestForQuoteAck.State.REJECTED;  ack.reason="rfq not supported";  send(session,ack);  }  }  private Instrument createEquity() {  Instrument equity = new Instrument();  equity.symbol="IBM";  equity.securityId ="1";  equity.type= Instrument.Type.EQUITY;  equity.minPriceIncrement=.01;  return equity;  }  private Instrument createCall() {  Instrument call = new Instrument();  call.symbol="IBM";  call.securityId ="2";  call.strike=100;  call.type= Instrument.Type.CALL;  call.underlyingSecurityId="1";  call.expiration=nextMonth();  call.minPriceIncrement=.25;  return call;  }  private Date nextMonth() {  Calendar c = Calendar.getInstance();  c.setTime(new Date());  c.add(Calendar.MONTH,1);  return c.getTime();  }  } |

# **Running Your CX Server**

The final step to getting a CX implementation working is running the CXServer code so that your CityServer instance can connect to it. Once the proper configuration is made server-side, CityServer will periodically poll the address and port that it expects to find your server instance running on. Once you've started your CXServer implementation, the two will connect and market data can flow between them.

## Starting the Server

Starting the server is fairly simple – just compile and run your classfile with the API jars in the classpath. For parameters, pass the CX server name and the port. The server name can be arbitrarily assigned.

|  |
| --- |
| freewayuser@localhost:/home/freewayuser$ java -cp <classpathfiles> TestCXExchange OPTIONSCITYCX 15000 |

# **Reading/Writing Local Files on Freeway Server**

Freeway supports writing local files on the server. This functionality is designed to allow you to write out your own datafiles or logfiles, import your own custom data or models, or store configuration files outside of your jobs. We provide two means of accessing these files – through standard Java file IO, and through SFTP. The directory itself that is permissioned for user reads and writes is called 'jobfiles' and is located in the working directory of your job.

## Accessing Your Local Directory

The directory that you have access to store files is located in the working directory and is called 'jobfiles'.

The following code reads a CSV located in this directory and outputs the contents to a log message.

|  |
| --- |
| import java.io.FileNotFoundException;  import java.io.FileReader;  import java.io.IOException;  import java.util.List;  import au.com.bytecode.opencsv.CSVReader;  import com.optionscity.freeway.api.AbstractJob;  import com.optionscity.freeway.api.IContainer;  import com.optionscity.freeway.api.IJobSetup;    public class ReadCSV extends AbstractJob {  CSVReader reader;  List<String[]> entries;  public void install(IJobSetup setup){  setup.setDefaultDescription("Reads a CSV and displays info");  }  public void begin(IContainer container){  super.begin(container);  try{  reader=new CSVReader(new FileReader("./jobfiles/mydata.csv"));  } catch (FileNotFoundException e) {  log("File not found " + e.toString());  }  try{  entries = reader.readAll();  } catch (IOException e){  log("IO Exception " + e);  }  for (String[] line : entries){  StringBuilder sb= new StringBuilder("Line: ");  for (String entry : line){  sb.append(entry+" ");  }  log(sb.toString());  }  }  } |

The directory also has write access, so similar code can easily write out data to a file for later retrieval by a job or through SFTP.

## SFTP Access

We also allow access to download and upload files in this directory through SFTP. To enable this, please contact OptionsCity support.

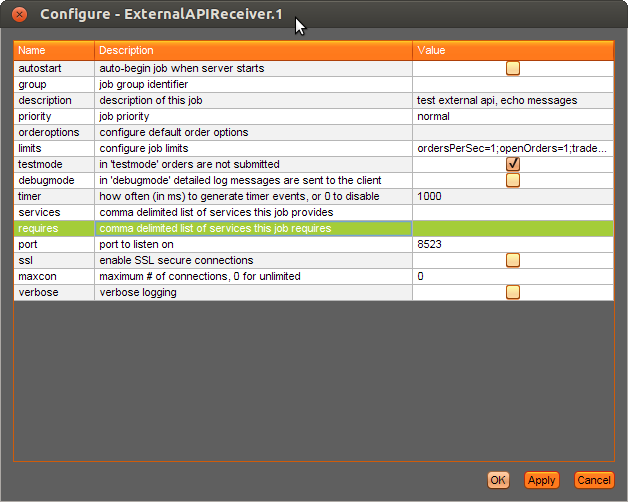
# **The ExternalAPI**

Freeway supports bidirectional connections to external services via the External API. The External API allows outside applications to connect directly to Freeway jobs, passing data over a socket connection. The data itself is formatted as UTF-16 characters, separated by a carriage return, line feed, or carriage return + line feed.

## The ExternalAPI Receiver Job

### Configuration

The recommended way of interacting with the External API is through a receiver job that extends the ExternalAPI abstract class. Extending this class automatically creates several configurations options on the receiver job that allow you to specify how to connect to the job.



Once you've loaded the job that extends ExternalAPI, your job will have 4 new settings. You must configure a port, and can optionally use SSL, limit connections, and enable verbose logging.

### Processing and Dispatching Messages

When new messages come into the port that you've configured your ExternalAPI job to listen on, you'll receive those in the onSignal() callback as ExternalAPIMessage objects. Messages are delimited by carriage returns, line feeds, or carriage return + line feed, so a new ExternalAPIMessage signal will be generated each time a new line break is sent.

Once your receiver job has received and processed a message from your external server, it can pass the appropriate information out to other jobs via standard signals.

## Code Samples

### Receiver Job

The following code sample illustrates the basics of receiving a message from the socket. Most of the work is done for you in the ExternalAPI superclass, so the only logic to implement in the receiver job is the message passing to the other jobs and any communication back with the originating external server.

|  |
| --- |
| import com.optionscity.freeway.api.IJobSetup;  import com.optionscity.freeway.api.jobs.ExternalAPI;    public class ExternalAPIReceiver extends ExternalAPI {  @Override  public void install(IJobSetup setup) {  super.install(setup);  setup.setDefaultDescription("test external api, echo messages");  }  public void onSignal(ExternalAPIMessage signal){  String[] parts = signal.message.split(",");  log(parts[0] + " - " + parts[1]);    // Our sample logic just checks for an arbitrary formatting of "echo, message"  // and communicates with the server whether the message was formatted properly  try {  if(parts.length!=2 || !"echo".equals(parts[0])){  send(signal.session,"error, format is echo,somemessage");  } else {  send(signal.session,"reply,you said '"+parts[1]+"'");  }  } catch(UnknownSession ex){}  }  } |

### SSL Socket External Sender Job (OrderSpeedTest)

This job is an external job that runs outside of Freeway. It opens a secure socket and sends data to the Freeway job.

|  |
| --- |
| package com.optionscity.freeway.externalapi;  import com.optionscity.citylibrary.monitoring.Timing;  import com.optionscity.citylibrary.scheduling.ManagedThread;  import javax.net.ssl.SSLContext;  import javax.net.ssl.SSLSocketFactory;  import javax.net.ssl.TrustManager;  import javax.net.ssl.X509TrustManager;  import java.io.\*;  import java.net.InetSocketAddress;  import java.net.Socket;  import java.security.KeyManagementException;  import java.security.NoSuchAlgorithmException;  import java.security.SecureRandom;  import java.security.cert.CertificateException;  import java.security.cert.X509Certificate;  import java.util.HashSet;  import java.util.Set;  /\*\*  \* test speed of sending orders via the External API to freeway  \*/  public class OrderSpeedTest {  public static void main(String[] args) throws IOException, InterruptedException, KeyManagementException, NoSuchAlgorithmException {  if(args.length>5 || args.length<4 || (args.length==5 && !"-ssl".equals(args[0]))){  System.err.println("usage: OrderSpeedTest {-ssl} host port #orders delay");  System.exit(1);  }  boolean ssl=false;  int pIndex=0;  if(args[pIndex].equals("-ssl")){  ssl=true;  pIndex++;  java.lang.System.setProperty("sun.security.ssl.allowUnsafeRenegotiation", "true");  }  Thread.currentThread().setPriority(Thread.MAX\_PRIORITY);  ManagedThread.configureCallingThread();  int norders = Integer.parseInt(args[pIndex+2]);  long delay = Integer.parseInt(args[pIndex+3]);  Socket s = ssl ? configureSSL().createSocket() : new Socket();  s.connect(new InetSocketAddress(args[pIndex+0],Integer.parseInt(args[pIndex+1])));  s.setTcpNoDelay(true);  s.setSendBufferSize(64 \* 1024);  s.setReceiveBufferSize(64\*1024);  BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));  BufferedReader br = new BufferedReader(new InputStreamReader(s.getInputStream()));  Timing roundtrip = new Timing(true);  long start = System.currentTimeMillis();  Set<Long> seen = new HashSet<Long>();  int left=norders;  while(left>0) {  long sendTime = System.nanoTime();  bw.write("order,IBM-E,1,100");  bw.newLine();  bw.flush();  while(true) {  String response = br.readLine();  if(response==null) {  System.err.println("error reading socket");  return;  }  if(response.endsWith("BOOKED")){  int space = response.indexOf(' ',13);  long orderId = Long.parseLong(response.substring(13,space));  if(!seen.add(orderId)){  // this can happen because the when the order manager gets the "new" notification, but then gets  // the order the status has already been book. Since the external order manager reports the "current"  // state we might see two booked, so just ignore  continue;  }  roundtrip.end(sendTime);  bw.write("cancel,"+orderId);  // System.out.println("canceled "+orderId);  bw.newLine();  bw.flush();  left--;  if(delay>0)  Thread.sleep(delay);  break;  } else if(response.startsWith("rejected,")){  System.out.println("error: "+response);  }  }  }  long end = System.currentTimeMillis();  System.out.println("processed "+norders+" in "+(end-start)+"ms, avg "+((end-start)/(double)norders));  System.out.println("roundtrip times "+roundtrip.snapshot());  s.getOutputStream().close();  s.close();  }  private static SSLSocketFactory configureSSL() throws NoSuchAlgorithmException, KeyManagementException {  SSLContext sslContext=SSLContext.getInstance("TLS");  TrustManager[] trustAllCerts = new TrustManager[] { new X509TrustManager() {  public java.security.cert.X509Certificate[] getAcceptedIssuers() {  return new java.security.cert.X509Certificate[] {};  }  public void checkClientTrusted(X509Certificate[] chain,  String authType) throws CertificateException {  }  public void checkServerTrusted(X509Certificate[] chain,  String authType) throws CertificateException {  }  } };  //init context  sslContext.init(  null,  trustAllCerts,  new SecureRandom()  );  return sslContext.getSocketFactory();  }  } |

### SSL Socket Receiver Freeway Job (OrderSpeedTest)

This is the corresponding code for the receiver job associated with the OrderSpeedTest above.

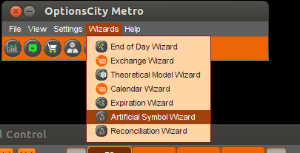
|  |
| --- |
| import com.optionscity.freeway.api.\*;  import com.optionscity.freeway.api.messages.MarketBidAskMessage;  import com.optionscity.freeway.api.messages.OrderMessage;  /\*\*  \* test the speed of submitting orders. a order is submitted. when the booked is received, the order is cancelled a new  \* order submitted  \*/  public class OrderSpeedTest extends AbstractJob {  public void install(IJobSetup setup) {  setup.addVariable("instrument","instrument to order","instruments","");  setup.addVariable("total","total orders","int","10");  setup.addVariable("marketdata","process market data","boolean","true");  }    String instrumentID;  int total;  long minTime = Long.MAX\_VALUE;  long maxTime = 0;  long totalTime =0;  long minNewTime = Long.MAX\_VALUE;  long maxNewTime = 0;  long totalNewTime =0;    long submitTime;  int nMarketData;  int nOrders=0;  long startTime, orderId, timer;  boolean sendNext, marketdata;  @Override  public void begin(IContainer container) {  super.begin(container);  container.subscribeToOrderMessages();  if(marketdata)  container.subscribeToMarketBidAskMessages();  instrumentID = instruments().getInstrumentId(getStringVar("instrument"));  log("instrument is "+instrumentID);  startTime=System.currentTimeMillis();  submitOrder();  }  @Override  public void onMarketBidAsk(MarketBidAskMessage msg) {  Prices p = instruments().getMarketPrices(msg.instrumentId);  nMarketData++;  }  public void onTimer() {  if(sendNext)  submitOrder();  }  public void onOrder(OrderMessage msg) {  if(msg.orderId!=orderId)  return;  if(msg.status== Order.Status.NEW){  long time = System.nanoTime()-submitTime;  totalNewTime+=time;  minNewTime = Math.min(time,minNewTime);  maxNewTime = Math.max(time,maxNewTime);  return;  }  if(msg.status!= Order.Status.BOOKED)  return;  long time = System.nanoTime()-submitTime;  totalTime+=time;  minTime = Math.min(time,minTime);  maxTime = Math.max(time,maxTime);  orders().cancel(orderId);  sendNext=true;  if(++nOrders==total){  long endTime = System.currentTimeMillis();  log("submitted "+nOrders+" orders in "+(endTime-startTime)+" ms");  log("processed "+nMarketData+" market data messages");  log("submit time per order, min "+minNewTime+", max "+maxNewTime+", avg "+(totalNewTime/total)+" nanos");  log("round trip per order, min "+minTime+", max "+maxTime+", avg "+(totalTime/total)+" nanos");  stopJob("complete");  }  if(timer==0)  submitOrder();  }  private void submitOrder() {  sendNext=false;  submitTime = System.nanoTime();  orderId = orders().submit(OrderRequest.buy(instrumentID,1,100.00));  }  } |

# **Artificial Symbols**

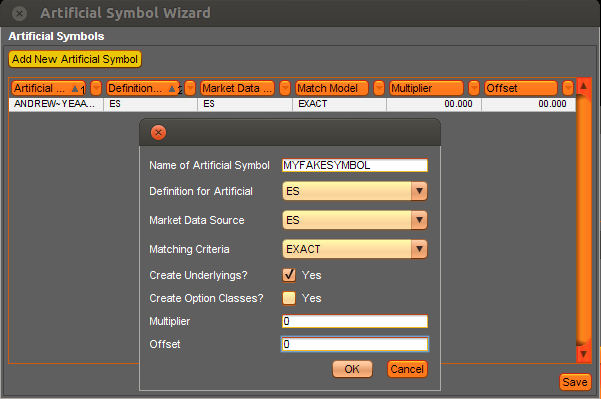
Freeway supports user-defined symbols that can be used as the underlying to quote options. These are called artificial symbols, and, once created, are managed and updated entirely by Freeway.

# **Creating an Artificial Symbol in Metro**

To create an artificial symbol, use the Artificial Symbol Wizard in Metro:



This will bring up the Artificial Symbol Wizard window. Click 'Add New Artificial Symbol' at the top.



In the Add window, set the following parameters:

**Name -** The name of the symbol class. Do not use special characters except "~" (tilde).

**Definition for Artificial -** If your artificial symbol is set up similarly to a real symbol, select it here. This will ensure other parameters, like futures expirations, match the real instrument.

**Market Data Source -** Set to the same instrument as above. If you're setting the artificial markets yourself (and have Multiplier and Offset set to 0), this will not be applied.

**Matching Criteria -** Set to Exact if you're setting markets yourself

**Create Underlyings -** Checking this will create the actual instruments for this symbol. In the example below, it will create MYFAKESYMBOL-20130315-F, MYFAKESYMBOL-20130621-F, etc. These correspond to the ES underlyings.

**Create Option Classes -** If you're using the artificial to drive pricing of real options, you likely want to leave this unchecked. Otherwise it will create options on the fake symbol.

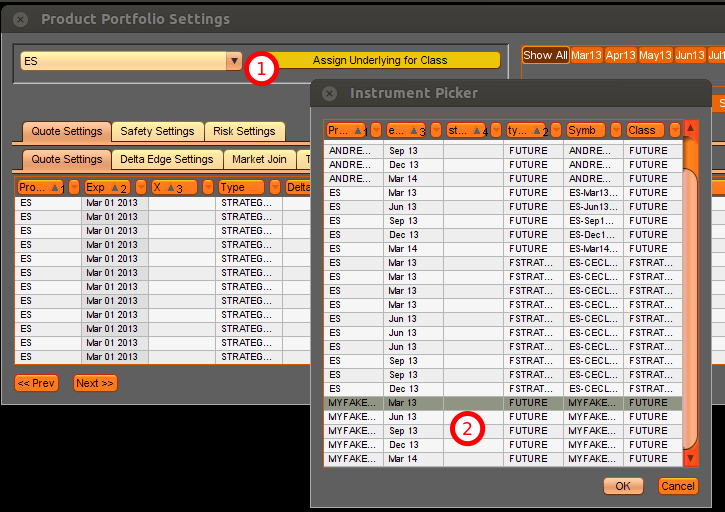
**Multiplier and Offset -** Set these to 0, since you will be controlling the pricing of the artificial symbol yourself.

**Tying Options to your Artificial Symbol in Metro**

Once you've created and saved the artificial symbol, exit out of the Artificial Symbol Wizard. You now have to set your options to use the artificial symbol as the underlying.

Open CityCenter, and click 'Settings'. Select the class of the instrument that you want to drive from the dropdown. In this case, we want to drive ES option off of my synthetic underlying.

Click 'Assign Underlying for Class', and then select the actual instrument to set for the underlying. Here, we're driving all ES options off of MYFAKESYMBOL-20130315-F instrument, which is the Mar 2013 expiration underlying for MYFAKESYMBOL.



# **Setting the Price of the Artificial in Freeway**

Now that the symbol is created and options are using the correct underlying, you can simply set the price in Freeway using instruments().setArtificialMarket():

|  |
| --- |
| //Basic example, just sets the artificial instrument to mimic the top of book for a real instrument  public void onMarketBidAsk(MarketBidAskMessage msg) {  Prices p = instruments().getMarketPrices(realInstrumentId);  instruments().setArtificialMarket(artificialInstrumentID,p.bid\_size,p.bid,p.ask\_size,p.ask);  } |

Every time the setArtificialMarket() function is called, you will see the value of the artificial symbol and the corresponding options change.

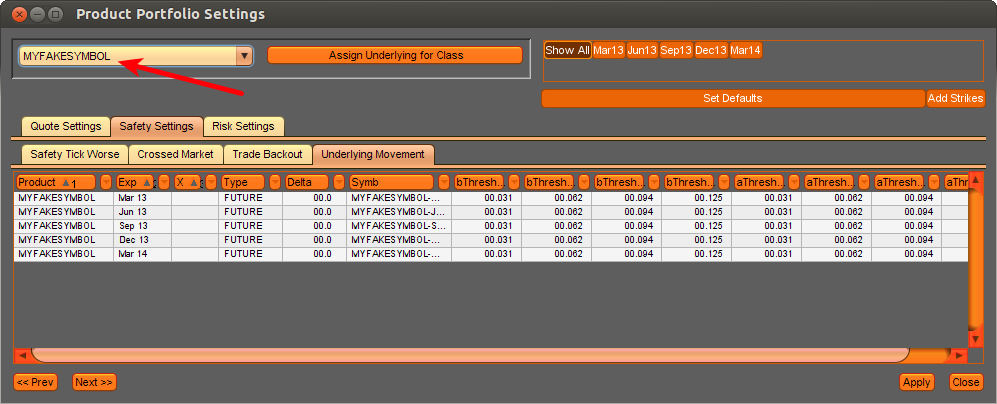
# **Safeties with Artificial Underlyings**

If you're using an artificial underlying, there are a few built-in safeties that will help to ensure proper handling of market irregularities.

## Quote Safeties

The standard quote safeties, such as underlying movement, trade backout and crossed market, apply when using an artificial underlying. With the exception of Underlying Movement, these are set the same way you would set the safeties for any option class.

The underlying movement threshold is set in the 'Settings' window in CityCenter. Select your symbol in the drop down, then choose the correct movement settings in Safety Settings -> Underlying Movement tabs.



## Stopping Your Artificial Quoting Job

If you stop the job that is driving your artificial symbol, all quotes using that artificial symbol will be pulled from the market. If there is any other cleanup you wish to do when stopping a job, you can implement the end() callback.

|  |
| --- |
| public void end(IContainer container) {  super.end(container);  //Do cleanup tasks here  log("Job stopped. Cleanup completed.");  } |

# **Position Tracking on an Event-Driven System**

Position tracking on a high-performance, event-driven system like Freeway can be a little more complex than on system that process a lower frequency of data and trades. The main consideration to be aware of is event synchronization. To illustrate this, here's some sample code where a trader waits for an onOrder status of FILLED, and then checks to see his current position:

|  |
| --- |
| public void onOrder(OrderMessage m){  if (m.status==Status.FILLED){  Order o = orders().getOrder(m.orderId);  long position=positions().getInstrumentRisk(o.instrumentId).dayTradePosition; // This call may not have the latest fill included  log("My position is: " + position );  }  } |

On the surface, this looks to be valid logic that will retrieve your position following the receipt of a FILLED order status. However, in practice, this approach runs into some inherent race conditions created by a multi-threaded system.

Freeway is designed to pass lightweight messages to the event handlers (onWhatever events). Often, these messages hit your code before the underlying system is fully updated with the data from that event. This is as designed, as it is important in a low-latency system to pass the messages through as fast as possible, without blocking on position updates or theoretical calculations. As a result, you may have the FILLED message before the PositionService [ positions() ] is updated with the fill.

## onOrder vs onTrade

The solution to the race condition in the case of fills is to avoid calling the PositionService when high-resolution fill data is desired. Instead, the onTrade method packages the filled quantity inside the message, which allows you to keep your own view of your position based on the fills you've seen. There are two advantages to this approach: first, you know that your position value is always consistent with the fills that your code has seen, and second, the TradeMessage is a light message and will result in higher performance than using the more fully featured PositionService.

The following code tracks position and increments a position variable every time a TradeMessage is seen:

|  |
| --- |
| public class MyTradingJob extends AbstractJob {  long position;    //  // ...begin and install methods and other trading logic here  //    public void onTrade(TradeMessage msg){  if (msg.side==Side.SELL)  position-=msg.quantity;  else  position+=msg.quantity;  }  } |

In general, onOrder should be used whenever you want to listen for status changes to a given order, such as NEW, BOOKED, FILLED, PARTIAL or REJECTED. onTrade should be used when you care about the actual fill details of an order.

# **Signals Reference**

Freeway and Freeway Analytics define some utility signals within the system to notify the user of certain events. Below is a list of system signals that jobs can listen for. For more information on listening to signals, see the [Signals](http://wiki.optionscity.com/display/FWPUB/Signals) document.

**Not all signals apply to all versions of the software – for a list of signals supported by your Freeway version, please refer to the Signal page on the Javadoc. The 'Direct Known Subclasses' will list all signals supported by your version of the API.**

## UI-Driven Signals

* [**Signal**](http://utilities.optionscity.com/freeway/javadoc/4.1.6/com/optionscity/freeway/api/messages/Signal.html) - The base class for all signals. Can be used as a lightweight way to send simple data – contains fields for signal class (called 'clazz'), message and sender.
* [**GridChangeSignal**](http://utilities.optionscity.com/freeway/javadoc/4.1.6/com/optionscity/freeway/api/messages/GridChangeSignal.html) - Signal sent when the grid has changed. Signal contains a collection of Change objects, each of which represent a changed cell in the grid.
* [**UserSignal**](http://utilities.optionscity.com/freeway/javadoc/4.1.6/com/optionscity/freeway/api/messages/UserSignal.html) - Sent when a user clicks a button that they have created in the Dashboard. Extends the base signal to support key/value pairs (see [Buttons](http://wiki.optionscity.com/display/FWPUB/Buttons) doc) in the message. Pairs can be accessed using the getter methods provided.
* [**ConfigurationChangeSignal**](http://utilities.optionscity.com/files/freeway/5.1/freeway_javadoc/com/optionscity/freeway/api/messages/ConfigurationChangeSignal.html) - Sent when a user changes the job configuration while a job is running. Jobs can use this to reload the configuration dynamically to pick up new inputs without a restart.

## Hedging Signals

* [**AutoHedgeRequest**](http://utilities.optionscity.com/freeway/javadoc/4.1.6/com/optionscity/freeway/api/messages/AutoHedgeRequest.html) - A utility signal that jobs can use to send to other jobs (or to the Auto Hedging Mission job on the Algo Store). This signal requires a receiver job to listen for it and hedge.
* [**AutoHedgeStatus**](http://utilities.optionscity.com/freeway/javadoc/4.1.6/com/optionscity/freeway/api/messages/AutoHedgeStatus.html) - A utility signal that jobs can send back to indicate hedging status on an order.

## Miscellaneous Signals

* [**Bulletin**](http://utilities.optionscity.com/files/freeway/5.1/freeway_javadoc/com/optionscity/freeway/api/messages/Bulletin.html)– A bulletin message sent by the exchange. Not all exchanges support this message
* [**ExternalAPI.ExternalAPIMessage**](http://utilities.optionscity.com/files/freeway/5.1/freeway_javadoc/com/optionscity/freeway/api/jobs/ExternalAPI.ExternalAPIMessage.html)- A signal that encapsulates data coming in on the socket for users using the External API. See [The External API (Socket Connections)](http://wiki.optionscity.com/pages/viewpage.action?pageId=5243072)for more details
* [**ExternalAPI.ExternalAPISessionClose**](http://utilities.optionscity.com/files/freeway/5.1/freeway_javadoc/com/optionscity/freeway/api/jobs/ExternalAPI.ExternalAPISessionClose.html)- Signals the close of an External API session
* [**ExternalAPI.ExternalAPISessionOpen**](http://utilities.optionscity.com/files/freeway/5.1/freeway_javadoc/com/optionscity/freeway/api/jobs/ExternalAPI.ExternalAPISessionOpen.html)- Signals the start of an External API session
* [**NoActiveUsers**](http://utilities.optionscity.com/files/freeway/5.1/freeway_javadoc/com/optionscity/freeway/api/messages/NoActiveUsers.html)- Signal emitted when the server no longer has any active users. This signal can be used to stop jobs if all clients get disconnected (either intentionally or unintentionally)
* [**PlaybackSignal**](http://utilities.optionscity.com/files/freeway/5.1/freeway_javadoc/com/optionscity/freeway/api/messages/PlaybackSignal.html)- Signal emitted when playback starts