API, Lifecycle, Configuration

ALGORITHM DEVELOPER' GUIDE

Execution Server (Ember)

CONTENTS

ALGORITHM DEVELOPER'S GUIDE

How to develop, test, and deploy algorithms in Deltix Execution Server 5.X (Ember).

OVERVIEW

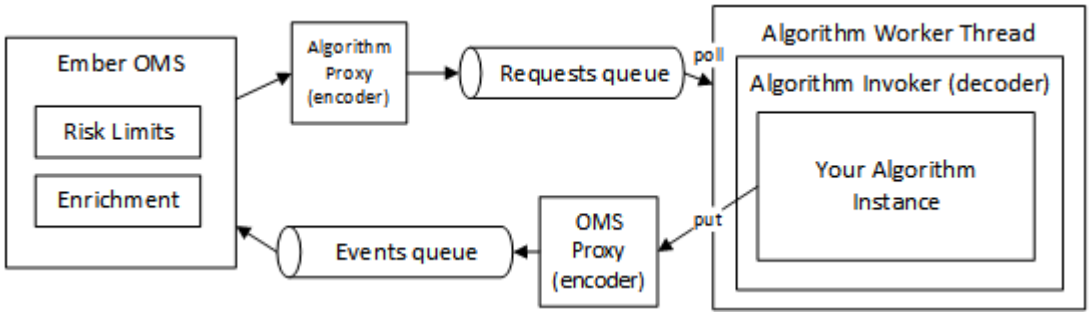
Execution Server was designed as a platform for custom trading algorithms. Each algorithm programmed to handle user orders according to specific logic under certain market conditions.

Algorithms in Deltix use Reactor design pattern shared with other Ember components. Each instance of execution algorithm has dedicated inbound request queue and processing thread. This system-managed thread runs in a loop that query various non-blocking event sources[1]:

- Market Data
- Trading Requests (for orders placed to this algorithm)
- Trading Events (for orders placed by this algorithm)
- Timers

In addition to trading orders and events, execution algorithm polls market data and timers. Each source will be described later in this section. State maintained by algorithm instance thread is not shared with any other threads in the system.

The following diagram shows how algorithm interacts with the rest of the system:



You may deploy multiple instances of each execution algorithm.

- Each algorithm instance is single-threaded and lock-free.
- Each algorithm receives Inbound [parent] Orders and emits Outbound [child] Orders.
- Each algorithm receives Inbound Events about child orders and emits Outbound Events regarding parent orders
- State of each algorithm is determined by combined state of its parent and child orders. Persistent state of each algorithm should not depend on market data or other external events[2].

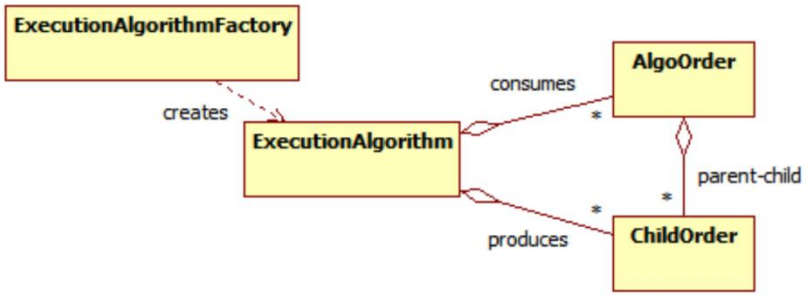
TUTORIAL: IMPLEMENTATION OF SIMPLE TWAP ALGORITHM

This section describes how to develop execution algorithm using TWAP (Time Weighted Average Price) order as an example. To keep code sections brief, only critical pieces are shown. Deltix provides full source code of this example with ES package.

Each execution Algorithm in Ember uses four classes:

[1] Custom event sources also supported by overriding doFirst()/doLast() callbacks.

[2] External events can of course influence order execution, but state of each order should be fully deterministic from set of trading events that were recorded for this order.



- ExecutionAlgorithmFactory is responsible for defining static parameters of algorithm instance and creating each instance. In most cases this is a boilerplate code.
- ExecutionAlgorithm defines algorithm execution logic. This is the main algorithm class handling market data, orders, trading events, and timers.
- AlgoOrder defines state of each parent order received and processed by the algorithm. Each execution algorithm typically creates its own subclass, where it keeps algorithms specific order state data.
- ChildOrder defines state of child orders emitted by execution algorithm. Usually these are Direct Market Access (DMA) orders, but in more complex cases these could be orders that are routed to another execution venue or algorithm.

ALGO ORDER

In this example, we will illustrate a simplified version of TWAP that takes three parameters:

- Start Time - when order should start executing
- End Time - when order should complete executing entire order quantity
- Drip Percentage - defines how to slice entire order size (e.g. when set to 5% entire order should execute in 5%). This parameter also determines interval between child orders.

To start, create a class TWAPOrder that extends standard class AlgoOrder and defines parameters above:

```
class TWAPOrder extends AlgoOrder {
  long startTime = TypeConstants.TIMESTAMP_NULL;
  long endTime = TypeConstants.TIMESTAMP_NULL;
  double dripPercentage;
  …
}
```

Inheritance from standard class AlgoOrder provides some basic runtime state. For example, list of child orders and helper methods to retrieve information like currently executed quantity, etc.

PARAMETER PARSING

ES API is heavily influenced by FIX protocol. Custom order parameters identified by numeric key. Let's define these keys and implement parsing code in our TWAPOrder class:

```
static final int START_TIME_ATTRIBUTE_KEY = 6021;
static final int END_TIME_ATTRIBUTE_KEY = 6022;
static final int DRIP_PERCENTAGE_ATTRIBUTE_KEY = 6023;

void copyExtraAttributes(OrderEntryRequest request) {
  if (request.hasAttributes()) {
    ObjectList<CustomAttribute> attributes = request.getAttributes();
    for (int i = 0, size = attributes.size(); i < size; i++) {
      CustomAttribute attribute = attributes.get(i);
      switch (attribute.getKey()) {
        case START_TIME_ATTRIBUTE_KEY:
```

```
          startTime = TimestampParser.parseTimestamp(attribute.getValue());
          break;
        case END_TIME_ATTRIBUTE_KEY:
          endTime= TimestampParser.parseTimestamp(attribute.getValue());
          break;
        case DRIP_PERCENTAGE_ATTRIBUTE_KEY:
          dripPercentage = CharSequenceParser.parseDouble(attribute.getValue());
          break;
      }
    }
  }
}
```

We will come back to Algo Order in later in this chapter when we go deeper into TWAP logic.

CHILD ORDER

We are going to use the standard class ChildOrder to represent the state of child orders emitted by TWAP execution algorithm. In more complex cases you can use subclass with some additional state data.

In Execution Server 5.2 each ChildOrder may belong to at most one parent AlgoOrder. Parent AlgoOrders may emit zero or more child orders. In special cases execution algorithm may send a child order that doesn't belong to any specific parent order (for example, order that closes combined position).

EXECUTION ALGORITHM

Now let's define class for TWAP execution Algorithm itself.

```
class TWAPAlgorithm extends AbstractAlgorithm<TWAPOrder> {
  TWAPAlgorithm(AlgorithmContext context, OrdersCacheSettings cacheSettings) {
    super(name, context, TWAPOrder::new, ChildOrder::new, cacheSettings);
  }
  …
}
```

In this example we extended standard class AbstractAlgorithm that provides boilerplate code to process trading events. The next three sections describe algorithm constructor parameters.

ALGORITHM CONTEXT

Algorithm Context contains configured algorithm name and provides various container services to algorithm:

- Ember service (e.g. to emit trading events)
- Timers (to define periodic jobs)
- Market Data
- Pricing Service (e.g. to lookup prices or security metadata about a contract)
- TimeBase API (e.g. to read market data)
- Counters (or unique sequence generators)

Each instance of execution algorithm has a unique name. This name identifies algorithm among other order destinations defined in the system. The name is typically defined by deployment configuration file (will be described later) and is passed to the algorithm with AlgorithmContext. It is possible to deploy multiple instances of the same algorithm running with different parameters.

ORDERS CACHE SETTINGS

`OrdersCacheSettings` parameter contains settings for caches used by algorithm. AbstractAlgorithm uses two separate caches for inbound and outbound orders. In this example these are represented by TWAPOrder and ChildOrder classes respectively. Each orders cache has two areas: unbound cache of active orders and fixed-size cache of recent inactive orders. Each cache uses object pool to avoid memory allocations (and associated GC) when processing new orders.

Order object instances go through this lifecycle:

a. When new order request comes in, algorithm takes Order instance from object pool and places it into active orders cache. It stays there while algorithm processes order events.

b. Once the order reaches the final state (e.g. CANCELLED or COMPLETELY_FILLED) it becomes inactive and algorithm moves it into Inactive Orders cache area. Inactive orders stay there for some time and help processing out-of-bands requests and events.

c. Eventually inactive order is pushed out of cache and recycled back to object pool.

Avoid keeping references to inactive orders in your algorithm state. Once recycled, these order objects will lose all stored information. Eventually recycled orders will contain information about different orders.

ADDITIONAL ALGORITHM PARAMETERS

Algorithm instance may be deployed with custom configuration parameters. See AlgorithmFactory section for more information.

ALGORITHM FACTORY

In the simplest form Algorithm Factory is a boilerplate:

```
public class TWAPAlgorithmFactory extends AbstractAlgorithmFactory<TWAPAlgorithm> {
   @Override
   public TWAPAlgorithm create(AlgorithmContext context) {
           return new TWAPAlgorithm(context, getCacheSettings());
   }
}
```

This factory class provides a way to specify deployment parameters. For example, if we want to define minimal order duration to be allowed by algorithm:

```
public class TwapAlgorithmFactory extends AbstractAlgorithmFactory<TwapAlgorithm> {

    private Duration minimumOrderDuration;

    public void setMinimumOrderDuration(Duration minimumOrderDuration) {
        this.minimumOrderDuration = minimumOrderDuration;
    }

    @Override
    public TWAPAlgorithm create(AlgorithmContext context) {
        return new TWAPAlgorithm(context, minimumOrderDuration, getCacheSettings());
    }
}
```

How to provide specific configuration parameters at deployment time will be described later in this document.

ALGORITHM DEPLOYMENT

To deploy an algorithm:

- Add algorithm library to Execution Server runtime CLASSPATH. One way of doing it is placing a library with compiled algorithm and supporting it classes under lib/custom folder of Ember installation. All JAR files found in this folder are automatically added to Java CLASSPATH by Ember startup scripts.
- Define algorithm in `ember.conf` configuration file[3] as shown below:

```
algorithms { // you may already have this section in your ember.conf
    TWAP : ${template.algorithm.default} {
```

[3] Ember configuration files use HOCON/JSON format described here.

```
        factory = "deltix.ember.service.algorithm.samples.twap.TwapAlgorithmFactory"
    }
}
```

The above configuration tells Ember that we have new algorithm instance "TWAP" and tells which factory class can create an instance of this algorithm. If we want to define other settings, we can expand this configuration:

```
algorithms { // you may already have this section in your ember.conf
    TWAP : ${template.algorithm.default} {
        factory = "deltix.ember.service.algorithm.samples.twap.TwapAlgorithmFactory"
        settings {
            minimumOrderDuration = 5 seconds // custom parameter
        }
    }
}
```

You can define custom deployment parameters for standard settings in the same manner. Use `ember-default.conf` as a reference (`/template/algorithm/default` node).

Custom Docker image

When Ember is used in Deltix CryptoCortex product there is a special docker container called "emberpack" that contains ember itself and library of standard connectors. You can extend this library with your custom algorithm:

Our sample provides an example of docker image generation. Dockerfile sample:

```
FROM https://nexus.deltixhub.com/deltix.docker/anvil/deltix-ember-pack:1.6.3
COPY custom /opt/deltix/ember/lib/custom
```

Gradle project shows how to build docker image. The script relies on the following environment variables that define credentials to access Deltix docker repository:

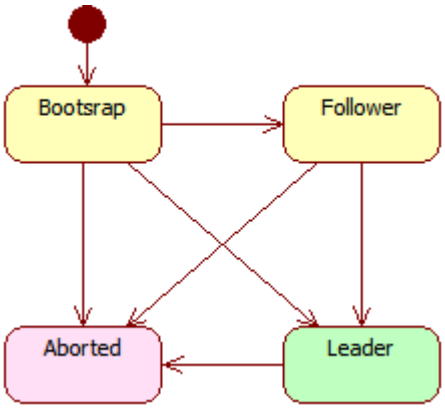
```
export DOCKER_USER=***************
export DOCKER_PASS=***************
export DOCKER_REPOSITORY=registry.deltixhub.com
```

Once you have these defined, you can run:

```
./gradlew dockerBuildImage -PdebugDockerBuild=true
```

ALGORITHM LIFECYCLE

Each Algorithm follows simple life cycle shown below:



- **Bootstrap** state: When server is started it restores state by replaying journal of historical trading transactions.
- **Follower** state: when High Availability mode is configured a server instance can act as a Follower In this state algorithms are on "stand by". They receive carbon copy of trading requests and events emitted on Leader node but can't emit their own messages. It is similar to Bootstrap state but received messages happen in real time. Trading is not allowed in this state as well.
- **Leader**: normal run time state in which algorithm receives real time messages and sends trade requests and events.
- **Aborted**: algorithm failed for whatever reason.

Note: Algorithm can trade only in Leader state. In Bootstrap and Follower state algorithm is responsible for keeping accurate trading state (state of parent and child orders).

The following table shows how trading messages are routed in different modes:

Mode	Inbound Order Requests	Inbound Order Events	Outbound Order Requests	Outbound Order Events
Bootstrap	OMS→Algorithm	OMS→Algorithm	OMS→Algorithm	OMS→Algorithm
Follower	OMS→Algorithm	OMS→Algorithm	OMS→Algorithm	OMS→Algorithm
Leader	OMS→Algorithm	OMS←Algorithm	OMS←Algorithm	OMS←Algorithm

Replaying historical trading messages back to algorithm can be used to restore trading state. For example, trades received during bootstrap can be used to restore your track of current position in each instrument.

Special case must be taken when processing time-sensitive order fields (such as expiration time, for example). We suggest suppressing these checks in Bootstrap mode.

The following API methods available in AbstractAlgorithm can be used to check current mode:

```
 /** @return true if current node (execution server) is a leader in a cluster or working live
standalone */
protected final boolean isLeader();

/** @return true if current node is restoring trading state after restart (processing journal messages)
*/
protected final boolean isBootstrap();
```

You could also override the following callback to react to mode changes:

```
/** Informs that cluster-related state of this node changed.
 * Also used to mark transition from initial BOOTSTRAP to LEADER
 * state for single-node installations */
@Override
public void onNodeStatusEvent(NodeStatusEvent event) {
   super.onNodeStatusEvent (event);
   final NodeStatus next = event.getNodeStatus();
   final NodeStatus previous = nodeStatus;
```

```
    LOGGER.info("Algorithm %s switched from %s to %s role")
            .with(name)
            .with(previous)
            .with(next);
}
```

SHUTDOWN BEHAVIOR

When system detects a shutdown request (e.g. SIGINT) it notifies all algorithms using onShutdownRequest callback of ShutdownRequestHandler interface that algorithms implement. Default handler from AbstractAlgorithm (shown below) simply acknowledges shutdown immediately using ShutdownResponse message:

```
@Override
public void onShutdownRequest(final ShutdownRequest request) {
    final MutableShutdownResponse response = new MutableShutdownResponse();
    response.setServiceId(id);

    context.getShutdownResponseHandler().onShutdownResponse(response);
}
```

In some cases algorithm can perform some additional actions before acknowledging shutdown. For example, active orders can be cancelled. Try to avoid performing any time-consuming actions at shutdown. Each algorithm has about 15 seconds to respond to shutdown request before system proceeds with process termination.

> Algorithms are not allowed to send any messages to OMS after they had acknowledged shutdown.

Once all services have acknowledged their shutdown system ember process terminates. Default acknowledgement timeout is 15 seconds.

TRADING HALT

When Trading Halt is activated it acts as a kill switch - any attempt to send new order or replace an existing order will be rejected by OMS. Cancellations are still allowed.

Trading can be halted for the following reasons:

- System operator manually halted trading (emergency kill switch in ES Monitor). This action is also available via Server API. See KillSwitchSample.
- Trading can be halted due to risk limit violation. For example, the number of rejected orders in the last second exceeds pre-configured limit.
- Trading can be halted due to some kind of abnormal situation. For example, by default Ember halts trading when connection to TimeBase is lost.

Algorithm implements KillSwitchHandler interface and is notified about halt/resume events using the following callback:

```
@Override
public void onKillSwitchEvent(KillSwitchEvent event) {
  if (isLeader()) {
    .. your action here
  }
}
```

EVENT HANDLERS

INBOUND ORDERS

There are four trading requests that normally each execution algorithm should handle:

- New order submission
- Cancellation of active order
- Modification of active order
- Order Status request

OrderRequestHandler interface implemented by agorithms has corresponding callbacks for these requests:

```
public interface OrderRequestHandler {
  void onNewOrderRequest(NewOrderRequest request);
  void onCancelOrderRequest(CancelOrderRequest request);
  void onReplaceOrderRequest(ReplaceOrderRequest request);
  void onOrderStatusRequest(OrderStatusRequest request);
}
```

To simplify algorithm development Ember provides boilerplate handler of these events in the class AbstactAlgorithm. This boilerplate class handles the following aspects:

- Maintains state of active orders based on their lifecycle events
- Provides boilerplate code to handle order workflows between parent orders and their child orders.

SECURITY MASTER

Deltix keeps information about all instruments in special TimeBase stream called Security Metadata (or "securities" for short). When the system starts each algorithm is provided with a snapshot of all known instruments via AlgorithmContext. While algorithm is running security metadata updates are visible via InstrumentUpdateHandler interface available to each algorithm. Out-of-the-box algorithm base class, AbstractAlgorithm, already keeps track of current instruments and provides SecurityMetadataProvider lookup interface.

MARKET DATA

Execution Server provides market data via TimeBase API.

Subscription - Streams

Algorithm configuration (ember.conf) defines which TimeBase stream each algorithm is using for market data.

```
algorithms {
  TWAP ${template.algorithm.default} {
    …
    subscription {
      streams = ["CMEFeed", "ICEFeed"]
    }
  }
...
```

By default algorithm subscribes to all data in selected streams, but it is possible to limit symbols and types of messages received from the stream with these attributes:

```
algorithms {
  TWAP ${template.algorithm.default} {
    …
```

```
    subscription {
      streams = ["CMEFeed", "ICEFeed"]
      symbols = ["IFLL ESU0", "IFLL L M2", "IFLL ERM2"]
      types = ["deltix.timebase.api.messages.TradeMessage"]
    }
  }
  ...
```

Subscription - Topics

Starting from QuantServer 5.2 Algorithms can receive market data from TimeBase Topics:

```
algorithms {
  TWAP ${template.algorithm.default} {
    …
    subscription {
      topic = "ticks"
    }
  }
…
```

The following table compares TimeBase Streams and Topics:

Feature	TimeBase Streams	TimeBase Topics
Maximum throughput	1.5 million messages/second	9 million messages/second
Latency - median	46 microseconds	0.35 microseconds
Latency - 99% percentile	62 microseconds	1.40 microseconds
Durability	Messages can be persisted using Durable Streams	Persistence is not supported as of QuantServer 5.2[4]
Subscription control	Supported via Entity and Type Subscription Controller	Not supported. Algorithms receive all data
Number of sources	TimeBase supports thousands of streams	Topics are very efficient but require a lot of resources. It is not recommended to have more than 10 topics per system.

Programmatic subscription

AlgorithmContext provides access to subscription via MarketSubscription interface. For example:

```
context.getMarketSubscription().subscribe("ESM9", InstrumentType.FUTURE);
```

[4] Deltix provides special Aggregator process to store messages of each given Topic into Durable TimeBase stream (disk can keep up with topic data rate).

Market Data Handler

Ember runtime (main algorithm processing thread) polls underlying TimeBase cursors and dispatches available market messages to algorithm via MarketMessageHandler interface:

```
public interface MarketMessageHandler {
    void onMarketMessage(InstrumentMessage message);
}
```

Algorithm can also subscribe to custom input channel. See Advanced Features section below.

TIMERS

Timers allow to schedule some piece of work to be done at a certain time or with some periodicity. Ember provides *allocation-free* timer API for algorithm developers. Timer jobs are executed in the main processing thread of each algorithm. Each timer job is defined using:

- Callback function that performs scheduled work
- Optional parameter passed to this callback
- Next execution time (scheduled time)

For example, here is a callback function that cancels order:

```
long onOrderTimeout(long now, Cookie cookie) {
   cancelOrder((Order) cookie);
   return DO_NOT_RESCHEDULE;
}
```

The first callback parameter provides current time (which may be slightly past scheduled time). The second parameter provides timer cookie parameter specified during job creation.

Use timer service to schedule a timer job in algorithm as shown below:

```
TimerJob cancelJob = getTimer().schedule(clock.time()+5000, this::onOrderTimeout, cookie);
```

Here we schedule a callback to be called 5 seconds from the current clock time. You can use returned TimerJob object if you want to cancel the job. For example, if the order is cancelled it may be a good idea to cancel order timeout timer:

```
cancelJob.cancel();
```

Finally, if you want to perform periodic tasks simply return next execution time from the callback. Otherwise return negative value and the timer will not be rescheduled.

See Appendix A "Implementing order timeout" for a complete end-to-end example of timer code.

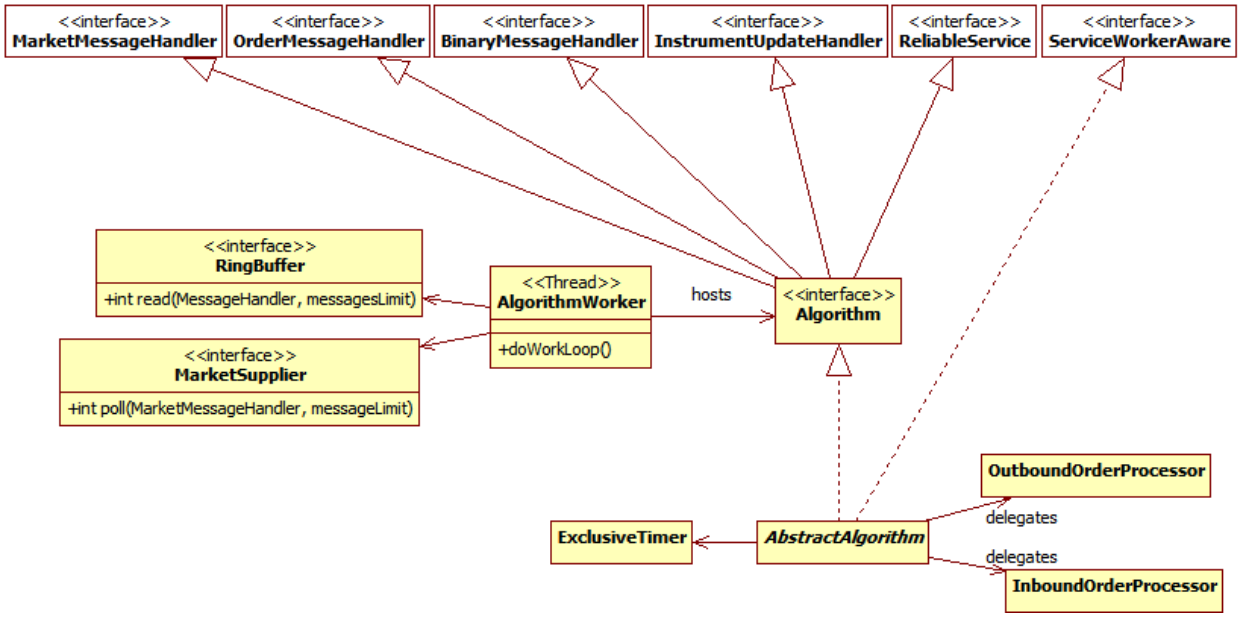
INBOUND / OUTBOUND ORDERS

Inbound orders are requests directed to algorithm logic. For example, a request to execute new client order or cancel previously transmitted order. Algorithms are fully responsible for the lifecycle of these orders. Execution Algorithm must provide some kind of acknowledgement upon receiving each inbound order. When order processing is not expected to take a lot of time, acknowledgement event can be skipped, providing that algorithm can guarantee some other feedback event. For example, algorithm that executes Immediate Or Cancel (IOC) orders may skip order acknowledgement and respond with FILL, CANCEL, or REJECT event depending on the market.

Ember Risk module has Order Acknowledgment timeout checker. This component will halt trading if execution algorithm does not acknowledge an order in certain time frame (usually a few seconds). All inbound requests require corresponding ACK / NACK from execution algorithm, not just child order submission. Execution Algorithm API provides helper methods that simplify this task.

SUMMARY

We can finish this chapter with more detailed design diagram of Algorithm and supporting classes:



For more information refer to TimedOrderAlgorithm or TWAPAlgorithm samples.

ADVANCED FEATURES

MEMORY ALLOCATION

Algorithms should not allocate new objects in working cycle. Memory allocations lead to periodic Garbage Collection which in turn causes unpredictable pauses in execution of entire Execution Server. Avoid per-order and similar memory allocations at all costs. Allocate all objects during algorithm initialization, use memory pools (e.g. deltix.anvil.util.ObjectPool).

EAGER INITIALIZATION

While "lazy" initialization is widely used practice in modern software, we recommend initializing any resource that algorithm may need during startup and initialization.

ALGORITHM CPU AFFINITY

Each algorithm instance that require the lowest message processing latency should be pinned to isolated CPU core. The following ember.conf fragment assigns CPU core 10 to the event processing thread of algorithm "TWAP":

```
affinity {
    algorithm-TWAP =[10]
    ...
}
```

See User's Guide and Linux Tuning Guide for more information.

CUSTOM INPUTS

In addition to main market data subscription algorithm may receive data from *custom* TimeBase streams. For example, you may transmit some control or configuration messages to your algorithm.

As you may remember Ember uses non-blocking I/O and custom input channels are not an exception. Algorithm framework provides you doFirst() / doLast() callbacks for this purpose. Algorithm's worker thread keeps calling these from the main loop. Algorithm API provides you the ability to poll custom input source(s) and call user-provided callback for each message received from each custom source.

```
// callback that will process custom messages
Consumer<InstrumentMessage> messageConsumer = this::onCustomInputMessage;
```

```
// here we create a poller for TimeBase stream identified by given stream key
customSourcePoller = context.createInputPoller(customStreamKey, messageConsumer);

/** Callback that will process custom messages */
private void onCustomInputMessage(InstrumentMessage message) {
   // This is just a sample. In real life avoid logging high rate messages
   LOGGER.info("Custom message received %s %s")
    .withTimestamp(message.getTimeStampMs()).with(message.getSymbol());
}
```

Now the following method override does the main work of polling your source and dispatching messages to the callback we defined above:

```
@Override
public int doLast(int workDone) {
  int newWorkDone = super.doLast(workDone); // don't forget this - otherwise things like timers won't work

   // check if input stream has any new messages
   newWorkDone += customSourcePoller.poll(32);

   return newWorkDone;
}
```

And finally, don't forget to close your poller at algorithm shutdown to free up Timebase resources:

```
@Override
public void close() {
   CloseHelper.close(customSourcePoller);
}
```

TRACKING TRADING CONNECTOR STATUS

```
@Override
public void onSessionStatusEvent(final SessionStatusEvent event) {
    if (isLeader()) {
        logger.info("Connector %s switched to %s state")
                .withAlphanumeric(event.getSourceId())
                .with(event.getStatus());

    }
}
```

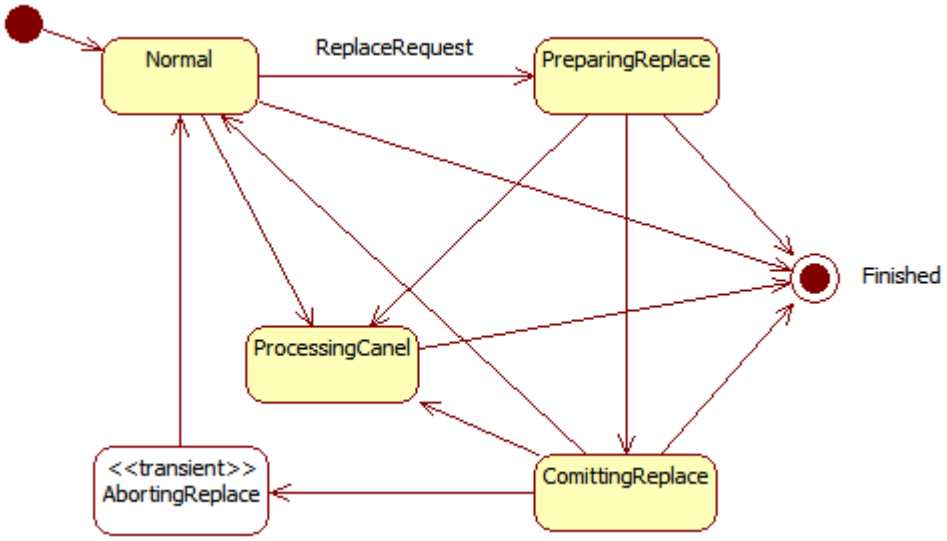
ORDER SLICING ALGORITHMS

Execution Algorithms that slice a large order into small portions (clips) for optimal execution are very common. Deltix developed a set of helper classes to simplify development of such algorithms. General idea is: let developer focus on specific business logic of execution algorithm, while standard boilerplate code will be done by helper classes. For example, when developing ICEBERG order a specific implementation should tell us how exactly to cut order into smaller pieces (using display quantity") and helper classes will do the rest:

- Propagate executions of child orders (slices) as execution of parent order
- Automatically handle modification (Cancel-Replace workflow) of parent order
- Automatically handle cancellation workflow of parent order

To use Slicing Algorithm, extend your implementation from abstract class SlicingAlgorithm. This helper class implements state machine driven by trading messages (e.g. OrderReplaceRequest or OrderRejectEvent).

Slicing Algo Order States



State	Description
Normal	Algo order is working. It may or may not have active child orders on the market.
PreparingReplace	The first phase of cancel-replace - some/all active children are cancelled at this step. New CancelReplace requests are still allowed at this stage (will be accumulated until Commit phase). Order leaves this state as soon as there are no child orders pending cancellation (in some cases immediately).
CommittingReplace	The second phase of cancel-replace - some/all active children are submitted/replaced. New CancelReplace requests for parent order are not allowed at this stage. Order leaves this state as soon as all child orders are confirmed/accepted by exchange (in some cases immediately).
AbortingReplace	Transient state that algorithm passes when replacement fails (e.g. child order cannot be replaced).
PreparingCancel	Pre-terminal phase: in this stage algorithm is working to cancel child orders. Note: order enters this state immediately upon cancellation request. There may be pending replacement request for parent or child orders in this state.
Final	Terminal stage: order is cancelled, completely filled, or rejected

APPENDIX A. SAMPLE IMPLEMENTING ORDER TIMEOUT

In this section we are going to show how to implement order timeout.

Imagine that our API allows user to specify optional order duration attribute. We will show how to parse order duration, how to define a timer and how to cancel order.

Assumption: for simplicity we are going to assume that order duration is immutable order attribute.

STEP 1: PARSE ORDER ATTRIBUTE

First we will add custom attribute parsing to our Algorithm. We will be using custom attribute key 6002 for order duration.

```
private static final int DURATION_ATTRIBUTE_KEY = 6002;
```

One good place to handle custom order attributes is processNewOrder callback. This method provides access to inbound order request (containing attributes) and order state.

```
@Override
protected CharSequence processNewOrder(final IcebergOrder order, final OrderNewRequest request) {
  ...
  // Schedule cancellation timer if necessary
  if (request.hasAttributes()) {
    ObjectList<CustomAttribute> attributes = request.getAttributes();
    for (int i=0; i < attributes.size(); i++) {
      CustomAttribute attribute = attributes.get(i);
      if (attribute.getKey() == DURATION_ATTRIBUTE_KEY) {
          long duration = OrderAttributesParser.parseDuration(DURATION_ATTRIBUTE_KEY, attribute.getValue());
          scheduleOrderExpirationTimer (parent, duration);
          break; // not interested in other attributes here
      }
    }
  }
  return super.processNewOrder(order, request);
}
```

Here we use OrderAttributesParser utility class that has handly method of parsing duration represented in HH:MM:SS.sss textual format to corresponding number of milliseconds.

STEP 2: SCHEDULE ORDER EXPIRATION TIMER

Let's define algorithm callback method that will be called from expiration timer.

```
private @Timestamp long onOrderTimeout(long now, IcebergOrder order) {
  if (order.isActive())
     cancelAlgoOrder(order, "Order expired");

  return TimerCallback.DO_NOT_RESCHEDULE;
}
```

See Algorithm Timers section above for more information about timer callbacks.

To avoid memory allocations, it is recommended to define callback reference as final field:

```
private final TimerCallback<IcebergOrder> cancelOrderCallback = this::onOrderTimeout;
```

Now we can schedule timer to call our callback:

```
private void scheduleOrderExpirationTimer(IcebergOrder order, long duration) {
   @Timestamp long orderExpirationTimestamp = getClock().time() + duration;
   order.setCancelJob(getTimer().schedule(orderExpirationTimestamp , cancelOrderCallback, order));
}
```

We keep TimerJob handle returned by timer schedule in our order. This allows us to cancel timer if needed.

STEP 3: CANCELLING TIMER

We need to cancel timer upon order completion. Order deactivation callback is a good place for this logic. Add the following to your custom order class:

```
@Override
public void onDeactivate(OrderEvent finalEvent) {
   super.onDeactivate(finalEvent);
   if (cancelJob != null) {
       cancelJob.cancel();
       cancelJob = null;
   }
}
```

Also, it may be a good idea to cleanup TimerJob handle we keep during order recycling:

```
@Override
public void clear() {
   super.clear();
   cancelJob = null;
}
```

REVISION HISTORY					
Ver.	Description of Change	Author	Date	Approved	
				Name	Effective Date
1.9	Ported into EPAM document format	Andy Malakov	5-March-2020		5-March-2020