VELA

METRO FREEWAY TECHNICAL GUIDE

TABLE OF CONTENTS

SESSION I: INTRODUCTION TO FREEWAY

What is Freeway?

Freeway is a platform used to develop and deploy custom trading strategies. The rich Java based API allows access to Metro services including market making, price taking, position, risk management and analytics. The APIs fill the gap and between the out-of-box Metro functionality and the demands of the individual user.

Both Freeway API and Widget API (Frontend GUI API) allow users to build bespoke automated trading applications with custom backend logic and unique and efficient frontend presentation. If required, third party libraries are supported too.

The Freeway algorithmic engine is embedded in the Metro process itself having access to most of the services that Metro has. The Freeway algos (jobs) run in-process in Metro and are single-threaded (maintaining its own queue) thereby taking advantage of the low-latency, high-throughput platform.

Jobs can communicate with each other inter and intra-server and with widgets. Native Metro widgets also can communicate with jobs.

Due to the intuitive nature of the API there is a reduction in time from concept to code to market and within this document we will provide many use cases to demonstrate just some of the possibilities of what can be done with the API(s).

Environment pre-requisites

There are host of Integrated Design Environments for Java development however in this document we will use Intellij. Download is available from JetBrains here: https://www.jetbrains.com/idea/download/

As of June 2020, we build using Java JDK 8 with the download available from Oracle here:
https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html

The Java Runtime Environment (JRE) is also required and can be downloaded from here:
https://www.oracle.com/java/technologies/javase-jre8-downloads.html

The Freeway API and Widget API (frontend GUI) can be found here together with the JavaDoc:
http://utilities.optionscity.com/files/widgets/
http://utilities.optionscity.com/files/freeway/

Basic Project Structure

The aim of this section is to allow the user to create a simple job and upload it to the Metro server. From the outset lets define some basic nomenclature:

- We call any back-end, server-side algos as ‘jobs’ and they are extensions of AbstractJob (Freeway API)
- We call any front-end, client-side GUIs as ‘widgets or apps’ and they are extensions of AbstractWidget (Swing-based Widget API).

Open IntelliJ and create a new project with select Java 8 JDK. A typical job project looks like the following (all details will be explained later!). You may be asked to create a build directory on project set up. Simply create one in the main project path. We will build using a special build.xml (see Appendix 1) which expects a structure like the one that we will describe. For this example the project is called metro-custom-freeway-template.

There are 4 main sub-directories* that need to be created from the main project directory:

- algo
- common
- widgets
- libs
- resources

* NB. algo, common, widget and libs can be created as modules (which have the same directory structure) however in this case we will have a single project module for simplicity.

algo - algo/src contains all instance of back-end, server-side jobs (classes that extend AbstractJob).
common - common/src contains all helper classes required for algo and/or widget. Contains Signals too (which will be explained later).
widgets - widgets/src contains all front-end, client-side GUI classes (extends AbstractWidget). Widgets are optional however if the directory is not present then there will be a build error.

Further breakdown of widget directory includes the file: widgets/META-INF/services/com.optionscity.freeway.onramp.api.IWidget which contains a list of the widgets that you are uploading (in this case just one called BasicWidget):

```
widget.BasicWidget
```

and widgets/META-INF/MANIFEST.MF contains a list of dependencies:

```
Manifest-Version: 1.0
Class-Path: freeway-6.2.0.jar widgets-6.2.0.jar
```

libs - libs is split into two subdirectories:

- **system/${VERSION}** – Contains the system API jars. The build.xml has a stanza to download a specific API version and create a system
- /${VERSION} directory if not present.
- **app** – all third party jars

NB we distinguish between the server-side, backend API eg. freeway-6.2.0.jar and client-side, GUI API eg. widgets-6.2.0.jar

resources - resources/setup contains the file default.setup that allows you to set default job configuration eg.

```
instances:MyJob=1
variable:MyJob.1.autostart=false
```

project.properties - Contains export variables for project and build.xml. As you can see current version is defined below:

```
BUNDLE_NAME=freeway.template
TARGET_OC_VERSION=6.2.0
```

We will address the building, bundling and uploading of your algos later in this chapter.

Quick setup!

In order to create the correct directories and download the API jars simply follow the following steps:

- Create new project in IntelliJ
- Copy build.xml (appendix 1) to project main directory
- Copy project.properties (appendix 1) file in main directory - set to latest OC version Run 'all' target via the ant panel in right hand side of IntelliJ.

Basic job structure

As stated earlier, a job is class that extends AbstractJob. It is an event-driven Java process that runs from the Metro server. Jobs (and widgets) can communicate with each other via signals and widget signals. On a high level its life-cycle is described as follows:

- install – must be implemented. Specifies what happens when you create an instance of the job. Job variables can set here via the Freeway GUI (onRamp).
- begin – must be implemented. Specifies what happens when the jobs starts. subscribe to messages – in the begin method
- implement callbacks on message end

The Freeway server and jobs run in-process with Metro. The many types of messages based on system and market events can be subscribed to.

In general, the system will send ‘tickler’ messages stating that an event has occurred then, via the Freeway services (detailed further in the case-studies), the job can then retrieve the relevant information.

We will introduce some important concepts by looking at a basic case of printing the book depth for a group of instruments. Consider the following example:

```
import com.optionscity.freeway.api.*;
import
com.optionscity.freeway.api.messages.BookDepthMessage
; import java.util.Collection;

public class BookLogger extends
  AbstractJob { private
  Collection<String> instrumentIDs;
    @Override
    public void install(IJobSetup setup) {
        setup.setDefaultDescription("My
        first job.");
        setup.addVariable("Instruments", "Instruments to log", "instruments","");
    }



    public void begin(IContainer
        container){
        super.begin(container);
        instrumentIDs
=instruments().getInstrumentIds(container.getVariable("Instrum
        ents")); container.filterMarketMessages("EW");
        container.subscribeToBookDepthMessages();
    }



    public void onBookDepth(BookDepthMessage message)
    {
        if (instrumentIDs.contains(message.instrumentId))
        {
            log("------------------------------------------------------------ ");
            log("Received BookDepthMessage for " +
            message.instrumentId); Book book =
            instruments().getBook(message.instrumentId);
            for (int i=0 ; i < book.bid.length ; i++)
            {
                Book.BookEntry bidEntry = book.bid[i];
                log("For bid price:" + bidEntry.price + "
quantity: " + bidEntry.quantity + " at level: " + i);
            }
            log("------------------------------------------------------------ ");
            for (int i=0 ; i < book.ask.length ; i++)
            {
                Book.BookEntry askEntry = book.ask[i];
                log("For bid price:" + askEntry.price + "
quantity: " + askEntry.quantity + " at level: " + i);
            }
            log("------------------------------------------------------------ ");
        }
    }
}
```

This job will take a user configured symbol set and output to log the book (10 levels bid and ask) for each instrument when a book update occurs. As stated, all jobs extend AbstractJob and must implement the install and begin methods.

```
@Override
    public void install(IJobSetup setup) {
        setup.setDefaultDescription("My first job.");
        setup.addVariable("Instruments", "Instruments to log", "instruments","");
    }
```

The argument of install is an IJobSetup object we name setup. We can add a simple job description. System variables can also be set and in this case we will allow the user to enter an instrument list that the job can utilise.

- “Instruments” – name of variables
- “Instruments to log” – description of variables
- “instruments” – type of variable (Freeway recognises ‘instruments’ as a type but user can also use ‘String’ or ‘boolean’ etc.) “”
- - default value.

```
public void begin(IContainer container){
        super.begin(container);
        instrumentIDs
=instruments().getInstrumentIds(container.getVariable("Instruments"));
        container.filterMarketMessages("EW");
        container.subscribeToBookDepthMessages();
    }
```

When the job is started the most important action is super.begin(container) which attaches the job to the Freeway server. Omitting this prevents the job from running.

After setting our variable in the install method (actual instrument selection is done via. OnRamp which we will show at the end of this chapter), we now pass this variable to the job itself. In our case the variable name is Instruments and we pass this into a Collection called instrumentIDs.

Freeway can subscribe to many types of system messages eg. market messages:

- BookDepthMessages – market message when book depth changes
- MarketBidAskMessage – market message when bid or ask changes
- MarketLastMessage – market message for last trade
- TheoMessage – market message when theos change

There are many other types messages that we will examine in the individual case studies.

It is easy to subscribe to a message type by container.subscribeToXXXX() and implementing the appropriate callback.

```
public void onBookDepth(BookDepthMessage message)
    {
        if (instrumentIDs.contains(message.instrumentId))
        {
            log("------------------------------------------------------------ ");
            log("Received BookDepthMessage for " + message.instrumentId);
            Book book = instruments().getBook(message.instrumentId);
            for (int i=0 ; i < book.bid.length ; i++)
            {
                Book.BookEntry bidEntry = book.bid[i];
                log("For bid price:" + bidEntry.price + " quantity: " +
bidEntry.quantity + " at level: " + i);
            }
            log("------------------------------------------------------------ ");
            for (int i=0 ; i < book.ask.length ; i++)
            {
                Book.BookEntry askEntry = book.ask[i];
                log("For bid price:" + askEntry.price + " quantity: " +
askEntry.quantity + " at level: " + i);
            }
            log("------------------------------------------------------------ ");
        }
    }
}
```

Consider the intial BookDepthMessage and its callback onBookDepth. It is important to note that BookDepthMessage does not contain any book information simply an instrumentID and sequence number (timestamp and machine from its super class also).

The BookDepthMessage is a ‘tickler message’. It notifies you that an event has occurred, it is now up to user to decide what do with that. In our case, a book depth change has occurred for a certain instrument.

In order to display this information we must create a book first. Freeway’s classes are intuitively named with the book object is called ‘Book’. In order to populate the book with the current levels we make use of one of the Freeway services through the convenience method instruments().

instruments() allows access to the Instrument Service which is used to retrieve instrument details and market data. There are many methods in this service (see JavaDoc >> services >> IInstrumentService) but we will limit ourselves to getBook(String instrumentID). This retrieves the book data for a specified instrument.

Having retrieved the book we now want to display that information. Freeway allows the user to output to system log and this can be done through the log() method which takes a String. There is also the debug() method which outputs to log if debug mode is set eg. in default.setup file or through the onRamp GUI checkbox in job configuration.

default.setup file config for debug logging:

```
instances:MyJob=1
variable:MyJob.1.debugmode=true
```

The Book consists of two arrays (bid and ask) composed of Book.Entry objects and turn these are composed of a price and a quantity. The job simply iterates over each array and outputs the price and quantity.

Going back to the begin method I omitted the container subscription filtering:

```
container.filterMarketMessages("EW");
```

By default you get a message for every symbol. You can filter market messages BEFORE it reaches the callback in the begin method. I have included this in this job but also included filtering in the callback itself for completeness. It is recommended to filter before any callbacks for performance reasons. In initial job testing you can leave it out but it is good to get into the habit of removing any unnecessary processing before any methods are reached.

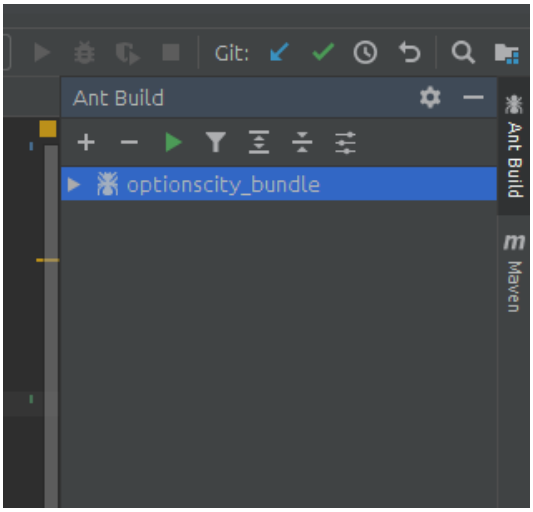
The above will filter on a product set however you can utilise the Instruments variable you have previously defined and just filter on a specific subset of symbols:

```
public void begin(IContainer container){
 super.begin(container);
    instrumentIDs
=instruments().getInstrumentIds(container.getVariable("Instruments"));

 for (String instID : instrumentIDs){
        container.filterMarketMessages(instID);
    }
}
```

Building and bundling

Building the job is easy and can be done using Ant. Either run via the command line ‘ant all’ (must be in build.xml dir) or by adding build.xml to Ant Build on right hand side of IDE and double clicking all:



As stated, the special build script (Appendix 1) will do the following:

- Remove old compiled classes and artifacts
- Fetch API fron Vela
- Compile classes
- Create bundles which contain libs, job and widgets classes.

The bundle file can then be uploaded to Metro server via onRamp.

OnRamp

OnRamp is the GUI for Freeway development. It is used to upload, configure, manage, test and monitor Freeway jobs. The user may have multiple instances of the same job allowing parellelisation of work.

OnRamp has 4 panels:

- Jobs – The main panel shows details of job instances, it description, where it is in the job lifecycle and some performance metrics. (A siml ar panel can be accessed from Metro Front).
- Log – Freeway and job logs output here (A similar panel can be accessed from Metro Front). System status – Details system metrics
- Notifications – Displays system notifications including warnings and errors.

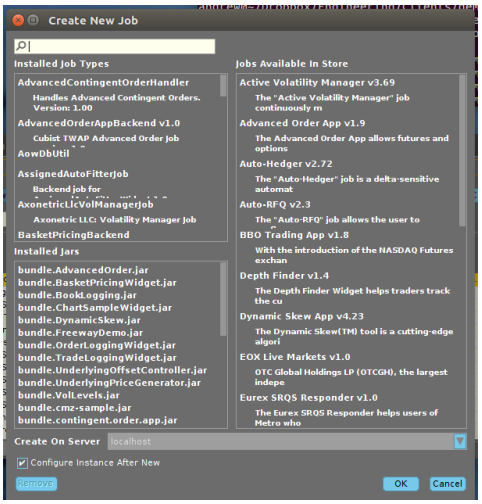
Uploading

You can either upload just the jobs you want to run if the dependencies are already present on the server or else you can simply upload the created bundle which already includes the necessary dependencies as per the build.xml.

In Jobs panel click upload and either select the single job or select the bundle. The advantage of creating bundles is that it is the same build and upload process whether or not you include a widget. If it is just jobs and no widgets then the widget.jar will just be empty.

On upload you will be notified immediately of any compilation errors. If successful you will be notified in the Notification panel.

The next step is to create and instance of the job itself. This is done by clicking new.



Select BookLogger and click OK. The job configuration will appear and with a new entry in the jobs grid for BookLogger. Configuration menu is also accessible by clicking on the job instance and clicking Confgure.

There are two panels in the Configure window:

- **System defined configs** – All instances of AbstractJob have access to these configurations and include inter alia:
    - add to or create a job group
    - limits on the jobs (safeties that will stop the job extending its limits)
    - timer setting (set to 1 sec by default)
    - test mode (prevents the job from submitting orders)
    - debug (whether to output debugs logs)
    - auto-start – job starts up whenever the server starts
- **Job specific configs** – this is where you define the variables that you created in setup.addVariable in the begin method

Back to the case of BookLogger. Clicking on the 'Instruments' config will open up a special instrument selection GUI that has access to any group of instruments on the server. This appears because Instruments is defined as 'instruments' type in the addVariable method. If it was eg. a String then it would simply be a text box.

Double click the empty box and select a Symbol, Type and enter an Expiry (can be numeric or alpha-number – see below). You can be more precise however this is just for demonstration purposes. You can see the number of matches found in the system decrease as you get more specific in your choice.

eg. ES, Future, DEC20 (can be 202012):

Click OK, highlight the job and click Start. If all is well and the you should see book information in the Log panel. Every time there is an update in the book it will be printed off to the log.

You can use the search bar in the Log panel to look for a specific log line and if there are multiple jobs running you can select the particular job th at you are interested in.

Now try to create another instance of the job and but this time with another instrument. This can be done by selecting New however if you want to use the same configuration then click Clone and just change the instrument that you are interested in. This new job will run concurrently with the other jobs(s).

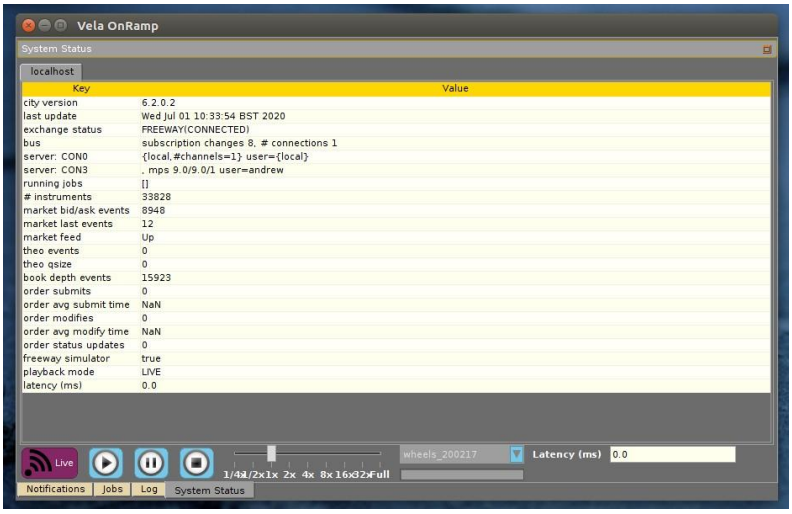
Each job is single threaded with its own message queue which get filled by the server. Each job thread polls each message off at a time and processes it via the callback implementation.

Stopping the job is simple. Simply click Stop (this can be done programmatically by another job via another Freeway service called IJobService that we discuss later.)

Playbacks and back-testing

You have the ability to back-test your strategies. Changing from live market to recorded data is done in the System Status panel.

In order to move from live to recorded data simply click live to pause, select your canned data file then click play. The rate of playback is controlled via the slider on the bottom and ranges from ¼ speed to 32 times and to maximum rate. Progress of the playback is seen in the progress bar under the file selection dropdown menu.



Out-of-the-box we provide a job that captures market data in Vela's own proprietary format (MarketRecorder). Select New and type MarketRecord er. Provide a filename, select whether you want to append the date to the recording and then select the instrument set you are interested in. Of course that capture must contain any symbols of interest if your own job is to see any callbacks on events with those instruments. It is also important to note that the recording job must be stopped before the data is played back.

You also have the ability to upload your own simulated data set (the .csv format is specified further in Appendix 2).

Freeway provides another service for automated testing via the Playback Service interface to allow for repeated runs on multiple datasets to enhance and optimise any trading strategies.

Selecting Live again switches you back to live market data again.

Data Visualisation and Presentation

OnRamp provides a GUI called a 'Dashboard' to allow you to visualise certain data values within a job without the need to create a widget. It must said that the Swing based widgets and Widget API are extremely powerful and customisable and we will address this thoroughly in an upcoming chapter.

We provide an example on how to populate a grid with values from your job. In our example we will create a simple job that displays the Greek values for a particular set of instruments. Consider the following job:

```
import java.util.Collection;
import com.optionscity.freeway.api.AbstractJob; import
com.optionscity.freeway.api.Greeks; import
com.optionscity.freeway.api.IContainer; import com.optionscity.freeway.api.IGrid;
import com.optionscity.freeway.api.IJobSetup;

public class GridExample extends AbstractJob { Collection<String> iids;
    String instruments, gridname; IGrid grid;
    double vegaFactor,deltaFactor,gammaFactor,thetaFactor;

    public void install(IJobSetup setup){
        setup.setDefaultDescription("Adds all instruments to a grid and lists
greeks for those instruments");

 setup.addVariable("instruments","instruments list","instruments","");

        setup.setVariable("timer", "10000");
        setup.addVariable("gridname","Grid
        name","string","InstrumentGreeksGrid");
        setup.addVariable("vegaFactor","vega factor","double",".01");
        setup.addVariable("deltaFactor","delta factor","double","100");
        setup.addVariable("gammaFactor","gamma factor","double","100");
        setup.addVariable("thetaFactor","theta factor (Default is
1/365)","double","0.00273972602");
    }

    public void begin(IContainer container){ super.begin(container);
        grid = container.addGrid(gridname, new String[]
{"Theo","Vega","Gamma","Delta","Theta"});
container.filterMarketMessages(getStringVar("instruments"));
        log("Shows bid greeks in a grid. To use, create a grid called " +
gridname); iids=instruments().getInstrumentIds(instruments);
        onTimer();
    }



    public void onTimer() {
        for (String iid : iids){
            Greeks g = theos().getBidGreeks(iid); grid.set(iid, "Theo", g.theo);
            grid.set(iid, "Vega", g.vega * vegaFactor);
            grid.set(iid, "Gamma", g.gamma * gammaFactor); grid.set(iid, "Delta",
            g.delta   *   deltaFactor);   grid.set(iid,   "Theta",   g.theta   *
            thetaFactor);
        }
    }
}
```

A breakdown of the above:

- Create an IGrid object – the name will be configurable through the user interface and is important when we come to display this in onRamp
- Attach to job container and define the values that you wish to populate – we call our grid InstrumentGreeksGrid. Populate entries within the grid with values (during callback usually).

The values are populated in a map. In our case we set the values (Theo, Vega, Gamma, Delta, Theta) per instrument ie.

```
public void onTimer() {
        for (String iid : iids){
            Greeks g = theos().getBidGreeks(iid);
            grid.set(iid, "Theo", g.theo);
            grid.set(iid, "Vega", g.vega * vegaFactor);
            grid.set(iid, "Gamma", g.gamma * gammaFactor);
            grid.set(iid, "Delta", g.delta * deltaFactor);
            grid.set(iid, "Theta", g.theta * thetaFactor);
        }
    }
```

NB. We are using the timer (set to 1 sec period by default and can be changed in the system configs in the Configure option for the job) and we are using another Freeway service via the convenience method theos() to access theoretical data.

Configure the GridExample job by selecting a set of instruments (eg. I have selected the ES-DEC20-700 call and put options). Leave the grid name by default 'InstrumentGreeksGrid'. Start the job and check all is well with the red/green light indicator.

Now we come to actually displaying these values. Select the large blue icon in top right of onRamp 'Jobs' panel and select 'New Dashboard'. Select Cog icon and select 'Add Grid'

In the 'Configure Dashboard Grid' set grid name to the one in the Freeway job ie. InstrumentGreeksGrid. Add each column as per below and select OK. The values will now be populated.

With values populated:

Default configs for deployment

Job settings can be set before deployment in the resources/setup/default.setup file. We addressed this earlier when setting the debug logging to true. A typical example of pre-deployment settings is:

```
instances:MyJob=2
variable:MyJob.1.autostart=true
variable:MyJob.1.testmode=false
variable:MyJob.1.debugmode=true
variable:MyJob.2.autostart=true
variable:MyJob.2.testmode=false
variable:MyJob.2.debugmode=true
```

Two instances of the job are created both auto-starting with debug logging enabled.

Appendices

Appendix 1

build.xml

build.xml for compiling and building project bundle ready to be uploaded to Metro via onRamp.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="optionscity_bundle" default="all">
    <property environment="env"/>
    <property file="project.properties" />

    <target name="all" depends="clean, init, initWidgets, retrieveApiJars,
unzipLibs, compile, packageWidget, packageCommon, packageBundle"/>
    <target name="clean">
        <delete dir="${basedir}/build/widget"/>
 <delete dir="${basedir}/build/common"/>
        <delete dir="${basedir}/build/algo"/>
        <delete dir="${basedir}/artifacts"/>
    </target>

    <target name="init">
     <mkdir dir="${basedir}/build/widget"/>
        <mkdir dir="${basedir}/build/common"/>
        <mkdir dir="${basedir}/build/algo"/>
        <mkdir dir="${basedir}/artifacts"/>
     <mkdir dir="${basedir}/artifacts/widget"/>
        <mkdir dir="${basedir}/artifacts/common"/>
        <mkdir dir="${basedir}/algo"/>
        <mkdir dir="${basedir}/algo/src"/>
        <mkdir dir="${basedir}/common"/>
        <mkdir dir="${basedir}/common/src"/>
        <mkdir dir="${basedir}/libs"/>
        <mkdir dir="${basedir}/libs/system"/>
        <mkdir dir="${basedir}/libs/app"/>
        <mkdir dir="${basedir}/resources"/>
        <mkdir dir="${basedir}/resources/dashboards"/>
        <mkdir dir="${basedir}/resources/setup"/>
        <touch file="${basedir}/resources/setup/default.setup"/>
    </target>

    <!--
    initWidgets creates necessary files and dirs to
    create widgets User should to create package
    'widget' in widget/src
    -->

    <target name="initWidgets">
        <mkdir dir="${basedir}/widgets"/>
        <mkdir dir="${basedir}/widgets/META-INF"/>
        <touch file="${basedir}/widgets/META-INF/MANIFEST.MF"/>
```

```
        <mkdir dir="${basedir}/widgets/META-INF/services"/>
        <touch
file="${basedir}/widgets/META-
INF/services/com.optionscity.freeway.onramp.api.IWidget"
/>
```

```
        <mkdir dir="${basedir}/widgets/src"/>
    </target>

    <!--
    initCurves creates necessary files and dirs to create custom
    vol curves User needs to create packages 'curve' and 'slide'
    in common/src
    -->

    <target name="initCurves">
        <mkdir dir="${basedir}/common"/>
        <mkdir dir="${basedir}/common/META-INF"/>
        <touch file="${basedir}/common/META-INF/MANIFEST.MF"/>
        <mkdir dir="${basedir}/common/META-INF/services"/>
        <touch
file="${basedir}/common/META-
INF/services/com.optionscity.freeway.api.VolatilityCurve$ Custom"/>
        <touch
file="${basedir}/common/META-
INF/services/com.optionscity.freeway.api.VolatilityCurve$
VolatilitySlide$Custom"/>
        <mkdir dir="${basedir}/common/src"/>
    </target>

    <target name="retrieveApiJars">
        <mkdir dir="${basedir}/libs/system/${TARGET_OC_VERSION}"/>
        <get
src="http://utilities.optionscity.com/files/freeway/${TARGET_OC_VERSION}/fr
eeway-${TAR GET_OC_VERSION}.jar"

dest="${basedir}/libs/system/${TARGET_OC_VERSION}/freeway-
${TARGET_OC_VERSION}.jar" usetimestamp="true"/>
 <get
src="http://utilities.optionscity.com/files/widgets/${TARGET_OC_VERSION}/wi
dgets-${TAR GET_OC_VERSION}.jar"

dest="${basedir}/libs/system/${TARGET_OC_VERSION}/widgets-
${TARGET_OC_VERSION}.jar" usetimestamp="true"/>
    </target>

    <target name="unzipLibs">
        <unzip dest="${basedir}/build/common">
            <fileset dir="${basedir}/libs/app">
                <include name="*.jar"/>
            </fileset>
        </unzip>
    </target>

    <target name="compile" depends="init">
        <javac nowarn="on" fork="false" srcdir="${basedir}/common"
destdir="${basedir}/build/common" debug="true" optimize="on"
includeantruntime="false"
>
```

```
<compilerarg value="-Xlint:all"/>
<classpath>
    <fileset dir="${basedir}/libs/system/${TARGET_OC_VERSION}">
        <include name="freeway-${TARGET_OC_VERSION}.jar"/>
        <include name="widgets-${TARGET_OC_VERSION}.jar"/>
    </fileset>

    <fileset dir="${basedir}/libs/app">
        <include name="*.jar"/>
```

```
                </fileset>
            </classpath>
        </javac>

 <javac nowarn="on" fork="true" srcdir="${basedir}/widgets"
destdir="${basedir}/build/widget" memoryInitialSize="512m"
memorymaximumsize="768m" debug="true" optimize="on"
includeantruntime="false" >
             <compilerarg value="-Xlint:all"/>
             <classpath>
                 <fileset dir="${basedir}/libs/system/${TARGET_OC_VERSION}">
                     <include name="freeway-${TARGET_OC_VERSION}.jar"/>
                     <include name="widgets-${TARGET_OC_VERSION}.jar"/>
                 </fileset>

                 <fileset dir="${basedir}/libs/app">
                     <include name="*.jar"/>
                 </fileset>
                 <pathelement location="${basedir}/build/common"/>
             </classpath>
        </javac>

        <javac nowarn="on" fork="true" srcdir="${basedir}/algo"
destdir="${basedir}/build/algo" memoryInitialSize="512m"
memorymaximumsize="768m" debug="true" optimize="on"
includeantruntime="false" >
            <compilerarg value="-Xlint:all"/>
            <classpath>
                <fileset dir="${basedir}/libs/system/${TARGET_OC_VERSION}">
                    <include name="freeway-${TARGET_OC_VERSION}.jar"/>
                    <include name="widgets-${TARGET_OC_VERSION}.jar"/>
                </fileset>

                <fileset dir="${basedir}/libs/app">
                    <include name="*.jar"/>
                </fileset>

                <pathelement location="${basedir}/build/common"/>
            </classpath>
        </javac>
    </target>

          <target
    name="packageCommo
            n">
                                    <jar
     destfile="${basedir}/artifacts/common/common.${BUNDLE_NAME}.jar">
       <fileset dir="${basedir}/build/common" includes="**/*.class"/>
       <fileset dir="${basedir}/build/common" includes="**/*.properties"/>
    </j
    ar>
   <
   /
   t
```

```
a
r
g
e
t
>

 <target name="packageCurves">
     <jar destfile="${basedir}/artifacts/common/common.curves.jar">
         <fileset dir="${basedir}/build/common/curve" includes="**/*.class"/>
         <fileset dir="${basedir}/build/common/curve"
         includes="**/*.properties"/>
         <fileset dir="${basedir}/build/common/slide" includes="**/*.class"/>
         <fileset dir="${basedir}/build/common/slide"
         includes="**/*.properties"/>
         <fileset dir="${basedir}/common/META-INF" includes="**/*"/>
     </jar>
 </target>

       <target
 name="packageWidge
        t">
                                     <jar
   destfile="${basedir}/artifacts/widget/widgets.${BUNDLE_NAME}.jar">
```

```
            <fileset dir="${basedir}/build/widget" includes="**/*.class"/>
            <fileset dir="${basedir}/widgets/" includes="META-INF/**"/>
            <fileset dir="${basedir}/libs/app" includes="*.jar"/>
        </jar>
    </target>

    <target name="packageBundle">
        <jar destfile="${basedir}/artifacts/bundle.${BUNDLE_NAME}.jar">
            <fileset
dir="${basedir}/artifacts/common"
includes="common.${BUNDLE_NAME}.jar"/>
            <fileset
dir="${basedir}/artifacts/widget"
includes="widgets.${BUNDLE_NAME}.jar"/>
            <fileset dir="${basedir}/build/algo" includes="*.class"/>
            <fileset dir="${basedir}/resources/dashboards"
            includes="*.dashboard"/>
            <fileset dir="${basedir}/resources/setup" includes="default.setup"/>
            <fileset dir="${basedir}/libs/app" includes="*.jar"/>
        </jar>
    </target>

    <target name="propsTest">
        <echo>Bundle name is ${BUNDLE_NAME}</echo>
```

```
    </target>

</project>
```

project.properties file

```
BUNDLE_NAME=MyBundle
TARGET_OC_VERSION=6.3.0
```

Appendix 2 - Playback csv

You have the ability to import market data files. Below are details of the format of such a file:

1. Name the file xxxxx.csv where xxxxx is a description name.
2. Move the file to the "home/recordings" directory of the installation.
3. The file will automatically be imported, and will appear as xxxxx in the list of available playback files when using the Freeway exchange. NB. th e original source file will be deleted, so make a copy when testing.

Below is an example of the file:

```
# clear the IBM book
B,0,IBM-E,0,0
# wait
5
secon
ds
L,+50
00,IB
M-
E,10,
101
L,+0,I
BM-
E,9,1
00.99
L,+0,IBM-E,8,100.88
```

```
L,+0,IBM-E,7,100.77
L,+0,IBM-E,6,100.66
L,+0,IBM-E,5,100.55
L,+0,IBM-E,4,100.44
L,+0,IBM-E,3,100.33
L,+0,IBM-E,2,100.22
L,+0,IBM-E,1,100.11
# wait 5 seconds between each book
top of book update T,+5000,IBM-
E,10,100.00,15,100.20
T,+5000,IBM-
E,11,100.10,16,100
.30 T,+5000,IBM-
E,12,100.20,17,100
.40
# wait 5 seconds before the full book update
```

```
B,+5000,IBM-
E,3,1,100.10,2,100.20,3,100.30,4,5,100.50,6,100.60,7,100.70,8,100.80
```

The file format is a comma delimited text file. Lines should be terminated with a newline. Commands and arguments are case sensitive. The basic format is:

command, timestamp, symbol, arg0, arg1, ... argn

The command is one of 'T' (top of book), 'L' (market last), 'B' (book depth), 'Q' (request for quote). You can place comments in the import file by starting the line with #

Timestamping details:

The timestamp can be represented in 4 ways:

1. Use 0 to use the current time when importing
2. Use +nnn to add nnn milliseconds to the last timestamp, with +0 meaning to use the same timestamp as the last
3. Use nnnn for a UTC timestamp represented in milliseconds
4. Use MMddyyyy HHmmss.SSS where MM is the month, dd is the day of the month, yyyy is the year, HH is the hour (0-23), mm is the minute (0-59), ss is the seconds (0-59), and SSS is the milliseconds (0-999(

The timestamp format can be different on a per line basis, which is useful when creating an import file by hand and must be greater than or equal to the previous timestamp or an import exception will occur.

The symbol is the 'idSymbol', which is the 'instrument identifier' in Freeway. Top Of Book entry:

```
T,timestamp,symbol,bidSize,bidPrice,askSize,askPrice
```

Market Last entry:

```
L,timestamp,symbol,quantity,price
```

Book Depth entry:

```
B,timestamp,symbol,bidLevels,[bidSize1,bidPrice1...
bidSizeN,bidPriceN],askLevels,[askSize1,as kPrice1... askSizeN,askPriceN]
```

Clear the book for an instrument:

```
B,timestamp,symbol,0,0
```

Request For Quote entry: Q,timestamp,symbol,exchangeIdentifier,quantity

SESSION II - DATA AND SERVICES

JavaDoc and Freeway API organisation

The Freeway and Widget API JavaDocs can be found here:

http://utilities.optionscity.com/files/freeway/6.3.0/freeway_javadoc/
http://utilities.optionscity.com/files/widgets/6.3.0/widget_javadoc/

In this section we focus on the Freeway API in the context of Freeway jobs (defined as classes which are extensions of AbstractJob). The Freeway API is divided into 6 main packages.

- **freeway.api** - general objects within the API
- **helpers** - helping classes for pricing and formatting usually abstracted from the user.
- **jobs** - shows different types of Freeway jobs (see Session 4)
- **messages** - shows all different message types
- **playback** - playback and back-testing objects listed here
- **services** - probably the most important package - contains methods to interact with the system, affect the state of the system and get information about the system from a Freeway job

Messages and subscription

In general, Freeway is an event driven API. The users subscribes to a particular event type and then implements the corresponding event handler.

There are many types of message generated by intrinsic or extrinsic events (see package com.optionscity.freeway.api.messages in JavaDoc). Metro can take data from many exchanges however the internal messages are all normalised into various types. These include:

Market data events eg. :

- AuctionMessage - an auction (request for price) message
- BookDepthMessage - market message when book depth changes
- RequestForQuoteMessage - request for quote

System events eg.:

- ConfigurationChangeSignal - a signal sent when the job configuration is changed while running GridChangeSignal - signal sent when a grid change is requested
- JobStarted - a signal sent when a job instance is started

Safety events eg:

- TradeRisk Signal - sent by the system when a trade risk (too many trades) is encountered.
- MarketRisk Signal - sent by the system when a market risk (e.g. exchange QPS, protocol error), etc.) is encountered.

In general, the job container sends a message to a job upon an event taking place.

Just to clarify nomenclature, we ask the question the difference between messages and signals? We would conventionally relate a message to a market data event and signals to an internal Metro event. The user can extend the Signal class to make instances of custom signals (addressed in Session III). Both are

in the same package with respect to the API (api.messages).

Subscribing to messages

The usual job design will involve the following steps:

1. Subscribe to the message of interest
2. Invoke the callback of that message type
3. Utilise the Freeway Container service to access or change Metro data/analytics.

All message subscriptions are done via methods in the container object called in the begin() method. A typical example is below:

```
public void begin(IContainer container){
    super.begin(container);
    container.subscribeToOrderMessages(); // Subscribe to OrderMessages
}
```

There are a host subscribe methods in the container for each message type:

Method	Detail
subscribeToAuctionMessages()	Tells the container to send any auction (request for price) notifications to this job.
subscribeToBookDepthMessages()	Tells the container to send any book depth update notifications to this job.
subscribeToInstrumentMessages()	Tells the container to send any instrument notifications to this job.
subscribeToMarketBidAskMessages()	Tells the container to send any BidAsk notifications to this job.
subscribeToMarketStatsMessages()	Tells the container to send any market statistics notifications to this job.
subscribeToOrderMessages()	Tells the container to send any order notifications to this job
subscribeToQuoteMessages()	Tells the container to send any quote messages to this job
subscribeToRequestForQuoteMessages()	Tells the container to send any RFQ notifications to this job
subscribeToSignals()	Tells the container to send any signals to this job
subscribeToTheoMessages()	Tells the container to send any theoretical data notifications to this job
subscribeToTradeMessages()	Tells the container to send any trade notifications to this job

If the user subscribes to an event then they must then implement the appropriate callback for that event in the main Job class.

Method	Detail
onAuction(AuctionMessage msg)	Called upon an auction event in the server.
onBookDepth(BookDepthMessage msg)	Called upon change to market book depth.
onInstrument(InstrumentMessage message)	Called when the server receives a new instrument or an instrument update
onMarketBidAsk(MarketBidAskMessage msg)	Called when a BidAsk
onMarketLast(MarketLastMessage msg)	Called upon change in market data
onMarketStats(MarketStatsMessage msg)	Called upon change in market statistics data.
onOrder(OrderMessage msg)	Called when upon an order event in the server.
onQuote(QuoteMessage message)	Called when quote data is generated by the server.
onRequestForQuote(RequestForQuoteMessage msg)	Called upon RequestForQuote event in the server.
onSignal(Signal signal)	Called upon any generated Signal event for this job.
onTheo(TheoMessage msg)	Called upon change theoretical data.
onTrade(TradeMessage msg)	Called when a trade is executed on the server.

In the context of the above example an implementation would look like this:

```
public void begin(IContainer container){
    super.begin(container);
    container.subscribeToBookDepthMessages();
}

public void onBookDepth(BookDepthMessage msg){
    // Do something here
}
```

Messages are light-weight objects which usually contain only information to tell that an event has happened eg consider the MarketBidAskMessa ge. This message is sent by the container to the job when bid or ask changes. This message contains only the following fields:

instrumentId, seqnum, machine, timestamp, priceChange

ie. does not contain the actual market bid or ask for that instrument. In order to access the actual market prices they must be accessed through one of the relevant Freeway services (described in the next chapter). These services are usually accessed through convenience methods within the AbstractJob eg. instruments() accesses the Instrument service to retrieve the market prices.

```
public void begin(IContainer container){
    super.begin(container);
    container.subscribeToBookDepthMessages();
}

public void onBookDepth(BookDepthMessage msg){

Book myBook = instruments().getBook(message.instrumentId);

// instruments() is the convenience method that references the job container's
IInstrumentService

    // Do something with myBook eg.

    for (int i=0 ; i < book.bid.length ; i++)
            {
                Book.BookEntry bidEntry = book.bid[i];
                log("Bid price:" + bidEntry.price);
            }

            for (int i=0 ; i < book.ask.length ; i++)
            {
                Book.BookEntry askEntry = book.ask[i];
                log("Ask price:" + askEntry.price);
            }
}
```

Filtering

Filtering is an important concept in each job that allows the user to improve and optimise job performance. For best performance, it is important to limit the data that the system processes events for. The container has the ability to filter messages and instruments that it is only interested in BEFORE it reaches a callback within the job.

Filtering instruments - Existing instruments

For example, if you're only trading ES futures, then you would likely not want the overhead of also processing options events. The filterMarketMes sages() method allows your job to designate which instruments it wants to handle events for. The snippet below demonstrates filtering based on a configuration variable ("instruments") that the user sets.

```
public void install(IJobSetup setup){
          setup.addVariable("instrument", "instruments to enable (others will be
filtered out)", "instruments", "");
}

public void begin(IContainer container){
 ...  container.filterMarketMessages(getStringVar("instruments"));
 ...
}
```

Put an example in of what if filtered in. Provide specific instruments and show union of filters applies when multiple filters are used. The filtering affects market, book, theo and quote messages. This method may be called multiple times and the filtering will be the union of all requests.

If this message is called, ALL pending market messages will be removed from the job's message queue. Please note that there are performance implications when using this method other than from begin().

Filtering instruments - New instruments

NB. The filterMarketMessages() method only filters market data for instruments that already exist in the system. For performance reasons this method removes non-matching instruments once at runtime and does not check instruments again as market data arrives. Because of this, NEW instruments (either dynamically added and received via the onInstrument() callback or created by the exchange and received via the onRequestF orQuote() callback) will not be filtered, and your job will receive all market data for these instruments.

Therefore, if you server is receiving data for ES and GC, and you filter out all GC contracts and are listening for RFQs in ES, any new GC RFQ that didn't exist at the first filtering will pass through your filters. To filter market data for new instruments, there are two additional calls that you can use: container.filterRequestForQuoteAsMarketMessage() and container.filterInstrumentsAsMarketMessage().

Adding these (along with the filter expression) in your begin() method will force any new RFQ instruments

or dynamic instruments to pass the market data filtering expression. Here's a sample that filters all existing market data, new RFQs and new instruments:

```
public void begin(IContainer container){
    super.begin(container);

    // Filter market data for instruments we know about
 container.filterMarketMessages(getStringVar("instruments"));

    // Filter market data for new RFQs on newly created instruments
 container.filterRequestForQuoteAsMarketMessage();

    // Filter market data for dynamically added symbols
    container.filterInstrumentsAsMarketMessage();
}
```

Filtering messages

You define the instruments of interest in the above section. Using those filters you can filter RFQs and new InstrumentMessages too via:

```
container.filterRequestForQuoteAsMarketMessage(true);
container.filterInstrumentsAsMarketMessage(true);
```

You can also filter on messages relevant to your job:

```
container.filterOnlyMyOrders(true) // filter orders messages received by the job, to
only orders submitted by the job
container.filterOnlyMyTrades(true) // filter trade messages received by the job, to
only orders submitted by the job
```

Resetting filters

Finally, you can reset all filters if you wish by using the following method:

```
resetMarketMessageFilters();
```

which removes all active market message filters. Filtering affects market, book, theo and quote

messages. A simple filtering demonstation job can be seen in Appendix 1.

Container Services

The power of Freeway comes from the Services that are provided to the user. These Services allow access and interaction with the Metro trading platform directly and a feature rich array of functions.

Services can be used to check state, extrapolate data, submit orders, and even adjust theoretical values or volatility curves on the fly. While some services, such as the InstrumentService, provide both specific and general details or helper methods, many of the services are closely related to a particular type of event. For example, if a job is notified of a theoretical event and receives a TheoMessage, you can then utilize the TheoService to look up additional details if desired. Many services also have one or more particular models that they provide for the requested data, such as a Price model, Greek model, etc.

In general, the event-driven process takes the form:

- Subscribe to relevant messages
- Implement callbacks
- Interface with Freeway service based on event type

Services are usually accessed through convenience methods within the AbstractJob. Details of the Services can be found by clicking on the api.s ervices package in the JavaDoc. We will look at a few cases and provide an example of a particular job.

Instrument Service

We start with this service as it is used to retrieve details and market data for a particular instrument. The results returned with this are usually used in conjunction with other Services.

It is used to retrieve, amongst other things, the market prices (top of book or full book), theoretical values, instrument details which include instrument IDs, active symbols, finding and adding strategies, expiration details and getting and setting settlement details. The objects returned in most cases are Freeway objects and can be manipulated in turn with other services.

Calling the InstrumentService is done via the convenience method instruments() (which returns this.container.getInstrumentService()) in AbstractJ ob.

A useful starter job is this:

- We use the onRamp confiuration to select a subset of instruments
- The instrument service is the used to get the collection of internal instrumentIDs
- We use each instrumentID again to retrieve the InstrumentDetails object for each instrument and log out certain fields.

```
public class ListInstrumentDetailsJob extends AbstractJob {

    private Collection<String> instrumentIDs;
    private String comma = ", ";

    @Override
    public void install(IJobSetup setup) {
    setup.addVariable("instruments", "Instruments to log", "instruments","");
    }

    public void begin(IContainer container){
    super.begin(container);
    instrumentIDs =
instruments().getInstrumentIds(container.getVariable("instruments"));
    queryInstrumentDetails(instrumentIDs);

    }

    public void queryInstrumentDetails(Collection<String> myArray){
    log(">> Checking details...");
        for (String st : myArray) {
                InstrumentDetails id = instruments().getInstrumentDetails(st);
                String toLog = id.instrumentId + comma
                        + id.displaySymbol + comma
                        + id.instrumentMonth + comma
                        + id.displayExpiration + comma
                        + id.maturityDate + comma
                        + id.symbol;
                log(">> " + toLog);
        }
    }
```

Notice in this case there is no callback implemented and the job is run from begin(). In the context of callbacks we can subscribe to MarketBidAsk messages and get prices for certain instruments:

```
public void onMarketBidAsk(MarketBidAskMessage m) {

    Prices
prices=instruments().getMarketPrices(m.instrumentId);

    double ask=prices.ask;
    double bid=prices.bid;
    log("Bid: " + bid);    log("Ask: " + ask);
}
```

With the Price object being an immutable struct that has the following fields:

public final double bid;
public final double ask;
public final int bid_size;
public final int ask_size;
public final double last;
public final int last_size;
public final double theo_bid;
public final double theo_ask;
public final double underlying_bid;
public final double underlying_ask;
public final double underlying_bid_size;
public final double underlying_ask_size;

The class InstrumentDetails returned from getInstrumentDetails() provides many useful objects including:

- InstrumentDetails.Type eg. (CALL, PUT, FUTURE)
- InstrumentDetails.LegDetails[] - You can return the instrument legs details legs when working with strategies.
- Instrument.Month - used in Freeway when handling the Option Month.
- Tick data (tickValue) is returned too via this object which is of use when converting to option notional value (useful for PnL algos).
- The underlying ID is also available in this object (useful for hedging scenarios).

We have used this service when querying the Book levels in previous examples in Session 1.

Usually the results of the Instrument Service are used in conjunction with other services. We will look at the example of this in respect to the Order Service.

Order Service

An extremely powerful application of the API is to submit, handle, monitor, modify and cancel orders through Freeway. This can be done based on triggers or events outside the job or within the market and it is entirely customisable. Orders can even be submitted to Freeway from Metro and the algo can perform various steps of logic before deciding what to do next.

As before, the OrderService is accessed through the convenience method orders().

NB. By selecting ‘test-mode’ in the onRamp config orders are not actually submitted to the exchange, and are local to the job.

Like before, you subscribe to Order messages in the system via. container.subscribeToOrderMessages() and implement its callback onOrder(Ord erMessage):

```
public void begin(IContainer container){
    super.begin(container);
    container.subscribeToOrderMessages();
}

public void onOrder(Order msg){
    Long orderID = msg.orderId;
    //Retrieve the order using the orderId contained within the OrderMessage
    Order myOrder = orders().getOrder(orderID);
}
```

The Order Message contains among other things the orderID which can then be used to retrieve the Freeway Order object. This immutable object contains many fields including booked price and quantity, filled quantity, order status, order type, edge and hedge details.

Through the orders() convenience method you can modify and cancel existing orders:

- **Modify orders** – orders().modify(long orderId, int quantity, double price)
- **Cancel orders** – orders.cancel(long orderID)

In order to submit an order you need to create an OrderRequest object:

OrderRequest(Order.Type type, Order.Side side, java.lang.String instrumentId, double price, int quantity) which can be submitted via. orders().submit(OrderRequest request) eg.

```
private void submitOrder(Order.Type type, Order.Side side, String instrumentId, double
price, int quantity){
     OrderRequest req = new OrderRequest(type, side, instrumentId, price, quantity);
     Orders().submit(req);
}
```

The order request contains many fields beyond its contructor including attaching a user defined label to the order for reference elsewhere, setting hedge mode, setting edge protection as well as price rounding helpers. Please see com.optionscity.freeway.api.OrderRequest in JavaDoc for more information.

As alluded to earlier, orders can be submitted from Metro to Freeway and manipulated therein before being sent to market. When you route the order to Freeway via the Metro Order ticket an OrderRequestSignal is submitted which contains the OrderRequest object. The job therefore needs to intercept these signals by:

- Subscribing to general signals in the begin() method – container.subscribeToSignals() Implement the callback – private void onSignal(OrderRequestSignal sig)

Position Service

This Service is used to retrieve the latest risk and position information .Custom risk management algos can be made utilising the Freeway Positio n Service. This can be used for many scenarios including PnL monitoring and order filtering.

The convenience method to access the PositionService in AbstractJob is: **positions()**

The service is primarily gets position risk for specific instruments, portfolio or trading account. What is returned is the Freeway object PositionRisk. This immutable object which includes the following aggregate fields:

- **committedPosition**
- **committedProfitAndLoss** - defined as P&L for the open position: difference between the current theoretical price (for options) current price (for futures) and last day's settlement price
- **dayTradePosition**

- **dayTradeProfitAndLoss** - defined as P&L for today's trades: difference between the current theoretical price (for options) or current price (for futures) and the trade price.

An example of displaying total position and delta is:

```
Collection<Position> myPositions = positions().getPositions();

for (Position position : myPositions) {

 instruments().getInstrumentDetails(position.instrumentId).underlyingId;
   Prices prices = instruments().getAllPrices(underlyingSymbol);
            underlyingBidPrice = prices.bid;
   double delta = theos().getGreeks(position.instrumentId).delta;
   double totalPosition = position.committed + position.dayTrade;

debug("Days to expiry for (Position) " + position.instrumentId + " is : " +
instruments().getDaysToExpiration(position.instrumentId));
        debug("Delta for " + position.instrumentId + " is: " + +delta);
        debug("Total Contract for " + position.instrumentId + " is: " +
totalPosition);
}
```

A usage for the above would be in creating a pre-trade risk algo that would prvent orders being sent if positional risk went beyond a certain defined level or a cancellation of orders if risk limit was overstepped.

This would be achieved by routing the orders to Freeway, catching the OrderRequest, sending to a risk algo which would cross reference the position and then submit order if all conditions were met.

Appendices

Appendix 1 - Filtering Job

The example demonstrates the use of container filtering and demonstates that the over all job filter is the union of filters applied at the instrument level.

```
import com.optionscity.freeway.api.AbstractJob;
import com.optionscity.freeway.api.IContainer;
import com.optionscity.freeway.api.IJobSetup;
import com.optionscity.freeway.api.messages.MarketBidAskMessage;

import java.util.Collection;

public class FilteringExample extends AbstractJob {

    private Collection<String> instrumentIDs;

    @Override
    public void install(IJobSetup setup) {
        setup.setDefaultDescription("Sample job to demonstrate services.");
        setup.addVariable("Instruments", "Instruments to log", "instruments","");
    }

    public void begin(IContainer container){
        super.begin(container);
        container.subscribeToMarketBidAskMessages();
        instrumentIDs =
instruments().getInstrumentIds(container.getVariable("Instruments"));

        //Filtering symbol set selected by user via onRamp
        //NB The filtering is a white list and will be the union of filters applied to
the job.
        for (String st: instrumentIDs) {
            log(">> Filtering: " + st);
            container.filterMarketMessages(st);
        }
    }

    public void onMarketBidAsk(MarketBidAskMessage msg){
       log("MarketBidAskMessage received for :" + msg.instrumentId);
    }

}
```

Instruments can be selected via onRamp. Below is an example of selecting ODAX call and put options with a DEC20 expiry:

NB. A better way is to do the following and pass in the job setup variable:

```
import com.optionscity.freeway.api.AbstractJob;
import com.optionscity.freeway.api.IContainer;
import com.optionscity.freeway.api.IJobSetup;
import com.optionscity.freeway.api.messages.BookDepthMessage;


public class BetterFilteringExample extends AbstractJob {

    @Override
    public void install(IJobSetup setup) {
        setup.addVariable("instruments", "instruments to filter", "instruments", "");
    }

    public void begin(IContainer container){
        super.begin(container);
        container.filterMarketMessages(getStringVar("instruments"));
    }

    @Override
    public void onBookDepth(BookDepthMessage msg) {
        log("Px received for " + msg.instrumentId);
    }

}
```

Appendix 2 - Order Service Job

The following is a simple job that makes ue of the Order Service. The job will either display all open orders in the system or if 'cancel' is selected in the onRamp config will cancel all open orders too. The orders() convenience method returns a snapshot of orders in the system and this object can be iterated over to perform manipulations.

```
import com.optionscity.freeway.api.AbstractJob;
import com.optionscity.freeway.api.IContainer;
import com.optionscity.freeway.api.IJobSetup;
import com.optionscity.freeway.api.Order;

import java.util.SortedSet;

public class CancelAllOrders extends AbstractJob {

    boolean cancel = false;

    @Override
    public void install(IJobSetup setup) {
        setup.setDefaultDescription("Job to cancel all open orders");
        setup.addVariable("cancel", "set to true to cancel all open orders",
"boolean", "false");
    }

    public void begin(IContainer container) {
        super.begin(container);
```

```
        container.subscribeToOrder
        Messages();
        container.subscribeToSigna
        ls();

        cancel =
        getBooleanVar("cancel"
        ); if (cancel) {
            cancelOpenOrders();
        } else {
            displayOpenOrders();
        }
        //container.stopJob("Cancel job complete.");


    }

    public void

        cancelOpenOrders() {

        log("Cancelling open

        orders...");
        log("Orders in the system: " + orders().snapshot().size());

        //Cancel booked or partial orders.

        if (orders().snapshot().size() > 0) {

            SortedSet<Order> orderSnap =
            orders().snapshot(); for (Order o :
            orderSnap) {
                if (o.status.equals(Order.Status.PARTIAL) ||
o.status.equals(Order.Status.BOOKED)) {
                    orders().cancel(o.orderId)
                    ; log("Cancelled order: "
                    + o.orderId);
                }
            }

            //Count the number of booked or partial orders after the above
            cancel.

            int count = 0;
            orderSnap =
            orders().snapshot();
            for (Order o :
            orderSnap) {
                if (o.status.equals(Order.Status.PARTIAL) ||
o.status.equals(Order.Status.BOOKED)) {
                    count++;
```

```
            }
        }
        log("Number of open orders is: " + count);

    } else {
        log("No orders in system!");
    }
}


public void

    displayOpenOrders() {

    log("Displaying open

    orders...");
    log("Orders in the system: " +

    orders().snapshot().size()); if

    (orders().snapshot().size() > 0) {

        SortedSet<Order> orderSnap = orders().snapshot();
```

```
            for (Order o :
                orderSnap) {
                log(o.toStri
                ng());
                if
(o.status.equals(Order.Status.PARTIAL) ||
o.status.equals(Order.Status.BOOKED)) {
                    log("Order : " + o.toString() + ". Status: " + o.status);
                }
            }
        } else {
            log("No orders in system!");
```

```
        }
    }
}
```

Appendix 3 - Position Service Job

The follow two job makes use of the Position Service.

- CommitPositionTest - commits day trade position into the system
- PositionRiskDisplay - Displays the updated position.

```
import com.optionscity.freeway.api.*;
import java.util.Date;
/**
 * test committing day trade position
 * CL-20191120-F - commit this and then dispay using PositionRiskDisplay
 */
public class CommitPositionTest extends AbstractJob {
    @Override
    public void install(IJobSetup setup) {
        setup.addVariable("instrument","instrument to test with","instruments","");
    }
    String instrumentId;
    @Override
    public void begin(IContainer container) {
        super.begin(container);
        instrumentId = instruments().getInstrumentId(getStringVar("instrument"));
        PositionRisk risk = positions().getInstrumentRisk(instrumentId);
        if(risk.dayTradePosition<=0)
            failJob("position must be non-zero to test, enter some manual trades");
        positions().commitPositions(new Date());
        risk = positions().getInstrumentRisk(instrumentId);
        if(risk.dayTradePosition>0)
            failJob("position should be zero");
        stopJob("commit position worked successfully");
    }
}
```

```
import com.optionscity.freeway.api.*;
import com.optionscity.freeway.api.helpers.Parsing;
/**
 * sample job to display the position risk
 * CL-20191120-F - commit first using the CommitPositionTest
 */
public class PositionRiskDisplay extends AbstractJob {
    @Override
    public void install(IJobSetup setup) {
        setup.addVariable("accounts","comma delimited list of trade accounts, or blank
for all","string","");
    }
    String accounts;
    @Override
    public void begin(IContainer container) {
        super.begin(container);
        if(Parsing.isEmpty(accounts)) {
            PositionRisk risk = positions().getPositionRisk();
            log("position risk all accounts is "+risk);
        } else {
            for(String account : Parsing.toArray(accounts)){
                PositionRisk risk = positions().getPositionRisk(account);
                log("position risk for account "+account+" is "+risk);
                for(Position p : positions().getPositions(account)){
                    log("position is "+p);
                }
            }
        }
    }
}
```

SESSION III - SIGNALS AND WIDGETS

Signals

In the previous section we described how Freeway is an event based API. Freeway also provides a mechanism for jobs to communicate with each other job via message publishing. These 'Signals' are light-weight extensible, objects that a job can publish or subscribe to. Signals are found in the com.optionscity.freeway.api.messages package within the API.

System signals

There are many system signals sent which are caused by internal events or user driven events eg.

- **JobStarted/JobStopped** - a signal sent when a job instance is started/stopped.
- **MarketRisk** - sent by the system when a market risk (e.g. exchange QPS, protocol error), etc.) is encountered. The standard system response is to stop all automated trading in response to a market risk. It is a job's responsibility to take appropriate action in response to this signal.
- **SafetyTriggered** - sent by the system when a "safety" is triggered. The standard system response is to stop all automated trading in the affected instruments. It is a job's responsibility to take appropriate action in response to this signal.
- **StopAllOrders** - sent by the system when a user click 'Stop All Orders' in Metro. It is a system configuration option as to whether this user action stops Freeway orders as well. If automatic stopping of Freeway orders is disabled, a job may listen for this signal and take appropriate action. The 'sender' property contains the user id that performed the Metro action.

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(); // cf Session 2
}
```

It is important to note that the handling of Signals is a little different than the other events. 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.

Custom signals

The Signal class can be used as is however the power, flexibility and performance comes from customising each Signal into a new type based on event and overloading the callback accordingly. You may have an algo running that is monitoring market prices and you are interested in a specific target. You may only want another job to be notified if that target is reached and not have to specifically monitor the prices itself. It is easy to create a signal just for that event which can then be published.
Consider TargetPriceSignal class below:

```
public class TargetPriceSignal extends Signal {

 public final String instrumentId;
 public final double price;

 public TargetPriceSignal(String instrumentId, double price) {
  this.price = price;
  this.instrumentId = instrumentId;
 }
}
```

This class would usually reside in the common module of the project and then can be called by other classes/jobs/widgets.

Publishing

All signals are published via the signal method in the job container. Consider the example job below.

- The user defines a targetPrice they are interested
- monitors prices for a subset of instruments
- if the bid (could be any price) is greater than the target price then publish a TargetPriceSignal.

```
public class SenderJob extends AbstractJob {
 double targetPrice;

 public void install(IJobSetup setup) {
  setup.setDefaultDescription("simple sender job");
  setup.addVariable("instruments", "list of instruments", "instruments", "");
  setup.addVariable("targetPx", "target price", "double", "");
 }

 public void begin(IContainer container) {
  super.begin(container);
  container.subscribeToMarketBidAskMessages();
  container.filterMarketMessages(getStringVar("instruments"));
  targetPrice = getDoubleVar("targetPrice");
 }


 public void onMarketBidAsk(MarketBidAskMessage msg) {
  if(instruments().getPrices(msg.instrumentId).bid >= targetPrice ){
  //Signal sent below
  container.signal(new TargetPriceSignal(msg.instrumentId,
instruments().getPrices(msg.instrumentId).bid));
 }
}
```

Receiving

A typical receiving job looks like this:

```
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 received. Doing something...");
 }
}
```

However it is important note that 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. Therefore applying it to the above TargetPriceSignal example the onSignal callback will be:

```
public void onSignal(TargetPriceSignal signal) {
 log("TargetPriceSignal received. Doing something else...");
}
```

Signal handling - Safety and Risk

Metro has pre-defined safety parameters and will automatically take preventative action when triggered.

- **Safety Tick Worse** - Safety Tick Worse defines the amount by which a quote price will be reduced, or faded, when a Crossed Market, Trade Backout or Underlying Movement safety is triggered.
- **Crossed Market** - The Crossed Market safety will evaluate the outbound quote against the existing market. If the outbound quote crosses the current market by the amount specified in the bid/ask thresholds, the corresponding Safety Tick Worse setting will be applied. **Trade Backout** - The Trade Backout defines the number of contracts that need to be traded for a Tick Worse level to be applied.
- **Underlying Movement** - Movement monitors the market bid, ask and last prices. If any price moves by an amount set in any of the thresholds, the corresponding Tick Worse level will be applied.

- **Underlying Timeout** - The Underlying Timeout monitors the time between trades in an underlying. The setting is applied to specific underlying instruments, not the underlying class (all underlying instruments) to provide flexibility for products that have discrepancies in liquidity over the various underlyings that would be used for pricing.

The following signals will be fired in an adverse event:

- **SafetyTriggered Signal** is sent by the system when a Safety is triggered.
- **MarketRisk Signals** are sent by the system when a market risk (e.g. exchange QPS, protocol error), etc.) is encountered.
- **TradeRisk signals** are sent by the system when a trade risk event (too many trades) is encountered.

For SafetyTriggered, MarketRisk and TradeRisk signals:

NB. The standard system response is to STOP all automated trading in response to a market risk. It is a job's responsibility to take appropriate action in response to this signal.

If the user selects 'Stop All Orders' or 'Stop All Quotes' in the Heath Center then the following Signals will be sent:

- **StopAllOrders** - sent by the system when a user click 'Stop All Orders' in Metro. It is a system configuration option as to whether this user action stops Freeway orders as well. If automatic stopping of Freeway orders is disabled, a job may listen for this signal and take appropriate action. The 'sender' property contains the user id that performed the Metro action.
- **StopAllQuotes** - sent by the system when a user click 'Stop All Quotes' in Metro. By design, Freeway jobs are not automatically affected by this, nor are their manual quotes, but automatic quoting initiated by a Freeway job will be stopped. jobs can receive this signal and take appropriate action if any. The 'sender' property contains the user id that performed the Metro action.

```
import
com.optionscity.freeway.api.Abstract
Job; import
com.optionscity.freeway.api.IContain
er; import
com.optionscity.freeway.api.IJobSetu
p; import
com.optionscity.freeway.api.messages
.*;


/**
 * Simple job to demonstrate a job handling triggered safety events in Freeway
 * NB. SafetyTriggered is sent by the system when a "safety" is
triggered. The standard system response is to stop all
 * automated trading in the affected instruments. It is a job's
responsibility to take appropriate action in
 * response to this signal.
 */
public class SafetyTriggers extends AbstractJob {

    String
    space =
    " ";
    @Overrid
    e
    public void install(IJobSetup setup) {
        setup.setDefaultDescription("Listens to Safety and Market/Trade
Risk messages sent by system.");
    }
    @Override
    public void begin (IContainer
        container){
        super.begin(container);
        container.subscribeToSigna
        ls();
    }
    public void onSignal(SafetyTriggered msg){
        String logString = "Safety triggered. Details: " + msg.causedBy + space +
msg.description + space
                + msg.reason + space +
        msg.affected ; log(logString);
    }
    public void onSignal(MarketRisk msg){
        String logString = "MarketRisk triggered. Details: " + msg.description +
        space
+ msg.reason ;
        log(logString);
    }
    public void onSignal(TradeRisk msg){
        String logString = "TradeRisk triggered. Details: " + msg.description +
        space
```

```
+  msg.reason ;
        log(logString);
    }
    public void onSignal(JobStopped msg){
        String logString = "JobStopped message received. Details: " +
                                                                    m
        sg.reason ; log(logString);
    }
    public void onSignal(StopAllQuotes msg){
        String logString = "Stop all quotes button
        pressed" ; log(logString);
    }
    public void onSignal(StopAllOrders msg){
        String logString = "Stop all orders button
        pressed" ; log(logString);
    }
}
```

Signal handling - Orders routed to Freeway

Another important signal to handle is the **SubmitOrderSignal** sent from Metro to Freeway when the option is selected in the Metro Order ticket. This signal is targetted towards a specific job from a dropdown menu Contained with SubmitOrderSignal is an OrderRequest object which can be handled and manipulated by the algo before being submitted to market or rejected.
An outline usage handling this signal is:

```
public void onSignal(SubmitOrderSignal signal){
 OrderRequest or = signal.request;

 if(or.quantity > limit){
  //reject order and send notification to manager job
 } else {
  submitOrderRequest(or);
 }
}

 private void submitOrderRequest(or){
  //submit order logic
 }
}
```

A note on serialization

As stated Signals are extensible objects and therefore can contain user defined classes as payloads.
It is important to note that all classes (including sub-classes) must be defined as Serializable or else a de-serialization error is thrown when the signal is handled on the receiving side.

Design

Whereas each job has and manages its own queue we recommend splitting tasks between DIFFERENT tasks and allow each job to communicate over signals.

The main advantages are the queue size is shorter to allow for greater throughput and lower overall latency and critical tasks can be separated out from lower priority tasks.

Case study:
Consider an algo that take orders routed from freeway, monitors real-time positional and unrealised risk, calculated risk impact if order is submitted and rejects if risk limit is exceeded.

This job uses both the Order Service and the Positional Service therefore it is logical to split this into two jobs.

(a) Order manager job - Handles incoming order requests and submits to market or rejects

(b) Risk Calculation job - Calculates risk parameters and passes to Order manager job when

required.

Each job can then work separately but communicate via Signals on demand.

Widgets

Widget API is client side API that allows you to create customizable JComponents within the Java Swing framework to construct client-side GUI tools Widgets can communicate to jobs (via Signals) and conversly widget can subscribe to widgetSignals sent from job and can be be displayed via onRamp or loading from Freeway icon on Metro Front.

- Widget provide an intuitive and highly customizable user interface that compliments trading algos and provides the user with greater control and information that is not obtained with the out-of-box solutions. The API itself can be found here:http://utilities.optionscity.com/files/widgets/6.3.0/widget_javadoc/

Project Structure

The widget classes are built in the widgets/src/<widget.package>/ directory in the project (see Session I for more context on overall project structure).

In the above example there are two widgets created within the same project. You can of course just have one widget. In our case, ChartSampleWidget in widget.charting sample package and VolCurvePlottingWidget in widget.volcurveplotting package.

The class loader requires widgets/META-INF/services/com.optionscity.freeway.onramp.api.IWidget.

The contents of which are:

```
widget.chartingsample.ChartSampleWidget
widget.volcurveplotting.VolCurvePlottingWidget
```

MANIFEST.MF contains a list of dependencies eg.

```
Manifest-Version: 1.0
Class-Path: jcommon-1.0.16.jar jfreechart-1.0.13.jar freeway-6.3.0.jar
widgets-6.3.0.jar
```

Third Party Dependencies

The Metro server contains many third party java jars however there will be a case where you need to upload your own. The ant build script will bundle up the approriate jars and the class loader will distibute them accordingly on the server however they must be placed in the libs/app directory eg.

Widget Structure

We define a Widget as an extention of AbstractWidget in the same sense a job is an extension of AbstractJob. Let's examine a widget code in its most reduced form:

```
public class MyWidget extends AbstractWidget {
 JPanel mainPanel;

 @Override
 public String getDescription() {
  return "My first widget.";
 }

 @Override
 public JComponent begin(final IWidgetContainer container, IWidgetConfiguration
config) {
  super.begin(container, config);
  initialiseComponents();
  return mainPanel;
 }

 private void initialiseComponents(){
  JPanel mainPanel = new JPanel();
 }
}
```

The widget, like AbstractJob, has a begin() method which returns the JComponent of interest and in this case the JPanel mainPanel. The minimum requirement is to return a JComponent in begin method. You can add to the JComponent, of course, by adding objects to it eg. a Box within a Box etc. as described in Java Swing documentation NB. super.begin(container, config) is required to attach the container. Unlike Freeway, the container object is not used as extensively except to send signals.

Widget example - A Charting Sample Widget

We will now look at a practical example of a widget that charts electronic trade prices for instrument(s) of choice. This widget relies on a backend job for price details which will be sent of special signals for widgets, WidgetSignals. The widget also makes use of third party libraries also.

This Charting Sample GUI consists of the following components:

- **ChartingSampleJob** - Backend job subscribing to market data messages and sending widget signals that contain price information (algo/src/<package name if applicable>)
- **ChartSampleWidget** - The GUI itself that will receive trade prices (via widget signals) and display on screen (widgets/src/widget/<package name if applicable>)
- **MarketLastWidgetSignal** - Custom WidgetSignal that contains instrument and trade price information (common/src/<package name if applicable>)
- **Third party dependencies** - These jar files are stored in libs/app directory of the project.

Note that both the widget and the backend job depend on the MarketLastWidgetSignal class therefore it is contained within the **common** module.

ChartingSampleJob

Most widgets tend to have a job running on the Server to handle market data, calculations and signalling. In our case we have the following simple job:

```
import com.optionscity.freeway.api.AbstractJob;
import com.optionscity.freeway.api.IContainer;
import com.optionscity.freeway.api.IJobSetup;
import com.optionscity.freeway.api.messages.MarketLastMessage;

/**
 * sample job to listen to trades and fire a widget signal based on trade event
 */
public class ChartingSampleJob extends AbstractJob {


    @Override
    public void install(IJobSetup setup) {
        setup.setDefaultDescription("Charting sample job.");
        setup.addVariable("instruments", "instruments to watch", "instruments", "");
    }

    public void begin(IContainer container) {
        super.begin(container);
        log("Starting charting sample job.");

        container.subscribeToTradeMessages();
        container.filterMarketMessages(getStringVar("instruments"));
    }

// for electronic trades
    public void onMarketLast(MarketLastMessage msg) {

        log ("Received market last message for " + msg.instrumentId);
        log("Details: " +msg.instrumentId  +" , " +  msg.price + " , " + msg.quantity
+" , " + msg.seqnum);

        container.signalWidgets(null, new
signals.MarketLastWidgetSignal(msg.instrumentId,
                msg.price,
                msg.quantity,
                msg.seqnum));
    }

}
```

In essence this job simply subscribes to MarketLastMessages, filters on instrument(s) of choice and then publishes a MarketLastWidgetSignal with instrumentID, price, quantity and seqnum.
Notice in this case we are just passing the message on to the widget. This job is in algo/src of the project.
eg.:

ChartSampleWidget

Next we come to the widget itself.

```
package widget.chartingsample;
```

```
import
com.optionscity.freeway.onramp.api.AbstractWidget;
import
com.optionscity.freeway.onramp.api.IWidgetConfigurat
ion; import
com.optionscity.freeway.onramp.api.IWidgetContainer;
import org.jfree.chart.axis.DateAxis;
import
org.jfree.chart.plot.PlotOrientat
ion; import
org.jfree.data.time.Millisecond;
import
org.jfree.data.time.TimeSeries;
import
org.jfree.data.time.TimeSeriesCollect
ion; import
org.jfree.data.xy.XYSeries;
import
org.jfree.data.xy.XYSeriesCollect
ion; import signals.*;

import javax.swing.*;
import
org.jfree.chart.ChartPanel;
import
org.jfree.chart.ChartFactor
y; import
org.jfree.chart.JFreeChart;
import
org.jfree.chart.plot.XYPlot
; import
org.jfree.data.xy.XYDataset
; import
org.jfree.ui.ApplicationFra
me; import
org.jfree.ui.RefineryUtilit
ies;

import java.awt.*;
import
java.awt.event.ActionEvent
; import
java.awt.event.ActionListe
ner; import
java.text.SimpleDateFormat
;

/
*
*
*
/
```

```
public class ChartSampleWidget extends

    AbstractWidget { public String

    instrumentID;
    public double
    price, quantity;
    long time ;


    // Data collections and series
    public XYSeriesCollection xyDataSeriesCollection = new
    XYSeriesCollection(); public XYSeries xyDataSeries = new
    XYSeries("Data Series");
    public TimeSeriesCollection xyTimeSeriesCollection = new
    TimeSeriesCollection(); public TimeSeries xyTimeSeries = new
    TimeSeries("Time Series");

    @Override
    public String getDescription() {
        return "Simple widget to demonstrate charting.";
    }

    @Override
    public JComponent begin(final IWidgetContainer container,
IWidgetConfiguration config) {
        super.begin(container, config);

        JPanel mainPanel = new
        JPanel();
        mainPanel.setLayout(new
        BorderLayout());
        final ChartPanel chartPanel = createTimeChartPanel("Trade prices vs
        Time");

        JButton floatBtn = new
        JButton("Button");
        floatBtn.addActionListener(new
        ActionListener() {
```

```
        @Override
        public void actionPerformed(ActionEvent
            e) {
            displayFloatingChartPanel("Metro",
            chartPanel);
        }
    });

    mainPanel.add(chartPanel,
    BorderLayout.CENTER);
    mainPanel.add(floatBtn,
    BorderLayout.SOUTH); return
    mainPanel;
}

//XYChart creation - epoch time
private ChartPanel createChartPanel(String chartTitle) {
    JFreeChart xylineChart =
            ChartFactory.createXYLineChart(chartTitle,
            "Time",
            "Price",
            addDataSeriesToCo
            llection(),
            PlotOrientation.V
            ERTICAL, true,
            t
            r
            u
            e
            ,

            f
            a
            l
            s
            e
            )
            ;
    return new ChartPanel(xylineChart);
}

//Time chart creation - human readable time
private ChartPanel createTimeChartPanel(String chartTitle) {
    JFreeChart xylineChart =
            ChartFactory.createTimeSeriesChart(chartTitle,
            "Time",
            "Price",
            addTimeSeriesToCo
            llection(), true,
            t
            r
            u
```

```
            e
            ,

            f
            a
            l
            s
            e
            )
            ;
    // Domain settings eg. time
    format XYPlot plot =
    xylineChart.getXYPlot();
    DateAxis domain = new
    DateAxis();
    domain.setDateFormatOverride(new
    SimpleDateFormat("hh:mm:ss"));
    domain.setAutoRange(true);
    plot.setDomainAxis(domain);
    return new ChartPanel(xylineChart);
}

private XYDataset
    addDataSeriesToCollection () {
    xyDataSeriesCollection.addSeries(xyD
    ataSeries); return
    xyDataSeriesCollection;
}

private TimeSeriesCollection
    addTimeSeriesToCollection () {
    xyTimeSeriesCollection.addSeries(xyTimeSerie
    s);
    return xyTimeSeriesCollection;
}

//Display classes and methods
private void displayFloatingChartPanel(String appTitle, ChartPanel
    chartPanel) { JFrame frame = new XYLineChart_AWT(appTitle,
    chartPanel);
    frame.pack();
    RefineryUtilities.centerFrameOnScree
    n( frame ); frame.setVisible( true
    );
}
```

```
    public class XYLineChart_AWT extends ApplicationFrame {

        public XYLineChart_AWT( String applicationTitle, ChartPanel chartPanel ) {
            super(applicationTitle);
            chartPanel.setPreferredSize(new java.awt.Dimension(560, 367));
            final XYPlot plot = chartPanel.getChart().getXYPlot(); // todo - doesn't
appear to do anything
            setContentPane( chartPanel );
        }
    }

    // Signals - Electronic trades
    public void onSignal(MarketLastWidgetSignal signal) {
        time = signal.timestamp;
        instrumentID = signal.instrumentID ;
        price = signal.price ;
        quantity = signal.quantity;
        xyTimeSeries.add(new Millisecond(),price);
    }

}
```

The JComponent that we are returning in the begin method is a JPanel. We have overloaded the onSignal callback to filter only the singals we are interested in ie. our custom MarketLastWidgetSignals. The callback then updates the time chart with the latest trade price. A button has been added with an action listener. This does nothing but it is left as an exercise for the user to make this a 'Clear' button to clear the graph.

MarketLastWidgetSignal

The MarketLastWidgetSignal is an example of a custom-made WidgetSignal made just to send trade information to a widget.

This class is used by the job and the widget therefore has to be included in the common module so that both classes have access to it. onRamp will notify of any compilation errors upon loading.

In the same sense that custom signals are extensions of AbstractSignal, custom WidgetSignals are extensions of AbstractWidgetSignal. All included classes contained within the WidgetSignal or payload of the signal must be serialised correctly or else an error occurs when the signal is received. Typically signals would be in their own package in the project eg:

```
package signals;

import com.optionscity.freeway.api.WidgetSignal;

/**
 * signal to inform client of last price on electronic trades
 */

public class  MarketLastWidgetSignal extends WidgetSignal {

    public final String instrumentID;
    public final double price;
    public int quantity;
    public long seqnum, timestamp;

    public MarketLastWidgetSignal(String _instrumentID, double _price, int _quantity,
long _seqnum) {
       this.instrumentID = _instrumentID;
       this.price = _price;
       this.quantity = _quantity;
       this.seqnum = _seqnum;
    }

}
```

Third party dependencies

As stated earlier, any third-party jar must be included within the widget bundle. This is easily done by placing them in libs/app directory and exporting them correctly. IDEs will usually include the import statement at the beginning of the class eg.

```
import org.jfree.chart.axis.DateAxis;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.time.Millisecond;
import org.jfree.data.time.TimeSeries;
import org.jfree.data.time.TimeSeriesCollection;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;


import org.jfree.chart.ChartPanel;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.XYPlot;
import org.jfree.data.xy.XYDataset;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;
```

Building and Uploading

Building and uploading widgets is the same as described when building the job example. Using the ant build script allows the bundle to be formed in the way the Server expects. As a note the class loader in onRamp requires the bundle to be have a widgets.${BUNDLE_NAME}.jar and common.${BUNDLE_NAME}.jar in order to send the classes and dependencies to the client for widget installation. If they are not included in the bundle jar then the widget will not be installed on the client-wide there the recommendation is to use the build.xml provided. Simply run ant all or click the right hand panel in IntelliJ.

In onRamp select 'Upload' and navigate to the bundle.

By default when a widget is uploaded a prompt will be given to each client asking whether they would like to upgrade.

Configure backend job

The only configuration for ChartingSampleJob is the instruments(s) of choice. We have used the Freeway instrument matcher and therefore you only thing to do is to select the instruments in the usual way eg.

Displaying the Widget

With the backend job configured and running, the widget can be viewed in either onRamp or in the Metro Now front end.

- Via onRamp - Select blue quare in upper right > Select new dashboard > Select Widget..
- Metro NOW - Click Freeway Icon > Select widget (user can check box to pin a button to the widget panel).

You will see something similar to this:

Widget bundle source code

Attached is the above widget bundle.

- Untar and upload bundle.ChartingWidget.jar via onRamp
- Select ChartingSampleJob.java > Configuration > Select instrument(s) of choice
- Select blue square on upper right > New Dashboard > Widget... > ChartingSampleWidget

SESSION IV - MISCELLANEOUS TOPICS

This session will try to gather miscellaneous topics that we deem important.
It must be said that the Freeway documentation website contains a wealth of information not mentioned in this on boarding guide and should be used a first stop for consulation.

Pre-deployment Configuration

Job setup configs can be set in the resources/default.setup file. An example of this is below:

```
instances:MyJob=2


variable:MyJob.1.autostart=true
variable:MyJob.1.testmode=false
variable:MyJob.1.debugmode=false
variable:MyJob.1.owner="A Developer"
variable:MyJob.1.group=custom
variable:MyJob.1.description="Test job 1 for Freeway dev"
variable:MyJob.1.priority=0
variable:MyJob.1.timer=500


variable:MyJob.2.autostart=true
variable:MyJob.2.testmode=false
variable:MyJob.2.debugmode=false
variable:MyJob.2.owner="A Developer"
variable:MyJob.2.group=custom
variable:MyJob.2.description="Test job 2 for Freeway"
variable:MyJob.2.priority=2
variable:MyJob.2.timer=500
```

- **instances** - defines the number of instances of a particular job to initiate
- **testmode** - if set to true then orders and quotes are NOT submitted. Setting to false enables quote and order submission
- **debugmode** - If set to true debug logs will be published to log otherwise not.
- **owner** - defines the job owner
- **group** - defines the onRamp job grouping you wish this job to be placed
- **description** - simple note that will be displayed in onRamp
- **priority** - job priority takes an integer value from 0 to 3 (inclusive). When resources are low, higher priority is given to jobs with a larger priority value.
- **timer** - the timer() method's default period is 1 second. This can be changed to n where n is an integer and represents the period in milliseconds.

Conflation

The following events are conflated if the system is unable to process the events as they come in ensuring that the code will receive the most recent update.

Event	Event Handler
Market Bid/Ask Event	onMarketBidAsk(MarketBidAskMessage m)
Theoretical Event	onTheo(TheoMessage msg)
Book Depth Event	onBookDepth(BookDepthMessage msg)
Quote Event	onQuote(QuoteMessage message)

The following events are NOT conflated:

Event	Event Handler
Request For Quote Event	onRequestForQuote(RequestForQuoteMessage msg)
Order Event	onOrder(OrderMessage msg)
Trade Event	onTrade(TradeMessage msg)
Signal Event	onSignal(Signal signal)

NB. Signals are not conflated, so they are not a recommended way to send realtime trading values in very active instruments because any queuing could result in stale signals arriving at your job.

Threading, CPU affinity and Job Performance

Metro runs as a single java process on the server with Freeway jobs being single threaded polling from their own queue. Job priority setting (from low to ultra) can be set in onRamp and effectively sets the niceness of the job thread. It is possible to assign specific jobs to certain cores or even set High/Ultra jobs to specific cores on their own away from lower priority jobs. This is done in the affinity.conf configuration file.

CPU and other system stats are displayed in the Jobs and System Status tabs in onRamp.
Support staff will advise and make changes if you suspect any performance issue or wish to discuss performance further.

As stated early, job design should be done with consideration of separating heavy/critical tasks into separate jobs to improve efficiency and reduce queue length during high load periods.

Further information on design can be found in the Vela documentation portal.

Risk and Safety Behaviour

Recovery

In general Metro and Freeway takes a cautious approach to algorithmic trading. The System has default behaviour in the event of adverse events. Orders, trades and positions are backed up to database for record.

If there is a market risk/trade risk event a safety is triggered and the default behaviour is to stop algorithmic trading, cancelling any open orders and terminate quoting.

If a job stops or crashes then all open orders are cancelled too and quoting ceases too. Due to persistence of market data information, jobs can retrieve the relevant service snapshot of interest through the API when the job is restarted eg.

Service	Snapshot
orders()	orders().snapshot()
trades()	trades().snapshot()
positions()	positions().getPositions()

Safety Behaviour and Signals

As mentioned earlier different adverse event will cause a Safety to be fired an an accompanying system message will also be triggered.

The default behaviour is to stop all automated trading. The signals can be handled too and therefore, if the callback is implemented, the responsibility of the job to handle such an event.

The scenarios and corresponding signals are detailed below.

Scenario	Safety Signal
Crossed market / trade backout / underlying market movement beyond pre-defined (TickWorse) tolerances.	SafetyTriggered
Underlying Movement Timeout	SafetyTriggered and MarketRisk
Market Risk event (connectivity)	MarketRisk
Trade Risk event (too many trades)	TradeRisk

Callback cutoff

Another important point is that the job will terminate if a callback method takes longer than 15 seconds to complete. This excludes the begin method which is used only on job start up.

Recording and back-testing

FW supports recording market data and playing it back. This can be done programmatically via the IPlaybackService or manually through onRamp.

Market data recording is done through a recorder job (see Appendix 4 for an example) and is saved in a proprietary format to the recordings directory on the server.

Going from live to recorded data is very simple through the "System Status" tab. Simply select the file of interest and press play! You can control the playback rate from 1/4 to as fast as possible.

The IPlaybackService can be used to programatically use the playback and record files and use the server cron to start and stop recordings. More information can be seen on the documentation portal. If you are interested in this functionality then please read this.

IDB and data persistence

Freeway allows you to persist values in an IDB database and retrive them on demand. The values take the form (key,value) = (String, Serialised Object).
Below is an example job to show how to:

- Create the IDB
- Write to database (put) Retrieve from database (get) Make value null in
- Clear database

```
public class SimpleDBJob extends AbstractJob {

 public void install(IJobSetup setup) {
 }


 IDB test; //Create IDB here.
 int count=0;


 public void begin(IContainer container) {
  super.begin(container);
  test = container.getDB("test");

  test.put("string","value0");
  test.put("integer",123);
  test.put("double",123.456);

  removeEntry("string");
  makeNull("integer");
  clearDB();
 }

 public void removeEntry(String key){
  test.remove(key);
 }

 public void makeNull(String key){
  test.put(key, null);
 }

 public void clearDB(){
  test.removeAll();
 }
```

It is important to reiterate that the value object must be Serialised properly (including subclasses and included classes) or else an error is thown.

Widget Smart Linking

It is possible for out-of-box Metro widgets (eg. Trade Sheet, City Market, Order Ladder) to send current instrument selection to custom widgets.

Eg. If you highlight certain cells in Trade Sheet say then the instruments that have been selected can be passed to a widget. This is especially useful as a dynamic instrument input for a widget.

A LinkedGroupSelectionChanged Signal is sent when the instrument selection changes. Contained within this object is:

java.lang.String	group
java.util.Set<java.lang.String>	instrumentIds
java.lang.String	source

You must set the group name for a particular widget by going to the menu Widget > Settings > Smart Link and defining it there.

It is advised that for each open widget that you create a new link eg. Link1, Link2, Link3 etc.

The widget will then listen to these signal with the callback below and you can then filter on widget group of interest eg.

```
public void onSignal(LinkedGroupSelectionChanged signal){
    if (signal.group.equals("Link1")){
        doSomethingWithNewInstrumentSelection(signal);
    } else if (signal.group.equals("Link2")){
        doSomethingElseWithNewInstrumentSelection(signal);
    }

 // etc.

}
```

Custom Volatility Curves

Building and uploading custom volatility curves

Appendix 1 contains the same build.xml from in Session I.

The class loader expects a bundle called common.curves.jar which contains META-INF/services directory with the following files:

- com.optionscity.freeway.api.VolatilityCurve$Custom
- com.optionscity.freeway.api.VolatilityCurve$VolatilitySlide$Custom

The contents of which are the qualified path of the custom curves and slides respectively.
Attached is a source code example to contruct, build and bundle package ready for uploading. Simply run:

'ant all' then 'ant packageCurves' which will create /artifacts/common/common.curves.jar ready for upload via. onramp

NB custom curves and slides are NOT instances of AbstractJob or AbstractWidget therefore bundling the classes must be done as described above.

Setting Custom Curves

The Volatility Curve can be set in the Theoretrical Wizard

ExternalAPI

FW job acts as an integration point with an external application over an open socket.

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 is formatted as UTF-16 characters, separated by a carriage return, line feed, or carriage return + line feed.

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.

You must configure a port, and can optionally use SSL, limit connections, and enable verbose logging.

In Appendix 2 is an example of a receiver job that receives messages from a socket. 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.

To run the job take the following steps:

- Set job port configuration to 10000 and enable SSL connection
- Open SSL client to that IP:PORT

```
openssl s_client -connect 127.0.0.1:10000
```

- Send a String command eg. in this case we are placing an order on FGBL-20191206-F for 1@12103.5

```
order,FGBL-20191206-F,1,12103.5
```

- Output will be a response from the onOrder callback eg.

```
status,order 100021 on FGBL-20191206-F, LIMIT BUY 1 @ 12103.5, hedge NONE, filled 0 @ 0.0, tif DAY, status = NEW
status,order 100021 on FGBL-20191206-F, LIMIT BUY 1 @ 12103.5, hedge NONE, filled 0 @ 0.0, tif DAY, status = BOOKED
status,order 100021 on FGBL-20191206-F, LIMIT BUY 1 @ 12103.5, hedge NONE, filled 1 @
```

173.78, tif DAY, status = FILLED

NB. Vela documentation site has an example of an SSL socket application (not Freeway) that can be used to send to the ExternalAPI job

Appendices

Appendix 1 - Custom Vol Curves

build.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="optionscity_bundle" default="all">
    <property environment="env"/>
    <property file="project.properties" />
    <target name="all" depends="clean, init, retrieveApiJars,
unzipLibs, compile, packageWidget, packageCommon, packageBundle"/>
    <target name="clean">
        <delete dir="${basedir}/build/widget"/>
 <delete dir="${basedir}/build/common"/>
        <delete dir="${basedir}/build/algo"/>
        <delete dir="${basedir}/artifacts"/>
    </target>
    <target name="init">
     <mkdir dir="${basedir}/build/widget"/>
        <mkdir dir="${basedir}/build/common"/>
        <mkdir dir="${basedir}/build/algo"/>
        <mkdir dir="${basedir}/artifacts"/>
     <mkdir dir="${basedir}/artifacts/widget"/>
        <mkdir dir="${basedir}/artifacts/common"/>
        <mkdir dir="${basedir}/algo"/>
        <mkdir dir="${basedir}/algo/src"/>
        <mkdir dir="${basedir}/common"/>
        <mkdir dir="${basedir}/common/src"/>
        <mkdir dir="${basedir}/libs"/>
        <mkdir dir="${basedir}/libs/system"/>
        <mkdir dir="${basedir}/libs/app"/>
        <mkdir dir="${basedir}/resources"/>
        <mkdir dir="${basedir}/resources/dashboards"/>
        <mkdir dir="${basedir}/resources/setup"/>
```

```
        <touch file="${basedir}/resources/setup/default.setup"/>
    </target>
    <!--
    initWidgets creates necessary files and dirs to create
    custom widgets User should create package 'widget' in
    widget/src
    NB. contents of IWidget file must be qualified path to
widget(s) eg. widget.myWidget
    -->
    <target name="initWidgets">
        <mkdir dir="${basedir}/widgets"/>
        <mkdir dir="${basedir}/widgets/META-INF"/>
        <touch file="${basedir}/widgets/META-INF/MANIFEST.MF"/>
        <mkdir dir="${basedir}/widgets/META-INF/services"/>
        <touch
file="${basedir}/widgets/META-
INF/services/com.optionscity.freeway.onramp.api.IWidget"
/>
        <mkdir dir="${basedir}/widgets/src"/>
    </target>
    <!--
    initCurves creates necessary files and dirs to create custom
    vol curves User needs to create packages 'curve' and 'slide'
    in common/src
    NB. contents of VolatilityCurve$Custom must be qualified path to
curve(s) eg. curve.myCurve and
    contents of VolatilityCurve$VolatilitySlide$Custom must be
qualified path to slides(s) eg. slides.mySlide
    -->
    <target name="initCurves">
        <mkdir dir="${basedir}/common"/>
        <mkdir dir="${basedir}/common/META-INF"/>
        <touch file="${basedir}/common/META-INF/MANIFEST.MF"/>
        <mkdir dir="${basedir}/common/META-INF/services"/>
        <touch
file="${basedir}/common/META-
INF/services/com.optionscity.freeway.api.VolatilityCurve$ Custom"/>
        <touch
file="${basedir}/common/META-
INF/services/com.optionscity.freeway.api.VolatilityCurve$
VolatilitySlide$Custom"/>
        <mkdir dir="${basedir}/common/src"/>
    </target>
    <target name="retrieveApiJars">
        <mkdir dir="${basedir}/libs/system/${TARGET_OC_VERSION}"/>
        <get
src="http://utilities.optionscity.com/files/freeway/${TARGET_OC_VERSION}/fr
eeway-${TAR GET_OC_VERSION}.jar"

dest="${basedir}/libs/system/${TARGET_OC_VERSION}/freeway-
${TARGET_OC_VERSION}.jar" usetimestamp="true"/>
 <get
```

```
src="http://utilities.optionscity.com/files/widgets/${TARGET_OC_VERSION}/wi
dgets-${TAR GET_OC_VERSION}.jar"

dest="${basedir}/libs/system/${TARGET_OC_VERSION}/widgets-
${TARGET_OC_VERSION}.jar" usetimestamp="true"/>
    </target>
    <target name="unzipLibs">
        <unzip dest="${basedir}/build/common">
            <fileset dir="${basedir}/libs/app">
                <include name="*.jar"/>
            </fileset>
```

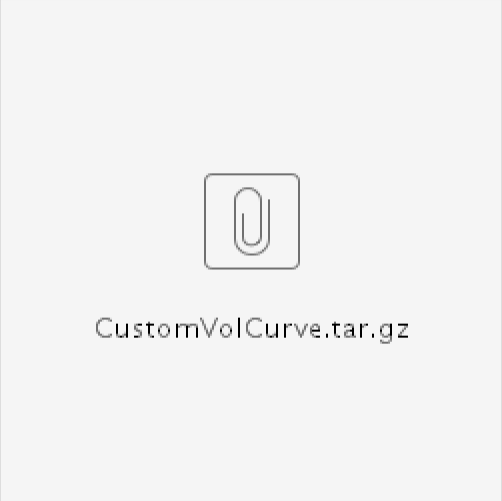
```
        </unzip>
    </target>
    <target name="compile" depends="init">
        <javac nowarn="on" fork="false" srcdir="${basedir}/common"
destdir="${basedir}/build/common" debug="true" optimize="on"
includeantruntime="false"
>
            <compilerarg value="-Xlint:all"/>
            <classpath>
                <fileset dir="${basedir}/libs/system/${TARGET_OC_VERSION}">
              <include name="freeway-${TARGET_OC_VERSION}.jar"/>
              <include name="widgets-${TARGET_OC_VERSION}.jar"/>
                </fileset>
                <fileset dir="${basedir}/libs/app">
                    <include name="*.jar"/>
                </fileset>
            </classpath>
        </javac>
 <javac nowarn="on" fork="true" srcdir="${basedir}/widgets"
destdir="${basedir}/build/widget" memoryInitialSize="512m"
memorymaximumsize="768m" debug="true" optimize="on"
includeantruntime="false" >
             <compilerarg value="-Xlint:all"/>
             <classpath>
                 <fileset dir="${basedir}/libs/system/${TARGET_OC_VERSION}">
                     <include name="freeway-${TARGET_OC_VERSION}.jar"/>
                     <include name="widgets-${TARGET_OC_VERSION}.jar"/>
                 </fileset>
                 <fileset dir="${basedir}/libs/app">
                     <include name="*.jar"/>
                 </fileset>
                 <pathelement location="${basedir}/build/common"/>
             </classpath>
        </javac>
        <javac nowarn="on" fork="true" srcdir="${basedir}/algo"
destdir="${basedir}/build/algo" memoryInitialSize="512m"
memorymaximumsize="768m" debug="true" optimize="on"
includeantruntime="false" >
            <compilerarg value="-Xlint:all"/>
            <classpath>
                <fileset dir="${basedir}/libs/system/${TARGET_OC_VERSION}">
              <include name="freeway-${TARGET_OC_VERSION}.jar"/>
              <include name="widgets-${TARGET_OC_VERSION}.jar"/>
                </fileset>
                <fileset dir="${basedir}/libs/app">
                    <include name="*.jar"/>
                </fileset>
                <pathelement location="${basedir}/build/common"/>
            </classpath>
        </javac>
    </target>
    <target name="packageCommon">
```

```
                                    <jar
 destfile="${basedir}/artifacts/common/common.${BUNDLE_NAME}.jar">
   <fileset dir="${basedir}/build/common" includes="**/*.class"/>
   <fileset dir="${basedir}/build/common" includes="**/*.properties"/>
</j
ar>
</target>
<target name="packageCurves">
    <jar destfile="${basedir}/artifacts/common/common.curves.jar">
        <fileset dir="${basedir}/build/common/curve" includes="**/*.class"/>
        <fileset dir="${basedir}/build/common/curve"
        includes="**/*.properties"/>
        <fileset dir="${basedir}/build/common/slide" includes="**/*.class"/>
```

```
            <fileset dir="${basedir}/build/common/slide"
            includes="**/*.properties"/>
            <fileset dir="${basedir}/common/META-INF" includes="**/*"/>
        </jar>
    </target>
    <target name="packageWidget">
         <jar destfile="${basedir}/artifacts/widget/widgets.${BUNDLE_NAME}.jar">
             <fileset dir="${basedir}/build/widget" includes="**/*.class"/>
             <fileset dir="${basedir}/widgets/" includes="META-INF/**"/>
             <fileset dir="${basedir}/libs/app" includes="*.jar"/>
         </jar>
    </target>
    <target name="packageBundle">
         <jar destfile="${basedir}/artifacts/bundle.${BUNDLE_NAME}.jar">
             <fileset
dir="${basedir}/artifacts/common"
includes="common.${BUNDLE_NAME}.jar"/>
             <fileset
dir="${basedir}/artifacts/widget"
includes="widgets.${BUNDLE_NAME}.jar"/>
             <fileset dir="${basedir}/build/algo" includes="*.class"/>
             <fileset dir="${basedir}/resources/dashboards"
             includes="*.dashboard"/>
             <fileset dir="${basedir}/resources/setup" includes="default.setup"/>
             <fileset dir="${basedir}/libs/app" includes="*.jar"/>
         </jar>
    </target>
    <target name="propsTest">
```

```
        <echo>Bundle name is ${BUNDLE_NAME}</echo>
    </target>
</project>
```

Custom Volatility Curves code example



Appendix 2 - ExternalAPI job

```
import com.optionscity.freeway.api.*;
import
com.optionscity.freeway.api.jobs.ExternalAPI;
import
com.optionscity.freeway.api.messages.OrderMess
age;

public class OrderManagerExternalAPI extends ExternalAPI {
    public static final JobPermission[] PERMISSIONS = {
            JobPermission.NETWORK_ACCESS, JobPermission.ORDER_SEND,
            JobPermission.SEND_SIGNALS };
    public void
        install(IJobSetup
        setup) {
        super.install(setup);
        setup.setDefaultDescription("order manager using external api");
    }
    public void begin(IContainer
        container){
        super.begin(container);
        container.subscribeToOrder
```

```
    Messages();
}
public void
    onOrder(OrderMessage
    msg) {
    super.onOrder(msg);
    Order order = orders().getOrder(msg.orderId);
    // the current order status is reported, not the event status, so
    // duplicate "statuses" are possible. maybe include the
    event status ? send("status,"+order);
}
public void
    onSignal(ExternalAPIMessage
    signal){ String[] parts =
    signal.message.split(",");
    if(parts.length==0)
```

```
            r
        e
        t
        u
        r
        n
        ;

        t
        r
        y

        {
            processRequest(signal.session,parts);
        } catch (UnknownSession
            unknownSession) {
            log(unknownSession.toStrin
            g());
        }
    }
    private void processRequest(String session,String[] parts) throws
        UnknownSession { if("order".equals(parts[0])) {
            //
            order,instrument,quanti
            ty,price
            if(parts.length!=4){
                send(session,"rejected, format is
                order,instrument,quantity,price"); return;
      }
      int quantity =
      Integer.parseInt(parts[2]); double
      price = Double.parseDouble(parts[3]);
      if(quantity<0)
          orders().submit(OrderRequest.sell(parts[1],Math.abs(quantity),price));
      else
pr        orders().submit(OrderRequest.buy(parts[1], Math.abs(quantity),
ic
e)
);
       } else if("modify".equals(parts[0])){
            //
            modify,orderid,quanti
            ty,price
            if(parts.length!=4){
                send(session,"rejected, format is
                modify,orderid,quantity,price"); return;
            }

orders().modify(Long.parseLong(parts[1]),Integer.parseInt(parts[2]),Double.
parseDouble (parts[3]));
       } else if("cancel".equals(parts[0])){
            //
            cancel,or
```

```
    derid
    if(parts.
    length!=2
    ){
        send(session,"rejected, format is
        cancel,orderid"); return;
    }
    orders().cancel(Long.parseLong(parts[1]));
} else {
    send(session,"rejected, command is order, modify, or cancel");
```

```
            }
        }
    }
```

Appendix 3 - Safety Signal Handling

```
import
com.optionscity.freeway.api.Abstract
Job; import
com.optionscity.freeway.api.IContain
er; import
com.optionscity.freeway.api.IJobSetu
p; import
com.optionscity.freeway.api.messages
.*;
/**
 * Simple job to demonstrate a job handling triggered safety events in Freeway
 * NB. SafetyTriggered is sent by the system when a "safety" is
triggered. The standard system response is to stop all
 * automated trading in the affected instruments. It is a job's
responsibility to take appropriate action in
 * response to this signal.
 */
public class SafetyTriggers extends AbstractJob {

    String
    space =
    " ";
    @Overrid
    e
    public void install(IJobSetup setup) {
        setup.setDefaultDescription("Listens to Safety and Market/Trade
Risk messages sent by system.");
    }
    @Override
    public void begin (IContainer
        container){
        super.begin(container);
        container.subscribeToSigna
        ls();
    }

    public void onSignal(SafetyTriggered msg){
        String logString = "Safety triggered. Details: " + msg.causedBy + space +
msg.description + space
                + msg.reason + space +
        msg.affected ; log(logString);
    }
    public void onSignal(MarketRisk msg){
        String logString = "MarketRisk triggered. Details: " + msg.description +
        space
+ msg.reason ;
        log(logString);
    }
    public void onSignal(TradeRisk msg){
        String logString = "TradeRisk triggered. Details: " + msg.description +
        space
+  msg.reason ;
```

```
        log(logString);
    }
    public void onSignal(JobStopped msg){
        String logString = "JobStopped message received. Details: " +
                                                                    m
        sg.reason ; log(logString);
    }
    public void onSignal(StopAllQuotes msg){
        String logString = "Stop all quotes button
        pressed" ; log(logString);
    }
    public void onSignal(StopAllOrders msg){
        String logString = "Stop all orders button
        pressed" ; log(logString);
    }
}
```

Appendix 4 - Recorder and Playback Jobs

Recorder job

```
import com.optionscity.freeway.api.AbstractJob;
import com.optionscity.freeway.api.IContainer;
import com.optionscity.freeway.api.IJobSetup;
import com.optionscity.freeway.api.helpers.Formatting;
import java.util.Date;
/**
 * job to write market data to playback file
 */
public class MarketRecorder extends AbstractJob {
    public void install(IJobSetup setup) {
        setup.addVariable("filename","filename","string","");
        setup.addVariable("append","append date to filename?","boolean","true");
        setup.addVariable("instruments","instruments to record","instruments","");
        setup.setDefaultDescription("record market data for playback");
    }
    String filename;
    @Override
    public void begin(IContainer container) {
        super.begin(container);
        filename = getStringVar("filename");
        if(getBooleanVar("append")){
            filename = filename + "_" + Formatting.toYYMMDDHH(new Date());
        }
        log("starting recording to file '"+filename+"'");
        container.getPlaybackService().record(filename,getStringVar("instruments"));
// Files in recordings/ directory
    }
    @Override
    public void end(IContainer container) {
        super.end(container);
        log("stopping recording to file '"+filename+"'");
        container.getPlaybackService().stop(filename);
    }
}
```

Playback job

```
import com.optionscity.freeway.api.AbstractJob;
import com.optionscity.freeway.api.Book;
import com.optionscity.freeway.api.IContainer;
import com.optionscity.freeway.api.IJobSetup;
import com.optionscity.freeway.api.messages.BookDepthMessage;

public class DisplayRecordedPrices extends AbstractJob {
    @Override
    public void install(IJobSetup setup) {
        setup.addVariable("instruments", "instruments to filter", "instruments", "");
    }
    public void begin(IContainer container){
        super.begin(container);
        //container.filterMarketMessages(getStringVar("instruments"));
    }
    @Override
    public void onBookDepth(BookDepthMessage msg) {
        Book bk = instruments().getBook(msg.instrumentId);
        log("Instrument: " + msg.instrumentId + ", bid:" + bk.bid[0] + ", ask: " +
bk.ask[0]);
    }
}
```

Output:

```
localhost 2020-09-03 13:03:23,279.484 DisplayRecordedPrices.1 Instrument:
ES-20200918-F, bid:3566.0(40), ask: 3566.25(24)
localhost 2020-09-03 13:03:24,759.654 DisplayRecordedPrices.1 Instrument:
ES-20200918-F, bid:3566.0(40), ask: 3566.25(25)
localhost 2020-09-03 13:03:24,891.595 DisplayRecordedPrices.1 Instrument:
ES-20200918-F, bid:3566.0(40), ask: 3566.25(25)
localhost 2020-09-03 13:03:26,359.730 DisplayRecordedPrices.1 Instrument:
ES-20200918-F, bid:3566.0(40), ask: 3566.25(25)
localhost 2020-09-03 13:03:26,763.847 DisplayRecordedPrices.1 Instrument:
ES-20200918-F, bid:3566.0(44), ask: 3566.25(25)
localhost 2020-09-03 13:03:26,899.854 DisplayRecordedPrices.1 Instrument:
ES-20200918-F, bid:3566.0(44), ask: 3566.25(25)
localhost 2020-09-03 13:03:28,503.994 DisplayRecordedPrices.1 Instrument:
ES-20200918-F, bid:3566.0(40), ask: 3566.25(26)
localhost 2020-09-03 13:03:28,764.011 DisplayRecordedPrices.1 Instrument:
ES-20200918-F, bid:3566.0(40), ask: 3566.25(26)
localhost 2020-09-03 13:03:29,164.087 DisplayRecordedPrices.1 Instrument:
ES-20200918-F, bid:3566.0(40), ask: 3566.25(27)
localhost 2020-09-03 13:03:29,308.216 DisplayRecordedPrices.1 Instrument:
ES-20200918-F, bid:3566.0(40), ask: 3566.25(27)
localhost 2020-09-03 13:03:29,968.305 DisplayRecordedPrices.1 Instrument:
ES-20201218-F, bid:3555.5(4), ask: 3556.0(2)
```