Deploy Anywhere Routing Engine User Guide

Overview

The Deploy Anywhere Routing Engine, or DARE, is responsible for managing sessions with external brokers and exchanges, sending orders, receiving execution reports, and maintaining order history. DARE has a JMS bus for order and execution report traffic and a web services API for order history and positions. DARE is a hardware appliance with software components pre-installed and configured. DARE is available via commercial license to be installed in your datacenter or in the cloud. It is possible to license DARE without the hardware appliance, if necessary.

Running DARE

DARE is a command-line application with no UI. DARE acts as part of the server component in the Marketcetera Automated Trading Platform stack.

Windows

On Windows, DARE is started from the Start Menu: Marketcetera-2.4.0 DARE->Start Marketcetera Server Components.

```
03 Jan 2014 06:27:01,218  INFO [main] marketcetera.ors.OrderRoutingSystem (OrderRoutingSystem.java:357) -
Copyright (c) 2006-2013 Marketcetera, Inc.
03 Jan 2014 06:27:01,245  INFO [main] marketcetera.ors.OrderRoutingSystem (OrderRoutingSystem.java:358) - ORS
version '2.4.0' (build 'cc.build.335')
03 Jan 2014 06:27:01,245  INFO [main] marketcetera.ors.OrderRoutingSystem (OrderRoutingSystem.java:361) - ORS
is starting
03 Jan 2014 06:27:10,930  INFO [main] marketcetera.ors.OrderRoutingSystem (OrderRoutingSystem.java:380) - ORS
started successfully. Ctrl-C to exit
```

Linux

```
$ ./startServerComponents.sh
nohup: redirecting stderr to stdout
Starting mysqld daemon with databases from /home/colin/marketcetera/marketcetera-2.4.0/mysql/data
06 Jan 2014 10:43:02,401  INFO [main] marketcetera.ors.OrderRoutingSystem (OrderRoutingSystem.java:357) -
Copyright (c) 2006-2013 Marketcetera, Inc.
06 Jan 2014 10:43:02,426  INFO [main] marketcetera.ors.OrderRoutingSystem (OrderRoutingSystem.java:358) - ORS
version '2.4.0' (build 'cc.build.335')
06 Jan 2014 10:43:02,427  INFO [main] marketcetera.ors.OrderRoutingSystem (OrderRoutingSystem.java:361) - ORS
is starting
Jan 6, 2014 10:43:03 AM java.util.prefs.FileSystemPreferences$7 run
06 Jan 2014 10:43:12,850  INFO [main] marketcetera.ors.OrderRoutingSystem (OrderRoutingSystem.java:380) - ORS
started successfully. Ctrl-C to exit
```

Broker and Exchange Connections

DARE maintains connections to one or more brokers and exchanges via FIX (Financial Information eXchange) sessions. It is possible to connect to non-FIX destinations with some custom configuration. Upon start, the DARE attempts to establish connections to all configured order destinations. If connection is lost, Marketcetera Automated Trading Platform client applications are notified of the loss of connectivity and DARE attempts to reconnect at regular intervals.

Brokers and exchanges are identified in DARE configuration. Upon installation, DARE is already configured to connect to the Marketcetera simulated exchange. The configuration files for a broker define the FIX session settings for that broker, gateway host and port, like start of session, end of session, and FIX version. You can also define optional attributes like message modifiers and response modifiers.

Message Modifiers

A message modifier is used to modify or selectively modify orders before they are released to a broker or exchange. A message modifier is simply a unit of code that receives the order before it is sent and has the opportunity to modify it or throw an exception to prevent it being sent at all. Multiple message modifiers can be applied in the sequence defined in the configuration files. Message modifiers are defined per broker.

In addition to message modifiers, there are also pre-send message modifiers. The distinction between the two are simply when the modifiers are activated. Message modifiers are applied before routing modification. Pre-send message modifiers are applied after routing modification.

To create a message modifier or a pre-send message modifier, write a Java class that extends `org.marketcetera.ors.filters.MessageModifier`. Build the class into a JAR and deploy the it to DARE lib directory. After the next restart, the message modifier will be available for use. To activate the message modifier, edit the `main.xml` file for your broker. Make sure this entry exists:

```
<property name="modifiers" ref="your_modifiers"/>
```

or, for pre-send modifiers, this:

```
<property name="preSendModifiers" ref="your_ps_modifiers"/>
```

The ref can be any name you want as long as it matches an entry in either `modifiers.xml` or `ps_modifiers.xml`, as appropriate. The contents of `modifiers.xml` might look like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema
/beans/spring-beans-3.2.xsd">
  <bean id="your_modifiers" class="org.marketcetera.ors.filters.MessageModifierManager">
    <property name="messageModifiers">
      <list>
        <bean id="your_modifier_1" class="com.acme.yourcompany.MyMessageModifier">
          <property name="property1" value="some value here"/>
        </bean>
        <bean id="your_modifier_2" class="com.acme.yourcompany.MyMessageModifier">
          <property name="property1" value="some value here"/>
        </bean>
      </list>
    </property>
  </bean>
</beans>
```

This would define two message modifiers, to be applied in the order indicated. Each modifier can have its own properties, or none. In addition to the `MessageModifier` interface, there is also the `SessionAwareMessageModifier` interface. The operation of the modifier is the same, but the session-aware version has access to the session info of the order sender.

Here is the implementation of FieldDuplicatorMessageModifier:

```
public class FieldDuplicatorMessageModifier
        implements MessageModifier
{
    /**
     * Create a new FieldDuplicatorMessageModifier instance.
     *
     * @param sourceField an <code>int</code> value
     * @param destField an <code>int</code> value
     */
    public FieldDuplicatorMessageModifier(int sourceField,
                                          int destField)
    {
        this.sourceField = sourceField;
        this.destField = destField;
    }
    /* (non-Javadoc)
     * @see org.marketcetera.ors.filters.MessageModifier#modifyMessage(quickfix.Message, org.marketcetera.ors.
history.ReportHistoryServices, org.marketcetera.quickfix.messagefactory.FIXMessageAugmentor)
     */
    @Override
    public boolean modifyMessage(Message inMessage,
                                 ReportHistoryServices inHistoryServices,
                                 FIXMessageAugmentor inAugmentor)
            throws CoreException
    {
        try {
            if(inMessage.isSetField(sourceField)) {
                String value = inMessage.getString(sourceField);
                inMessage.setField(new StringField(destField,value));
                return true;
            }
            return false;
        } catch (FieldNotFound fieldNotFound) {
            throw new CoreException(fieldNotFound);
        }
    }
    /**
     * field to which to duplicate
     */
    private int destField;
    /**
     * field from which to duplicate
     */
    private int sourceField;
}
```

This message modifier copies the value from one field to another, if the source field is set. This can be useful for brokers that might require the symbol (field 55) to also be in another field, for example. The modifier returns `true` or `false` depending on whether the message was modified or not. To prevent the order from being sent, the modifier can throw an exception. Note that you can implement risk management using this technique, if you like.

There are several pre-defined message modifiers you can use as-is or as a model for your own modifiers.

- ConditionalFieldRemoverMessageModifier - removes a field from a message if a given condition is met
- ConditionalMessageModifier - modifies a message if a given condition is met
- DefaultMessageModifier - sets specified fields to specified values in all messages
- FieldDuplicatorMessageModifier - copies the string value of one field to another
- FieldOverrideMessageModifier - sets the value of the given fields regardless of the original value
- OrderTagRecorder - records specified fields from outgoing orders, used with OrderTagRemapper
- OrderTagRemapper - remaps saved order tags onto execution reports, used with OrderTagRecorder. The use case here is if you want to retain some tags on orders that your broker won't accept (identification tags not present in the canonical FIX spec, for example). You use the OrderTagRecorder and the OrderTagRemapper on outgoing orders and incoming execution reports, respectively. The tags are transparently removed from outgoing orders and added back to incoming execution reports without the broker ever seeing them. You can then sort or filter on that tag.
- TransactionTimeInsertMessageModifier - inserts the TransactTime field into an order if it's not currently present
- UserInfoMessageModifier - sets the SenderSubID on outgoing messages according to the Marketcetera user that initiated the order

Response Modifiers

A response modifier is similar to a message modifier and implements the same interface. It is used to modify incoming execution reports. To activate a response modifier, make sure this property is in the broker main.xml file:

```
<property name="responseModifiers" ref="your_response_modifiers"/>
```

The ref can be any name you want as long as it matches an entry in resp_modifiers.xml. The contents of resp_modifiers.xml might look like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema
/beans/spring-beans-3.2.xsd">
  <bean id="your_response_modifiers" class="org.marketcetera.ors.filters.MessageModifierManager">
    <property name="messageModifiers">
      <list>
        <bean id="your_response_modifier_1" class="com.yourcompany.MyMessageModifier">
          <property name="property1" value="some value here"/>
        </bean>
        <bean id="your_response_modifier_2" class="com.yourcompany.MyMessageModifier">
          <property name="property1" value="some value here"/>
        </bean>
      </list>
    </property>
  </bean>
</beans>
```

Order Routing

Order routing in DARE is the process of choosing to which of the available destinations an order should be sent. A given order may be sent to any destination. The order as created in the Marketcetera Automated Trading Platform will be customized as necessary depending on the chosen destination.

When an order is created, you may explicitly choose a destination in your strategy or in the UI, or you may leave it to DARE to route the order. DARE is configured with a default destination. You may also establish your own custom rules for order routing.

The precedence for order routing is as follows:

1. Destination selected when the order is created
2. Destination selected from configurable routing rules
3. Default destination

The default destination is defined in ors/conf/brokers/selector.xml in the property defaultBroker:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema
/beans/spring-beans-3.2.xsd">
  <bean id="metc_selector" class="org.marketcetera.ors.brokers.SpringSelector">
    <!-- Default broker. -->
    <property name="defaultBroker" ref="metc_broker"/>
  </bean>
</beans>
```

Set the default broker to the id of a broker in one of the broker main.xml files.

To establish your own routing rules, you need to create one or more classes that extend org.marketcetera.ors.brokers.SpringSelectorEntry. When routing an order, DARE passes the outgoing order to each class you define/ The selector entry you create returns a boolean value that determines whether the order should go to the broker associated with the selector entry. If all entries return false, the order is routed to the default broker.

The selector.xml file with selectors looks like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema
/beans/spring-beans-3.2.xsd">
  <bean id="metc_selector" class="org.marketcetera.ors.brokers.SpringSelector">
    <!-- Routing entries -->
    <property name="entries">
      <list>
        <bean class="com.yourcompany.RoutingSelector1">
          <property name="broker" ref="broker1"/> <!-- matches the id of a broker bean -->
          <property name="skipIfUnavailable" value="true"/>
        </bean>
        <bean class="com.yourcompany.RoutingSelector2">
          <property name="broker" ref="broker2"/> <!-- matches the id of a broker bean -->
          <property name="skipIfUnavailable" value="true"/>
        </bean>
      </list>
    </property>
    <!-- Default broker -->
    <property name="defaultBroker" ref="broker3"/>
  </bean>
</beans>
```

Here is a sample broker selector that routes by security type:

```
package org.marketcetera.ors.brokers;


import org.apache.commons.lang.Validate;
import org.marketcetera.trade.Order;
import org.marketcetera.trade.SecurityType;
import org.springframework.beans.factory.InitializingBean;


/**
 * Selects a target broker by {@link SecurityType}.
 *
 * @author <a href="mailto:colin@marketcetera.com">Colin DuPlantis</a>
 * @version $Id$
 * @since $Release$
 */
public class SecurityTypeSelectorEntry
        implements SpringSelectorEntry, InitializingBean
{
    /* (non-Javadoc)
     * @see org.marketcetera.ors.brokers.SpringSelectorEntry#routeToBroker(org.marketcetera.trade.Order)
     */
    @Override
    public boolean routeToBroker(Order inOrder)
    {
        SecurityType orderType = inOrder.getSecurityType();
        if((orderType != null) && orderType != SecurityType.Unknown) {
            if(targetType.equals(orderType)) {
                return true;
            }
        }
        return false;
    }
    /* (non-Javadoc)
     * @see org.marketcetera.ors.brokers.SpringSelectorEntry#setBroker(org.marketcetera.ors.brokers.
SpringBroker)
     */
    @Override
    public void setBroker(SpringBroker inBroker)
    {
        broker = inBroker;
    }
```

```
    /* (non-Javadoc)
     * @see org.marketcetera.ors.brokers.SpringSelectorEntry#getBroker()
     */
    @Override
    public SpringBroker getBroker()
    {
        return broker;
    }
    /* (non-Javadoc)
     * @see org.marketcetera.ors.brokers.SpringSelectorEntry#setSkipIfUnavailable(boolean)
     */
    @Override
    public void setSkipIfUnavailable(boolean inSkipIfUnavailable)
    {
        skipIfUnavailable = inSkipIfUnavailable;
    }
    /* (non-Javadoc)
     * @see org.marketcetera.ors.brokers.SpringSelectorEntry#getSkipIfUnavailable()
     */
    @Override
    public boolean getSkipIfUnavailable()
    {
        return skipIfUnavailable;
    }
    /* (non-Javadoc)
     * @see org.springframework.beans.factory.InitializingBean#afterPropertiesSet()
     */
    @Override
    public void afterPropertiesSet()
            throws Exception
    {
        Validate.notNull(targetType);
        Validate.notNull(broker);
    }
    /**
     * Sets the targetType value.
     *
     * @param inTargetType a <code>String</code> value
     */
    public void setTargetType(String inTargetType)
    {
        targetType = SecurityType.getInstanceForFIXValue(inTargetType);
    }
    /**
     * target broker
     */
    private SpringBroker broker;
    /**
     * indicates if this routing should be skipped if unavailable
     */
    private boolean skipIfUnavailable = true;
    /**
     * target security type
     */
    private SecurityType targetType;
}
```

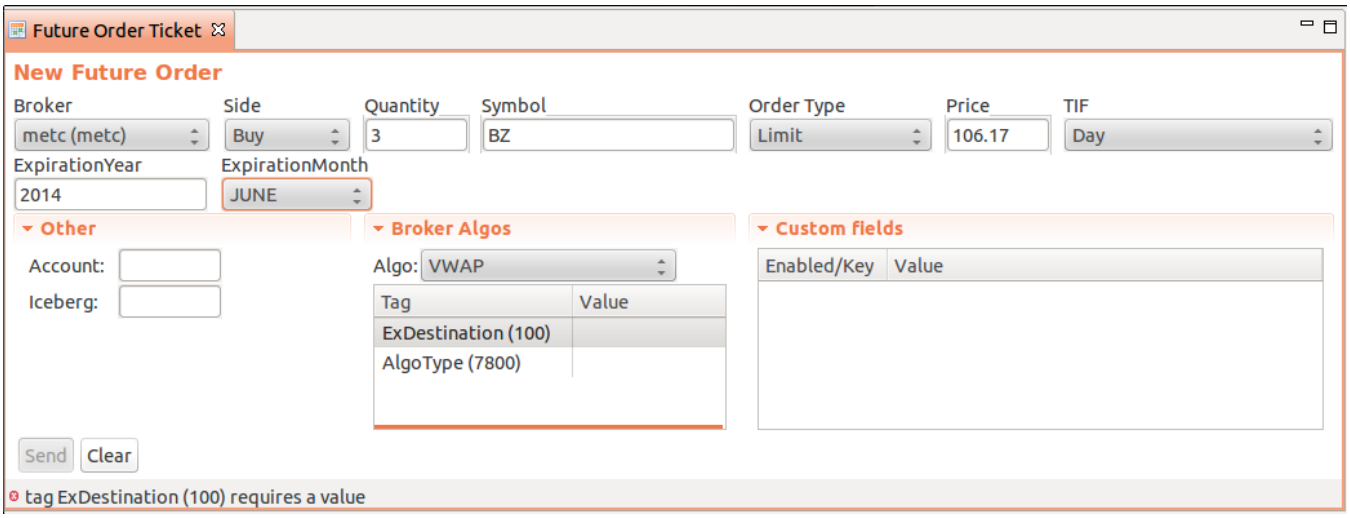
You could use this selector to send, for example, all equities to broker1 and all options to broker2. Simply create two instances of this bean in selector.xml:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema
/beans/spring-beans-3.2.xsd">
  <bean id="metc_selector" class="org.marketcetera.ors.brokers.SpringSelector">
    <!-- Routing entries -->
    <property name="entries">
      <list>
        <bean class="org.marketcetera.ors.brokers.SecurityTypeSelectorEntry">
          <property name="broker" ref="broker1"/> <!-- matches the id of a broker bean -->
          <property name="targetType" value="CS"/> <!-- 'CS' == Equities (Common Stock) -->
          <property name="skipIfUnavailable" value="true"/>
        </bean>
        <bean class="org.marketcetera.ors.brokers.SecurityTypeSelectorEntry">
          <property name="broker" ref="broker2"/> <!-- matches the id of a broker bean -->
          <property name="targetType" value="OPT"/>
          <property name="skipIfUnavailable" value="true"/>
        </bean>
      </list>
    </property>
    <!-- Default broker -->
    <property name="defaultBroker" ref="broker3"/>
  </bean>
</beans>
```

The skipIfUnavailable property regulates if the entry should be skipped if the broker is offline.

Broker Algorithms

Brokers will often offer algorithms that you can use when placing orders with them, like VWAP, TWAP, etc. The Marketcetera Order Routing System can make these broker algorithms available to Photon and Strategy Agent strategies, making it easier to use take advantage of the broker offerings.



In Photon, available broker algorithms are shown in the order ticket. The fields that the broker requires are listed. For strategies, each broker advertises its available algorithms along with the specifications necessary to take advantage of them. Take a look at this code snippet to see how to execute a broker algorithm from a strategy:

```
import java.math.BigDecimal;
import java.util.Set;
import org.marketcetera.algo.BrokerAlgo;
import org.marketcetera.algo.BrokerAlgoSpec;
import org.marketcetera.algo.BrokerAlgoTag;
import org.marketcetera.algo.BrokerAlgoTagSpec;
import org.marketcetera.client.ClientManager;
import org.marketcetera.client.brokers.BrokerStatus;
import org.marketcetera.event.BidEvent;
import org.marketcetera.strategy.java.Strategy;
import org.marketcetera.trade.Factory;
import org.marketcetera.trade.OrderSingle;
import com.google.common.collect.Sets;
/**
 * Demonstrates how to execute a broker algorithm order.
 *
 * @author <a href="mailto:colin@marketcetera.com">Colin DuPlantis</a>
 * @version $Id$
 * @since $Release$
 */
public class TestStrategy
        extends Strategy
{
    /* (non-Javadoc)
     * @see org.marketcetera.strategy.java.Strategy#onBid(org.marketcetera.event.BidEvent)
     */
    @Override
    public void onBid(BidEvent inBid)
    {
        // get an arbitrary broker, you'll want to do whatever you do to pick your broker
        BrokerStatus myBroker = ClientManager.getInstance().getBrokersStatus().getBrokers().get(0);
        Set<BrokerAlgoSpec> allAlgoSpecs = myBroker.getBrokerAlgos();
        // again, picking an arbitrary algo, you'll want to pick one in a more sensible way
        BrokerAlgoSpec myAlgoSpec = allAlgoSpecs.iterator().next();
        // for each tag in the algo, set its values
        Set<BrokerAlgoTag> myTags = Sets.newHashSet();
        for(BrokerAlgoTagSpec tagSpec : myAlgoSpec.getAlgoTagSpecs()) {
            // obviously, you can set tag specs even if they're not mandatory, this is just to show you
            //  that the specs can tell you whether they're required or not
            if(tagSpec.getIsMandatory()) {
                BrokerAlgoTag tag = new BrokerAlgoTag(tagSpec,
                                                      "this value");
                // this step is optional - this is what Photon does to make sure you've typed in a reasonable
value
                tagSpec.getValidator().validate(tag);
            }
        }
        BrokerAlgo myAlgo = new BrokerAlgo(myAlgoSpec,
                                           myTags);
        OrderSingle order = Factory.getInstance().createOrderSingle();
        order.setPrice(new BigDecimal("100.50"));
        /*
         * set other order fields here
         */
        order.setBrokerAlgo(myAlgo);
        send(order);
    }
}
```

Broker algorithms are specified in the broker configuration files for DARE. In the broker main.xml, define an entry for broker algorithms:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema
/beans/spring-beans-3.2.xsd">
  <import resource="qdescriptor.xml"/>
  <import resource="modifiers.xml"/>
  <import resource="routes.xml"/>
  <import resource="ps_modifiers.xml"/>
  <import resource="resp_modifiers.xml"/>
  <import resource="algos.xml"/>
  <bean id="metc_broker" class="org.marketcetera.ors.brokers.SpringBroker">
    <!-- The broker name (an arbitrary string). -->
    <property name="name" value="${metc.broker.name}"/>
    <!-- The broker ID (an arbitrary string, but a short one is best). -->
    <property name="id" value="${metc.broker.id}"/>
    <!-- Indicates if the broker requires a FIX Logout message on disconnect -->
    <property name="fixLogoutRequired" value="${metc.broker.fixlogoutrequired}"/>
    <!-- The QuickFIX/J session descriptor. -->
    <property name="descriptor" ref="metc_qdescriptor"/>
    <!-- The message modifiers. -->
    <!--
    <property name="modifiers" ref="metc_modifiers"/>
    -->
    <!-- The routing filter. -->
    <!--
    <property name="routes" ref="metc_routes"/>
    -->
    <!-- The pre-sending message modifiers. -->
    <property name="preSendModifiers" ref="metc_ps_modifiers"/>
    <!-- The response message modifiers. -->
    <!--
    <property name="responseModifiers" ref="metc_resp_modifiers"/>
    -->
    <!--  broker algos -->
    <property name="brokerAlgos">
      <set>
        <ref bean="vwapAlgoSpec"/>
        <ref bean="participationAlgoSpec"/>
      </set>
    </property>
  </bean>
</beans>
```

This configuration supports two broker algorithms: VWAP and Participation. The actual specifications are in the algos.xml file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema
/beans/spring-beans-3.2.xsd">
  <bean id="vwapAlgoSpec" class="org.marketcetera.algo.BrokerAlgoSpec">
    <property name="name" value="VWAP"/>
    <property name="algoTagSpecs">
      <set>
        <ref bean="exDestinationTagSpec"/>
        <ref bean="algoTypeTagSpec"/>
      </set>
    </property>
  </bean>
  <bean id="participationAlgoSpec" class="org.marketcetera.algo.BrokerAlgoSpec">
    <property name="name" value="Participation"/>
    <property name="algoTagSpecs">
      <set>
        <ref bean="maxParticipationTagSpec"/>
        <ref bean="algoModeTagSpec"/>
      </set>
    </property>
  </bean>
  <bean id="exDestinationTagSpec" class="org.marketcetera.algo.BrokerAlgoTagSpec">
    <property name="label" value="ExDestination (100)"/>
    <property name="description" value="Execution destination as defined by institution when order is entered"/>
    <property name="tag" value="100"/>
    <property name="isMandatory" value="true"/>
    <property name="pattern" value="^\w*$"/>
  </bean>
  <bean id="maxParticipationTagSpec" class="org.marketcetera.algo.BrokerAlgoTagSpec">
    <property name="label" value="MaxParticipation (7807)"/>
    <property name="description" value="Maximum participation"/>
    <property name="tag" value="7807"/>
    <property name="isMandatory" value="true"/>
    <property name="pattern" value="^\d*$"/>
  </bean>
  <bean id="algoTypeTagSpec" class="org.marketcetera.algo.BrokerAlgoTagSpec">
    <property name="label" value="AlgoType (7800)"/>
    <property name="description" value="Algorithm type as defined by the broker"/>
    <property name="tag" value="7800"/>
    <property name="isMandatory" value="true"/>
    <property name="pattern" value="^[X|Y|Z]$"/>
  </bean>
  <bean id="algoModeTagSpec" class="org.marketcetera.algo.BrokerAlgoTagSpec">
    <property name="label" value="AlgoMode (7803)"/>
    <property name="description" value="Algorithm mode as defined by the broker"/>
    <property name="tag" value="7803"/>
    <property name="isMandatory" value="true"/>
    <property name="pattern" value="^[2|5|8]$"/>
    <property name="options">
        <map>
          <entry key="Patient" value="2"/>
          <entry key="Normal" value="5"/>
          <entry key="Aggressive" value="8"/>
        </map>
    </property>
  </bean>
</beans>
```

A broker algorithm is established as a broker algorithm specification (BrokerAlgoSpec). The specification is the template for the algorithm. The BrokerAlgoSpec is composed of a number of BrokerAlgoTagSpec objects. The algorithm tag specifications include the following properties:

- label - human-readable short of the tag
- description - human-readable description of the purpose of the tag
- tag - integer value indicating which FIX field this tag specification corresponds to. Match this to the FIX specification you received from the broker.
- isMandatory - true/false indicating if this tag is required for this algorithm or optional
- pattern (optional) - if specified, regular expression used to validate the tag value
- options (optional) - if specified, presents the series of possible values this tag may hold

Define a BrokerAlgoSpec for each algorithm the broker provides that you want to make available to clients. For sell-side installations, you can define your own algorithms and present them in this way.

Order History

DARE provides access to order history via web services. In Photon, order history is presented automatically in the various FIX Message Views. From Strategies, order history is available via the org.marketcetera.client.Client interface.

```
import java.util.Date;
import org.marketcetera.client.ClientManager;
import org.marketcetera.strategy.java.Strategy;
import org.marketcetera.trade.ReportBase;
/**
 * Demonstrates the retrieval of order history.
 *
 * @author <a href="mailto:colin@marketcetera.com">Colin DuPlantis</a>
 * @version $Id$
 * @since $Release$
 */
public class OrderHistory
        extends Strategy
{
    /* (non-Javadoc)
     * @see org.marketcetera.strategy.java.Strategy#onStart()
     */
    @Override
    public void onStart()
    {
        // get execution reports received in the last hour
        ReportBase[] lastHoursReports = ClientManager.getInstance().getReportsSince(new Date(System.
currentTimeMillis()-1000*60*60));
    }
}
```

The reports that you receive are limited to the ones that the strategy user is entitled to see.

Positions

DARE provides access to positions by instrument via web services. In Photon, positions are presented automatically in the Positions View. From strategies, positions are available via the strategy framework.

```
import java.math.BigDecimal;
import java.util.Date;
import java.util.Map;
import java.util.Map.Entry;
import org.marketcetera.core.position.PositionKey;
import org.marketcetera.strategy.java.Strategy;
import org.marketcetera.trade.Currency;
/**
 * Demonstrates the retrieval of positions.
 *
 * @author <a href="mailto:colin@marketcetera.com">Colin DuPlantis</a>
 * @version $Id$
 * @since $Release$
 */
public class Positions
        extends Strategy
{
    /* (non-Javadoc)
     * @see org.marketcetera.strategy.java.Strategy#onStart()
     */
    @Override
    public void onStart()
    {
        // get FX position as of now
        Map<PositionKey<Currency>,BigDecimal> positions = getAllCurrencyPositionsAsOf(new Date());
        for(Entry<PositionKey<Currency>,BigDecimal> entry : positions.entrySet()) {
            System.out.println("Position of " + entry.getKey().getInstrument() + " for " + entry.getKey().
getTraderId() + " in account " + entry.getKey().getAccount() + ": " + entry.getValue());
        }
        BigDecimal singlePosition = getCurrencyPositionAsOf(new Date(),"BTC/USD");
    }
}
```

Positions are available for other asset classes, as well. Positions are limited to the ones that the strategy user is entitled to see.

Symbol Resolution

DARE provides configurable symbol resolution service. Symbol resolution resolves a basic string symbol to an instrument, which is Marketcetera's internal representation of a tradable instrument. Configuring DARE to resolve instruments can simplify integration with external components like legacy systems and market data providers.

By default, DARE is configured using the PatternSymbolResolver, which tries to match instruments to Futures in the form SYMBOL-YYYYMM, Options in OSI format, Currencies with symbols that contain the '/' character, as in BTC/USD, and, then returns an Equity. To define your own symbol resolution scheme, modify ors/conf/server.xml, in particular, the symbolResolverServices bean:

```
<property name="symbolResolverServices">
  <bean class="org.marketcetera.ors.symbol.IterativeSymbolResolver">
    <property name="symbolResolvers">
      <list>
        <bean class="org.marketcetera.ors.symbol.PatternSymbolResolver"/>
      </list>
    </property>
  </bean>
</property>
```

In the default configuration, there is a single SymbolResolver that always returns an Instrument. You can define your own resolvers creating one or more classes that implement org.marketcetera.ors.symbol.SymbolResolver and adding them to the symbolResolvers list. The order is key as they symbol resolvers are invoked in the given order. The first resolver that returns non-null wins.

Authentication and Authorization

Users

In Marketcetera, there are two classes of users, superuser and non-superuser. Superusers can see all orders and positions, non-superusers can see their orders and positions only. Each client connection (Photon, Strategy Agent, Order Loader, etc.) is owned by a particular user. All actions undertaken by that client are performed by that user.

ORSAdmin

To provision users, use the command-line user admin tool.

```
$ ors/bin/orsadmin.sh
Missing required options: u, [--listUsers List all users, --changeSuperuser Change Superuser Flag, --addUser
Add a new user, --deleteUser Delete User, --changePassword Change Password, --restoreUser Restore User]
usage: orsadmin -u <login> -p <password> --listUsers [-a y|n]
               orsadmin -u <login> -p <password> --addUser -n <user name>  -w <user password>  [-s
               y|n]
               orsadmin -u <login> -p <password> --deleteUser -n <user name>
               orsadmin -u <login> -p <password> --restoreUser -n <user name>
               orsadmin -u <login> -p <password> --changePassword -w <user password>  [-n <user
               name> ]
               orsadmin -u <login> -p <password> --changeSuperuser -n <user name>  -s y|n
Options:
    -a,--active <user active flag>         Active flag of user being operated on
       --addUser                           Add a new user
       --changePassword                    Change Password
       --changeSuperuser                   Change Superuser Flag
       --deleteUser                        Delete User
       --listUsers                         List all users
    -n,--username <user name>              Name of user being operated on
    -p <password>                          password of the user running the CLI
       --restoreUser                       Restore User
    -s,--superuser <user superuser flag>   Superuser flag of user being operated on
    -u <login>                             user name of the user running the CLI
    -w,--password <user password>          Password of user being operated on
```

Brokers by User

Access to brokers can be controller per user by specifying a user whitelist or blacklist for that broker. If a whitelist is specified for a broker, a user must exist in the whitelist to use the broker. Any users not included in the whitelist cannot use that broker. If a blacklist is specified for a broker, a user cannot exist in the blacklist to use that broker. Any users included in the blacklist cannot use that broker. If both a whitelist and a blacklist are specified for a broker, the user must both not exist in the blacklist and exist in the whitelist.

To specify a whitelist or blacklist for a broker, modify the main.xml for that broker:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema
/beans/spring-beans-3.2.xsd">
  <import resource="qdescriptor.xml"/>
  <import resource="modifiers.xml"/>
  <import resource="routes.xml"/>
  <import resource="ps_modifiers.xml"/>
  <import resource="resp_modifiers.xml"/>
  <import resource="algos.xml"/>
  <bean id="metc_broker" class="org.marketcetera.ors.brokers.SpringBroker">
    <!-- The broker name (an arbitrary string). -->
    <property name="name" value="${metc.broker.name}"/>
    <!-- The broker ID (an arbitrary string, but a short one is best). -->
    <property name="id" value="${metc.broker.id}"/>
    <!-- Indicates if the broker requires a FIX Logout message on disconnect -->
    <property name="fixLogoutRequired" value="${metc.broker.fixlogoutrequired}"/>
    <!-- The QuickFIX/J session descriptor. -->
    <property name="descriptor" ref="metc_qdescriptor"/>
    <property name="userWhitelist">
      <list>
        <value>user1</value>
        <value>user2</value>
      </list>
    </property>
    <property name="userBlacklist">
      <list>
        <value>user3</value>
        <value>user4</value>
      </list>
    </property>
  </bean>
</beans>
```

Order Filters

Outgoing orders can be filtered by DARE, preventing orders from going out as specified by the user configuration. By default, no orders are filtered. You can specify your own filters by creating one or more classes that extend org.marketcetera.ors.filters.OrderLimitFilter. There are several predefined filters. One in particular, the AssetClassFilter, restricts given users from using asset classes as desired using whitelists and blacklists by security type. To enable order filters, modify ors/conf/server.xml and make sure the following bean exists:

```
<property name="allowedOrders">
  <list>
    <ref bean="metc_allowed_orders"/>
    <ref bean="metc_restricted_users"/>
   </list>
</property>
```

Here are the definitions of the beans:

```
<!-- indicates users allowed or disallowed to send orders of a particular asset class -->
<bean id="metc_restricted_users" class="org.marketcetera.ors.filters.AssetClassFilter">
  <property name="userlists">
      <map>
        <entry key="CS"> <!-- restrictions for 'CS', i.e. equities -->
          <bean class="org.marketcetera.ors.filters.UserList">
              <!-- whitelist explicitly allows users - if specified, anyone not on the whitelist is not allowed
-->
              <property name="whitelist">
                  <set>
                      <value>bob</value>
                      <value>alice</value>
                  </set>
              </property>
              <!-- blacklist explicitly disallows users - if specified, anyone not on the blacklist is allowed
```

```
-->
              <property name="blacklist">
                  <set>
                      <value>evilbob</value>
                      <value>evilalice</value>
                  </set>
              </property>
          </bean>
        </entry>
        <entry key="FUT"> <!-- restrictions for 'FUT', i.e. futures -->
          <bean class="org.marketcetera.ors.filters.UserList">
              <!-- whitelist explicitly allows users - if specified, anyone not on the whitelist is not allowed
-->
              <property name="whitelist">
                  <set>
                      <value>bob</value>
                      <value>alice</value>
                  </set>
              </property>
              <!-- blacklist explicitly disallows users - if specified, anyone not on the blacklist is allowed
-->
              <property name="blacklist">
                  <set>
                      <value>evilbob</value>
                      <value>evilalice</value>
                  </set>
              </property>
          </bean>
        </entry>
        <!-- asset classes not included, like 'OPT' and 'CUR' have no restrictions -->
      </map>
  </property>
</bean>
<bean id="metc_allowed_orders" class="org.marketcetera.ors.filters.OrderLimitFilter">
  <!--
    - If true, orders with type 'MARKET' are rejected. If omitted, false
    - is assumed.
    -->
  <property name="disallowMarketOrders" value="false"/>
  <!--
    - Minimum order price; it is checked only if a price is set.
    - If omitted, no minimum price check is performed.
    -->
  <property name="minPrice" value="0"/>
  <!--
    - Maximum order price; it is checked only if a price is set.
    - If omitted, no maximum price check is performed.
    -->
  <property name="maxPrice" value="100"/>
  <!--
    - Maximum order quantity; it is checked only if a quantity is set.
    - If omitted, no maximum quantity check is performed.
    -->
  <property name="maxQuantityPerOrder" value="10000"/>
  <!--
    - Maximum order notional (product of price and quantity); it is
    - checked only if both a price and a quantity are set. If omitted,
    - no maximum notional check is performed.
    -->
  <property name="maxNotionalPerOrder" value="100000"/>
</bean>
```

You can add as many order filters as you like. Outgoing orders must pass through all active order filters.