# RuleMSX Documentation

### Overview

RuleMSX is a library which provides the core functionality of a rule engine. It is designed to inter-operate with the EasyMSX and EasyMKT libraries which use the Bloomberg API to access Bloomberg EMSX and market data.  
  
This functionality is provided in the shape of RuleSets, DataSets and Actions. By defining Rules and the conditions that must exist for these Rules to be triggered, the user can build complex reasoning based on the content of a DataSet, and how that DataSet changes over time. The Actions are the tasks performed as a result of a Rule being triggered.

### RuleSets

RuleSets are named entities that represent a collection of RuleSet objects. This is only used to organise rules into logical groupings. A RuleSet is a named collection of Rules.

And example of a RuleSet would be to route new orders to a particular broker code, based on certain criteria, such as the exchange. We will call this the “AutoRoute” ruleset.

Once we have a RuleSet and a DataSet object , we can execute the RuleSet. RuleSets need one or more supporting DataSets to operate against.

### Rules

Each Rule in a RuleSet is a named collection of RuleConditions and RuleActions. When all conditions in a Rule evaluate to True, the associated actions are executed.

Following the above example, and single rule within the AutoRoute ruleset would be “RouteUStoBB”, which would route any orders on the “US” exchange code to the broker known as “BB”. Another rule example could be “RouteLNtoBMTB”.

### RuleConditions

A RuleCondition is a named item within a Rule, which evaluates to either True or False. It does this through client-side code using a RuleEvaluator. A single Rule can have multiple RuleConditions, and they must all evaluate to True for the associated RuleActions to be executed.

For our RouteUStoBB example, we would have a condition called “MustBeUSExchange” that checked the order to ensure that it was for the “US” exchange. Another condition would be that the order must be in a NEW state, perhaps called “CheckNEWState”, to ensure that this rule is only triggered once.

### RuleEvaluator

A RuleEvaluator is an abstract class that must be implemented in the client-side code. This abstract class has an “Evaluate” method that must be overridden. This method must return True or False. When the Evaluate method is called, it is passed the current DataSet as a parameter, to support the determination of the return value.

The abstract class also provides a mechanism for creating a dependency between a Rule and named DataPoints. To do this, we call the AddDependantDataPointName method of the class, as follows :-  
  
 this.AddDependantDataPointName("OrderStatus")

In this case, we are saying that this particular Rule uses the value of the "OrderStatus" DataPoint. The purpose of using this mechanism is to ensure that if the value of "OrderStatus" in any DataSet changes, any WorkingRules add queue to be re-tested in the next cycle. The change to the value of a DataPoint is indicated by calling the SetStale method (see DataPointSource).

### RuleActions

A Rule can have many RuleActions. Each RuleAction has a client-side component called an ActionExecutor. When a Rule evaluates to True, all associated RuleActions are executed.

For example, we would have a RuleAction called “RouteOrdertoBB”, which would be called as a consequence of the “RouteUStoBB” rules all evaluating to True.

### ActionExecutors

An ActionExecutor is the client-side code that is run when an Action is executed. It is an abstract class that contains an Execute method that must be overridden.

When the “RouteOrdertoBB” action is executed, the Execute method of the instance of the abstract class would be called. This is the code that would create and send the route to the broker. Just as with the RuleCondition evaluators, the executors are passed the current dataset as a parameter when they are called.

### DataSets

DataSets are named entities that represent a collection of DataPoint objects. They are only used to organise DataPoints into logical groupings.

In our current example, we would create a DataSet object for each order. Once the DataSet object is defined, we can add it to the list of DataSets being run through a RuleSet by the ExecutionAgent.

### DataPoints

A DataPoint is an object that represents a single piece of data. Fundamentally, it is a simple key-value pair. A DataPoint doesn’t have value itself, but rather has an underlying DataPointSource which is used to provide the value.

Examples of DataPoints would be OrderNumber, OrderStatus, OrderExchange, etc.

### DataPointSource

A DataPointSource is the client-side code that provides a value for a named DataPoint. It is an abstract class with a GetValue method that must be overridden. It also provides a SetStale method that is used to indicate to the ExecutionAgent that the value must be re-examined. This will cause any WorkingRules for Rule that have a dependency on this DataPoint to be queue for re-evaluation on the next cycle.

The DataPointSources for the above example DataPoints would access the EMSX data to return the correct EMSX\_SEQUENCE and EMSX\_STATUS, and perhaps use the reference data service to get the exchange code for the ticker on the order.

### **The ExecutionAgent process**

When the application has completed the configuration of all the main elements (Rules, RuleConditions, Evaluators, Action, Executors, etc.), one or more RuleSets can be executed.  
  
This involves taking a DataSet and asking the RuleSet to be executed against that DataSet: -  
  
 myRuleSet.Execute(dataSet\_1);  
  
If this is the first time this RuleSet has been executed, a new ExecutionAgent will be created for the RuleSet. If the RuleSet already has an ExecutionAgent, it will be reused. The specified DataSet is then passed to the RuleSet's ExecutionAgent: -  
  
 executionAgent = new ExecutionAgent(myRuleSet, dataSet\_1);  
  
or  
  
 executionAgent.AddDataSet(dataSet\_1);  
  
Each ExecutionAgent has a DataSetQueue. Adding a DataSet to an ExecutionAgent simply adds the DataSet reference into the DataSetQueue. This is used to ensure that new DataSets are only ingested at the correct time, and not at the mid-point of a cycle.

A new ExecutionAgent will create a new internal thread that will operate a WorkingSetAgent. This WorkingSetAgent is the main loop that controls execution of the rules and actions for a RuleSet, and it continues to run until stopped by an external request (a call to the stop() method).   
  
Each cycle of the WorkingSetAgent begins with ingesting any DataSets in the ExecutionAgent's DataSetQueue. This is the process of creating a WorkingRule for each Rule in the RuleSet and the specified DataSet.

To create a WorkingRule, a Rule and a DataSet are required. A process known as dereferencing takes place, which has two steps. The first step is to take each Action associated with the Rule, and add the ActionExecutor references to the WorkingRule’s Executors collection.  
  
The second part of the dereferencing process is to iterate each RuleCondition of the Rule, and add it’s RuleEvaluator to the Evaluators collection of the WorkingRule. Each RuleEvaluator has a collection of DataPoint names that it depends on. For each of these dependant data point names, we find the actual DataPoint in the DataSet that matches the name. The WorkingRule is then added to the AssociatedWorkingRules collection of the DataPoint’s DataPointSource object.  
  
The reason for doing this is that when a DataPointSource’s value changes, its SetStale() method is (should be) fired. This forces each WorkingRule dependency of the DataPointSource to be added to the OpenSetQueue in the WorkingSetAgent for execution in the next cycle, unless the WorkingRule is already in the OpenSetQueue.   
  
Following the ingestion process, the current OpenSetQueue becomes the OpenSet, and the OpenSetQueue is then reset to empty.  
  
The OpenSet is now iterated, and each WorkingRule in the queue is processed. Each Evaluator in the WorkingRule is fired, passing it the WorkingRule’s DataSet. If all Evaluators in the WorkingRule return true, then the action process begins. Each action associated with the WorkingRule is executed.