VAIL Reference Guide

Overview

Vantiq is a platform for building Event-Driven Applications (EDA). EDA's continuously process streams of data, in the form of events, using that data to derive intelligence, effect changes to the state of the application's data model, and make decisions. EDA's are also sometimes known as "Reactive Applications" due to the fact that they are constantly reacting as new events occur. Vantiq uses rules, services, and procedures to construct an EDA.

- rules - A **rule** is triggered by an event such as the arrival of new application data, a change to the state of the application's data model or the expiration of a timer. Once triggered, the body of the rule is executed. The triggering event is supplied as a parameter to the body of the rule.
- services - A **service** encapsulates the behavior needed to perform a specific application task, including access to any data required or produced by that task and the production and consumption of events. Services provide this behavior via their interface which consists of one or more **procedures** and/or **event types**. Services can also be accessed remotely, allowing one application to use services offered by another.
- procedures - A **procedure** defines zero or more parameters and an optional return value. When invoked, the body of the procedure is executed using the supplied parameter values.

Rules, services, and procedures operate in the context of the application data model, querying for its current state, receiving events that signal changes to that state, and effecting changes of its own. The application data model consists of in-memory and persistent data managed by the Vantiq platform and data from external systems. The following resources are used to describe the application data model:

- types – A **type** describes the structure (aka schema) of data being processed by an application. This includes data contained in events, data processed by services, and data that is stored persistently.
- topics – A **topic** is an application defined channel to which events can be published and from which they can be received.
- sources – A **source** defines a connection to some external system. This allows the application to receive events generated by that system, publish events to that system, and query the current state of that system.

Rules, services, and procedures are constructed using a common set of declarative constructs provided by a language called **VAIL**. VAIL is:

- A domain specific language (DSL), designed to simplify the learning curve for building Event-Driven Applications.
- A JavaScript like language with support for embedded SQL for access to the persistent part of the application's data model.
- A dynamic language with runtime enforced (aka "duck") typing.

This document is a reference manual for VAIL, explaining how to use VAIL to construct the rules, services, and procedures that make up an EDA.

Packages

A package is a scoping/grouping mechanism that organizes a set of related application resources. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because an EDA can be composed of many resources, it makes sense to keep things organized by placing related resources into packages.

Programmers can define their own packages to bundle a group of application resources. It is a good practice to group related resources implemented by you so that a programmer can easily determine that they are related.

Each package creates a new "name scope" for the resources it contains (distinct from the "global" scope that exists without packages). As a result there won't be any name conflicts with names in other packages. Using packages, it is easier to locate and reuse the related resources.

Declaring Packages

Package names consist of two or more identifiers separated by the "dot" (`.`) character, such as *com.mycompany.myapp*. "Simple" package names (those with no "dot", such as *mypackage*) are not permitted. To help avoid collisions between packages that may be shared across or between companies, we recommend the following naming conventions:

- Package names are written in all lower case to avoid conflict with the names of other resources.
- Companies use their reversed Internet domain name to begin their package names. For example, *com.example.mypackage* for a package named *mypackage* created by a programmer at *example.com*.
- Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name (for example, *com.example.region.mypackage*).
- Packages provided by Vantiq will always begin with *io.vantiq* (which is reserved and cannot be used).

In some cases, the Internet domain name may not be a valid package name. This can occur if the domain name contains a hyphen or other special character, if the package name begins with a digit or other character that is illegal to use as the beginning an identifier, or if the package name contains a reserved Java keyword, such as `int`. In this event, the suggested convention is to add an underscore. For example:

Domain Name	Package Name Prefix
hyphenated-name.example.org	org.example.hyphenated_name

Domain Name	Package Name Prefix
example.int	int_.example
123name.example.com	com.example._123name

Placing VAIL Resources in Packages

VAIL resources are declared to be in a package using the package statement. If present, this must be the first, non-comment line in the VAIL text declaring the resource. For example:

```
// A top level comment is OK here
package com.company.factory.robots

rule processSensorData
when event occurs on "/topics/sensor"
... rest of rule definition ...
```

Here we are defining the rule *processSensorData* in the package *com.company.factory.robots*.

Placing Data Model Resources in Packages

The application data model resources are declared to be in a package by prepending the package name to the resource name (separated by a "dot" (`.`)). For example, creating a type with the name *com.company.factory.robots.Sensor* declares the type *Sensor* in the package *com.company.factory.robots*.

> Modelo presents the simple name (*Sensor*) and the package (*com.company.factory.robots.Sensor*) in two distinct fields, even though it is stored as a single, qualified name in the underlying resource model.

When declaring a topic in a package, the package name is contained in the topic's first path segment. For example, the topic */com.company.factory.robots/sensor* declares the topic */sensor* in the package *com.company.factory.robots*.

Package Scoping and Name Resolution

In addition to defining the package for all declared resources, the package statement also alters the default resolution for any referenced resources. Rather than treating unqualified names as being in the global scope, they are instead assumed to be in the declared package's scope. For example:

```
// A top level comment is OK here
package com.company.factory.robots

rule processSensorData
when event occurs on "/topics/sensor"

var robotStatus = select exactly one from Robots where id == event.value.robotId
```

The above rule references the topic */sensor* and the type *Robots*. In each case, these references would be automatically qualified with the package scope *com.company.factory.robots*. As a result, the system would look for these resources in the package (and only in the package) and not in the global scope as is done when there is no package.

Referencing Resources in Other Scopes

Resources outside of the local package (the one declared in the package statement) can be referenced by using their fully qualified name. However, doing so makes for some very verbose code. In addition resources in the global scope cannot be referenced this way, since their "qualified" form is identical to a simple name. The import statement can be used to address both of these issues.

Using an import statement allows the resource instance to be referenced using just its simple name (typically the part after the last ".") or by the specified alias if one is provided. For example, to reference the type *com.company.factory.robots.Robots* from a package other than *com.company.factory.robots* you could use the following import statement:

```
import type com.company.factory.robots.Robots
```

This would allow the type to be referenced using the symbol *Robots*. This import statement:

```
import type com.company.factory.robots.Robots as Robbie
```

would allow the type to be referenced using the symbol *Robbie*.

It is not legal for more than one import of the same resource to use the same alias. Nor is it legal to use an alias which overlaps with an existing local name. The use of aliases is optional, except when resolving an overlap between an imported resource instance and a local one or between two imported resources. In these cases, an alias is required to avoid any such overlap. For example, if there exists a local type with the name *Robots* then the following import is illegal:

```
import type com.company.factory.robots.Robots
```

Similarly, the following two imports would be illegal without an alias to disambiguate them:

```
import type com.company.factory.robots.Robots
import type com.company.space.Robots
```

Global resources can also be referenced through the use of imports. The following import:

```
import type Robots
```

provides access to the type *Robots* from the global scope.

VAIL Declarations

Rules, services, and procedures are created by submitting a VAIL declaration of the resource. The body of a rule or procedure consists of a sequence of VAIL statements.

A complete description of VAIL syntax and semantics can be found in VAIL Syntax and Semantics.

Rules

Rules provide a way for an application to respond when events related to the application data model occur. These events might indicate the arrival of data from an external system, the receipt of data from another application, or a change to the application's persistent data.

A rule declaration consists of the `RULE` keyword followed by the rule name, an optional version, optional selection criteria, an optional activation constraint, a triggering condition, and a set of VAIL statements representing the body of the rule. The syntax of the RULE statement is:

```
RULE <name>[:<version> WHERE <versionSelectionCriteria>] [ACTIVATIONCONSTRAINT <activationConstraint>]
WHEN <triggeringCondition>
[VAIL Statement]*
```

The specifics of name, version, versionSelectionCriteria, activation constraint, and when clause are described below.

For example:

```
RULE myFirstRule
WHEN EVENT OCCURS ON "/types/Order/insert"

var order = event.value
for (mgr in SELECT ONE * FROM Employee WHERE name == order.salesPersonManager) {
    PUBLISH { message: "There is an order for your review: " + order.orderId }
        TO SOURCE corporateEmail
        USING { recipients: [mgr.emailAddress] }

    if (order.customerName == "myFavoriteCustomer") {
        applyDiscount("10%", order)
    }
}
```

The rule is named "myFirstRule" as specified in the `RULE` statement.

The triggering condition is an INSERT into the **Order** resource as specified in the `WHEN` clause. The order instance will appear as an implicit variable named *order* (case sensitive) that is available for use in subsequent rules statements.

The `SELECT` statement searches for the manager of the sales person assigned to the *order*. If the manager exists, the body of the `FOR` statement is executed with the **Employee** instance representing the manager assigned to the local variable *mgr*.

The first statement in the block emails the manager a request to approve the order.

The second statement states that if the customer's name is "myFavoriteCustomer" the system will apply a 10% discount to the order by calling the **applyDiscount** procedure.

In the example, and all subsequent examples, the convention is to use lower case for JavaScript style keywords and UPPER CASE for all SQL style keywords (however all VAIL keywords are case insensitive).

Rule Name

The name of a rule may consist of alphanumeric characters and the underscore character. The rule name must start with an alphabetic or underscore character. It may not start with a numeric character. In addition, all names that start with `ars_` are reserved for system use.

Rules may be versioned and multiple versions of a rule may simultaneously be defined. If embedded in a name, the colon (':') character delimits a version number. The character string that precedes the colon is considered the proper name of the rule. The character string that follows the colon is considered the version number. Versions have special semantics as described in the subsection below. Version numbers may contain the characters: alphanumeric, '.', '-', '_'.

An example of a fully specified name containing both a name and a version identifier is `myRule:02.03.044`. The name of the rule is `myRule`. The version of the rule is `02.03.044`.

Versioning

Rules may be versioned to simplify the maintenance of an EDA as the data it processes changes. A collection of rules that differ only in their version number are considered to be the same rule and only one rule in the collection will be evaluated when a triggering event occurs.

By default, the "most recent" rule version is activated in response to an event. However, by incorporating a `WHERE` clause on the event binding, more sophisticated behavior can be achieved. For example, it may be desirable to introduce a new version of a rule but only apply it to a limited set of triggering events - perhaps those associated with a specific set of test data.

Evaluation occurs as follows:

- When an event occurs, the rule dispatcher evaluates the most recent version of the rule.
- If the event binding contains no constraint (via a `WHERE` clause), the version of the rule is evaluated unconditionally.
- If the rule statement contains a `WHERE` clause, the clause is evaluated.
- If the `WHERE` clause evaluates to true, the rule is fully evaluated.
- If the `WHERE` clause evaluates to false, this version of the rule is discarded and the next "youngest" version of the rule is selected and evaluated.

This process continues until a version of the rule is found that can be evaluated or until all versions have failed the evaluation constraint.

The notion of the "most recent" version of a rule is determined by the case insensitive, lexical ordering of the version numbers with the lexically largest value considered the most recent or youngest and the lexically smallest value the oldest.

A lexical ordering for version numbers is illustrated below:

- 02.03.044
- 02.01.033
- 01.01.02
- 01.00.04
- 0.0.0

Caution is recommended because there are perverse cases that must be avoided. For example, the following lexical ordering is accurately represented even though considering the ordering of the rules based on their numeric values would result in different ordering:

- 1.00.00
- 02.00.00

In order to eliminate such inadvertent surprise orderings, specifying versions in a fixed format with all version identifiers containing the same number of characters is highly recommended.

For example, given the following two versions of a rule:

```
RULE myRule:V1
WHEN EVENT OCCURS ON "/types/Employee/insert"
...

RULE myRule:V2 WHERE event.value.salary < 100
WHEN EVENT OCCURS ON "/types/Employee/insert"
...
```

The following `INSERT` will trigger the evaluation of **myRule:V1**:

```
INSERT Employee(name: "Harriet", salary: 1000)
```

However, the following `INSERT` will trigger the evaluation of **myRule:V2**:

```
INSERT Employee(name:"Stanley", salary: 50)
```

All versions of a rule **MUST** be triggered by the same `WHEN` clause. Only the `WHERE` clause of the event bindings may vary between versions of the same rule set.

Activation Constraint

A rule may declare an activation constraint. The activation constraint is used by the deployment tools to determine if the rule should be installed as active or inactive on a target node. By default, each rule is provisioned in its default state on each node on which it is deployed. However, there may be times when a rule should be provisioned as active on a select set of nodes, but provisioned as inactive on all other nodes. The activation constraint can be used to accomplish this.

The activation constraint is applied to the resource instance for each node on which the rule is deployed. If the constraint evaluates to true, the rule is configured to be active on that node. If the constraint evaluates to false, the rule is configured to be inactive. The activation constraint is expressed as a JSON string containing a query constraint (aka `WHERE` clause) that can be evaluated against the nodes resource. For example, given the following activation constraint:

```
{
    "ars_properties.status": "experimental"
}
```

The rule would only be active on those systems whose Nodes resource had a property "status" with the value "experimental". In this way you could cause the rule to be inactive on all systems except the "experimental" ones.

WHEN Clause

The WHEN clause is used to specify the triggering condition for a rule. It specifies what events must occur and what conditions they must meet (if any) in order for the rule's body to be executed. If the triggering condition specifies more than one event, the rule is only triggered after all events involved have been received and evaluated. If the triggering condition is satisfied, the body of the rule is evaluated. If the triggering condition is not satisfied, the rule is dismissed.

The formal syntax is:

```
whenClause := WHEN <triggeringCondition>

triggeringCondition := <eventBinding> [<eventCorrelation>] [<expectation>] | <eventMerge>

eventCorrelation := <temporalOperator> <triggeringCondition> WITHIN <windowInterval>
    [<correlationConstraint>]

eventMerge := <eventBinding> OR <triggeringCondition> [<correlationConstraint>] [<expectation>]

correlationConstraint := CONSTRAIN TO <queryCondition>

expectation := EXPECT (EVERY | WITHIN) <intervalLiteral>

eventBinding := EVENT OCCURS ON <eventPath> [AS <alias>] [ WHERE <eventCondition>]

temporalOperator := BEFORE | AFTER | AND

windowInterval := ... any interval literal ...

eventCondition := ... logical expression over event instance ...

queryCondition := ... logical expression over correlated event instances ...
```

Event Binding

All events in Vantiq originate from a specific resource instance in the application data model. They represent an operation which has been performed on that instance. The resources which may generate events are:

- Types – types defined in the Vantiq automation model will generate events for the `INSERT`, `UPDATE`, and `DELETE` operations.
- Sources – Vantiq sources will generate an event any time a message is received by the source.
- Topics – topics generate an event for every `PUBLISH` operation performed on the topic.
- Services – services can define *OUTBOUND* event types which will generate events based on semantics determined by the service's implementation.

Rules are bound to these events using an event binding of the following form:

```
EVENT OCCURS ON <eventPath> [AS <alias>] [ WHERE <eventCondition>]
```

The `eventPath` is a `String` literal with the following format: `/<resource>/<instance id>[/<operation>]`. This classifies each event based on the resource, instance, and operation that it represents. The `operation` is only required for type events since both sources and topics have only one possible operation.

The `alias` identifies an alternate variable name bound to the event object(s). If no alias is provided then the event will be bound to the local variable `event`. At runtime, the bound variable will contain the event instance being processed.

Events have the following properties:

- uuid – a unique identifier for the specific event instance.
- path – the path of the event (as described above).
- value – the data associated with the event. The format of this property depends on the resource and operation. The various data formats are described in the resource specific binding sections below.
- createdAt – the date and time of the event's creation.
- topic – for MQTT, AMQP and Kafka Sources, the topic the event was published on

The optional `WHERE` clause specifies a query condition that the event triggering the evaluation must satisfy before the rule can be activated. The form of this expression is identical to that used by the `WHERE` clause of the `SELECT` operation.

Correlating Events

The triggering condition for a `WHEN` clause can trigger rule evaluation by correlating more than one event. Correlation occurs when multiple triggering events are specified and connected by the temporal operators. For example:

```
RULE multipleConditions
WHEN
  EVENT OCCURS ON "/types/Customer/insert" AS Customer
  BEFORE
    EVENT OCCURS ON "/types/Order/insert" AS Order
    WITHIN 30 seconds
    CONSTRAIN TO Order.value.customer == Customer.value.name
```

In this example the rule body is only evaluated if an insert on **Customer** is followed by an insert on **Order** with the second event occurring within 30 seconds of the first event and both events referencing the same customer object.

The general form of a correlating trigger condition is:

```
<eventBinding> <temporalOperator> <triggeringCondition> WITHIN <windowInterval> [CONSTRAIN TO <queryCondition>]
```

The temporal operator in the above example, `BEFORE`, indicates that one event happens before another event. The complete set of temporal operators is:

- `BEFORE` The left event occurs before the right event
- `AFTER` The left event occurs after the right event
- `AND` The left and right events occur in either order

WITHIN

The `WITHIN` clause specifies how closely in time the two events must occur. This time interval is typically specified as an interval literal.

For example,

```
WITHIN 12 seconds
```

This states that the second event must arrive within 12 seconds of the first event. Note that because of the vagaries of the scheduling algorithms used in the operating system and the underlying infrastructure, the 12 second interval will not be precisely enforced to the microsecond. The developer should think of the interval more appropriately as at least 12 seconds plus any delays associated with scheduling the rule for evaluation and then completing the evaluation.

If the interval specified in a `WITHIN` clause expires, the rule may need to take an action. This is accomplished by specifying a TIMEOUT section in the rule body. When the `WITHIN` interval expires, the statements included in the `TIMEOUT` section are executed. See `TIMEOUT` clause for a detailed definition of `TIMEOUT`.

CONSTRAIN TO

The `CONSTRAIN TO` clause specifies joint conditions the correlated events must satisfy. In the example:

```
RULE multipleConditions
WHEN
  EVENT OCCURS ON "/types/Customer/insert" AS Customer
  BEFORE
    EVENT OCCURS ON "/types/Order/insert" AS Order
    WITHIN 30 seconds
    CONSTRAIN TO Order.value.customer == Customer.value.name
```

The customer name in the **Customer** event must match the customer in the **Order** event. If the condition is not satisfied by the pair of events, the rule body is not evaluated. For this example, these semantics correspond to the intuitive notion that it doesn’t make sense to process an event on one specific customer with an order for a different customer.

The `CONSTRAIN TO` clause may contain an arbitrarily complex logical expression comparing values in the two events being correlated. It is also possible to include expressions that constrain a single event although these are more properly specified in the event binding using a `WHERE` clause. This affords the system more optimization alternatives. For example:

```
RULE multipleConditions
WHEN
  EVENT OCCURS ON "/types/Customer/insert" AS Customer
  BEFORE
    EVENT OCCURS ON "/types/Order/insert" AS Order
    WITHIN 30 seconds
    CONSTRAIN TO Order.value.customer == Customer.value.name AND Customer.value.name == "paul"
```

is a valid `CONSTRAIN TO` clause but the preferred expression would be:

```
RULE multipleConditions
WHEN
  EVENT OCCURS ON "/types/Customer/insert" AS Customer WHERE Customer.value.name == "paul"
  BEFORE
    EVENT OCCURS ON "/types/Order/insert" AS Order
    WITHIN 30 seconds
    CONSTRAIN TO Order.value.customer == Customer.value.name
```

Compound Correlation

Temporal events can be composed to express more complex conditions. The conditions are evaluated left to right in a manner similar to arithmetic expression evaluation. For example:

```
RULE multipleConditions
WHEN
  EVENT OCCURS ON "/types/Customer/insert" AS Customer
  BEFORE
    EVENT OCCURS ON "/types/Order/insert" AS Order
    WITHIN 30 seconds
    CONSTRAIN TO Order.value.customer == Customer.value.name
  BEFORE
    EVENT OCCURS ON "/types/Shipment/insert" AS Shipment
    WITHIN 5 minutes
    CONSTRAIN TO Order.value.orderNo == Shipment.value.orderNo
```

The system evaluates the first condition, then waits for the second condition and, after the first two conditions are satisfied, waits for the third condition. The precedence can be modified using parenthesis in a manner similar to arithmetic expressions:

```
RULE multipleConditions
WHEN
  EVENT OCCURS ON "/types/Customer/insert" AS Customer
  BEFORE
    (
      EVENT OCCURS ON "/types/Order/insert" AS Order
      BEFORE
        EVENT OCCURS ON "/types/Shipment/insert" AS Shipment
        WITHIN 5 minutes
        CONSTRAIN TO Order.value.orderNo == Shipment.value.orderNo
    )
WITHIN 30 seconds
CONSTRAIN TO Order.customer == Customer.name
```

In this modified example, the precedence will cause the first condition to be evaluated and then combined with the result of evaluating the subsequent two conditions. Note that the `CONSTRAIN TO` and `WITHIN` clauses that previously followed the second condition have been moved to the end of the expression because the second and third conditions are being evaluated as a single condition with respect to the first temporal operator. Also note that the `WITHIN` clauses are now somewhat inconsistent since they specify a wait time of 5 minutes between the second and third conditions but only 30 seconds between the first condition and the COMBINATION of the second and third conditions.

`WHEN` clauses with compound triggering conditions are a powerful tool for correlating data arriving from multiple event streams. This is common in both consumer and industrial situations where data is being received from more than one sensor. For example, in a retail setting information may be received from BLE beacons, an indoor location system and the consumer's smart-phone. In an industrial setting sensor data is being received from multiple sensors on separate channels and, possibly, separate machines where the readings need to be correlated in time.

Merging Event Streams

Sometimes you have multiple event streams which are producing the same data. For example, you might have a source which provides sensor data and multiple instances of the source which provide data from different physical locations. In this case rather than writing separate rules to bind to each source, you can instead merge the event streams and process them in a single rule. This is done using the `OR` operator in a WHEN clause. For example:

```
RULE processMergedStream
WHEN
  EVENT OCCURS ON "/sources/EasternSource"
  OR
  EVENT OCCURS ON "/sources/WesternSource/"
```

In this example the rule body is evaluated whenever we receive a message from either EasternSource or WesternSource. The messages will be processed as they arrive, so they will be interleaved with each other without regard to where they originated.

The general form of a merging trigger condition is:

```
<eventBinding> OR <targetCondition>
```

The event streams being merged must have the same event schema type (see sources and topics for how to specify the event schema type). They must also be bound to the same local variable (either the default `event` or a common alias).

Detecting Missing Events

In addition to processing events as they occur, there may be times when a rule needs to run code when events don't occur when expected. The WITHIN clause accomplishes this in the context of a correlation, but what about detecting the lack of events from a single event source? To do this you use the EXPECT clause. This clause has the following syntax:

```
<eventBinding> [EXPECT <expectation> <interval>]
```

Where `expectation` describes how to determine if an event is "missing" and `interval` specifies how long to wait for the expectation to be fulfilled (typically as an interval literal). The possible expectations are:

- `EVERY` – we expect to see at least one event during each `interval`. In any given `interval` when no event is seen the system will generate a timeout.
- `WITHIN` – after any given event, we expect to see another event within the specified `interval`. Whenever we don't see the next event in `interval` time, the system will generate a timeout.

Whenever the specified expectation is not fulfilled, the rule may need to take an action. This is accomplished by specifying a TIMEOUT section in the rule body. Whenever the expectation produces a timeout, the statements included in the TIMEOUT section are executed. See TIMEOUT clause for a detailed definition of `TIMEOUT`.

For example, let's say you have an MQTT source which should produce messages continuously. If you want to know whenever it stops producing messages for at least a minute you could use:

```
WHEN EVENT OCCURS ON "/sources/myMQTTSource" EXPECT EVERY 1 minute
... do work on message ...
TIMEOUT
... do work for lack of data ...
```

In this example, if the source stopped sending messages we would call the TIMEOUT section of the rule once every minute. If we just wanted it to be called once after the minute passes then we could use `EXPECT WITHIN`.

TIMEOUT clause

The `TIMEOUT` clause declares an alternate variant of the rule body which will be executed if the correlated event specified by the rule's `WHEN` clause does not occur within the specified time period. For example:

```
RULE multipleConditions
WHEN
  EVENT OCCURS ON "/types/Customer/insert" AS Customer WHERE Customer.value.name == "paul"
  BEFORE
    EVENT OCCURS ON "/types/Order/insert" AS Order
    WITHIN 30 seconds
    CONSTRAIN TO Order.value.customer == Customer.value.name

    INSERT INTO Log(msg: "The event occurred.")

TIMEOUT

    INSERT INTO Log(msg: "The event did not occur.")
```

If the correlated event occurs in the 30 second time interval specified then the main body of the rule will be executed (in this case the first `INSERT` statement). However, if the event does not occur then the body specified after the `TIMEOUT` keyword will be executed (in this case the second `INSERT` statement).

Before Rules

Before Rules are used to extend the default validations provided for resource types. They have the following syntax:

```
RULE <name>
BEFORE (INSERT | UPDATE | DELETE) ON <resourceType>
[VAIL Statement]*
```

The resource instance being operated on is bound to an implicitly defined variable with the same name as the target *resourceType*. The VAIL statements are evaluated synchronously, immediately before the associated operation. The value produced by the rule is used as the actual resource instance when the operation is finally applied. This allows the rule to alter the instance to add, remove, or change properties as appropriate. It also allows the rule to cancel the operation by returning a `null` value. This can be accomplished by setting the variable representing the target type to `null` at the end of rule execution or by using an explicit `RETURN null` statement.

Services

Services encapsulate behavior associated with a specific functional aspect of an application. Services expose that behavior via their interface, which consists of a list of procedures and/or Event Types which are available to consumers of the Service.

Procedures

Procedures are used to implement the behavior of Procedure Signatures defined in the Service Interface (aka service procedures) and to modularize VAIL code to make it easier to reuse and help organize the functions that support an EDA (aka utility procedures). Procedure invocations must supply the procedure name and a list of parameters (positionally) or a set of named parameters.

Service Procedures

A service procedure declaration consists of an optional visibility modifier, an optional state access modifier, a qualified procedure name combining the name of the service and the name of the procedure, formal parameters, an optional return type, an optional `WITH` clause and a list of VAIL statements representing the body of the procedure. The formal syntax is:

```
[PRIVATE] [GLOBAL | MULTI PARTITION | STATELESS] PROCEDURE <serviceName>.<name>(<parameters>)[:<returnType>]
[WITH properties = <procedureProperties>,
      profiles = ["<localProfile>", "<systemProfile>"],
      ars_dependentResource = "/<resource>/<resourceId>",
      ars_public = [true|false],
      ars_group = "<groupName>"
]
<VAIL Statements>*
```

Any comment at the top of a procedure declaration becomes the *description* for the procedure. This description will be passed to any use of the procedure or its service in the Submit Prompt ActivityPattern, LLM.submitPrompt procedure or Tool GenAI Component. The top-level comment/description must be before any Package Declaration for the procedure, if there is one.

For example:

```
// changes the salary for the employee with name "empName"
PROCEDURE Payroll.changeSalary(empName String, newSalary Decimal): Employee

UPDATE Employee (salary: newSalary) WHERE name == empName
```

This procedure is part of the *Payroll* service. It accepts two parameters, an employee name and a new salary for the employee. The top-level comment becomes its description. When invoked it updates the employee assigning the new salary to the employee and returns the updated employee instance. Of course, a more realistic example would deal with exceptional conditions such as the employee failing to exist or a salary specified that is outside the standard salary range for the employee’s position.

Procedure Modifiers

A service procedure may have an optional visibility modifier and/or an optional state access modifier (if the service is stateful).

Visibility Modifiers

The procedure visibility modifier governs whether or not a procedure can be invoked from code that is not part of the current service. By default all procedures are considered “public”. This means that they can be invoked from any context (from the same service, from a different service, from a standalone procedure, from a rule, etc…).

> This use of the term “public” is distinct from the use of the term in the context of request authorization. Public resources are resources which can be accessed without requiring any authorization. Since this includes service procedures, you will see the term public procedure applied in this context. The overlap of terms is unfortunate, but you can usually tell from context whether the reference is to the *visibility* of a procedure at development time or the *accessibility* of a procedure at runtime.

- `PRIVATE` – indicates that the procedure can only be invoked from another procedure in the same service or from a service event handler defined by the service. Useful for modularizing internal service behavior without having to make it part of the service interface. Private procedures cannot be invoked remotely. Invocation of a stateful private procedure by a procedure with a conflicting state access modifier (for example partitioned from stateless) is legal, but requires the use of the EXECUTE statement.

State Access Modifiers

Service procedures have access to any state that is defined by their service. There are 4 possible state access patterns:

- Global – global procedures have access to the service’s global state.

- Single Partition – single partition procedures have access to the service's partitioned state. They require at least one parameter which is used to identify which partition should be accessed by a given invocation of the procedure.
- Multi Partition – multi-partition procedures have access to the service's partitioned state. When invoked they execute once for each partition that exists (these invocations may occur concurrently).
- Stateless – stateless procedures are procedures that do not access any of the Stateful Service's state properties. Stateless procedures defined as part of a partitioned Service do not require a partitionKey parameter.

More details about these access patterns can be found in Stateful Services and In-memory State Management.

If a service declares only global state, then all of the service's procedures are "global". If it declares only partitioned state, then all procedures are "single partition". If a service declares both global and partitioned state, then procedures without a modifier default to being "single partition". The default behavior can be altered by applying the following state access modifiers to the procedure:

- `GLOBAL` – the procedure uses the global state access pattern and has access to the service's global state. This modifier is only legal if the service declares global state.
- `MULTI PARTITION` – the procedure uses the multi-partition state access pattern and has access to the service's partitioned state. This modifier is only legal if the service declares partitioned state.
- `STATELESS` – the procedure is stateless and cannot access any of the service's state (global or partitioned).

> There is no `SINGLE PARTITION` modifier since if it existed, it would always be optional (given the default behavior).

Qualified Name

The declaration of a service procedure includes a qualified procedure name consisting of the name of the service and a simple procedure name, separated by a dot (`.`). When referencing a service procedure in VAIL code, its fully qualified name must be used. Both names must be legal Java identifiers which means that they:

- May consist of alphanumeric characters and the underscore character.
- May not start with a numeric character

In addition, all names that start with "ars_" are reserved for system use.

Declaration of a service procedure will create both the procedure and the containing service, assuming it does not already exist. Services created in this manner will always be stateless. See Services documentation for how to declare stateful services.

Recalling our previous example:

```
PROCEDURE Payroll.changeSalary(empName String, newSalary Decimal): Employee

UPDATE Employee (salary: newSalary) WHERE name == empName
```

This declares a procedure named *changeSalary* which is part of the service *Payroll*. To call this procedure one would write VAIL code like this:

```
var employee = <... create or select employee instance ...>
var updatedEmployee = Payroll.changeSalary(employee.name, 50000)
```

Formal Parameters

The formal parameters of a procedure consist of a comma separated list of zero or more parameter declarations. Each parameter declaration consists of the parameter's name and type, followed by zero or more parameter modifiers. The formal syntax is:

```
parameterList := [<parameterDecl>](, <parameterDecl)*

parameterDecl := <parameterName> [<parameterType>] <parameterModifier>*

parameterModifier := ARRAY | REQUIRED | DEFAULT <defaultValue> | DESCRIPTION <stringConstant>
```

The parameter name must be a legal VAIL Identifier and the parameter type must be a legal VAIL Type. If the procedure is a single partition procedure then it must have at least one parameter whose type must be one of: `String`, `Integer`, `Decimal`, `Currency`, `DateTime`, or `ResourceReference`.

For example:

```
PROCEDURE Payroll.changeSalary(empName String REQUIRED, newSalary Decimal(2) REQUIRED): Employee

UPDATE Employee (salary: newSalary) WHERE name == empName
```

The procedure has 2 parameters, *empName* which must be a `String` and *newSalary* which must be a `Decimal` with a "precision" of 2. Both parameters are declared as `REQUIRED` (more on that in a bit).

Parameter Type

Although optional, it is highly encouraged to declare a type for all parameters (note that mixing typed and untyped parameters is not permitted). When present the system will validate that any values provided for the parameter are legal values of the declared type and will automatically perform any supported type conversions. Since VAIL is dynamically typed, these validations/conversions occur at runtime. In addition, the Vantiq system will confirm that any requested operations can be performed and produce an error if they cannot (also known as "duck" typing). This check will occur whether or not the parameter has a declared type.

Parameter Modifiers

Parameters may have any of the following modifiers:

- `ARRAY` – declares that the value being passed must be an array of the declared type (if there is one) or any array if there is no declared type. Sequences are not currently supported as procedure parameters.
- `REQUIRED` – declares that the parameter is required and that any caller must supply a value (though that value can be `null`).
- `DEFAULT` – declares that if a value is not supplied for the parameter, then the specified value should be used.
- `DESCRIPTION` – provides a description for the parameter which is used for documentation purposes.

Here is an example of using the `ARRAY` modifier:

```
PROCEDURE Payroll.changeSalary(empName String ARRAY, newSalary Real ARRAY)

FOR (i in range (0, empName.size(), 1)) {
    UPDATE Employee (salary: newSalary[i]) WHERE name == empName[i]
}
```

This procedure accepts an array of names and salaries and updates all the employees in the two arrays. This example assumes the two arrays are the same size. If the newSalary array is shorter than the empName array, a runtime error will be produced.

Parameter Passing

Procedures that are part of the service's interface (aka non-private procedures) implement **call by value** semantics, where objects and arrays are treated as values. That is, if a parameter value is an array or an object, a copy of the array or object is made and the copy is passed as the actual value of the parameter. This implies that changes to an array or an object made within the procedure will NOT be seen by the caller unless the updated values are explicitly returned to the caller as part of the procedure's return value. These semantics make remote procedure invocations behave identically to local procedure invocations, enabling transparent distributed processing.

Private procedures implement **call by object reference** semantics, where object and array *references* are treated as values. That is, if a parameter value is an array or an object, the parameter will be bound to a reference. This means that the procedure can change the contents of the array or object, but it cannot change *which* array or object is being referenced. These semantics are more efficient and used because there is no possibility of remote invocation.

> Note that the semantics used for private procedures are the same as those used by Java. This is often also referred to as call by value. The distinction between the two is what is considered a value (subtle, but important).

Return Type

Procedures produce a result upon completion. By default this will be the value of the last statement executed. This default behavior can be altered with the use of the `RETURN` statement (which must itself be the last statement of the procedure). Procedures may optionally declare a return type. If provided, the system will use the type to validate and, if legal, convert any value produced by the procedure.

Recall our previous example:

```
PROCEDURE Payroll.changeSalary(empName String REQUIRED, newSalary Decimal(2) REQUIRED): Employee

UPDATE Employee (salary: newSalary) WHERE name == empName
```

The procedure returns the result of the `UPDATE` statement and that return will be validated against the declared type – `Employee`.

The return type can be qualified using both the `Array` and `Sequence` keywords. Explicitly declaring that a procedure returns a sequence allows the system to "stream" the procedure's results as they are produced. This will happen when either:

- A VAIL caller uses an EXECUTE statement with a processing block.
- A VAIL caller uses an iteration statement to process the sequence.
- The `stream` URL parameter is set when using the REST binding.

Here is an example of a procedure returning a sequence:

```
PROCEDURE employeesWithMinSalary(minSalary Real): Object Sequence

SELECT SEQUENCE FROM Employee WHERE salary >= minSalary
```

The procedure returns a sequence containing all employees with the given minimum salary.

WITH Clause

The procedure `WITH` clause allows a procedure to be configured via VAIL syntax. The `WITH` clause supports the following properties:

- `properties` - Corresponds to the `ars_properties` value on the underlying procedure definition. Properties can be used to select the procedure in a dynamic `EXECUTE` statement.
- `profiles` - An array of profiles that override the caller's profiles at execution time for purposes of authorizing operations performed by the procedure. The creator of the procedure must have the authorizations necessary to grant the specified profiles.
- `ars_public` - Used to indicate whether or not the procedure should be available to all users, without authentication. If not specified, then `ars_public` defaults to false.
- `ars_dependentResource` - Specifies a dependency between the procedure and another resource instance using a Resource Reference. When specified, if the dependent resource is deleted the procedure will be deleted as well.
- `ars_group` - Indicates the security group to which the procedure belongs. When specified, a group value restricts access to the procedure to members of the group. Namespace administrators can always access the procedure.
- `ars_relationships` - Specifies the explicit relationships that the Rule or Procedure has to other resources in the Namespace.

Public Procedures

Procedures that have the `ars_public` option set to true in the `WITH` clause can be executed by anyone, without any check for authorizations. Public procedures should only be used to implement publicly accessible functionality, most commonly to support server side functionality of Public Clients.

All public procedures must make use of the `profiles` property to specify the execution profiles applied at runtime. If `profiles` is not specified it is impossible to determine what level of privilege to run with when a public procedure is invoked by an unknown user.

Procedures executed publicly are executed under an anonymous `publicExecutor` identity, which is the username that will appear in any audit records or `ars_createdBy` or `ars_modifiedBy` values in any records created during the execution of that procedure.

Note that marking a procedure as publicly accessible does not require that it is executed publicly. It can still be executed by an authenticated and authorized user, in which case the procedure will execute under that identity rather than the `publicExecutor` identity.

For more information on how to invoke public procedures via the REST API, please refer to the API Reference Guide.

Group Restricted Procedures

By default, all procedures in a namespace are executable by all users in a namespace. To restrict access to a Procedure, set the `ars_group` value of the procedure in the `WITH` clause to any Group. Once set, the procedure is only visible and executable to members of the group and namespace administrators (who bypass group restrictions generally).

> Once set, the `ars_group` value of a procedure is immutable. If you must change the group, delete the procedure and recreate the procedure with the new group specification.

Utility Procedures

A utility procedure declaration consists of a name, the formal parameters, an optional return type, an optional `WITH` clause and a list of VAIL statements representing the body of the procedure. The declaration syntax is:

```
<name>(<parameters>)[:<returnType>]
[WITH properties = <procedureProperties>,
      profiles = ["<localProfile>", "<systemProfile>"],
      ars_dependentResource = "/<resource>/<resourceId>",
      ars_public = [true|false],
      ars_group = "<groupName>"
      ars_relationships = [{"to": "/<resource>/<resourceId>", "type": <relationship type>}...]
]
<VAIL Statements>*
```

The procedure name must be a legal Java identifier consisting of alphanumeric characters and the underscore character. It cannot begin with a numeric character. All of the options available work as they do for service procedures.

Hiding System Procedures

Although it is not necessarily recommended, it is possible to define both service and utility procedures which have the same name as those defined in the system namespace (with the exception of specific, "built-in" procedures). When this happens the locally defined procedure effectively "hides" the system procedure of the same name. Any references to the procedure will invoke the local version and not the system version. It is still possible to use the system version of the procedure by further qualifying its name using the `system.` prefix. For example, the following references a system version of our previously defined "changeSalary" procedure (assuming such a thing existed):

```
system.Payroll.changeSalary()
```

Data Model Declarations

The resources that make up the application data model are created using the platform's resource API. See the Resource Reference Guide for more information. Once created these resources are globally visible within a given namespace and can be referenced by the resources declared using VAIL. Together they define the state that will be managed by the application in order to provide its functionality.

This state encompasses both the events to which the application will react and any passive data to be acted upon by the application (as part of the response to the events). The data model spans three distinct domains – in-memory data, persistent data, and data from external systems.

In-Memory State

The application's in-memory state is defined by topics which declare a channel on which events can be sent and received and stateful services which manage passive, in-memory state and provide an API through which that state can be manipulated.

Topics

Topics provide a way to send and receive application defined events that are internal to the Vantiq platform. These events may be specific to a single application or span multiple applications, through the use of the Event Catalog. Each topic declares the structure of the data carried by its events using a type. An application receives these events via rules with an event binding of the form:

```
WHEN EVENT OCCURS ON "/topics/<topicName>"
```

Topic names themselves take the form of "paths", such as `/engineSensor/rpm/max`.

Applications send events via a topic using the VAIL `PUBLISH` statement.

In-Memory State Management

Stateful Services encapsulate access to in-memory state the structure of which is defined using types. Access to the state is provided by the procedures that make up the service's interface, thus allowing each service to provide whatever API makes sense for the state being managed. A key aspect of managing the in-memory state is dealing with concurrent access requests which can (and will) occur when multiple "clients" of the service (be they rules or other services) invoke the service at the same time. This is done using the VAIL built-in Concurrent service and function expressions. For example, here is a procedure which will compute the average for a supplied `Real` value in a way that handles concurrent access:

```
PROCEDURE ComputeSensorStats.recordStatsForSensor(sensorId String, sensorValue Real)
// The property "statsBySensorId" is defined in the service's partitioned type.
// It is typed as "Map" and implicitly initialized to "Concurrent.Map()"
statsBySensorId.compute(sensorId, (key, stats) => {
    // If this is the first time we've seen this sensor id, set up the defaults
    if (stats == null) {
        stats = {
            total: 0.0,
            count: 0
        }
    }

    # Compute the average based on the supplied value
    stats.total += sensorValue
    stats.count += 1
    stats.avg = stats.total / stats.count
    return stats
})
```

The above example shows a procedure designed to update the in-memory state, but of course services can also provide access to that state as well, for example:

```
PROCEDURE ComputeSensorStats.getSensorStats(sensorId String):Object
return statsBySensorId.get(sensorId)
```

Persistent State

The application's persistent state is defined by resource types which can include those predefined by Vantiq or application specific types. Once defined they can be used to manage resource instances which generate events and support a standard set of data manipulation operations. These operations use a SQL-like syntax which should be familiar to most developers.

Extending Data Validation

This feature is disabled for any resource type with a value of `true` for *rulesSuppressed*

Each resource type declares the properties that it supports. These declarations include information that the system uses to validate the structure and contents of the submitted resource instances. For example, the **Employee** type might declare that it expects to see a first, last, and middle name for the employee, as `String` values, and that first and last name are required, but middle name is optional. Given this the system would reject any attempt to insert or update an **Employee** instance which did not meet these requirements.

Sometimes however, it is necessary to define more complex or domain specific validations. This can be done using Before Rules. Before rules are bound directly to their associated operation and can influence the operation's outcome (including causing the operation to be canceled entirely).

Type Event Generation

> This feature is disabled for any resource type with a value of `true` for *rulesSuppressed*

The persistent resource instances generate events to indicate changes in their state. The general form of the event path for such events is: `/types/<resourceTypeName>/<operation>`. Due to the fact that these events are associated with the **types** resource, they are often referred to as "type events". The event path includes the name of the resource type associated with the changed instance and the operation that caused the change. The possible operations are:

- insert – indicates that an instance of the resource was created
- update – indicates that one or more instances of the resource were updated
- delete – indicates that one or more instances of the resource were deleted

In each case the event data carries more detailed information about the actual change. In the case of a change to a single instance, the event will contain a copy of the affected instance. The update and delete operations have "bulk" versions designed to improve performance by avoiding the need to manipulate the affected instances. In those cases the event will contain the following operation metadata:

- ars_bulkOperation – a boolean property which will be set to `true`
- count – a count of the affected instances
- qual – the query used to select the affected instances

The *ars_bulkOperation* property can be used in `WHERE` clause of an event binding to ensure that a rule either does or does not receive events related to bulk operations.

Persistent State Management

As is the case with the in-memory state, one of the major concerns when managing persistent state is properly handling concurrent operations. The Vantiq persistent data model ensures that each individual operation is handled atomically, however, it does not support transactions, so it cannot extend that guarantee to sequences of operations or operations performed simultaneously. This can lead to some very common "race conditions".

Simultaneous Updates

One such condition arises when an application attempts to perform simultaneous updates of a resource instance. For example, given the following VAIL statements:

```
// The variable "orderId" is assumed to be populated with the id of the target order instance
var order = SELECT * FROM Orders WHERE orderId == orderId
UPDATE Orders(total: order.total + 10) WHERE orderId == orderId
```

If this code is executed concurrently twice, the end result may be that the total is increment by either 10 or 20. To help address this issue, the Vantiq platform supports a technique known as optimistic concurrency control (OCC).

The idea behind OCC is to proceed on the assumption that most of the time, the update will succeed, because most of the time there will be no actual concurrency. If this turns out to be incorrect, then the system will generate a well-known exception so the application can adjust however necessary and potentially retry the operation. The error code for this exception will always be *io.vantiq.update.authorization.or.version.conflict*.

For this to work, the system must have a way to tell if the update is being applied to an "out of date" instance of the resource. Vantiq accomplishes this through the use of the *ars_version* system property. This property is automatically maintained by the Vantiq platform and is incremented any time an update is made to a resource instance. By including this property in any `UPDATE` operation the application tells Vantiq to use OCC to confirm the validity of the update. The other requirement is that the `WHERE` clause of the `UPDATE` must reference exactly one instance. This means that it must use either the system's built-in *_id* property or the resource type's natural key.

For example:

```
var updateComplete = false
for (i in range(0, 5) until updateComplete) {
    try {
        // The variable "orderId" is assumed to be populated with the id of the target order instance
        var order = SELECT * FROM Orders WHERE orderId == orderId
        UPDATE Orders(total: order.total + 10, ars_version: order.ars_version) WHERE orderId == orderId

        // If we get here the update succeeded, so we want to terminate the loop
        updateComplete = true
    } catch (error) {
        // If this isn't a version conflict, report it
        if (error.code != "io.vantiq.update.authorization.or.version.conflict") {
            rethrow(error)
        }

        // Since this is a version conflict, just fall through so we retry
    }
}

// Check to see if we completed the update successfully
if (!updateComplete) {
    // This is an error condition, so report it somehow...
}
```

Here we have included the *ars_version* property, so if we attempt to perform a concurrent update the result will either be correct or an explicit error. We are using a `TRY` statement to intercept the error so we can check to see if it was a version conflict and if so, we fall through and let the `FOR` statement take us around for another try (up to 5 times in this example).

Simultaneous Inserts

Another form of race condition involves trying to insert “the same” resource instance (as determined by its natural key) more than once. When this happens, the result is an exception with the error code `io.vantiq.resource.duplicate.key`. The situation cannot be avoided by using a `SELECT` statement to first check for the existing instance since if there are concurrent executions they might both see the instance as “missing” and thus both try to perform the insert. To help address this scenario, VAIL provides the `UPSERT` statement. `UPSERT` guarantees that it will insert the instance if it does not exist and update it if it does. It handles any necessary retries automatically.

External State

The application’s external state is defined by sources which declare a connection from the Vantiq platform to an external system using a specified protocol. The complete list of supported implementations can be found in the External Source Reference Guide. Once declared, a source can be used to send and receive events and to query the state of the external system (not all of these are available for all source implementations).

Sources may produce events as data of interest becomes available. The form the event path is `/sources/<sourceName>`. Sources can declare the structure of the data contained in their events using a message type.

Sources may permit the sending of events using the VAIL `PUBLISH` statement.

Source may permit querying the state of their data model using the VAIL `SELECT` statement.

The exact operations available, formats used, and data available is source specific.

VAIL Syntax and Semantics

Syntax

The VAIL syntax derives from JavaScript for all procedural elements and from SQL for most interactions with the application data model. The VAIL resources are defined by an ordered list of VAIL statements, stored as text using the `UTF-8` character set. These statements are executed sequentially with the current statement completing before the next statement is evaluated.

VAIL has no statement terminator and permits statements to be combined on the same line or split across multiple lines. However, for readability it is recommended that there be at most one statement per line and that a reasonable line length limit be observed (wrapping statements to the next line when needed).

Comments

VAIL supports both single line and multi-line comments. Single line comments start with `//` and can be found at any position in the line. The characters following `//`, until the end of the line, are considered part of the comment.

```
// This is a single line comment
var a = 1 + 1 // This is also a single line comment
```

Multi-line comments start with `/*` and can be found at any position in the line. The characters following `/*` will be considered part of the comment, including new line characters, up to the first `*/` closing the comment. Multi-line comments can thus be put at the end of a statement, or even inside a statement.

```
/* a standalone multi-line comment
   spanning two lines */
var hello = "hello" /* a multi-line comment starting
                   at the end of a statement */
var a = 1 /* one */ + 2 /* two */
```

Keywords

VAIL keywords are case insensitive. The following keywords are reserved and cannot be used as resource names (types, rules, procedures, etc…) or as variable names, parameter names, or aliases:

alter	before	component	create	delete	drop
else	event_stream	execute	filter	for	function
if	insert	java	map	not	procedure
publish	return	rule	ruleset	select	system
timeout	update	upsert	var	when	

The following keywords may only appear at the beginning of a new statement:

alter	create	drop	filter	for	if
map	return	select	try	upsert	var

Identifiers

Identifiers start with a letter or an underscore. They cannot start with a number.

A letter can be in the following ranges:

- 'a' to 'z' (lowercase ascii letter)
- 'A' to 'Z' (uppercase ascii letter)

Then following characters can contain letters and numbers.

Here are a few examples of valid identifiers (here, variable names):

```
var name
var item3
var with_underscore
var _underscoreStart
```

But the following ones are invalid identifiers:

```
var 3tier
var a+b
var a#b
```

All keywords are also valid identifiers when used after a reference operator:

```
foo.if
foo["return"]
foo.upsert
```

Statements

Each statement describes an action to be taken by the VAIL runtime system. The legal statements are:

- Package Declaration
- Variable Declaration
- Variable Assignment
- Type Literals
- Any Expression
- Flow Control
- Iteration
- Procedure Execution

- RETURN
- Event Sending
- Data Manipulation
- Resource Definition

VAIL Types

VAIL type names are case sensitive, so the type `String` is not the same as the type `string`.

Standard Types

- `String` – a string of characters
- `Integer` – an integer value represented as a 64-bit quantity
- `Real` – a real value represented as a 128-bit floating point number
- `Decimal` – a fixed point decimal number
- `Currency` – a currency type indicator and a fixed point decimal number
- `Boolean` – `true` or `false`.
- `DateTime` – an instant in time. A DateTime is represented externally as an ISO date.
- `GeoJSON` – a location as a GeoJSON object
- `Object` – an object of *name, value* pairs
- `ResourceReference` – a reference to a Vantiq resource stored as a document with the properties *resource* and *resourceId*.
- `Map` – holds a mapping *key* to *value* for one or more arbitrary keys.
- `Value` – holds an arbitrary *value*.
- `Any` – indicates that the actual type is unknown (or unknowable). Note that this can only be used as part of a procedure signature. It can be used to type either a procedure parameter or its return type.

The types do not necessarily correspond directly to SQL types or to JavaScript types. For example, there is no JavaScript-like **Number** type that supports both integer and floating point numbers. However, the general behavior of strings, integers, reals, dates and objects will be familiar to both JavaScript and SQL users.

The specific definitions are:

String

Strings are represented in the form of a chain of UTF-8 characters. VAIL string literals are enclosed in double quotes (`"`), for example:

```
var a = "string"
myProcedure("this is a string")
```

String Concatenation

All the VAIL strings can be concatenated with the `+` operator:

```
var a = "one part" + " of a string"
```

Multi-line Strings

Strings can span multiple lines and will retain all newline and white space characters between the starting and ending quotes:

```
var multiLine = "line one
line two
    indented line three"
```

Escape Characters

VAIL strings support embedded *escape* sequences to represent characters that are either difficult to type or would be ambiguous. Escaped characters begin with a backslash (`\`). For example, to embed a double quote in a string you would use:

```
var embeddedQuote = "string with a \" in it"
```

The recognized escape characters are:

Escape Sequence	Character
\b	backspace
\f	formfeed
\n	newline

Escape Sequence	Character
\r	carriage return
\s	single space
\t	tabulation
\\	backslash
\“	double quote

Unicode Escape Sequences

For characters that are not present on your keyboard, you can use Unicode escape sequences: a backslash, followed by ‘u’, then 4 hexadecimal digits.

For example, the Euro currency symbol can be represented with:

```
var unicodeEmbed = "The Euro currency symbol: \u20AC"
```

Integer

A signed, 64-bit value. The permissible value range is `-9,223,372,036,854,775,808` (-2**63) to `9,223,372,036,854,775,807` (2**63 -1). The runtime representation of `Integer` is equivalent to Java `Long`.

> When using any JavaScript client (such as the Vantiq IDE), the maximum integer displayable is the JavaScript Number.MAX_SAFE_INTEGER value, which is 9,007,199,254,740,991 or (2**53 -1). This can cause display issues when returning a value larger than MAX_SAFE_INTEGER to be shown in a UI dialog. *This is a display-only problem.* In general, the value displayed will have less precision that the actual value–for example, if a procedure returns Long.MAX_VALUE, it might be displayed in the IDE as 9223372036854776000. For display in JavaScript or the IDE, you can use the VAIL `toString()` method to work around this JavaScript limitation. For example, use `return myValue.toString()`.

Integer literal values are declared using a sequence of numeric characters as in:

```
var myInt = 12345
```

Real

A double-precision 64-bit IEEE 754 floating point value. The runtime representation of `Real` is equivalent to Java `Double`.

Real literal values are declared using a sequence of numeric characters containing a decimal point. For example:

```
var myReal = 123.45
var myRealToo = 12345.0
```

Decimal

A fixed point decimal number. If a property or parameter is declared as Decimal a *scale* SHOULD be specified. The scale is an `Integer` that declares the number of digits of precision to carry after the decimal point. If no scale is provided then a scale of 0 is used, resulting in an `Integer` value. For example:

```
procedure Payroll.changeSalary(empName String, salary Decimal(2))
```

The parameter *salary* is declared as a `Decimal` value with 2 digits of precision.

It is not possible to declare a `Decimal` value as a literal. Instead the procedure **toDecimal** must be used on a `Real` literal or a `String` literal.

The JSON representation of a `Decimal` value is a `String`. The use of strings eliminates any potential rounding errors that might occur in transforming the values to/from their JSON representations. The JSON form is also what should be used in a query constraint.

> This means that when receiving events containing Decimal data from an *external* client (such as the Vantiq IDE or a Vantiq Client), the Decimal values will be represented by Strings. These values will not be automatically converted, even if the target topic or source defines a message schema. To compensate for that, you can use toDecimal() to convert the string back to a Decimal for use in comparisons & arithmetic. For example, *toDecimal(event.Temperature, 2) > 500* in a *condition* expression for an Event Stream Activity Pattern or in an IF or WHERE clause. See also Decimal and Currency Support for details about supported operators.

Currency

Represents currency values (dollars, euros, etc.). It consists of a currency code and a `Decimal` value. If a property or parameter is declared as `Currency` a *scale* SHOULD be specified. If no scale is provided then a scale of 0 is used, resulting in an `Integer` value.

It is not possible to declare a `Currency` value as a literal. Instead the procedure **toCurrency** must be used on a `String` literal.

The JSON representation of a `Currency` value is a `String` of the following form:

```
<currency code>:<value>
```

where

- *currency code* – the ISO 4217 code assigned to all currencies (eg. GBP, USD, JPY, CNY, …)
- *value* – a decimal number representing the currency value.

A property may contain values in multiple currencies but it is the user's responsibility not to attempt nonsensical operations such as summing a property that contains values in different currencies. In such cases the user should sum the values grouping by the currency code. This is facilitated by the representation used for currency. For example, a user wishing to find sales > $100 would issue the query (in pseudo query form):

```
select name, sales from store where sales > "USD:100.00"
```

The fact that a currency MUST be specified limits the result to ONLY stores where sales are recorded in USD.

Boolean

`Boolean` is a special data type that is used to represent truth values: `true` and `false`. `Boolean` values can be stored in variables, assigned into fields, just like any other data type:

```
var myBooleanVariable = true
var untypedBooleanVar = false
obj.booleanField = true
```

`true` and `false` are the only two literal boolean values. But more complex boolean expressions can be represented using logical operators.

DateTime

An instantaneous point on the time-line. `DateTime` values can be represented down to nanosecond precision, but generally anything finer than millisecond is rarely used. The runtime representation of `DateTime` is equivalent to Java `Instant`. All times are normalized to UT so that there is no ambiguity when operating across multiple timezones.

It is not possible to declare a `DateTime` value as a literal. Instead the procedure **toDate()** must be used to convert from either a `String` value or an `Integer` value.

The JSON representation of a `DateTime` is a `String` using the ISO-8601 representation:

```
yyyy-MM-dd'T'HH:mm:ss[.SSS]'Z'
```

When accepting `DateTime` values through the Vantiq Resource API or in query constraints the system will accept either the above `String` form or an `Integer` containing an "epoch time" value (in milliseconds). Conversion from a higher precision representation to a lower precision representation truncates higher precision components of the date.

> "Epoch time" refers to the amount of time elapsed since `1970-01-01T00:00:00Z`. So "epoch milliseconds" is the number of milliseconds since that time.

Interval Literals

Interval literals (aka intervals) represent a length of time such as one minute, three days or 25 hours. Intervals always have the same magnitude, regardless of when they are created. So 1 day is always 24 hours and one week is always 7 days.

Intervals can be specified in one of the following units:

- milliseconds
- seconds
- minutes
- hours
- days
- weeks

Examples:

```
23 milliseconds
100 seconds
5 weeks
```

At runtime an interval value is represented as an `Integer` containing the number of milliseconds in the interval.

ResourceReference

A reference to a Vantiq resource instance. The runtime representation of a reference is an object with the properties:

- **resource** – the resource being referenced
- **resourceId** – the resource id of the instance being referenced

It is not possible to declare a `ResourceReference` value as a literal. Instead the procedure **Utils.buildResourceRef()** must be used.

The JSON representation of a `ResourceReference` is a `String` of the form:

```
/<resource>/<resourceId>
```

When using the REST API the above form can be used to construct a URI which references the resource instance by prepending the standard resource API prefix.

When using a `ResourceReference` value in a query constraint you can use the JSON representation to query for exact matches. It is also possible to query for individual properties of a `ResourceReference` as follows:

```
SELECT * from MyType WHERE entity.resource == "myresource"
```

When inserting/updating a property of type ResourceReference the value can be presented in either the external “string” form or the internal “document” form. Both of these are legal:

```
INSERT MyType(entity: {resource: "types", resourceId: "MyType"})
INSERT MyType(entity: "/types/MyType")
```

Object

A JSON object. An `Object` instance contains zero or more properties, each of which is bound to a value. Object instances may be created using an Object literal expression using the following syntax:

```
objectLiteral := { [<propertyDeclaration>][, <propertyDeclaration>]* }

propertyDeclaration := <propertyName> : <propertyValue>

propertyName := any legal identifier

propertyValue := any value expression
```

For example:

```
var myObj = {
    prop1: "property value",
    prop2: myProc()
}
```

GeoJSON

A Point, LineString or Polygon. The value is represented as a JSON object containing two properties:

```
{"type": ["Point" | "LineString" | "Polygon"], "coordinates": <valueSpec>}
```

The *valueSpec* contains the specification of the point, line or polygon as an array of geocoordinates with each geocoordinate consisting of an array containing two or three values: the longitude and the latitude (in that order), and, optionally, the altitude in meters. Note that for a Point, the coordinates are a single array containing two (or three) values. However, for LineSegments and Polygons the coordinates are an array containing an array of geocoordinates (each nested array containing two (or three) values representing longitude and latitude (and, optionally, the altitude in meters), respectively).

Observe that the coordinate *type* **MUST** be properly capitalized and, specifically, must be one of the strings: “Point”, “LineString” or “Polygon”.

GeoJSON literals are `Object` literals with the appropriate properties, for example:

```
var aPoint = {
  type: "Point",
  coordinates: [-122.4194155, 37.7749295]
}

var polygon = {
  type: "Polygon",
  coordinates: [
    [ [84.0, 40.0], [84.0, 41.0], [88.0, 41.0], [88.0, 40.0], [84.0, 40.0] ]
  ]
}

// Point with altitude

aPoint = {
  type: "Point",
  coordinates: [-151.00708, 63.06909, 6190]
}
```

Map

A `Map` instance contains zero or more keys, each of which is bound to a non-null value. `Map` instances are created using the built-in Concurrent.Map() or Concurrent.Cache() constructors and support all the documented procedures. Properties of this type:

- Support replication
- May *not* be persisted

Value

A `Value` instance contains a single value, which may be `null`. `Value` instances are created using the built-in Concurrent.Value() constructor and support all the documented procedures. Properties of this type:

- Support replication
- May *not* be persisted

Application Defined Types

An EDA can define its own types to represent the data that it processes. These types can be used anywhere a standard type can be used. At runtime these are treated as `Object` for purposes of validation and conversion.

Collections

VAIL supports collections in the form of `Arrays`, similar to JavaScript arrays, and `Sequences`. It also allows `Object` values to be treated as a collection of key/value pairs.

Arrays

An array is a dynamically expanding set of elements with each element placed in a “slot” in the array. The slots are directly address with an index number. The slots are numbered from 0 to N where N is the size of the array - 1. New elements may be added to either the beginning or end of the array. Arrays may be iterated over using the `FOR` statement. We sometimes use the term *list* to refer to an array.

Array literals are declared using the following syntax:

```
"[" [<valueExpression>] [, <valueExpression>]* "]"
```

For example:

```
var intArray = [1, 2, 3, 4, 5]
var emptyArray = []
```

Arrays are stored in memory. Therefore, the maximum size of an array is limited. Vantiq **STRONGLY** recommends that arrays contain less than 10,000 elements. In a future release the 10,000 element limit on the size of an array **WILL BE ENFORCED** by the runtime system.

Sequences

A sequence is also a collection of elements but, in contrast to arrays, sequences are:

- unbounded – a sequence may contain an arbitrary number of elements.
- lazy – the elements in a sequence are materialized as they are requested.
- immutable – the contents of a sequence cannot be altered and sequences cannot be randomly accessed.

Sequences may be transformed into a new sequence by using statements such as `FILTER` and `MAP`, and they may be iterated over using `FOR`. See Iteration for more details.

While there are some restrictions, sequences can, in general, be transmitted remotely. It is best to explicitly declare when a procedure will return a sequence by defining a return type for the procedure.

Objects

Any `Object` value can be treated as a collection and processed using iteration. One item is produced for each property in the object. These items have the properties *key* and *value* (corresponding to the name of the property and the property's value). For example:

```
// Create object instance
var myObj = {
    prop1: "property value",
    prop2: 10
}

// Iterate over the object's properties
for (prop in myObj) {
    log.info("Prop Name " + prop.key + " value: " + prop.value)
}
```

Type Conversions

Vantiq performs the following automatic type conversions:

From\To	String	Integer	Real	Decimal	Currency	Boolean	DateTime
String	✔	✔[1]	✔[1]	✔[1]	✔[1]	✔[4]	✔[1]
Integer	✔	✔	✔	✔		✔[2]	
Real	✔		✔	✔		✔[2]	
Decimal	✔		✔	✔			
Boolean	✔					✔	
Currency	✔		✔	✔	✔		
DateTime	✔	✔[3]					✔

1. Contents of string must conform to format of target type.
2. Only supported values are 0 (`false`) and 1 (`true`).
3. Converted to epoch milliseconds (any nanoseconds are truncated).
4. Converted to `true` if the string is not null and is equal, ignoring case, to the string "true". Otherwise, converts to `false`.

Values of type `Object` and `GeoJSON` can be converted between each other, but not to/from any other type. When converting from `Object` to `GeoJSON`, the system confirms that the necessary properties are present.

These conversions occur when:

- Storing a value in the property of a resource instance.
- Passing a value to a procedure parameter with a declared type.
- Returning a value from a procedure with a return type.

See also Decimal and Currency Support for details about supported operators.

Operators

Arithmetic Operators

VAIL supports the usual familiar arithmetic operators you find in mathematics and in other programming languages like JavaScript.

Binary Operators

Operator	Purpose
`+`	Addition
`-`	Subtraction

Operator	Purpose
`*`	Multiplication
`/`	Division
`%`	Remainder
`**`	Power

Unary Operators

Both `+` and `-` are available as prefix, unary operators. For example:

```
var minusOne = -1
var positiveOne = -(-1)
positiveOne = +1
```

In terms of unary arithmetic operators, the `++` (increment) and `--` (decrement) operators are available in postfix notation. They return the value *before* increment/decrement.

Assignment Arithmetic Operators

The binary arithmetic operators we have seen above are also available in an assignment form (except for power (`**`)):

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`

For example:

```
var a = 3
a += 4 // result is 7
a -= 1 // result is 6
a *= 2 // result is 12
a /= 3 // result is 4
a %= 3 // result is 1
```

Relational Operators

Relational operators allow comparisons between values, to know if two values are the same or different, or if one is greater than, less than, or equal to the other.

The following operators are available:

Operator	Purpose
`==`	equal
`!=`	different
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal

VAIL does not support JavaScript's *identity* operator (`===`).

These operators can be applied to values of any type. Each type applies its own semantics to the comparison. For example:

```
var isFalse = 2 > 11 // Numerically 2 is not greater than 11
var isTrue = "2" > "11" // Lexically the string "2" is greater than the string "11"
```

Logical Operators

VAIL offers three logical operators for `Boolean` expressions:

- `&&` – logical "and"
- `||` – logical "or"
- `!` – logical "not"

For example:

```
var isTrue = true && true
isTrue = true || false
isTrue = !false
```

Precedence

Logical "not" (`!`) has a higher precedence than logical "and" (`&&`).

```
var isFalse = (!false && false)
```

Logical "and" (`&&`) has a higher precedence than logical "or" (`||`).

```
var isTrue = true || true && false
```

> Unlike JavaScript, VAIL *does not* implement short-circuiting of the logical operators in most cases. Of particular importance, it will *not* short circuit procedure calls used in logical expressions. For example, given the expression `(true || myProc())`, VAIL *will* execute the procedure `myProc`.

Conditional Operators

Ternary Operator

The ternary operator is a shortcut expression that is equivalent to an if/else branch assigning some value to a variable.

Instead of:

```
if (string !=null && string.length() > 0) {
    result = "Found"
} else {
    result = "Not found"
}
```

You can write:

```
result = (string != null && string.length() > 0) ? "Found" : "Not found"
```

Reference Operators

Subscript

The subscript operator (`[]`) can be used to both access and store data in `Object` or `Array` values. It can also be used to access (but not store) data in `Map` values. The syntax is (in this case the `[` and `]` characters should be interpreted as literals):

```
subscriptExpression := <object, map or array>[<subscript>]

subscript := <numeric value> | <alphanumeric value>
```

When used with an `Object` or `Map` value, the subscript is interpreted as a `String` which corresponds to a property name/key. When used with an `Array` value, the subscript is interpreted as an `Integer` which corresponds to the index of an item in the array. Accessing an item that does not exist will return `null`. Setting a value into an `Array` at an index that does not yet exist will result in the prior indexes being "filled" with `null` values.

The operator can appear in any expression and on either side of an assignment. For example:

```
// Here we are accessing properties of an object
var obj = {}
obj["prop1"] = 42
var prop1Value = obj["prop1"]
var nullValue = obj["noSuchPoperty"]

// Here we are accessing into an array by index
var arrayValue = []
arrayValue[10] = 42
var setValue = arrayValue[10]
var nullValue = arrayValue[15]
var size = arrayValue.size() // the result here is 11
```

As noted above, use of the reference operator on a `Map` value is limited to accessing data, but not storing it. Therefore, the following is legal:

```
var myMap = Concurrent.Map()
myMap.put("key", "value")
var value = myMap["key"]
```

But this is not:

```
var myMap = Concurrent.Map()
myMap["key"] = "value"
```

Property Reference

The property reference operator (`.`) is used to both access and store data in the properties of an `Object` value and to access data from a `Map` value. Property references take the form:

```
<variableName>.<propertyName>
```

> This is equivalent to the subscript operator when applied to an `Object` or `Map` value.

References to properties and elements of an array can be combined:

```
<variableName[<indexValue>].<propertyName>[<indexValue].<subPropName>
```

Lambda Operator

The lambda operator (`=>`) is used to define function expressions of the form:

```
<functionParams> => <VAIL expression> | { [VAIL Statements]* }

functionParams := () | <parameterName> | ([<parameterName>[, <parameterName>]*])

parameterName := VAIL Identifier
```

The left hand side of the expression defines the parameters expected by the function. There can be zero or more such parameters. Each one declared will define an implicit variable name which is visible to the body of the function (which may be an expression or a sequence of VAIL statements). The primary use of function expressions is in combination with the Concurrent service. For example:

```
var concurrentMap = Concurrent.Map()
concurrentMap.compute("key1", (key, currentValue) => {
    if (currentValue == null) {
        return 0
    }
    return currentValue + 5
})
```

Here we have a function expression as the second parameter to the `compute` procedure. The function is invoked with 2 parameters: *key* and *currentValue* and returns an `Integer` value based on the state of the map (though in this example that's always known).

Decimal and Currency Support

The arithmetic operators are supported on the `Currency` and `Decimal` types with the exception of the remainder (`%`) operator. Arithmetic on mixed numeric types is supported between `Decimal` / `Currency` values and any `Integer`, `Real`, `Decimal` or `Currency` values whether stored in a variable or expressed as a constant. For example, assuming *myDec* is a variable containing a `Decimal` value and *myCur* is a variable containing a `Currency` value, the following expressions are supported:

```
myDec * myDec
myDec + 12
myDec - 100
myDec * 2

myCur * myCur
myCur + 12
myCur - 100
2 * myCur
myCur / 4
myCur ** 2

"USD:12.0" + myCur
"20.0" * myDec
```

Relational operators always determine the type of the comparison by the type of the left operand. Thus, if the left operand is a `Decimal` or `Currency` value, the right operand is converted to a `Decimal` or `Currency` value and the comparison made. However, if the left operand is `Integer`, `Real` or `String` an attempt is made to convert the right operand to the corresponding type. If the right operand is a `Decimal` or `Currency` value, this conversion will fail resulting in an **IllegalArgumentException**. The simplest approach is to always make a `Decimal` or `Currency` value the left operand in any comparison. If this is not possible, the left operand should be explicitly converted to `Decimal` using the **toDecimal()** procedure (conversion to `Currency` from `Number` is not currently supported).

Examples of valid comparisons to `Decimal` and `Currency`:

```
myDec >= "100.42"
myCur < "USD:100.00"
toDecimal("100.42") < myDec
toCurrency("USD:100.00") == "USD:100.42"
myCur == myCur
```

The following are examples of unsupported comparisons:

```
"USD:100.00" <= myCur
50.0 < "22.25"
```

When two `Decimal` or `Currency` values are involved in the computation, the scale of the result is set as follows:

- **Addition** – the larger of the scales assigned to the two operands
- **Subtraction** – the larger of the scales assigned to the two operands
- **Multiplication** – the product of the scales assigned to the two operands
- **Division** – the scale of the divisor is subtracted from the scale of the dividend

Of course, if the result value is then assigned to a property whose type is `Decimal` or `Currency`, the scale is set to the scale declared for the property.

Packages

Package Declaration

The `package` declaration statement must be the first non-comment line of the text used to declare a VAIL resource. It has the syntax:

```
packageStatement := package <packageName>

packageName := <identifier>[.<identifier>]*
```

The package names `io.vantiq` and `system` are reserved and may not be used.

Resource Import

Using an `import` statement allows the resource instance to be referenced using just its simple name (typically the part after the last ".") or by the specified alias if one is provided. The syntax is:

```
resourceImport := import <resource> <resourceId> [as <alias>]

alias := <simple resource id>
```

The value for *resource* can be any one of: **service**, **procedure**, **type**, **topic**, **source**, and **eventstream.** The format and expectations for the *resourceId* and *alias* depend on the resource being imported. The alias must always be the unqualified form for the target resource.

> To access a service procedure, you must import the *service*. Importing individual service procedures is not supported.

Import statements are only legal if there is also a package statement present. They must appear after the package statement and before the resource declaration.

Variables

Variable Declarations

Variables created within the body of a rule or procedure must be declared before they are used. A variable is declared using the variable declaration syntax:

```
var <variableName>
```

This creates a new variable named *variableName*. Variable names are case sensitive and must consist of alphanumeric characters and the underscore. The name may not begin with a number.

As a convenience, a variable may be declared and assigned a value in a single statement using the notation:

```
var <variableName> = <expression>
```

Variables are globally visible within the rule or procedure in which they are defined. They may not "hide" any variables defined outside of that scope, including:

- The event variable declared by a `WHEN` clause.
- Any declared procedure parameters.
- Any service state variables.

Re-declaration of variables is possible, but not recommended.

Implicit Variable Declarations

There are several circumstances that result in implicit declaration of a variables:

- Event variables defined in the `WHEN` clause of a rule.
- The properties of a service's state types.
- Result variable defined as part of a `SELECT` statement.
- The parameters defined by a procedure declaration.
- The iteration variable declared in a `FOR` statement.

Variable References

Several VAIL operations support the use of variable references in order to dynamically specify the target of the statement. Variable references have the form `@<variableName>` and indicate that the system should use the current value of the variable as the name of the target resource. For example, the following code:

```
var variableName = "Sensor"
SELECT * FROM @variableName AS s
```

results in the system issuing a SELECT against the type **Sensor** and is equivalent to:

```
SELECT * FROM Sensor as s
```

The documentation of each operation will indicate when/if this construct is explicitly supported (no mention means that there is no support in that context).

Assignment

A traditional assignment statement is available to assign a value to a variable. The example assignment:

```
myVar = 5 + 10
```

will assign the existing variable named **myVar** to the integer value 15. More generally:

```
<variableName> = <expression>
```

Flow Control

IF

The `IF` statement provides conditional execution. It consists of several clauses (only one of which is required). The syntax is:

```
if (<logicalExpression>) {
    <VAIL Statements>*
} [ else if (<logicalExpression>) {
    <VAIL Statements>*
}]*
[ else {
  <VAIL Statements>*
}]
```

The `logicalExpression` must result in a `Boolean` value. It may contain expressions and relational operators combined using the logical operators.

The logical expressions are evaluated one at a time, starting with the initial `IF` clause and proceeding through each `ELSE IF` clause, in the order written. The first one to evaluate to `true` will cause the associated block of VAIL statements to be executed. No further evaluations are performed in that case. If none of the conditions evaluate to `true` then the statements associated with the `ELSE` clause (if any) will be executed.

TRY

The `TRY` statement provides a way to `CATCH` (intercept) errors that occur while running a rule or procedure. When an error occurs during the execution of a rule or procedure the system will generate an exception which causes the execution to terminate with an entry written to the current namespace's error log. However, if the error occurs in the context of a `TRY` statement then the error will be delivered to the `CATCH` block and processing will continue from there as if the error had not occurred. The general form of the statement is:

```
try {
    <VAIL Statements>*
} catch(errorVar) {
    <VAIL Statements>*
} [finally {
    <VAIL Statements>*
}]
```

Upon completion of the statements in the `CATCH` block processing will continue with any statements that follow the `TRY` statement, unless the error processing statements themselves generate an error (at which point control would pass to the `CATCH` block of any enclosing `TRY` statement or would result in termination if no such statement exists).

In addition to the `CATCH` block (which is mandatory, though it may be empty) there may be an optional `FINALLY` block. The statements found in this block will be executed any time the block is exited, whether that is due to normal completion of the statement or an error that has occurred during that execution.

Iteration

VAIL offers several ways to iterate through collections of values.

FOR

The `FOR` statement iterates over the elements of a collection using the following syntax:

```
for (<iterationVariable> in <collection> [until <terminationCondition>]) {
    <VAIL statements>*
}
```

The `iterationVariable` is an implicitly defined variable which is populated by each element in the `collection` as it is processed. The `collection` is an expression or variable which must evaluate to one of the legal collection types. If present, the optional `UNTIL` clause specifies a logical condition which will be evaluated after each iteration. If the condition evaluates to `true` then the iteration will terminate. Otherwise the statements in the body of the `FOR` will be executed once for each element in the collection.

For example,

```
var Orders = ... get a collection of orders ...
for (o in Orders) {
    if (o.total > 1000) {
        PUBLISH { message: "Thank you for your order" }
            TO SOURCE companyEmail
            USING { recipients: [ o.customerEmail ] }
    }
}
```

This will inspect every order and send thank you emails to each customer with a total value greater than 1,000. Here is a similar example using a SELECT expression:

```
for (o in SELECT * FROM Orders WHERE total > 1000) {
    PUBLISH { message: "Thank you for your order" }
        TO SOURCE companyEmail
        USING { recipients: [ o.customerEmail ] }
}
```

This will select orders with a total greater than 1,000 and send each customer a thank you email.

The `FOR` statement supports iteration over the properties of an object:

```
var foundPaul = false
var person = {name: "paul", address: "11 Main Street", salary: 200}
for (prop in person) {
    if (prop.key == "name" && prop.value == "paul") {
        foundPaul = true
    }
}
if (foundPaul == true) {
    person.status = "employee"
}
```

Although a bit contrived the above code fragment will search the object looking for the property named *name* and then check if the value of the property is the string `paul`. Note that if the test is successful, a new property is added to the object. The next time a `FOR` statement is used to iterate over the properties of the object the added property will be processed in the body of the `FOR`. Iterating over the properties of an object can be very useful in cases where you are processing objects whose structure may change from instance to instance.

Iteration with `FOR` can be terminated before all the elements in the collection have been processed, for example:

```
var total = 0
for (i in range(1,10,1) until total >= 12) {
    total += i
}
```

The value of total on completion is 15 computed as: 1 + 2 + 3 + 4 + 5. The until clause is evaluated at the end of each loop iteration, so the `FOR` loop above terminates after the iteration where total is greater than or equal to 12.

FILTER

The FILTER statement accepts a collection as its input and iterates over its elements producing a new sequence as its result. The syntax is:

```
filter (<iterationVariable> in <collection> [until <terminationCondition>]) {
    <VAIL statements>*
}
```

The `iterationVariable` is an implicitly defined variable which is populated by each element in the `collection` as it is processed. The `collection` is an expression or variable which must evaluate to one of the legal collection types. If present, the optional `UNTIL` clause specifies a logical condition which will be evaluated after each iteration. If the condition evaluates to `true` then the iteration will terminate. Otherwise the statements in the body of the `FILTER` will be executed once for each element in the collection. For each element processed, the VAIL statements in the body are executed to determine if the element should appear in the resulting sequence. A result of `true` means that the element will appear in the sequence; otherwise it will not.

For example:

```
var user = SELECT ONE * From MyUser as user WHERE name == "paul"
var locFilter = FILTER (r in SELECT * FROM PossibleActions) {
    if (r.location == user.location) {
        return true
    }
    else {
        return false
    }
}
```

Note that in this case we apply the filter directly to the `SELECT` statement instead of using an intermediate variable. This allows the system to treat the output of the SELECT as a sequence and avoid materializing all of its results. The value in the `locFilter` variable also contains a sequence, which can be further processed using a `FOR` or a `MAP` statement or event another `FILTER`, such as:

```
var typeFilter = FILTER (r in locFilter) {
    return r.type == user.type
}
```

The first filter will iterate over all objects in “PossibleActions” returning a result consisting of all possible actions that are appropriate in the user’s location. The second filter will iterate over the results in **locFilter** applying another constraint to further reduce the set of possible actions to only those that are relevant to the user type.

Using sequences in this manner and applying successive processing to them is the most efficient way to process large amounts of data (and the only way to process results sets which exceed the array size limit).

MAP

The MAP statement accepts a collection as its input and iterates over its elements producing a modified collection consisting of all the elements returned by the MAP body. The body of the MAP statement may modify the value of each element or even produce an entirely new element. The syntax is:

```
map (<iterationVariable> in <collection> [until <terminationCondition>]) {
    <VAIL statements>*
}
```

The `iterationVariable` is an implicitly defined variable which is populated by each element in the `collection` as it is processed. The `collection` is an expression or variable which must evaluate to one of the legal collection types. If present, the optional `UNTIL` clause specifies a logical condition which will be evaluated after each iteration. If the condition evaluates to `true` then the iteration will terminate. Otherwise the statements in the body of the `MAP` will be executed once for each element in the collection.

The following example produces a sequence of all employees, each with their salary increased by 10%.

```
var employeesWithRaises = MAP (r in SELECT * FROM Employee) {
    r.salary += (r.salary * 0.10)
    return r
}
```

The resulting sequence can be used anywhere they are supported such as in a `FOR` statement to process the contents further.

Procedure Execution

VAIL allows procedure invocations to occur as standalone statements or as part of an expression. There are three different forms used to express the invocation of a procedure.

Direct Invocation

Procedures can be invoked directly by specifying the procedure name and any parameters, enclosed in parenthesis. The parameters can be specified positionally or by name. The syntax is:

```
directInvocation := <targetProcedure>(<actualParameters>)

actualParameters := <emptyArguments> | <positionalArguments> | <namedArguments>

emptyArguments :=

positionalArguments = <valueExpression> [, <valueExpression>]*

namedArguments = <parameterName>: <valueExpression> [, <parameterName>: <valueExpression>]*
```

The `targetProcedure` is the fully qualified name of the procedure, including its service, if any. Positional parameters are bound to the declared parameters in the order specified. Named parameters are bound using the supplied `parameterName` and can therefore appear in any order. When specifying parameters (using either approach) it is possible to omit parameters, as long as they are not required (or have a default value). All parameters must be either named or positional. A mix of named and positional parameters is not supported.

For example, to invoke the previously shown **Payroll.changeSalary** procedure with two parameters, **empName** and **salary** the following invocation could be used:

```
Payroll.changeSalary("Some Employee", 30000.00)
```

Similarly, to supply named parameters for the same invocation:

```
Payroll.changeSalary(empName: "Some Employee", salary: 30000.00)
```

Method Style Invocation

Many of the built-in procedures can be called “method style” similar to JavaScript method invocations. The syntax is:

```
methodInvocation := <receiver>.<directInvocation>
```

When a procedure is called using this style, the `receiver` is always passed as the procedure's first parameter. A classic example is the **length** procedure applied to a string which is invoked as a procedure with the string as its first parameter:

```
var myString = "a string"
var l = length(myString)
```

Using the method style, the same procedure call takes the form:

```
var myString = "a string"
var l = myString.length()
```

EXECUTE

The previous two procedure invocation styles perform local, synchronous execution of a well-known procedure. The `EXECUTE` keyword expands on that by also supporting asynchronous execution, dynamic execution, partitioned execution, and remote execution. The syntax is:

```
EXECUTE [ASYNC] <targetProcedure> | * (<actualParameters>)
[WHERE <procedureConstraint>]
[WITH partitionKey = <value>]
[<processedByClause>]
```

Asynchronous Execution

By default all procedures are executed "synchronously", meaning that any statements or expressions following the procedure invocation will not be executed until the procedure has completed and returned any value that it produces. By using the `ASYNC` modifier the caller can indicate that the procedure should be invoked *asynchronously*. When this option is used, the caller will continue on to the next statement while the procedure is executed, potentially concurrently. When invoked asynchronously procedures:

- Must be a statement, they cannot be used as part of an expression.
- Are invoked using **call by value** parameter passing semantics.
- Are subject to all workflow management (aka quota) controls.
- Have any value they produce discarded.

For example:

```
EXECUTE ASYNC Payroll.changeSalary("Some Employee", 30000.00)
// At this point the procedure may not have started, may have completed,
// or may be still executing.
```

Dynamic Execution

There are times when the name of the procedure to be executed is not known until runtime. In such cases, the procedure name must be selected dynamically. `EXECUTE` supports a special notation for selecting a procedure name dynamically:

```
EXECUTE * (<actualParameters>) WHERE <procedureConstraint>
```

In this dynamic form of `EXECUTE`, the procedure name is not specified after the `EXECUTE` keyword. Only the actual parameters are specified. The procedure is selected by querying the procedures resource for a set of instances that match the specified procedure constraint (this will always exclude private procedures). All of the matched procedures are executed concurrently and the statement will complete when all executions have completed.

The result of issuing an `EXECUTE` statement with a `WHERE` clause will **ALWAYS** be an array of procedure results, one entry for each procedure that is executed. The order of execution is indeterminate since each procedure may be executed asynchronously. If any of the procedures generates an exception then the statement will produce that same exception (ignoring any values produced by previously executed procedures).

As a simple example select a procedure based on the value in a variable:

```
var procName = "paul"
if (a == 1) {
    procName = "sharon"
}
EXECUTE * ("parm1", "parm2") WHERE name == procName
```

This will execute either a procedure named: **paul** or a procedure named: **sharon** depending on the value of the variable: **a.**

A convenient technique for selecting procedures on criteria other than name is to qualify the request on the value of **properties** in the **procedures** resource. The **properties** value can be set when the procedure is defined using the `WITH` clause. For example, if **properties** for a procedure contains an object in the following form:

```
{ role: "manager" }
```

The following `EXECUTE` statement will invoke all procedures that have the value "manager" assigned as their role:

```
EXECUTE * (1, 2, 3) WHERE ars_properties.role == "manager"
```

See the procedures section for a detailed description of assigning values to **properties**.

Partitioned Execution

In most circumstances the system is able to determine when a procedure will be accessing partitioned state and ensure that it has the information needed to properly route the request (by using the first parameter as a partitioning key). However, when invoking a `private` procedure from a Service Event Handler or from a service procedure with a conflicting state access modifier, this is not possible. Instead, the partitioning key must be explicitly provided using a `WITH` clause, such as:

```
var partitionKey = event.value.keyProperty
EXECUTE privateProc() WITH partitionKey = partitionKey
```

In this example we are extracting a value from the current event and using that to route the procedure execution to the proper partition (the proper value to use is obviously application dependent).

Remote Invocation

`EXECUTE` supports remote invocation via the standard `PROCESSED BY` clause.

RETURN

IMPORTANT NOTE – in most languages, the `RETURN` statement (or its equivalent) alters the flow of control and causes execution to immediately terminate. This is not true in VAIL (read below for details).

VAIL procedures return a value when their execution completes (this is true whether or not a return type is declared). By default, the return value is the value produced by the last VAIL statement to execute. This can often be ambiguous since not all VAIL statements have a defined result value. In order to make return values more explicit the `RETURN` statement may be used to specify the returned value. The syntax is:

```
return <valueExpression>
```

The `RETURN` statement **MUST** be the last statement in the body of the procedure. The `RETURN` statement **CANNOT** be used to terminate execution of the procedure earlier in the procedure body. For example, this is a legal use of `RETURN`:

```
procedure addProperty(obj Object, propName String, value String): Boolean
var propertyAdded = false
if (!obj.has(propName)) {
    obj[propName] = value
    propertyAdded = true
}
return propertyAdded
```

The procedure will return `true` if the property was added to the object and `false` if it was not.

However, this is an illegal use:

```
procedure addProperty(obj Object, propName String, value String): Boolean
if (obj.has(propName)) {
    return false // This is ILLEGAL!!  The code will keep running past this point.
}
obj[propName] = value
return true
```

This procedure will always set the property in the object and always return `true`.

Event Sending

The `PUBLISH` statement allows the user to publish events on a specified topic or deliver a message to a specified source. The general syntax of the `PUBLISH` statement is:

```
PUBLISH <message> TO [TOPIC <topic> [SCHEDULE <schedule>] |
                      SERVICE EVENT <eventTypeRef> |
                      SOURCE <source> [USING <config>]]
[<processedByClause>]
```

Sending via a Topic

The syntax for sending a user defined event via a topic is:

```
PUBLISH <eventData> TO TOPIC <topic> [SCHEDULE <schedule>]
```

The `eventData` is an expression whose result becomes the data attached to the event (accessible via the event's `value` property). The `topic` must be either a `String` literal or a variable reference containing the name of the target topic. For example:

```
var topicName = "/myTopic/subtopic/new"
PUBLISH { name: "granite" } TO TOPIC @topicName
```

Scheduled Events

The `PUBLISH` statement can be used to create a Scheduled Event which will be delivered via a topic at a future point in time (and possibly re-delivered periodically). This is done by adding the optional `SCHEDULE` clause to the `PUBLISH` statement. For example:

```
PUBLISH { name: "granite" } TO TOPIC "/myTopic/subtopic/new" SCHEDULE {interval: 1 minute}
```

This statement will publish the event from the previous example, but instead of doing so immediately, the event will be delivered 1 minute after the statement executes. The `schedule` argument must be an expression which produces an `Object`. This can be an `Object` literal (as shown above) or it could be any other expression such as a variable reference or the invocation of a procedure. The properties of the schedule `Object` vary depending on whether the event should be scheduled for one-time or periodic delivery.

- One time delivery: the object should have either the *interval* or *occursAt* property (but not both). They may also be marked as "transient".
    - interval – specifies the amount of time that the system should wait until publishing the event. The value is a number of milliseconds (so you can use an interval literal as shown in the example).
    - occursAt – specifies a `DateTime` value when the event should be published. If the value given is in the past, then the event will be delivered immediately.
    - isTransient – if `true` then the event is stored in memory only. This is more efficient, but restricts the event to occurring within 5 minutes.
- Periodic delivery: the expected properties are:
    - name – the name of the event. This is mandatory. If it refers to an already existing event then the event will be updated with the remaining properties.
    - periodic – a `Boolean` value which must be set to `true`.
    - interval – the period of time to wait between repeated publishing of the event. The property is required and should specify a number of milliseconds (so you can use an interval literal).
    - occursAt – specifies a `DateTime` value when the event should be published. The property is optional. If not given, the initial delivery of the event will be based on the interval. If the value given is in the past, then the event will be delivered immediately and its next delivery will be the first point in the future which aligns with the given interval. For example, if the value is 7 minutes in the past and the interval is 5 minutes, then the event will be delivered immediately and scheduled for delivery in 3 minutes (i.e., at that value plus 10 minutes, aka 2 intervals).

The previous example showed the creation of a one-time event, here we are creating a periodic event:

```
PUBLISH { name: "granite" } TO TOPIC "/myTopic/subtopic/new"
SCHEDULE {name: "graniteEvent", periodic: true, interval: 5 minutes}
```

This event will be initially delivered 5 minutes after the request and will be repeated once every 5 minutes until it is deleted.

Once created via a `PUBLISH` statement, scheduled events are stored as resource instances and can be manipulated as such. For example, to delete the previously created event you would use:

```
DELETE system.scheduledevents WHERE name == "graniteEvent"
```

One-time events will be deleted automatically once they are delivered. Periodic events have their *occursAt* property updated as they are delivered to reflect when they should next be delivered. Both of these operations take place asynchronously, so they may not have occurred prior to when the event is actually processed.

Sending to a Service

The syntax for sending a user defined event to a service event type is:

```
PUBLISH <eventData> TO SERVICE EVENT <eventTypeRef>
```

The `eventData` is an expression whose result becomes the data attached to the event (accessible via the event's `value` property). The `eventTypeRef` must be either a `String` literal or a variable reference containing a reference to a service event type. Service event type references are of the form `<serviceName>/<eventTypeName>`.

For example:

```
var eventType = "PipelineMonitor/recordData"
PUBLISH { segmentId: "123", pressure: 50.0 } TO SERVICE EVENT @eventType
```

The referenced service event type must be *inbound*.

Sending to an External System

Events can be sent from Vantiq to an external system via a source. The syntax for publishing a message to a source:

```
PUBLISH <sendRequest> TO SOURCE <source> USING <configuration>
```

The `sendRequest` is an expression which produces an `Object` value. The properties of the value are source dependent and can be found in the external source reference guides. The `source` must be either an identifier or a variable reference containing the name of the target source. The `configuration` in the USING clause is an expression which produces an `Object` value. The properties of this value are merged with those of the `sendRequest` to produce the final request to the source.

Here is an example for sending a "message" to an MQTT source:

```
PUBLISH {
    message: {
        id: Device.id,
        condition: "The device is no longer operating within allowed tolerances."
    }
} TO SOURCE mqttSource USING { topic: "/myTopics/device/outOfRange" }
```

Remote Invocation

`PUBLISH` supports remote invocation via the standard `PROCESSED BY` clause.

Data Manipulation

VAIL provides several SQL-like operations used to access the application's persistent state and external state. With the exception of the `UPSERT` statement, each of these can be used as either a statement or as part of an expression.

SELECT

The `SELECT` operation retrieves data from either the application's persistent or external state, as specified. The formal syntax for `SELECT` is:

```
SELECT [SEQUENCE | ARRAY | [EXACTLY] ONE] [* | <propertyBinding> [, <propertyBinding>]*]
FROM [SOURCE ] <target> [WITH <options>]
[WHERE <queryCondition>]
[GROUP BY <prop1> [, <propN>]*]
[ORDER BY <prop1> [ASC | DESC] [, <propN> [ASC | DESC]]*]
[<processedByClause> | <processingBlock> ]

processingBlock := [UNTIL <expression>] {
    <VAIL Statement>*
}
```

For example, the triggering event might contain a product id that must be mapped to a product name for use in rule processing. A `SELECT` operation can be used to obtain the product name from the product type.

```
var product = SELECT EXACTLY ONE name FROM Product
    WHERE productId == event.newValue.productId
var name = product.name
```

Here the `SELECT` is used on the right-hand side of a variable declaration statement.

Query Results

The default query result is determined by the context of the `SELECT` statement. In most cases, the result will be an `Array`. However, when used as the target of a FOR, MAP, or FILTER statement, `SELECT` returns a `Sequence`. `SELECT` also produces a `Sequence` if it is immediately followed by a *processingBlock* (and optional termination condition).

These defaults can be altered by explicitly declaring the result type of the `SELECT` statement as being one of:

- SEQUENCE – the `SELECT` produces a `Sequence` containing all results which match the query condition.
- ARRAY – the `SELECT` produces an `Array` of results. By default the `Array` will be limited to 1000 objects. This size can be increased, up to a maximum 10,000 objects, through the use of the *limit* query option.
- ONE – the `SELECT` produces either a single result or no result. If the query would produce more than one result then it will raise an exception. If the `EXACTLY` modifier is included then the `SELECT` will also raise an exception if no results would be returned.

Property Bindings

Property bindings (`propertyBindings`) are used to specify which properties of the target to retrieve or how to compute a property using a VAIL expression. The full syntax for a property binding is:

```
<propertyName> [= <propertyExpression>]
```

The `propertyExpression` is a standard VAIL expression with two exceptions

- The expression is evaluated in the context of the query results. This means that is has access to any properties of that data, but does not have access to any declared variables or procedure parameters.
- In addition to the standard VAIL built-in services and procedures, the expression can also use a special set of aggregate procedures.

For example, the query:

```
SELECT name, total = quantity * price FROM LineItem
```

Selects the *name* property from the **LineItem** resource type and computes the property *total* by multiplying the *quantity* and *price* properties.

Aggregate Procedures

The expression defining a property binding may include the following aggregate procedures. When used in conjunction with a GROUP BY clause the system will compute one value for each unique group key. Otherwise they will compute a single aggregate value over the entire `SELECT` results.

- **count(*propertyName*)** – computes the count of documents. If the optional *propertyName* argument is provided, then only documents with a non-null value for that property will be counted.
- **sum(*propertyName*)** – computes the sum of the values of *propertyName* from each document. The property values must be non-null and numeric; otherwise an error will result.
- **avg(*propertyName*)** – computes the average of the values of *propertyName* from each document. The property values must be non-null and numeric; otherwise an error will result.
- **first(*propertyName*)** – records the first value of *propertyName* seen in the result set. Most useful when combined with ORDER BY.
- **last(*propertyName*)** – records the last value of *propertyName* seen in the result set. Most useful when combined with ORDER BY.
- **min(*propertyName*)** – computes the minimum of the values of *propertyName* in the result set. The property values must be comparable in some way that makes sense.
- **max(*propertyName*)** – computes the maximum of the values of *propertyName* in the result set. The property values must be comparable in some way that makes sense.

Externally Managed Types & Property Expressions

When types in the Vantiq system reference external data managed via storage manager, VAIL enables access to data store specific functionality by pushing property expressions into the storage manager for evaluation. In this case the property expression can be any standard VAIL expression that the underlying data management system is capable of processing. Applications may for example leverage any available aggregate function in the external system, extending the range of aggregate functionality well beyond the above list. Note that the Vantiq system also pushes all property binding expressions. If the external system cannot handle `sum`, `avg`, `min`, `max`, etc. then attempting to use those aggregate procedures with the associated type will result in errors.

target

The *target* identifies a resource type or source defined in the application's data model to which the `SELECT` will be applied. The *target* optionally contains a `WITH` clause containing any configuration information required for access to the type or source. The general syntax of a *target* is:

```
FROM [SOURCE] <name> [AS <alias>] [WITH <options>]
```

The `name` must be either an identifier or a variable reference containing the name of the target type or source. The `alias` can be used to override the default result binding. The `with` clause provides additional options used to modify the query execution.

The detailed syntax for types and sources is detailed below.

Select from a Resource Type

Performing a `SELECT` on a resource type will return the resource instances that match any supplied query condition. The query results will be either a collection of `Object` values or a single `Object` value, as requested.

By default the result is bound to a variable with the same name as the resource type. For example, in the following `SELECT` statement:

```
SELECT FROM Sensor
```

The results are bound to the variable *Sensor*. If the resource type name contains a dot (`.`) then we will instead use the part of the name after the last dot. In this example:

```
SELECT FROM com.vantiq.Sensor
```

The results are also bound to the variable *Sensor*. An alias can be used to change this default binding, for example this:

```
SELECT FROM Sensor AS s
```

Binds the result into a variable named `s`.

Two options may be specified in the `WITH` clause when selecting from a resource type:

- **LIMIT** <max> - declares the maximum number of objects to be returned by the `SELECT`. If the `SELECT` produces more than *max* rows, only the first *max* rows are returned.
- **SKIP** <count> - causes the `SELECT` request to discard the first *count* objects returned by the `SELECT` and begin returning rows starting at object *count* + 1.

The `LIMIT` and `SKIP` options are very convenient for “paginating” the query results. For example, to retrieve the resource instances 20 at a time, set `LIMIT=20` and `SKIP=0` to retrieve the first 20 instances. To receive instances 21 through 40 set `LIMIT=20` and `SKIP=20`; for instances 41 through 60 set `LIMIT=20` and `SKIP=40`.

Select from a Source

`SELECT` can be used to query the data model of an external source by adding the `SOURCE` modifier. This causes the target to be interpreted as the name of the source to query. The `WITH` options are used to provide source implementation specific settings. For example,

```
SELECT FROM SOURCE sampleRestSource WITH PATH = "/myResource"
```

Here we are querying a REMOTE source and setting the request `PATH` to `"/myResource`.

Any data returned by the source will first be processed according to the source’s configuration and then will be formatted as specified by the `SELECT` operation. For example, if we have a source which produces JSON data and a given query to that source produces a JSON array, the system will first deserialize the JSON which results in an Array. From there, the system will then format the array based on the requested result type, as follows:

- `ARRAY` – the result will be placed in an `Array` instance. In this case the result will be an `Array` with a single element which is also an `Array` (since that’s what the source produced).
- `SEQUENCE` – the result will be a `Sequence` containing the `Array` produced by the source.
- `ONE` – the result will be the `Array` produced by the source.

It is encouraged that source results be processed using either `SEQUENCE` or `ONE`.

`SELECT` may be run against a type managed by a storage manager. In this case, if the storage manager supports transactions, the `WITH` clause can be used to specify the in-progress transaction ID value. For example:

```
SELECT FROM Sensor WITH transaction = <transaction ID value>
```

The Vantiq system in concert with the underlying storage manager ensures the select runs in the context of the specified transaction. See Storage Manager Transactions for more information.

WHERE clause

> The `WHERE` clause can only be used when the target of the `SELECT` is a resource type.

The `WHERE` clause is used to provide a `Boolean` expression (known as a “query condition”) which will be evaluated for every instance of the target resource type. Only instances for which the expression evaluates to `true` will be included in the results of the `SELECT` operation.

Query conditions have a SQL-like structure so they are familiar to most database users. The query condition consists of one or more comparison clauses connected by the logical operators `AND` and `OR` (note that these are synonyms for `&&` and `||` and that either syntax can be used). The syntax is:

```
<propertyReference> <comparisonOperator> <constantValue>
```

Property Reference

The `propertyReference` is either a single identifier or a reference to a nested property using the property reference operator. The reference is applied to each resource instance to obtain the value used in the comparison.

Comparison Operators

The following comparison operators are supported:

Operator	Description	Example
==	exact match	`prop == 42`
!=	does not match	`prop != 42`
>	greater than	`prop > 42`
>=	greater than or equal to	`prop >= 42`

Operator	Description	Example
<	less than	`prop < 42`
<=	less than or equal to	`prop <= 42`
in	must be in the list of values	`prop in ["paul", "fred", "melanie", "susan"]`
NOT[1]	unary negation	`NOT (prop in ["paul", "fred", "melanie", "susan"])`
![1]	unary negation	`!(prop == 42 AND prop2 < 5)`
regex[2]	must match the regular expression pattern	`prop regex "^p\w*"`
geoNear[4]	must be within the specified distance of a given `GeoJSON` value	see below
geoIntersects[4]	must intersect the given `GeoJSON` value	see below
geoWithin[4]	must be contained within the given `GeoJSON` value	see below

1. Query condition being negated must be enclosed in parentheses.
2. Target property must hold a `String`.
3. Can be applied to either `String` or `Array` values. When applied to a `String` any whitespace is ignored.
4. Target property must hold a `GeoJSON` value. The operator will ignore any (optionally) provided altitude.

Any query condition involving negation will match instances where the target property is not set. When querying for optional properties it can be useful to add the sub-clause: `AND prop != null`. For example, the query condition:

```
NOT (prop in ["paul", "fred", "melanie", "susan"]) AND prop != null
```

Will ignore any instances where `prop` is `null` or not set.

Constant Value

The `constantValue` may be any literal value for any VAIL type. Types with a distinct JSON representation can use that form in any query condition. The system will also perform any necessary type conversions.

Example Queries

Here are some example query conditions. The first example assume we are doing a `SELECT` on a resource type with the properties *weight* (in kg), *type*, and *location*.

```
weight >= 5000
```

would select instances that weigh more than 5000 (kg)

```
type == "truck"
```

would select instances that are identified as trucks.

```
type == "truck" and weight >= 5000
```

would select instances that are trucks weighing at least 5000 (kg).

```
not (type == "truck") and weight >= 5000
```

would select instances that are not trucks and that weigh at least 5000 (kg).

```
location geoNear {
    "$geometry": {
        type: "Point",
        coordinates: [ 90.834, 34.1987 ]
    },
    "$minDistance": 100,
    "$maxDistance": 1000
}
```

would select instances where the location is no less than 100 units and no more than 1000 units from the specified "point".

```
location geoWithin {
    "$geometry": {
        type: "polygon",
        coordinates: [[ [0, 0], [0, 10], [10, 10], [10, 0], [0, 0] ]]
    }
}
```

would select instances where the location is somewhere inside the area specified by the given "polygon".

GROUP BY

> The `GROUP BY` clause can only be used when the target of the `SELECT` is a resource type.

The `GROUP BY` clause is used to aggregate the results of a select for sets of instances having the same values for the specified properties. For example:

```
SELECT customerId, count=count() FROM Order GROUP BY customerId
```

Selects the count of orders for each unique value of "customerId". Without the GROUP BY clause the system would produce a single value for the aggregate function over all of the selected instances.

ORDER BY

> The `ORDER BY` clause can only be used when the target of the `SELECT` is a resource type.

The `ORDER BY` clause is used to sort the results of the `SELECT`.

```
SELECT * FROM <type> WHERE <constraint> ORDER BY <prop1> [ASC | DESC], ...
```

The results are sorted on the properties specified in the ORDER BY clause. The order of the sort of each property is specified as ascending with the ASC modifier or descending with the DESC modifier. The default ordering is ascending. Properties with the following types **MAY NOT** be referenced by an `ORDER BY` clause: `Decimal`, `Currency`, `ResourceReference`, `Object`, and `Array`.

Processing Results

To immediately process the results of a `SELECT` it can be followed by a *processingBlock*. This is a list of VAIL statements which will be executed once for each element of the sequence produced by the `SELECT` statement. The processing block may be proceeded by an optional `UNTIL` clause which will terminate the sequence if/when it evaluates to `true` for a given element.

Remote Invocation

`SELECT` supports remote invocation via the standard `PROCESSED BY` clause.

INSERT

The `INSERT` operation inserts a new instance of the specified resource type. The syntax is:

```
INSERT [INTO] <typeName>(<valueSet>) [WITH <options>]
[<processedByClause>]
```

The `typeName` must be either an identifier or a variable reference containing the name of the target resource type.

The `valueSet` may be:

- A list of *name, value* pairs of the form: `<propertyName>: <value>`.
- An `Object` literal.
- A variable containing an `Object` or `Array` value.

For example, this:

```
INSERT Customer(name: "aCustomerName", address: "1400 Main St, SomeCity AK")
```

will insert a customer consisting of a name and address into the **Customer** type.

An alternate syntax is available for those more comfortable with SQL:

```
INSERT INTO Customer(name = "aCustomerName", address = "1400 Main St, SomeCity AK")
```

The same example using a variable containing the object to insert:

```
var customer = { name: "aCustomerName", address: "1400 Main St, SomeCity AK" }
INSERT Customer(customer)
```

`INSERT` may be run against a type managed by a storage manager. In this case, if the storage manager supports transactions, the `WITH` clause can be used to specify the transaction ID value of an in-progress transaction. For example:

```
INSERT Customer(name: "customerName", address: "1400 Main St., SomeCity AK") WITH transaction = <transaction ID value>
```

The Vantiq system in concert with the underlying storage manager ensures the insert runs in the context of the specified transaction. See Storage Manager Transactions for more information.

Bulk INSERT

`INSERT` supports processing of more than one object per request (aka "bulk insert"). This is done by specifying a variable bound to an `Array` value. The elements in the `Array` must be `Object` values. Each one will be processed independently as a resource instance and will generate a distinct type event. An exception during processing may result in some instances being processed and others being ignored.

Remote Invocation

`INSERT` supports remote invocation via the standard `PROCESSED BY` clause.

UPDATE

The `UPDATE` operation updates the specified instances(s) of the specified resource type. The syntax is:

```
UPDATE <typeName>(<valueSet>) [WITH <options>] WHERE <queryCondition>
[<processedByClause> | <errorHandler> ]

errorHandler := ON ERROR(<errorVariable>) {
    <VAIL Statement>*
}
```

The `typeName` must be either an identifier or a variable reference containing the name of the target resource type.

The `valueSet` may be:

- A list of *name, value* pairs of the form: `<propertyName>: <value>`.
- An `Object` literal.
- A variable containing an `Object` value.

The inclusion of the system property *ars_version* in the `valueSet` will trigger the use of optimistic concurrency control (OCC) to manage simultaneous updates.

The `WHERE` clause follows the same form as described earlier for `SELECT`.

For example, to update the name of an existing customer found in the Customer type:

```
UPDATE Customer(name: "newCustomerName") WHERE name == "aCustomerName"
```

The existing Customer with the name "aCustomerName" is assigned the new name "newCustomerName".

The same example using a variable containing the object to update:

```
var customerUpdate = { name: "newCustomerName" }
UPDATE Customer(customerUpdate) WHERE name == "aCustomerName"
```

`UPDATE` may be run against a type managed by a storage manager. In this case, if the storage manager supports transactions, the `WITH` clause can be used to specify the transaction ID value for an in-progress transaction. For example:

```
UPDATE Customer(address: "11 Main St.") WITH transaction = <transaction ID value> WHERE name == "aCustomerName"
```

The Vantiq system in concert with the underlying storage manager ensures the update runs in the context of the specified transaction. See Storage Manager Transactions for more information.

UPDATE via Instance Query

If the `WHERE` clause specifies the resource id (and only the resource id) of the target resource type, then it is known as an "instance query". Using an instance query indicates that the `UPDATE` should effect *exactly* one instance. If no matching instance is found, the operation will throw an exception. In all other cases, if no instances are found the operation is merely ignored.

Bulk UPDATE

The previous examples show the use of a `WHERE` clause that matches only one instance. It is also possible to perform an update where the `WHERE` clause matches more than one instance (aka a "bulk" update). When this occurs the update will be applied to all matching instances *without* having to directly access them individually. Bulk `UPDATE` operations generate a single type event when they complete.

Handling Update Errors

The `UPDATE` operation may specify an `ON ERROR` block that is used to handle errors that occur during processing. This is especially useful when performing a bulk update as it allows for incremental processing of errors as they occur rather than terminating the operation on the first error.

Remote Invocation

`UPDATE` supports remote invocation via the standard `PROCESSED BY` clause.

UPSERT

The `UPSERT` statement performs one of two functions on the specified instance of the specified resource type:

- If the instance does not currently exist it is inserted into the type.
- If the instance already exists, it is updated with the specified values.

The syntax of the UPSERT statement:

```
UPSERT <typeName>({<propertyName>: <value> [, <propertyName> : <value>]...}) [WITH <options>]
[<processedByClause>]
```

The `typeName` must be either an identifier or a variable reference containing the name of the target resource type. The resource type specified **MUST** have a *naturalKey* declared. The instance specified in the `UPSERT` operation **MUST** include a value for all of the properties that are part of the *naturalKey*.

The `valueSet` may be:

- A list of *name, value* pairs of the form: `<propertyName>: <value>`.
- An `Object` literal.
- A variable containing an `Object` or `Array` value.

For example:

```
UPSERT Customer(name: "customerName", address: "11 Main Street")
```

Assuming **Customer** has declared *name* as its natural key and a **Customer** with *name* equal to `customerName` does not exist, this new **Customer** will be inserted. However, if the **Customer** instance does exist, its address will be updated to "11 Main Street".

`UPSERT` may be run against a type managed by a storage manager. In this case, if the storage manager supports transactions, the `WITH` clause can be used to specify the transaction ID value of an in-progress transaction. For example:

```
UPSERT Customer(name: "customerName", address: "11 Main Street") WITH transaction = <transaction ID value>
```

The Vantiq system in concert with the underlying storage manager ensures the upsert runs in the context of the specified transaction. See Storage Manager Transactions for more information.

Bulk UPSERT

`UPSERT` supports processing of more than one object per request (aka "bulk upsert"). This is done by specifying a variable bound to an `Array` value. The elements in the `Array` must be `Object` values. Each one will be processed independently as a resource instance and will generate a distinct type event. An exception during processing may result in some instances being processed and others being ignored.

Remote Invocation

`UPSERT` supports remote invocation via the standard `PROCESSED BY` clause.

DELETE

The `DELETE` operation deletes the specified instance(s) of the specified resource type. The syntax is:

```
DELETE <typeName> [WITH <options>] WHERE <queryConstraint>
[<processedByClause>]
```

The `typeName` must be either an identifier or a variable reference containing the name of the target resource type.

The `WHERE` clause follows the same form as described earlier for `SELECT`.

For example, to delete the customer updated in the `UPDATE` operation example:

```
DELETE Customer WHERE name == "newCustomerName"
```

`DELETE` may be run against a type managed by a storage manager. In this case, if the storage manager supports transactions, the `WITH` clause can be used to specify the transaction ID value of an in-progress transaction. For example:

```
DELETE Customer WITH transaction = <transaction ID value> WHERE name == "newCustomerName"
```

The Vantiq system in concert with the underlying storage manager ensures the delete runs in the context of the specified transaction. See Storage Manager Transactions for more information.

DELETE via Instance Query

If the `WHERE` clause specifies the resource id (and only the resource id) of the target resource type, then it is known as an "instance query". Using an instance query indicates that the `DELETE` should effect *exactly* one instance. If no matching instance is found, the operation will throw an exception. In all other cases, if no instances are found the operation is merely ignored.

Bulk DELETE

The previous examples show the use of a `WHERE` clause that matches only one instance. It is also possible to perform a `DELETE` where the `WHERE` clause matches more than one instance (aka a "bulk" delete). When this occurs the operation will delete all matching instances *without* having to directly access them individually. Bulk `DELETE` operations generate a single type event when they complete.

Remote Invocation

`DELETE` supports remote invocation via the standard `PROCESSED BY` clause.

Error Handling

VAIL uses exceptions to handle errors and other exceptional events. An exception is an event, which occurs during the execution of a rule or procedure, that disrupts the normal flow of processing. When an error occurs during execution, the system records information about the error in an exception and immediately begins processing that exception (this is also known as "throwing an exception"). Exceptions are `Object` values with the following properties:

- **code** – a `String` value used to identify the error (and its associated message template). The "error codes" for exceptions produced by Vantiq begin with `io.vantiq` (there are also some legacy error codes beginning with `com.accessg2.ag2rs`).
- **message** – a `String` value which contains the fully formatted form of the error message.
- **params** – an `Array` value which contains the substitution parameters that were used to construct the fully formatted error message. The items in the `Array` may be of any legal VAIL type.
- **isFormatted** – a `Boolean` value which indicates whether the message has already been formatted. This is primarily for internal use. Generally, any value that comes from an exception event will have `isFormatted` set to `true`. If VAIL code builds exception values by hand, `isFormatted` should correspond to the state of the `message` property. If `true` then the `message` property must contain the fully formatted message. If `false` then the `message` property must contain the message template (see the Java MessageFormat class) and the `params` property must contain the substitution parameters. Omission of the `isFormatted` property is equivalent to setting it to `false`.

After an exception occurs, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of procedures that had been called to get to the procedure where the error occurred. The list of procedures is known as the *call stack*. In the case of the processing of a rule, the rule will always be at the top of this stack.

The runtime system searches the call stack for a block of code that can handle the exception. This block of code is called an *error handler*. The search begins with the procedure in which the error occurred and proceeds through the call stack in the reverse order in which the procedures were called. When an appropriate handler is found, the runtime system passes the exception to the handler. A handler is appropriate if:

- It is defined by the `CATCH` block of a currently active `TRY` statement.
- It is defined by the `ON [REMOTE] ERROR` block of a remote operation and the exception was thrown by that operation.
- It is defined in the `ON ERROR` block of an `UPDATE` operation and the exception was thrown by that operation.

An exception that is not handled by a procedure in the call stack causes that procedure to terminate (the exception is said to have "escaped" the procedure). An exception that "escapes" from an executing rule, will cause that rule to terminate. When a rule or a non-private procedure terminates due to an exception, the system will create an instance of the **ArsRuleSnapshot** resource with the *entryType* property set to `error`.

Application Errors

There may be conditions under which an application may detect an error condition and wish to report the error and potentially terminate the current procedure/rule. This is accomplished by invoking the built-in **exception()** procedure. For example:

```
exception("my.exception.code", "My error message: {0}", ["help"])
```

Global Error Processing

The procedures and rules that are in the call stack when an exception occurs perform "local" handling of the error (and can even correct the results). However, sometimes it is useful to find out about errors that occur more globally. This can be accomplished by creating a rule to process the creation of the **ArsRuleSnapshot** instances mentioned previously. For example, the following rule will run any time an error record is created:

```
RULE exceptionHandler
WHEN EVENT OCCURS ON "/types/ArsRuleSnapshot/insert"
<logic for handling exception>
```

While such a rule can record information about the error and even alter the state of the application's data model, it cannot influence the local processing of the exception.

Logging

VAIL supports the generation of application specific "log" messages through a system supplied logging service. Access to this service is provided in all VAIL execution contexts by the implicitly defined *log* variable (there is no other way to obtain access to this service).

Log messages are associated with a "level" which controls which messages will be produced at any given time. The available logging levels are:

- `ERROR` – level 40
- `WARN` – level 30
- `INFO` – level 20
- `DEBUG` – level 10
- `TRACE` – level 0

These levels are "ordered" such that enabling a "lower" level (a level with a lower associated value) also enables all "higher" levels. So enabling the `DEBUG` level will cause the levels `INFO`, `WARN` and `ERROR` to also be enabled. The system's default logging level is `INFO`. Logging levels can be set globally for a namespace and specifically for individual VAIL resources. This is done using the debugconfigurations resource.

The service's interface is:

- **error(message, args)** - Logs the given message after first performing variable substitution using the given array of arguments. The message will only be logged if the current logging configuration specifies a level of `ERROR` or lower.
- **warn(message, args)** - Logs the given message after first performing variable substitution using the given array of arguments. The message will only be logged if the current logging configuration specifies a level of `WARN` or lower.
- **info(message, args)** - Logs the given message after first performing variable substitution using the given array of arguments. The message will only be logged if the current logging configuration specifies a level of `INFO` or lower.
- **debug(message, args)** - Logs the given message after first performing variable substitution using the given array of arguments. The message will only be logged if the current logging configuration specifies a level of `DEBUG` or lower.
- **trace(message, args)** - Logs the given message after first performing variable substitution using the given array of arguments. The message will only be logged if the current logging configuration specifies a level of `TRACE`.

In all of the above cases the message formatting is done using the SLF4J MessageFormatter.

Log messages are stored as instances of the `logs` resource. Logged messages are available via the Vantiq Resource API or may be viewed through the Vantiq IDE. When viewing the logs be aware that the log entries are first sorted on the unique *invocationId* associated with each execution of the rule or procedure and then sorted on the timestamp assigned to the log entry.

For example, given the following rule:

```
RULE myRule
WHEN EVENT OCCURS ON "/types/myType/insert"
log.info("An insert occurred on myType!")
```

The message `An insert occurred on myType!` will be logged if the log level is configured to be at least `INFO`.

Log messages may be parameterized as illustrated in the following example:

```
RULE myRule
WHEN EVENT OCCURS ON "/types/myType/insert"
log.debug("This message contains p1: {} and p2: {}", ["first", "second"])
```

The message 'This message contains p1: first and p2: second' will be logged if the log level is configured to be at least DEBUG. Starting from the beginning of the message, each pair of braces, '{}', will be replaced by the next parameter to the log invocation.

Because `INFO` is the default level, Vantiq recommends that any log statements used to produce log entries during development be written as `DEBUG` or `TRACE` messages. Otherwise, the log statements will add large amounts of output to the production log when the rule or procedure is deployed. This will make it more difficult to filter the production log output from the unintended development diagnostic messages.

Resource Definition

The resource definition statements support create/update/delete of the following Vantiq resources:

- rules
- procedures
- types
- topics
- sources

- nodes

For all statements the resource definition presented is the one defined in the Resource Reference Guide. Most of the time this is a JSON representation, but for the VAIL resources the definitions are given as plain text.

CREATE and ALTER

The `CREATE` and `ALTER` statements support creating new instances of each resource and/or modifying the definition of an existing instance. Vantiq implements automatic “upsert” for all of these resources. As a result, `CREATE` and `ALTER` are identical in both syntax and semantics, the only difference being the initial keyword.

> In the syntax below we show only the `CREATE` keyword, but this can be replaced with `ALTER`.

Remote Invocation

`CREATE` supports remote invocation via the standard `PROCESSED BY` clause.

CREATE RULE

Create an instance of the rules resource.

```
CREATE RULE (<ruleDefiningText>) [WITH active = [true | false]]
[<processedByClause>]
```

The `Boolean` *active* property may be included to set the initial state of the rule. A `true` value declares the rule to be active; a `false` value declares the rule to be inactive. By default rules are active immediately after they are created.

CREATE PROCEDURE

Create an instance of the procedures resource.

```
CREATE PROCEDURE (<procedureDefiningText>)
[<processedByClause>]
```

CREATE TYPE

Create an instance of the types resource.

```
CREATE TYPE (<typeDefiningObject>)
[<processedByClause>]
```

CREATE TOPIC

Create an instance of the topics resource.

```
CREATE TOPIC (<topicDefiningObject>)
[<processedByClause>]
```

CREATE SOURCE

Create an instance of the sources resource.

```
CREATE SOURCE (<sourceDefiningObject>)
[<processedByClause>]
```

CREATE NODE

Create an instance of the nodes resource.

```
CREATE NODE(<nodeDefiningObject>)
[<processedByClause>]
```

DROP

The drop statement is used to remove a resource instance from the system. The specific statements are listed below.

Remote Invocation

`DROP` supports remote invocation via the standard `PROCESSED BY` clause.

DROP RULE

```
DROP RULE <ruleName>
[<processedByClause>]
```

DROP SOURCE

```
DROP SOURCE <sourceName>
[<processedByClause>]
```

DROP TYPE

```
DROP TYPE <typeName>
[<processedByClause>]
```

DROP TOPIC

```
DROP TOPIC <topicName>
[<processedByClause>]
```

DROP NODE

```
DROP NODE <nodeName>
[<processedByClause>]
```

Distributed Processing

Vantiq supports distributed execution of VAIL statements which will cause them to be executed by a set of remote Vantiq servers. The statements which support distributed processing are:

- SELECT
- INSERT
- UPDATE
- UPSERT
- DELETE
- PUBLISH
- EXECUTE
- CREATE
- ALTER
- DROP

When invoked remotely, these operations **MUST** be used as statements (assuming they can be used in any other context).

Defining Remote Connections

The nodes resource represents connections to remote installations of the Vantiq server. Each node contains the URI used to connect to the target installation and the credentials to used to authenticate any requests. As a result, each node grants the ability to access a specific namespace hosted by the target installation. Any remotely executed statements operate in that namespace, using the authorizations granted by the node's credentials.

PROCESSED BY clause

Remotely executable statements all support the `PROCESSED BY` clause, which describes any remote processing for the statement. The placement of the clause can be found in the syntax details for each supported keyword. Regardless of placement, the `PROCESSED BY` clause uses the following syntax:

```
[AS <resultVar>] PROCESSED BY [[EXACTLY | AT LEAST] ONE] (<queryConstraint> | ALL)
[CONTEXT TO <contextVar>]
[WITHIN <timeInterval>]
[UNTIL <terminatingCondition>] {
    <statements>
}
[ON [REMOTE] ERROR (<errorVar>) {
    <statements>
}]
```

The following example illustrates the use of `PROCESSED BY` with an `INSERT` statement:

```
INSERT MyType(name: "myName") AS result PROCESSED BY ars_properties.region == "West"
CONTEXT TO resultCtx {
    log.info("Inserted object with id {} on node {}", [result._id, resultCtx.node.name])
} ON REMOTE ERROR (errorCtx) {
    log.warn("Failed to insert on node {} due to error: {}", [errorCtx.node.name, errorCtx.error.message])
}
```

Result Variable

`PROCESSED BY` can define the variable used to hold the results using an `AS` clause. Depending on the statement being executed this clause may or may not be required. For example, the `SELECT` statement defines a default variable binding for its results which will be used if no explicit result variable is defined. However, the `EXECUTE` statement has no such default which means that the `AS` clause is required to process non-singleton results (see processing results for more details).

The example defines the variable `result` which will hold the result of the `INSERT` statement on each target node.

Selecting Remote Targets

The `queryConstraint` defined as part of the `PROCESSED BY` clause selects the node instances where the statement will be executed. The `queryConstraint` is a logical expression of the same form used by the `WHERE` clause of the `SELECT` statement. The `queryConstraint` is applied to the **nodes** resource in the local namespace. Any instances that satisfy the constraint will be used to execute the statement and return the results to the local node. The query constraint also recognizes the keyword `ALL` which causes the statement to be executed by all remote nodes.

Ideally the constraint should specify some "logical" characteristic of the target node(s) rather than a physical one such as its URI. The system defined *ars_properties* property can be used to hold this kind of information.

In our previous example, the `INSERT` will execute on all nodes where the *region* key in *ars_properties* has been set to "West".

Another alternative is to make use of *tag values* used by the UI to facilitate searching of resource instances. These values are stored in the `Array` value *ars_properties.tags* as individual elements (each tag value will have a prepended `#`). So if we wanted to target all nodes with the tag of "factoryNode" we would use the query constraint:

```
ars_properties.tags == "#factoryNode"
```

Node Cardinality Modifiers

By default `PROCESSED BY` will execute the statement on all nodes that match the query constraint, whether that's one, none, or dozens. The following node cardinality modifiers can be used to more precisely control the expected number of target nodes:

- `ONE` – the collection of target nodes cannot have more than one member (but may have zero).
- `AT LEAST ONE` – the collection of target nodes must have at least one member (but may have more).
- `EXACTLY ONE` – the collection of target nodes must have one, and only one, member.

If the size of the target node collection does not conform to the specified limits, then the system will throw an exception.

In addition to enforcing the specified cardinality, the `ONE` and `EXACTLY ONE` modifiers also trigger processing of a singleton result.

Bounding Request Time

The `WITHIN` sub-clause can be used to limit the amount of time that the system will wait for a remote node to produce its results. The `timeInterval` is specified as an `Integer` number of milliseconds (usually using an interval literal). Any node that does not produce a result within the specified amount of time will instead produce an exception with the error code `io.vantiq.federation.request.timeout`.

Processing Results

By default the results of a statement executed via `PROCESSED BY` are returned as a sequence, unless the modifiers `ONE` or `EXACTLY ONE` are used. If either of those modifiers is present then the result is returned directly, as a singleton value.

Sequence Results

In this form, the remote statement produces a sequence of results which are processed by the supplied processing block. For all statements other than `SELECT`, the sequence will contain one result for each node on which the statement was executed. For `SELECT` the results from all nodes are merged into a single sequence which is then processed. In either case, the order of the returned results is non-deterministic. The `CONTEXT TO` sub-clause can be used to obtain information about the node that produced the current result. Processing of the sequence will continue until it has been exhausted, unless the processing block includes an `UNTIL` sub-clause.

UNTIL

The `UNTIL` sub-clause specifies a logical expression which is evaluated after each invocation of the processing or error handling blocks. If the expression evaluates to `true` then processing of the sequence will be immediately terminated. See the `FOR` statement for examples of `UNTIL` processing.

CONTEXT TO

The `CONTEXT TO` sub-clause is used to specify a variable which will hold the request status for the sequence item currently being processed. The status is an `Object` value with the following properties:

- **status** – a `String` value indicating the status of the request. This is one of either "SUCCESS" or "FAILURE" though in this case it will always be "SUCCESS".
- **node** – the instance of the **nodes** resource for the node which produced the item being processed.

For example, here is the status from the node "node3":

```
    {
        "node": { "name" : "node3", ... },
        "status": "SUCCESS",
    },
```

Singleton Result

A special form exists when the `PROCESSED BY` clause includes the node cardinality modifiers `ONE` or `EXACTLY ONE`. In this case, the returned result will be formatted as it would be for a local execution of the equivalent statement. For example, given the following `SELECT ONE` statements:

```
var local = SELECT ONE * FROM procedures WHERE (query clause)
var remote = SELECT ONE * FROM procedures WHERE (query clause) PROCESSED BY EXACTLY ONE (node query clause)
```

When called without a `PROCESSED BY` the `SELECT ONE` statement returns a single `Object` value if an instance matches the query clause, or `null` if no instance matched. Since the `PROCESSED BY` uses `EXACTLY ONE` to require a maximum of one target node its return will have the same format, instead of producing the default sequence result.

There are only two effects that the `PROCESSED BY` has on the result format in this case:

- An exception is still thrown when an invalid number of nodes is found.
- When using `PROCESSED BY ONE` specifically, statements other than `SELECT` will return `null` if no nodes match the query clause. `SELECT` will return an empty array in this case.

Handling Remote Errors

When errors occur during the execution of the request and/or the production of the resulting sequence, they will be ignored, *unless* the `PROCESSED BY` clause includes an error handler defined via the `ON REMOTE ERROR` sub-clause. When the error handler is defined, it will be invoked for any such error and the declared error variable will be populated with a status object with the following properties:

- **status** – the status of the request. This is one of either "SUCCESS" or "FAILURE", though in this case it will always be "FAILURE".
- **node** – the instance of the **nodes** resource for the node on which the error occurred.
- **error** – an exception describing the error.

For example, here is the error status for a request to "node3" which failed due to an authentication problem.

```
    {
        "node": { "name" : "node3", ... },
        "status": "FAILURE",
        "error":  {
            "code": "io.vantiq.authentication.failed",
            "message": "Supplied authentication information is invalid."
        }
    },
```

VAIL Built-in Services and Procedures

Type Specific Procedures

The VAIL standard types support a number of system defined (aka "built-in") procedures. Unless otherwise noted, these procedures can be called using either direct or method-style invocation. The documentation lists the parameters used for direct invocation.

String Procedures

The following built-in procedures are available to conveniently operate on `String` values.

Type Conversion Procedures

The following procedures can be used to explicitly perform conversion of `String` values to another standard type:

- **toInteger(str)** - convert the `String` to a corresponding `Integer` value. Raises an exception if *str* cannot be converted to an `Integer`.
- **toReal(str)** - convert the `String` to a corresponding `Real` value. Raises an exception if *str* cannot be converted to a `Real`.

- **toDecimal(str, scale)** - convert the `String` to a corresponding `Decimal` value. Raises an exception if *str* cannot be converted to a `Decimal` value. The parameter `scale` is optional and defaults to 0 if not specified.
- **toCurrency(str, scale)** - convert the `String` to a corresponding `Currency` value. Raises an exception if *str* cannot be converted to a `Currency` value. The parameter `scale` is optional and defaults to 0 if not specified.
- **toDate(str)** - convert the `String` to a corresponding `DateTime` value. Raises an exception if *str* cannot be converted to a `DateTime`. The target *str* must be in ISO-8601 format.
- **toBoolean(str)** - convert the `String` to a corresponding `Boolean` value. The resulting value will be `true` if the `String` is `"true"` (ignoring case). Otherwise the value will be `false`.

JavaScript Compatible Procedures

The following procedures are designed to provide syntax and semantics that aligns with the corresponding JavaScript String methods:

- **concat(str, str1, … strN)** - concatenate the `String` values *str* through *strN* in that order returning the concatenated `String`. Strings can also be concatenated using the `+` operator.
- **endsWith(str, substr)** - return true if *str* ends with the `String` specified in *substr*.
- **includes(str, substr)** - return true if the *substr* is found anywhere in *str*. The search always starts with the first character of *str*.
- **indexOf(str, substr)** - return the position in *str* of the first occurrence of the `String` specified in *substr*. Return -1 if no matching string is found. The search always starts with the first character of *str*.
- **lastIndexOf(str, substr)** - return the position in *str* of the last occurrence of the `String` specified in *substr*. Return -1 if no matching string is found. The search always starts with the last character of *str*.
- **length(str)** - return the length of *str* in code units (for ASCII characters the number of code units equals the number of characters in the string).
- **match(str, pattern)** - return an `Array` containing the `String` values within *str* that match the regular expression specified in *pattern*. The return value is an `Array` of `String` values, or `null` if no matches were found.
- **matches(str, pattern)** - return true if *str* is a full match of the regular expression specified in *pattern*.
- **repeat(str, count)** - return a `String` containing *str* duplicated *count* times.
- **replace(str, pattern, newSubStr)** - replace all occurrences of *pattern* found in *str* with *newSubStr*. The *pattern* may be a regular expression or a `String` that will be parsed as a regular expression. The *newSubStr* must be a `String`, though the implementation requires that the `\` character be escaped as if it were a regular expression. The JavaScript special replacement patterns are not supported in *newSubStr*.
- **search(str, pattern)** - return the index of the first match of *pattern* within *str*. The *pattern* is a regular expression. If no match is found, return -1.
- **slice(str, start, end)** - return the substring of *str* starting at the index specified in *start* up to but not including the character at the index specified in *end*. If *end* is not specified, returns the substring starting at the index specified in *start* and including all subsequent characters in *str*.
- **split(str, pattern, limit)** - return an `Array` of `String` values representing *str* divided into substrings by the specified separator *pattern*. The separator characters are removed. *limit* is the maximum number of splits to produce.
- **startsWith(str, substr)** - return true if *substr* matches the characters starting at index 0 in *str*.
- **substr(str, begin, length)** - return the substring of *str* starting at the index specified in *begin* and containing *length* characters. If the value of *length* is greater than the string's actual length then an `IndexOutOfBounds` exception is raised.
- **substring(str, begin, end)** - return the substring of *str* starting at the index specified in *begin*, up to but not including the index specified in *end*. If *end* is not specified, include all characters to the end of the string.
- **toLowerCase(str)** - convert all upper case characters in *str* to lower case and return the converted `String`.
- **toUpperCase(str)** - convert all lower case characters in *str* to upper case and return the converted `String`.
- **trim(str)** - return *str* with any leading or trailing whitespace removed.

Utility Procedures

The following are additional `String` utility procedures.

- **io.vantiq.text.Strings.format(str, args)** - return a `String` produced by substituting the values of an arbitrary number of parameters passed as an array into the pattern specified in *str*. See String Formatting for details. The deprecated `format` procedure is limited in the number of parameters it can accept (15). Because io.vantiq.text.Strings.format accepts an array of parameters, it is not limited in the number of values it can accept.
- **format(str, p0, …)** [deprecated] - return a `String` produced by substituting the values of the parameters *p0* through *pN* into the pattern specified in *str*. See String Formatting for details.
- **isAlpha(str)** - return `true` if *str* contains only letters (a-zA-Z).
- **isAlphaNumeric(str)** - return `true` if *str* contains only alpha and numeric characters.
- **isHexadecimal(str)** - return `true` if *str* contains only (a-fA-F0-9).
- **isName(str)** - return `true` if *str* is a valid resource identifier (using the regular expression `[a-zA-Z0-9\$_-]*`).
- **isNumeric(str)** - return `true` if *str* contains only digits (0-9).
- **isReal(str)** - return `true` if *str* represents a legal `Real` value (containing only digits (0-9) and no more than one dot (`.`)).
- **trimLeft(str)** - remove leading whitespace from *str*. Returns the trimmed `String`.
- **trimRight(str)** - remove trailing whitespace from *str*. Returns the trimmed `String`.
- **whitespaceTo(str, char)** - force all whitespace characters to the value specified in *char*. The *char* parameter must be a `String` containing a single character. Returns the modified `String`.

Regular Expressions

Regular expression patterns are supported for use in procedures that support `String` search using a regular expression.

A regular expression pattern may be created by calling the **RegExp()** procedure.

- **regExp(patternString)** - produces a regular expression pattern from the regular expression represented by the string value supplied as the *patternString* parameter. For more details on the format of the pattern string, see Pattern.

When using a `String` literal as a pattern string, remember that VAIL perform its own escaping for all `\` characters. This means that the sequence `\\` will be turned into `\` *before* it is sent to **regExp()**. For example, if you wanted to target every `\` character in a string you would need to specify the pattern `"\\\\"` which will be seen as `"\\"` by the procedure (which according to the Pattern documentation will match `\`).

The resulting regular expression can be used as the value of the *pattern* parameter for the match, replace and search procedures.

Integer Procedures

The following built-in procedures are available to conveniently operate on `Integer` values.

Type Conversion Procedures

The following procedures can be used to explicitly perform conversion of `Integer` values to another standard type:

- **toReal(intValue)** - convert the `Integer` to a corresponding `Real` value.
- **toString(intValue)** - convert the `Integer` to a corresponding `String` value.
- **toDate(intValue)** - convert the `Integer` to a corresponding `DateTime` value. The target `intValue` must be an epoch millisecond value.
- **toBoolean(intValue)** - convert the `Integer` to a corresponding `Boolean` value. The resulting value will be `true` if the `Integer` value is `1` and `false` if the `Integer` value is `0`. Otherwise, it will raise an exception.

Utility Procedures

The following are additional `Integer` utility procedures.

- **range(from, to, increment)** - produces an iterable range of `Integer` values starting with *from*, incrementing by *increment*, and ending with the number that is less than the value of *to*. This can be useful for iterating over a range of numbers in the `FOR` statement.

Real Procedures

The following built-in procedures are available to conveniently operate on `Real` values.

Type Conversion Procedures

The following procedures can be used to explicitly perform conversion of `Real` values to another standard type:

- **toInteger(realValue)** - convert the `Real` to a corresponding `Integer` value. The `Real` value will be truncated if necessary.
- **toString(realValue)** - convert the `Real` to a corresponding `String` value.
- **toDecimal(realValue, scale)** - convert the `Real` to a corresponding `Decimal` value. The parameter `scale` is optional and defaults to 0 if not specified.
- **toBoolean(realValue)** - convert the `Real` to a corresponding `Boolean` value. The resulting value will be `true` if the `Real` value is `1.0` and `false` if the `Real` value is `0.0`. Otherwise, it will raise an exception.

DateTime Procedures

The following built-in procedures are available to conveniently operate on `DateTime` values. Note that methods from Java `Instant` are also available, like *myDate.toEpochMilli()*.

Type Conversion Procedures

The following procedures can be used to explicitly perform conversion of `DateTime` values to another standard type:

- **toInteger(date)** - convert the `DateTime` to a corresponding `Integer` value as epoch milliseconds.
- **toString(date)** - convert the `DateTime` to a corresponding `String` value. The resulting value will be in ISO-8601 format.

JavaScript Compatible Procedures

The following procedures are designed to provide syntax and semantics that aligns with the corresponding JavaScript date methods:

- **getMilliseconds(date)** - the milliseconds component of the date (0-999)
- **getSeconds(date)** - the seconds component of the date (0-59)
- **getMinutes(date)** - the minutes component of the date (0-59)
- **getHours(date)** - the hour component of the date (0-23)
- **getDate(date)** - the day component of the date returned as the day within the month (1-31)
- **getMonth(date)** - the month within the year (1-12)
- **getFullYear(date)** - the year
- **getDay(date)** - the day of the week numbered 1 - Monday through 7 - Sunday. This differs from the typical Javascript numbering of the days which is 0 through 6 starting on Sunday.
- **getTime(date)** - the time in milliseconds since the epoch (standard Java epoch)

Utility Procedures

The following are additional `DateTime` utility procedures:

- **date(value, sourceRepresentation, destinationRepresentation)** - converts the date supplied as **value** from its current representation to the specified destination representation. For example, convert a date represented as a default date to an ISO date string.

`date(now(), "date", "ISO")`

Another example: convert a date from epochMilliseconds to epochDays effectively truncating the time from the date to leave just the day component:

```
var ems = date(now(), "date", "epochMilliseconds")
date(ems, "epochMilliseconds", "epochDays")
```

Another typical example is converting a date from the standard representation to epochMinutes:

`date(now(), "date", "epochMinutes")`

The **sourceRepresentation** and **destinationRepresentation** parameters may be one of the following string values:

- date
- ISO
- epochDays
- epochHours
- epochMinutes
- epochSeconds
- epochMilliseconds
- epochMicroseconds

- **now()** - returns a `DateTime` value representing the current date and time.
- **parseDate(str, dateFormat)** - convert the `String` to a corresponding `DateTime` value using the given date format pattern. See SimpleDateFormat for the supported format patterns. If no timezone is provided, then UTC is assumed. If no time fields are provided, the start of day is assumed.
- **truncateTo(date, chronoUnit)** – truncate the given `DateTime` value to the specified `chronoUnit` (given as a `String` value). Supports all legal Java ChronoUnit values, up to DAYS.
- **addMonth(date, monthOffset, preserveLastDay)** – return a *date* corresponding to the same day of the month as the original date, adjusted by *monthOffset* months. If *monthOffset* is negative, the date moves into the past. The parameter *preserveLastDay* is optional and defaults to `false`.

  For example, assuming `monthOffset = 1`:

  - If the same day exists on the next month,

    ```
    2024-01-11T10:20:33 -> 2024-02-11T10:20:33
    ```

  - If the current day is greater than the next month last day, the next month last day is returned,

    ```
    2024-01-31T10:20:33 -> 2024-02-29T10:20:33
    ```

  - If the current day is the last day of the month, the outcome depends on the preserveLastDay value,

    `2024-02-29T10:20:33 -> 2024-03-29T10:20:33` with *preserveLastDay* = `false`
    `2024-02-29T10:20:33 -> 2024-03-31T10:20:33` with *preserveLastDay* = `true`

- **addYear(date, yearOffset, preserveLastDay)** – return a *date* corresponding to the same day of the month as the original date, adjusted by *yearOffset* years. If *yearOffset* is negative, the date moves into the past. The parameter *preserveLastDay* is optional and defaults to `false`. The same logic as `addMonth` is applied.

The following procedures extract portions of a date (as opposed to truncating the low order components of the date as explained for the date() operation):

- **week(date)** - the week within the current year (1-52)
- **DoW(date)** - the day of the week numbered 1 - Monday through 7 - Sunday
- **HoW(date)** - the hour within the week numbered from 0
- **MoW(date)** - the minute within the week numbered from 0
- **DoY(date)** - the day within the year numbered from 1
- **MoD(date)** - the minute of the day (0-1439)
- **SoD(date)** - the second of the day (0-86399)
- **SoH(date)** - the second of the hour (0-3599)

Interval Procedures

- **stringToInterval(str)** - return the number of milliseconds in a String containing an interval literal as an `Integer` value. For example, `stringToInterval("1 second")` would evaluate to 1000.
- **durationInMillis(str)** - returns the number of milliseconds specified by the given ISO 8601 duration string. For example:

```
var millisecondsInOneDay = durationInMillis("P1D")
```

Intervals may be added and subtracted from dates using the following **methods** defined on the standard **date** type:

- **<date>.plusMillis(interval)**

- **<date>.minusMillis(interval)**

Example:

```
var twoDaysFromNow = now().plusMillis(2 days)
```

The use of the **method** style invocation is required because plusMillis is a method defined on the underlying date type and not a registered VAIL procedure.

Timezone Procedures

- **timezoneOffset(zoneId, date)** - return the timezone offset for the zoneId at the time specified. For example:

```
var offset = timezoneOffset("America/New_York", now())

// return "-07:00"
timezoneOffset("America/Los_Angeles", date("2022-05-08T10:00:00.000Z", "ISO", "date"))
// return "-08:00"
timezoneOffset("America/Los_Angeles", date("2022-11-06T10:00:00.000Z", "ISO", "date"))
```

- **getZoneIds()** - return the list of zoneIds where a zoneId is represented as a String value.

Object Procedures

The following built-in procedures are available to conveniently operate on `Object` values.

Type Conversion Procedures

The following procedures can be used to explicitly perform conversion of `Object` values to another standard type:

- **toObject(obj)** - assert that an “untyped” value is an instance of type `Object`. Raises an exception if *obj* is not actually an `Object` instance.
- **toDecimal(obj, scale)** - convert the `Object` to a corresponding `Decimal` value. Raises an exception if *obj* cannot be converted to a `Decimal` value. The parameter `scale` is optional and defaults to 0 if not specified.
- **toCurrency(obj, scale)** - convert the `Object` to a corresponding `Currency` value. Raises an exception if *obj* cannot be converted to a `Currency` value. The parameter `scale` is optional and defaults to 0 if not specified.
- **toGeoJSON(obj)** - convert the `Object` value to a corresponding `GeoJSON` value.

JavaScript Compatible Procedures

The following procedures are designed to provide syntax and semantics that aligns with the corresponding JavaScript Object methods:

- **clear(obj)** - remove all properties defined on *obj*. The result returned by clear is the null value.
- **deleteKey(obj, key)** - remove the property named *key* from *obj*. The result returned by deleteKey() is the value deleted, if found, else a null value.
- **has(obj, key)** - return true if *obj* contains a property named *key*.

The following are general, utility procedures:

- **Object.clone(obj)** - return a copy of the given `Object` *obj*. Allows you to modify fields without affecting the original `Object`. Must be invoked as a procedure and not using the “method-style”.

ResourceReference Procedures

The following built-in procedures are available to conveniently operate on `ResourceReference` values.

Utility Procedures

- **Utils.buildResourceRef(resourceType, instance)** - constructs a resource reference `String` for a resource instance. The procedure will verify that the `resourceType` exists and use its definition to extract the property `resourceId` from the given `instance`. When dealing with system resources be sure to specify the qualified resource name and not the underlying type name (for example use `system.users` and not `ArsUser`).
- **Utils.buildResourceRefMap(resourceType, instance)** - same as **Utils.buildResourceRef()**, but returns the resource reference in its internal `Object` format ({ resource: …, resourceId: …}).
- **Utils.get(ref)** - fetch the resource instance identified by the resource reference *ref*.
- **Utils.getNamespaceAndProfiles()** returns the namespace and profiles for the current user/token.
- **Utils.packageReference(ref, forceBase64)** returns a *packaged reference* used in sending and receiving Vantiq Documents, Images, or Videos to sources. The `ref` parameter is required, and it must be a resource reference. The `forceBase64` parameter is an optional boolean value, and is used (*i.e.*, set to `true`) only when the resulting data must be base64 encoded. If not specified, the data from `ref` will be base64 encoded only if required (based on context and the `fileType` of the referenced Vantiq document, image, or video.

Array Procedures

The following built-in procedures are available to conveniently operate on `Array` values.

JavaScript Compatible Procedures

The following procedures are designed to provide syntax and semantics that aligns with the corresponding JavaScript Array methods:

- **concat(arr, joinedArray)** - append the values in *joinedArray* to those in *arr* returning the combined `Array`.
- **fill(arr, value, begin, end)** - fill the entries in *arr* beginning at index *begin* and ending at index (*end* - 1) with *value*. The result returned by fill() is the filled `Array`.
- **includes(arr, element)** - return `true` if the *element* is a member of *arr*.
- **indexOf(arr, element)** - return the index of the first entry in *arr* that contains *element*.
- **join(arr, separator)** - return a `String` consisting of the concatenation of all the entries in *arr* converted to `String` and separated by the `String` specified in *separator*.
- **lastIndexOf(arr, element)** - return the index of the last entry in *arr* that contains *element*.
- **length(arr)** - returns the length of *arr*.
- **pop(arr)** - return the last element of *arr* and remove it from *arr*.
- **push(arr, element)** - add element to the end of *arr*. Return `true` if the element is added.
- **reverse(arr)** - reverse the elements in *arr*.
- **shift(arr)** - return the first element in *arr* removing the element from *arr*.
- **slice(arr, begin, end)** - return an `Array` containing the elements of *arr* starting at the index specified in *begin* and ending at the index specified as (*end* - 1)
- **sort(arr)** - sort the elements in *arr* in ascending order.
- **sort(arr, property)** - sort the `Object` values in *arr* based on the ascending order of the value for the given *property*. For example, [{“a”:7},{“a”:3},{“a”:5}] will be sorted as [{“a”:3},{“a”:5},{“a”:7}] when sort(“a”) is called
- **splice(arr, begin, deleteCount, list)** - at the index specified by *begin*, delete the number of elements specified in *deleteCount* and then add the elements in *list*.
- **unshift(arr, element)** - add element as the first entry in *arr* shifting all existing elements such that their index is one greater than before the unshift call. No return value.

Utility Procedures

The following are additional `Array` utility procedures.

- **size(arr)** - returns the length of *arr*
- **removeAt(arr, index)** - remove the element at the specified *index* from *arr*. Returns the item removed from the `Array`.
- **remove(arr, element)** - remove all occurrences of *element* from *arr*. Returns `true` if the element is removed.
- **removeElement(arr, element)** - remove the first occurrence of *element* from array. Returns `true` if the element is removed.
- **addAll(arr, elementArr)** - add all the elements in *elementArr* to *arr*. Returns the `true` if the content of the array is changed.
- **flatten(arr)** - if *arr* contains arrays, remove all the contained arrays and add their elements to *arr*. Returns the modified `Array`.
- **clear(arr)** - remove all elements from *arr*. The result returned by clear() is the `null` value.
- **toSequence(arr)** – convert the given value into a Sequence value using standard type conversion rules.
- **toArray(arr)** – convert the given value into an Array value using standard type conversion rules.

General Use Procedures

For convenience, a number of commonly useful procedures are predefined.

JavaScript Compatible Procedures

The following procedures are designed to provide syntax and semantics that aligns with the corresponding JavaScript global methods:

URI Procedures

- **decodeUri(encodedUri)** - decode a URI previously encoded by encodeUri().
- **decodeUriComponent(encodedUriComponent)** - decode a URI component previously encoded by **encodeUriComponent()**.
- **encodeUri(str)** - encode the complete URI represented by *str*.
- **encodeUriComponent(str)** - encode the URI component represented by *str*.
- **escape(str)** - replace >, <, &, " with HTML entities like &gt;, &lt;, &amp;, &quot; etc.

Content Parsing Procedures

- **stringify(target, prettyPrint)** - converts *target* into a `String` containing its JSON representation. If *prettyPrint* is `true`, the resulting JSON `String` is formatted in a more easily read form. If *prettyPrint* is `false` the resulting `String` contains the minimum number of characters required to represent the JSON object.
- **parse(str)** - parses the contents of *str* as a JSON `String` producing the corresponding standard type value.
- **parseXml(str)** - parses the contents of *str* as an XML `String` producing an `Object` representation of the XML structure that can be manipulated using the Groovy GPath notation.
- **parseInt(str)** - parses the contents of *str* to produce an `Integer`. Equivalent to **toInteger**.
- **parseFloat(str)** - parses the contents of *str* to produce a `Real`. Equivalent to **toReal**.

Miscellaneous Procedures

- **typeOf(target)** - determine the runtime type of *target*. The value returned is a `String` containing the name of one of the VAIL standard types.
- **classOf(obj, fullyQualified)** - determine the underlying class of *obj*. Intended for in-depth debugging of GenAIFlow code. The value returned is a `String` containing the name of the class of *obj*, by default the simple class name (e.g. “String”). If *fullyQualified* is `true`, it will return the full name (e.g. “java.lang.String”).
- **uuid()** - generates a unique 128-bit identifier.

- **geoDistance(pointA, pointB)** - computes the spherical distance between pointA and pointB and returns the result in meters. The format for points A and B can be either GeoJSON like {“type”: “Point”, “coordinates”: [0, 0]} or can be in WKT string format like “POINT(0 0)”. (Note that this procedure does not take altitude (if specified) into account when computing the spherical distance.)

Utility Procedures

- **exception(code, message, params)** - *code* is a `String` containing an error code and *message* is a `String` containing the associated error message. The *message* value will be processed using Message Formatter along with the supplied *params*. The result of calling the **exception()** procedure is that the system will construct and throw an exception.
- **rethrow(exception)** - *exception* is an `Object` consisting of all the key/value pairs for an error value in the Vantiq system (see Error Handling). Typically, values for the exception parameter come from a try / catch code block in VAIL. **rethrow()** is useful when you want to catch an exception, perform some checks or other processing, and then rethrow it. Prior to the introduction of **rethrow()**, developers would call **exception()** passing the individual values from the caught error value to accomplish this. That is an antipattern and should be avoided in favor of using **rethrow()**. Doing so avoids double formatting of the error message which can be problematic for messages that contain unescaped special characters after the initial formatting completes. The **exception()** built-in should only be used when creating a new exception in VAIL.
- **threshold(previousValue, newValue, thresholdValue, direction)** - returns `true` if the *thresholdValue* specified has been crossed between the *previousValue* and the *newValue*. The *direction* value is a `String` containing one of:
  - `increasing` - the new value is greater than the old value
  - `decreasing` - the new value is less than the old value
  - `either` - the threshold was crossed in either the increasing or decreasing direction.

Built-In Services

As described in the Services Section of the Resource Guide, services are a resource used to organize collections of procedures. The following is a list of Built-In Services.

A2A

The **io.vantiq.ai.A2A** service provides utility procedures related to the Agent to Agent protocol.

- **A2A.getAgentCard(agentName String Required)** – returns the Agent Card for the named agent.

The following types are defined as part of the A2A framework:

- **io.vantiq.a2a.Task** – represents a stateful unit of work being processed by the agent. Can only be called when processing requests that were received via the standard agent framework. In the current release this will only occur when an agent is invoked during plan execution.

Agent

The **io.vantiq.ai.Agent** service provides procedures which provide support for the development of GenAI Agents.

The procedures in this service make use the following types:

- **io.vantiq.ai.PlanningGraph** – used to represent a plan to be executed as a directed, acyclic graph (DAG).
  - **nodes** (io.vantiq.ai.PlanningGraphNode[]) – an array of the graph nodes.
  - **edges** (io.vantiq.ai.PlanningGraphEdge[]) – an array of edges which describe the connections between the nodes.
- **io.vantiq.ai.PlanningGraphNode** – represents a node in a planning graph. The properties are:
  - **id** (String) – a unique identifier for the node.
  - **tool** (String) – the tool to use when computing the node’s value. The legal values are:
    - `user` – used to request input from the initial caller of the agent.
    - `llm` – used to submit a prompt to an LLM for processing.
    - `<name>` – used to invoke a named tool. The exact interpretation depends on the value of *toolType*.
  - **toolType** (String) – indicates how to interpret tool names. A value of `agent` (the default) means that the tool name refers to an agent which must exist. A value of `skill` means that the tool name refers to a skill provided by the current agent.
  - **input** – the input value to be provided to the invoked tool. The input may contain a reference to the output of another node in the form of a “variable”.
  - **desc** – a description of the goal for this node.
- **io.vantiq.ai.PlanningGraphEdge** – represents a directed edge between two nodes in the graph. The properties are:
  - **source** – the id of the source node for the edge.
  - **target** – the id of the target node for the edge.

The service procedures are:

- **Agent.executePlan(planGraph io.vantiq.ai.PlanningGraph Required, userRequest Any, requestContext String): Any** – Executes the given plan which is represented by a DAG. Execution starts at the roots and proceeds until a value has been calculated for each node. This is done by invoking the node’s *tool* using the specified *input*. The procedure’s return value is an `Object` with the following properties:
  - **evidence** – the values computed for each node in the graph. This is an `Object` with one property for each node id.
  - **response** (optional) – the final response to the user’s request. Only present if the user’s request was provided. This will be a `String` which is suitable for display to the user.
  - **inputs** (optional) – the input values used to create the response. Only present if the user’s request was provided. This will be an `Any[]` with one item for each leaf node value.

If a *requestContext* value is provided, then it will be used when calling the Agent's User Request Bridge Procedure. * **Agent.getCurrentTask(): io.vantiq.a2a.Task** – returns the currently active agent task. Only relevant when the agent is invoked as a planning tool. Will throw an exception if the agent is not currently processing a task. * **Agent.requestUserInput(requestData Any Required, requestContext Any): Any** – used to request input from the initiator of an agent request. The value of *requestData* will be provided and the resulting response will be returned by this procedure. If a value is provided for *requestContext* then it will be used when calling the Agent's User Request Bridge Procedure.

Callback

The **io.vantiq.Callback** service provides the means for a client to register a "callback" which can be invoked by another service. Once the callback has completed, it can return a value to the "caller" which will resume processing. Basically it provides the same capability as a service procedure for cases where the target of the invocation is *not* a service (most typically it will be a Vantiq Client). It contains the following procedures:

- **Callback.register(callbackId String Required): String** – Register a new callback with the given *callbackId*. The id value must be unique for the current namespace. The return value is the event path on which invocation events will arrive. The caller is expected to subscribe for these events and respond to them when received. The invocation events have the following properties:
  - `callbackId` – the id of the callback being invoked.
  - `invocationId` – the id of this invocation.
  - `data` – the data provided when the callback was invoked.
- **Callback.invoke(callbackId String Required, data Any Required, timeout Integer): Any** – Invoke the identified callback, sending the supplied data. The optional *timeout* value is an interval in milliseconds which defines how long the caller will wait for a response. By default the caller will wait indefinitely. If the invocation times out, it will generate an exception with the error code `io.vantiq.rulemgr.execution.callback.timeout`. The return value is the response provided by the callback invocation.
- **Callback.sendResponse(callbackId String Required, invocationId String, Required, response Any Required)** – Respond to the given invocation of the specified callback. The value provided by *response* will be sent to the caller of the callback.

Chat

The **Chat** service implements a collection of procedures for interacting with the messaging component of the Vantiq Mobile Apps. It contains the following procedures:

- **Chat.addUser(chatId, usersToBeAdded)** - adds the users in *usersToBeAdded* to the chatroom (**ArsChat** instance) identified by *chatId*
- **Chat.createChatroom(situationId, topic, users)** - creates a chatroom with the name *topic* related to the situation identified by *situationId* and sends an initial message to all users in the *users* list letting them know the chatroom was initialized.
- **Chat.deleteChatroom(chatId)** - delete the **ArsChat** instance identified by *chatId*.
- **Chat.removeUser(chatId, usersToRemove)** - removes all users in the *usersToRemove* list from the chatroom identified by *chatId*.
- **Chat.sendMessage(chatId, type, msg, metadata, userList, sender)** - send a message to the chatroom identified by *chatId* with *msg* as the message text. For a textual msg specify *type* as "text", for other message types *metadata* can be used to provide additional details about the message contents. *userList* can be used to specify the list of users to notify of the new message (if an empty list or null is provided, all users are notified). Override the senders name with *sender*.
- **Chat.updateChatrooms(username, oldChatRooms)** - used by the mobile apps to fetch changes in the chatrooms that a user specified by *username* is subscribed to.

ChatMessage

The **io.vantiq.ai.ChatMessage** service implements builders for instances of the **io.vantiq.ai.ChatMessage** schema type. This type has the following properties:

- **type** (String) – the type of message being provided. Legal values are:
  - `human` - the message contains input from a user.
  - `ai` - the message contains a response from an LLM.
  - `system` - the message contains instructions to an LLM to help guide/constrain its output.
  - `chat` - the message contains a chat response (overlaps with `human`).
  - `function` - (*deprecated*) the message contains the results of a requested function invocation.
  - `tool` - the message contains the results of a requested tool invocation.
- **content** (String, String[], or Object[]) – the (optional) message content.
- **ars_properties** (Object) - a collection of name/value pairs which can be used to store application specific information.
- **functionName** (String) - (*deprecated*) the name of a function. Only used when the message represents either a request to invoke a function or the result of a function invocation.
- **functionArguments** (Object) - (*deprecated*) an Object whose properties represent the values to use for named function arguments. Only used when the message represents a function call request.
- **name** (String) - the name of a tool. Specified within a *tool call* description in an LLM response message, and as the tool name in a tool result message.
- **tool_call_id** (String) - represent a unique tool identifier as provided by a *tool call* description within an LLM response message. This value is used by the LLM to correlate the tool invocation request (*tool call*) with the tool invocation result message.

ChatMessage instances are used by the ConversationalMemory, LLM and SemanticSearch services.

ChatMessage Builders

To assist in the creation of Chat Messages, Vantiq offers several "builders" which are built-in procedures used to create specific types of chat messages. The most commonly used builders are:

- **ChatMessage.buildHumanMessage(content Required, properties Object)** - builds a message of type `human` with the given content.

- **ChatMessage.buildSystemMessage(content Required, properties Object)** - build a message of type `system` with the given content.

These let you construct a prompt for the LLM to be interpreted as either a user request (human messages) or precondition/instruction for the LLM (system messages). In the simplest case, the content provided is a single string, containing a "prompt". However, the builders also supports arrays of strings or `Object` instances. The latter is used to construct more complex prompts if/when a model supports them. For example, "multi-modal" models (such as OpenAI's GPT-4o) provide the ability to include image data in the prompts for analysis. The following VAIL code constructs a message which includes both a text prompt and an image reference (in this case using a URL):

```
var msg = io.vantiq.ai.ChatMessage.buildHumanMessage([
    {type: "text", text: prompt},
    {type: "image_url", image_url: {url: imageUrl}},
])
```

The `image_url.url` value can be provided as a ResourceReference referring to a Document, Image, or Temp Blob instance. In these cases, the referenced resource will be materialized (subject to the `documentExpansion` quota) and submitted as part of the prompt. Information about the `documentExpansion` quota can be found in the Administrators Reference Guide.

The *properties* parameter is optional. When present, the values are placed in the *ars_properties* property of the Chat Message.

To help support the development of LLM "Tools", we also provide a builder used to communicate the output of a tool to the LLM:

- **ChatMessage.buildToolResultMessage(name String Required, content Required, toolCallId String Required, properties Object)** - constructs a message of type `tool` with the given content. See the LLM Reference Guide Tool section for an example.
- **ChatMessage.buildFunctionResultMessage(functionName String Required, result Required, properties Object)** - (*deprecated*) constructs a message of type `function` with the given result.

The service also defines builders which are more specialized and are not typically used outside of very specialized use cases or testing:

- **ChatMessage.buildAIMessage(content Required, properties Object)** - constructs a message of type `ai` with the given content.
- **ChatMessage.buildChatMessage(content Required, properties Object)** - constructs a message of type `chat` with the given content.
- **ChatMessage.buildFunctionCallMessage(functionName String Required, arguments Object, properties Object)** - constructs a message of type `ai` which represents a request to invoke the named function with the supplied arguments. **NOTE – typically messages of this type will be returned by an LLM and not fabricated by the application.**

Client

The **Client** service contains procedures for sending clients to mobile devices using push notifications. It contains the following procedures:

- **Client.sendByName(pushSourceName, title, subtitle, users, clientName)** - send a notification to the specified *users* using the push source identified by *pushSourceName*, which launches a client when opened. The *title* and *subtitle* will be used for the title and message of the push notification sent to the users phone.

Concurrent

The **Concurrent** service supports the creation of concurrent data structures which assist in the management of the application's in-memory state. It contains the following procedures:

- **Concurrent.Map()** - creates an instance of a concurrent map data structure. The instance conforms to the VAIL `Map` type and supports the following procedures (which must use method style invocation):
    - **size(): Integer** - returns the number of key/value pairs held by the map.
    - **isEmpty(): Boolean** - return `true` when there are no key/value pairs in the map.
    - **containsKey(key Object): Boolean** - return `true` if the map contains an entry for the given `key`.
    - **containsValue(value Object): Boolean** - return `true` if the map contains an entry with the given `value`.
    - **get(key Object): Object** - return the `value` associated with the given *key*. If the map contains no such mapping then return `null`.
    - **getOrDefault(key Object, defaultValue Object): Object** - return the *value* associated with the given *key*. If the map contains no such mapping, return `defaultValue`.
    - **keySet()** - return an iterable sequence of the keys held by the map.
    - **values()** - return an iterable sequence of the values held by the map.
    - **entrySet()** - return an iterable sequence of the map "entries" (each entry holds a *key* and its associated *value*).
    - **putIfAbsent(key Object, value Object): Object** - adds a mapping for the given *key* to the given *value* if there is no existing mapping for the *key*. Returns `null` if the mapping did not previously exist or the current value if it does.
    - **remove(key Object, value Object): Boolean** - removes the mapping for the given *key* if the current *value* matches the one given. Returns `true` if the mapping was removed.
    - **compute(key, remappingFunction): Object** - used to perform a thread-safe update to the value associated with a specific *key* in the map. The *remappingFunction* must be a function expression of the form: `(key, value) => { <VAIL statements>* }`. The function is invoked with the supplied *key* and the *value* current associated with that key (which may be `null`). The function is expected to return the updated *value* for the *key* or `null` if the *key* should be removed from the map. Returns the new *value* for the *key* or `null` if there is none.
    - **computeIfAbsent(key, remappingFunction): Object** - used to create a new value for a *key* that does not current exist in the map. The *remappingFunction* must be a function expression of the form: `key => { <VAIL statements>* }`. The function is invoked with the supplied *key*, but only if there is currently no value associated with it in the map. The function is expected to return a new *value* for the *key*. Returns the new *value* for the *key* or `null` if there is none.
    - **computeIfPresent(key, remappingFunction): Object** - used to perform a thread-safe update to the value associated with a specific *key* in the map. The *remappingFunction* must be a function expression of the form: `(key, value) => { <VAIL statements>* }`. The function is

invoked with the supplied *key* and the *value* current associated with that key, but only in the case where that value exists. The function is expected to return the updated *value* for the *key* or `null` if the *key* should be removed from the map. Returns the new *value* for the *key* or `null` if there is none.
- **merge(key Object, value Object, remappingFunction): Object** - if the specified *key* does not have a mapping, associates it with the given *value*. Otherwise, the supplied *remappingFunction* is invoked with the supplied *key* and the existing *value*. The function is expected to return the updated *value* for the *key* or `null` if the *key* should be removed from the map. Returns the new *value* for the *key* or `null` if there is none. The *remappingFunction* must be a function expression of the form: `(key, value) => { <VAIL statements>* }`.
- **put(key Object, value Object): Object** - sets the *value* associated with the given *key*. Returns the previous *value* associated with the *key* (or `null` if there wasn't one).
- **clear()** - removes all key/value mappings from the map.

- **Concurrent.Cache(maximumSize, expireAfterAccessMs, expireAfterWriteMs)** - creates an instance of an auto-evicting cache. The parameters are:

  - *maximumSize* - an (optional) Integer specifying the maximum number of entries that the cache will hold. Once this maximum size is reached, existing entries will be evicted to make room for new one. A value of `-1` (the default) means that no maximum size is set.
  - *expireAfterAccessMs* - an (optional) Integer specifying the access "time to live" of the entry, in milliseconds. If the time between the last access of the value and the current time exceeds the configured value, then the entry will be evicted from the cache. A value of `-1` (the default) means that no access TTL is enforced.
  - *expireAfterWriteMs* - an (optional) Integer specifying the update "time to live" of the entry, in milliseconds. If the time between the last update of the value and the current time exceeds the configured value, then the entry will be evicted from the cache. A value of `-1` (the default) means that no update TTL is enforced.

  The instance conforms to the VAIL `Map` type and supports all of the procedures supported by `Concurrent.Map()`. In addition, it also supports the following procedures (which must use method style invocation):

  - **notifyOnEviction(topicName)** - configures the cache instance to send an event any time an entry is evicted from the cache. The event will be sent to the specified topic (a value of `null` will disable notifications). The event data is a VAIL Object with the following properties:
    - **key** – the key of the evicted entry
    - **value** – the value of the evicted entry

  WARNING: If an exception occurs inside a Concurrent Cache's `compute*()` block the data being operated on may be lost. Make sure to validate any values before use and to avoid calls to `exception()`.

- **Concurrent.Lock()** - creates an instance of a lock that can be used to synchronize concurrent access to a block of code. The lock supports the following procedures (which must use method style invocation):

  - **synchronize(function)** - used to execute a block of code while holding an exclusive lock. The *function* parameter is any function expression. The lock must be acquired before the function will be executed and will be released once execution completes.
- **Concurrent.Value()** - creates an instance of a concurrent value-storing data structure. The instance conforms to the VAIL `Value` type and supports the following procedures (which must use method style invocation):
  - **getValue(): Object** - returns the currently stored value.
  - **getValueOrDefault(defaultValue Object): Object** - returns the currently stored value. If the currently stored value is `null` (aka unset), then `defaultValue` is returned instead.
  - **setValue(value Object): Object** - sets the stored *value* to the one given. Returns the stored *value*.
  - **updateAndGet(updateFunction): Object** - perform an update of the stored value and return the result. The *updateFunction* must be a function expression of the form: `(value) => { <VAIL statements>* }`. The *updateFunction* is expected to return a value which will become the one stored. Returns the resulting stored value.
  - **getAndUpdate(updateFunction): Object** – perform an update of the stored value and return the value prior to the update. The *updateFunction* must be a function expression of the form: `(value) => { <VAIL statements>* }`. The *updateFunction* is expected to return a value which will become the one stored. Returns the value stored prior to the update.

Context

The **Context** service contains a collection of procedures which provide access to the current execution context. It contains the following procedures:

- **Context.authProvider()** - returns the name of the identity provider. When the system is using its internal identity and authentication provider, the return value is "Vantiq".
- **Context.buildInfo()** - returns an `Object` containing the build version, date, and commit SHA for the Vantiq server.
- **Context.email()** – returns the email address (as a `String`) of the user on whose behalf the execution is occurring. If the user has no email address then the value will be `null`.
- **Context.isEmailVerified()** – returns a `Boolean` value which indicates whether the user's email address has been verified by the authorization provider. This value is only present when using OAuth to perform authorization.
- **Context.isInternalAuth()** - returns true if the system is using built-in identity management and authentication. Returns `false` if the system is using an external identity provider like Keycloak.
- **Context.isPartitionKeyOwner(partitionKey Required)** - returns `true` if the caller is running in a context where they "own" the given *partitionKey*. This procedure is designed to be called from a multi partition procedure or the partitioned state initializer. It is illegal to use from outside the context of a service.
- **Context.licenseInfo()** - returns the Vantiq server license as a set of claims. Expiration time (`exp` claim) and issuance time (`iat` claim) represent a date as seconds since Epoch. License information is only accessible to a sys admin user.
- **Context.namespace()** - returns the namespace in which the current execution is occurring as a string.
- **Context.preferredUsername()** - returns the preferredUsername (as a `String`) of the user on whose behalf the execution is occurring. The `preferredUsername` contains the name by which the user is most commonly known.
- **Context.profiles()** - returns the profiles that the current execution context is using to authorize all operations. This will be an array of strings, with each entry in the array being a namespace-prefixed (normalized) profile name.

- **Context.serverUri()** - returns the URI of the local server in which the current execution is occurring.
- **Context.username()** - returns the username (as a `String`) of the user on whose behalf the execution is occurring. The *username* is the resource id for the Users resource. It is guaranteed to be both stable and unique; however, it is not necessarily user friendly.

ConversationMemory

The **io.vantiq.ai.ConversationMemory** service supports management of "conversations" in which previous requests and responses provide context for subsequent requests. Conversations consist of an Array of `io.vantiq.ai.ChatMessage` instances. Each conversation has an opaque "bag" of properties which will always contain the following system properties:

- `ars_namespace` – the namespace to which the conversation belongs
- `ars_createdAt` – a DateTime value indicating when the conversation was started.
- `ars_createdBy` – the user who started the conversation
- `ars_modifiedAt` – a DateTime value indicating when the conversation was last updated.
- `ars_modifiedBy` – the user who performed the last modification.

The service has the following procedures:

- **ConversationMemory.startConversation(initialState io.vantiq.ai.ChatMessage Array, conversationId String, properties Object): String** – start an AI "conversation". If provided, the conversation will start with the given `initialState`; otherwise it will be empty. Once established, the conversation will keep track of any request/response pairs that occur as part of the conversation (indicated by passing the conversation's id to the io.vantiq.ai.SemanticSearch.answerQuestion or io.vantiq.ai.LLM.submitPrompt procedures). The *properties* parameter can be used to add user defined properties to the conversation's property bag. The return value is an opaque String referred to as a "conversation id". This value can be provided by the caller (useful when associating the conversation with an existing resource such as a collaboration). Otherwise, it will be generated by the system.
- **ConversationMemory.getConversation(conversationId String Required): io.vantiq.ai.ChatMessage Array** – returns the current state of the specified conversation.
- **ConversationMemory.addChatMessages(conversationId String Required, newMessages io.vantiq.ai.ChatMessage Array Required): io.vantiq.ai.ChatMessage Array** – adds the given new messages to the end of the conversation.
- **ConversationMemory.setConversation(conversationId String Required, converationState io.vantiq.ai.ChatMessage Array Required)** - set the current state of the specified conversation to the given messages array.
- **ConversationMemory.getConversationProperties(conversationId String Required): Object** - return the properties for the given conversation.
- **ConversationMemory.updateConversationProperties(conversationId String Required, properties Object Required, replace Boolean)** - update the properties of the given conversation. If *replace* is set to `true`, then the given properties will replace any existing ones (excluding system properties). Othwerise, the properties will be merged (the default).
- **ConversationMemory.endConversation(conversationId String Required) –** terminates the conversation with the given id. To help with resource management, conversations will automatically expire and be closed after 30 minutes of inactivity.

The service generates the following events:

- **conversationStarted** – emitted anytime a new conversation is started. The event properties are:
  - `conversationId` – the id of the started conversation
  - `properties` – the conversation properties
- **conversationUpdated** – emitted when a conversation is updated. The event properties are:
  - `conversationId` – the id of the updated conversation
  - `properties` – the conversation properties
  - `messageCount` – the number of messages in the conversation
- **conversationEnded** – emitted when a conversation is ended. The event properties are:
  - `conversationId` – the id of the conversation which was ended.

Deployment

The **Deployment** service contains a collection of procedures used to implement the configuration deployment tool in the Vantiq IDE. It contains the following procedures:

- **Deployment.deploy(configName, globalId, errorsOnly)** - deploys the **configurations** instance identified by *configName* to all nodes targeted by the configuration's provisioning constraint. The *globalId* will be used to identify the results of the deployment and should be a UUID generated by the caller. The *errorsOnly* parameter, when `true`, hides all results except for errors.
- **Deployment.deployPartitions(configName, globalId, errorsOnly)** - deploys the **deployconfigs** instance identified by *configName* to all nodes targeted by each partition's constraint. The *globalId* will be used to identify the results of the deployment and should be a UUID generated by the caller. The *errorsOnly* parameter, when `true`, hides all results except for errors. Calling Deployment.deployPartitions again with the same *globalId* will try to redeploy the same deployment to new nodes or failed nodes from the previous deploy attempt. The *globalId* can also be used to query deployment results from **system.deploystatus**.
- **Deployment.undeploy(configName, errorsOnly)** - reverses a deployment by removing all artifacts specified in the **configurations** instance (identified by *configName*) from the nodes targeted by the configuration's provisioning constraint. The *errorsOnly* parameter, when `true`, hides all results except for errors.
- **Deployment.undeployPartitions(configName, errorsOnly)** - reverses a deployment by removing all artifacts specified in the **deployconfigs** instance (identified by *configName*) from the nodes targeted by each partition's constraint. The *errorsOnly* parameter, when `true`, hides all results except for errors.

Encoding, Decoding, and Hashing

The **Encode**, **Decode** and **Hash** services are useful for interacting with external sources, where it's often necessary to encode, decode, and hash values before or after interacting with the source.

- **Encode.base64(val)** - base64 encodes the specified value (which can either be a `String` or byte array) into a `String`.
- **Encode.formUrl(val)** - performs a form URL encoding (`x-www-form-urlencoded`) of the set of key-value string pairs expressed as a VAIL object. Returns a string value.
- **Decode.base64(val)** - decodes a base64 encoded `String` into the decoded string.
- **Decode.base64Raw(val)** - like Decode.base64, this procedure decodes a base64 encoded value. However, `Decode.base64Raw` does *not* encode the result as a UTF-8 String. It instead returns the data as raw bytes. This is helpful when the original unencoded data was not a UTF-8 string (e.g. an image).
- **Decode.formUrl(val)** - decodes a form URL encoded string representing key-value string pairs and returns a VAIL object containing the corresponding set of key-value pairs.
- **Hash.sha1(val)** - hashes a `String` using the sha1 algorithm. The result is a byte array that can be encoded with the **Encode** service, or can be cast to a `String` with `.toString()`.
- **Hash.sha256(val)** - hashes a `String` using the sha256 algorithm. The result is a byte array that can be encoded with the **Encode** service, or can be cast to a `String` with `.toString()`.
- **Hash.sha384(val)** - hashes a `String` using the sha384 algorithm. The result is a byte array that can be encoded with the **Encode** service, or can be cast to a `String` with `.toString()`.
- **Hash.sha512(val)** - hashes a `String` using the sha512 algorithm. The result is a byte array that can be encoded with the **Encode** service, or can be cast to a `String` with `.toString()`.
- **Hash.md5(val)** - hashes a `String` using the md5 algorithm. The result is a byte array that can be encoded with the **Encode** service, or can be cast to a `String` with `.toString()`.
- **Hash.hmacSha1(key, val)** - hashes a `String` using the hmac-sha1 algorithm and specified *key* which can either be a `String` or byte array. The result is a byte array that can be encoded with the **Encode** service, or can be cast to a `String` with `.toString()`.
- **Hash.hmacSha256(key, val)** - hashes a `String` using the hmac-sha256 algorithm and specified *key* which can either be a `String` or byte array. The result is a byte array that can be encoded with the **Encode** service, or can be cast to a `String` with `.toString()`.
- **Hash.hmacSha512(key, val)** - hashes a `String` using the hmac-sha512 algorithm and specified *key* which can either be a `String` or byte array. The result is a byte array that can be encoded with the **Encode** service, or can be cast to a `String` with `.toString()`.

Event Processing

When a rule processes a reliable event the rule must acknowledge processing the event, or else the reliable messaging system will continue to redeliver the event, which can lead to duplicate rule executions. To acknowledge receipt of an event from a rule, use the built-in ack procedure. To know if an event is being redelivered, use the built-in isRedelivered procedure.

- **Event.ack()** - sends off an event acknowledging receipt of the triggering event. This is a no-op when called in a rule triggered off an unreliable resource.
- **Event.isRedelivered()** - returns true if the event is being redelivered.
- **Event.reply(response Any Required): Boolean** – Send a response to the caller of **Event.request** which triggered the current event. If the current event was not triggered via **Event.request** then this is a no-op. A return value of `true` indicates that the response was sent.
- **Event.request(serviceEventName String Required, data Any Required, timeout Integer): Any** – triggers the named service event and waits for a response to be sent using **Event.reply**. This can only be called from a service procedure. The optional *timeout* value is an interval in milliseconds which defines how long the caller will wait for a reply. By default the caller will wait indefinitely. If the invocation times out, it will generate an exception with the error code `io.vantiq.rulemgr.execution.callback.timeout`. The return value is the response provided by **Event.reply**.

Image Processing

As described in the Images Section of the Resource Guide, the **Image** resource can store images in the Vantiq database. To understand more about the usage of these services, please see the Image Processing Guide.

There are two services offered for Image processing: **VisionScriptBuilder** and **VisionScriptOperation**.

VisionScriptBuilder

The VisionScriptBuilder service provides an easy way to construct a VisionScript object.

- **VisionScriptBuilder.newScript(scriptName)** - constructs and returns a new script object with the specified name.
- **VisionScriptBuilder.addAction(script, action)** - adds the specified *action* to the *script*, returning the new script.
- **VisionScriptBuilder.addConvertToGrayScaleAction(script)** - adds an action to convert an image to grayscale (*i.e.* black and white), returning the new script.
- **VisionScriptBuilder.addCropAction(script, x, y, width, height)** - adds an action that will crop an image to the height & width specified based on the x, y coordinates, returning the new script.
- **VisionScriptBuilder.addDrawBoxesAction(script, boxList)** - adds an action that will draw boxes as specified in the *boxList* parameter, returning the new script.
- **VisionScriptBuilder.addBoxToList(boxList, x, y, width, height, thickness, color, label, labelDetails)** – add a box to the list. If the list is `null`, create the list. Return a new list.
  - `x` - `Integer` X coordinate of upper left corner of box
  - `y` - `Integer` Y coordinate of upper left corner of box
  - `width` - `Integer` width of box, measured (right) from X coordinate
  - `height` - `Integer` height of box, measure (down) from Y coordinate
  - `thickness` - Optional `Integer` thickness of the line. If not present, defaults to 2.

- `color` - Optional `Object` containing 3 `Integer` properties (values 0-255): red, green, and blue. For example, { red: 128, green: 128, blue: 128 }. If not present, defaults to red.
- `label` - Optional `String` with which to label the box
- `labelDetails` - Optional textDetail (see **VisionScriptBuilder.newTextDetails()**) for the label

- **VisionScriptBuilder.addDrawBoxesFromPreviousAction(script, actionId)** - creates & returns a boxList using the results of a previous action (currently, a **VisionScriptBuilder.findFaces()** action).
  - `actionId` - `String` identifying the `tagName` used for the previous action.
- **VisionScriptBuilder.newBoxListFromYOLOResult(yoloResult, thickness, colors, labelDetails)** - creates and returns a box list from the results of the interpretation of an image by a neural net (specifically, a YOLO model). YOLO models intepret an image, identifying objects (based on the model's training) in the image. From the model, you will get the object name & location. This procedure creates a box list where each box is labeled with the object name.
  - `yoloResult` - Result `Object` from a YOLO TensorFlow execution
  - `thickness` - Optional `Integer` specifying the thickness of the line surrounding the boxes
  - `colors` - Optional `Object Array` describing the colors to use. The default will be RED, and an empty color list is treated as if no colors were provided (*i.e.* RED).
  - `labelDetails` - Optional `Object` describing the text properties for the label (see **VisionScriptBuilder.newTextDetails()**).
- **VisionScriptBuilder.addDrawTextAction(script, text, x, y, details)** - adds an action to draw the specified text at the specified position, returning the new script.
  - `script` - VisionScript `Object` to which to add the action
  - `text` - `String` value to be added to the image
  - `x` - `Integer` X value of the upper left of the text to be added
  - `y` - `Integer` Y value of the upper left of text to be added
  - `details` - Optional `Object` describing presentation details of the text to be added (see **VisionScriptBuilder.newTextDetails()**).
- **VisionScriptBuilder.newTextDetails(font, italicize, thickness, fontScale, color)** - creates and returns an `Object` containing the text details for text used in other actions. Used in **VisionScriptBuilder.addBoxToList()**, **VisionScriptBuilder.newBoxListFromYOLOResult()**, and **VisionScriptBuilder.addDrawTextAction()**.
  - `font` - `String` naming the font to use for the text. Font choices are controlled by the underlying platform, and are as follows:
    - `HERSHEY_PLAIN`,
    - `HERSHEY_COMPLEX`,
    - `HERSHEY_TRIPLEX` (Italics ignored)
    - `HERSHEY_SIMPLEX`,
    - `HERSHEY_DUPLEX`,
    - `HERSHEY_COMPLEX_SMALL`,
    - `HERSHEY_SCRIPT_SIMPLEX`,
    - `HERSHEY_SCRIPT_COMPLEX`
  - `italicize` - `Boolean` indicating whether to italicize the text (if possible)
  - `thickness` - `Integer` thickness in pixels of the text
  - `fontScale` - `Real` number by which to scale the text
  - `color` - `Object` containing 3 `Integer` properties (values 0-255): red, green, and blue. For example, { red: 128, green: 128, blue: 128 }. If not present, defaults to red (`{red: 255, green: 0, blue: 0}`).
- **VisionScriptBuilder.addResizeAction(script, width, height)** - add a resize action to the script, returning the new script. The image is resized as specified by the `width` and `height` parameters.
- **VisionScriptBuilder.addSaveAction(script, imageName, imageType)** - adds an action that will save the current result of the script to `imageName` using the `imageType` to specify the type to save. If the `imageName` or `imageType` is missing, they are defaulted to the same value as the current image. (See **VisionScriptOperation.processImage()** below.)

Each of the top-level builders returns the new script. In the following example, we build a procedure that creates an action list that converts an image to black and white and then finds the faces in that image.

```
PROCEDURE VisionScriptTest.genActionScript1(scriptName String)
var as1 = VisionScriptBuilder.newScript(scriptName)
    as1 = VisionScriptBuilder.addConvertToGrayscaleAction(as1)
    as1 = VisionScriptBuilder.addFindFacesAction(as1)

return as1
```

In a second example, we create a procedure that draws boxes on an image surrounding the faces found.

```
PROCEDURE createFaceBoxer()
var as1 = VisionScriptBuilder.newScript("FaceBoxer")
    as1 = VisionScriptBuilder.addConvertToGrayscaleAction(as1)
    as1 = VisionScriptBuilder.addFindFacesAction(as1)
    as1 = VisionScriptBuilder.addDrawBoxesFromPreviousAction(as1, "findFaces")  // Use results from findFaces action
    as1 = VisionScriptBuilder.addSaveAction(as1, null, "image/jpeg")
return as1
```

VisionScriptOperation

The **VisionScriptOperation** service provides procedures that are used to run vision scripts (such as those built by the **VisionScriptBuilder** service).

- **VisionScriptOperation.processImage(image, script)** - Runs the script over the named image.
  - `image` - `String` containing the name of the image on which the script is to be run

- `script` - `Object` containing the script to be run

As an example, using the `createFaceBoxer` procedure from the previous section, we might run that script over an image named “familyWedding.jpg” as follows:

```
var faceBoxerScript = createFaceBoxer()
VisionScriptOperation.processImage("familyWedding.jpg", faceBoxerScript)
```

The result of running this code snippet would be that the “familyWedding.jpg” image would be replaced by black and white image with faces found marked with boxes.

For example usage, see the sample code in the Image Processing Guide.

JWT Procedures

The **JWT** service contains a collection of procedures that create and decode Json Web Tokens (JWTs) (see jwt.io and RFC 7519). These allow for the use of symmetric (HMAC), RSA, and Elliptical keys for JWT signature processing. Throughout the calls below, the following are supported as signing algorithms:

Algorithm	JWS alg	Description
HMAC256	HS256	HMAC with SHA-256
HMAC384	HS284	HMAC with SHA-384
HMAC512	HS512	HMAC with SHA-512
RSA256	RS256	RSASSA-PKCS1-v1_5 with SHA-256
RSA384	RS384	RSASSA-PKCS1-v1_5 with SHA-384
RSA512	RS512	RSASSA-PKCS1-v1_5 with SHA-512
ECDSA256	ES256	ECDSA with curve P-256 and SHA-256
ECDSA256	ES256K	ECDSA with curve P-256k1 and SHA-256
ECDSA384	ES384	ECDSA with curve P-384 and SHA-384
ECDSA512	ES512	ECDSA with curve P-521 and SHA-512
Unsigned	NONE	The JWT is unsigned

At a high level, the HMAC/HS values specify symmetric keys, the RSA/RS values use RSA keys, and the EC/ES are elliptical keys. VAIL does not provide key generation support.

From the table above, the JWS alg (Json Web Signature alg) column contains the value to be used as the signing algorithm parameter for the calls below.

The signing and verification keys (see below) are always String values in the Vantiq system. For HMAC/HS keys, these are the String values of the keys. For the RSA/RS and ECDSA/ES algoritm, the values are the Base64-encoded values. This is true both for keys passed directly and those provided by Vantiq resources.

In the following calls, the signing key provided for RSA/RS and ECDSA/ES algorithm will be the **private** key; for verification, provide the **public** key. For HMAC/HS algorithms, there is only one symmetric key.

- `JWT.createToken(signingAlgorithm, claims, signingKey)` – returns a String containing signed JWT containing the claims specified.
  - `signingAlgorithm` – a String containing the name of the signing algorithm to be used to generate the JWT. The name should be taken from the *JWS alg* column above.
  - `claims` – a VAIL Object containing the claims to be included in the JWT. The keys represent the claim names, and the values the claim values.
  - `signingKey` – a String containing the key value to be used for signing the JWT. The value for the **private** (if appropriate) key, converted to a string as specified above.
- `JWT.createTokenWithHeaders(headers, claims, signingKey)` – returns a String containing signed JWT containing the claims specified.
  - `headers` – a VAIL Object containing the headers to include in the JWT. The keys represent the header names, and the values the header values. The headers *MUST* include the `alg` header, and this must contain the name of the signing algorithm to be used to generate the JWT. The name should be taken from the *JWS alg* column above.
  - `claims` – a VAIL Object containing the claims to be included in the JWT. The keys represent the claim names, and the values the claim values.

- `signingKey` – a String containing the key value to be used for signing the JWT The value for the **private** (if appropriate) key, converted to a string as specified above.

* `JWT.createTokenUsingResource(signingAlgorithm, claims, resource, resourceId)` – returns a String containing signed JWT containing the claims specified.
    - `signingAlgorithm` – a String containing the name of the signing algorithm to be used to generate the JWT. The name should be taken from the *JWS alg* column above.
    - `claims` – a VAIL Object containing the claims to be included in the JWT. The keys represent the claim names, and the values the claim values.
    - `resource` – a String containing name of the Vantiq resource type containing key. The `system.documents` and `system.secrets` resources are supported.
    - `resourceId` – the name of the document or secret to be used as the key. This named resource must contain the String value of the key, as specified above.
* `JWT.createTokenWithHeadersUsingResource(headers, claims, resource, resourceId)` – returns a String containing signed JWT containing the claims specified.
    - `headers` – The headers to include in the JWT. The keys represent the header names, and the values the header values. The headers *MUST* include the `alg` header, and this must contain the name of the signing algorithm to be used to generate the JWT. The name should be taken from the *JWS alg* column above.
    - `claims` – a VAIL Object containing the claims to be included in the JWT. The keys represent the claim names, and the values the claim values.
    - `resource` – a String containing name of the Vantiq resource type containing key. The `system.documents` and `system.secrets` resources are supported.
    - `resourceId` – the name of the document or secret to be used as the key. This named resource must contain the String value of the key, as specified above.
* `JWT.decodeAndVerifyToken` – returns a VAIL object containing the decoded JWT. Optionally, an indication of whether the JWT was verified is included.
    - `token` – a String containing the JWT to be decoded.
    - `verificationKey` – (optional) a String containing the key to be used for JWT signature verification. The value for the **public** (if appropriate) key, converted to a string as specified above. If the `verificationKey` is not provided, no verification is performed.
    - This procedure returns a VAIL object containing the following properties:
        - `headers` – the headers contained in the JWT
        - `claims` – the claims contained in the JWT
        - `verified` – a boolean value indicating whether the verfication was successful. If no verification was requested (*i.e.,* no `verificationKey` was provided), this property will be absent.
* `JWT.decodeAndVerifyTokenUsingResource` – returns a VAIL object containing the decoded JWT. Optionally, an indication of whether the JWT was verified is included.
    - `token` – a String containing the JWT to be decoded.
    - `resource` – a String containing name of the Vantiq resource type containing key. The `system.documents` and `system.secrets` resources are supported.
    - `resourceId` – the name of the document or secret to be used as the key. This named resource must contain the String value of the **public** (if appropriate) key, as specified above.
    - This procedure returns a VAIL object containing the following properties:
        - `headers` – the headers contained in the JWT
        - `claims` – the claims contained in the JWT
        - `verified` – a boolean value indicating whether the verfication was successful.

For example, consider that we would like to create a JWT using HMAC256 signature algorithm. To do so, we might do something like the following:

```
var jwt = JWT.createToken("HS256",
                         { iss: "http://example.com", "http://example.com/name" : "fred" },
                          "mySecret")
```

This will result in a string that looks something like

```
0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vZmhjYXJ0ZXIuY29tIiwic29tZXRoaW5nIjoiZWxzZSJ9.T9Wx1TWUZzzP8JYs1LeW0JoP0vQZpdQv
```

To decode and verify such a token, use the following

```
var jwtDecoded = JWT.decodeAndVerifyToken(jwt, "mySecret")
```

This will return an object containing the following values:

```
{
    headers: {
        typ: "JWT",
        alg: "HS256"
    },
    claims: {
        iss: "https://example.com",
        "http://example.com/name" : "fred"
    },
    verified: true
}
```

To decode but not verify such a token, use the following

```
var jwtDecoded = JWT.decodeAndVerifyToken(jwt)
```

This will return an object containing the following values:

```
{
    headers: {
        typ: "JWT",
        alg: "HS256"
    },
    claims: {
        iss: "https://example.com",
        "http://example.com/name" : "fred"
    }
}
```

LLM

The **io.vantiq.ai.LLM** service provides methods used to interact with a Large Language Model. Vantiq has preconfigured support for the following generative LLMs: GPT-4 and GPT-3.5.

- **LLM.submitPrompt(llmName String, prompt, conversationId String, responseProperties Object, functions, runtimeConfig, functionAuthorizer, tools)** - submits the specified prompt to the named LLM and returns the generated response. Typically this response is a `String` which is synthesized by the LLM. The arguments are:
    - **llmName** - the name of the target LLM. This must refer to a *generative* LLM.
    - **prompt** - the prompt to submit to the LLM. This must be either a `String` or an `io.vantiq.ai.ChatMessage` Array.
    - **conversationId** - if provided, associates the prompt submission with a previously established "conversation" (see ConversationMemory). The current conversation state is submitted along with the provided prompt and the prompt and the resulting LLM response are appended to the conversation.
    - **responseProperties** - optional properties to attach to the `ChatMessage` instance representing the LLM's response when it is added to the active conversation (the value is ignored if there is no conversation).
    - **functions** - (*deprecated*, see `tools` parameter) an optional Array of AI function descriptors.
    - **runtimeConfig** - an optional Object that specifies LLM configuration settings for the `submitPrompt` invocation. When specified, *runtimeConfig* is merged with the LLM configuration, overriding its configuration properties. The merged configuration is then used for the `submitPrompt` call; the LLM configuration itself is not changed. For example, this argument can be used to specify a `temperature` or `max_tokens` value specific to a `submitPrompt` invocation.
    - **functionAuthorizer** - an optional reference to a procedure that determines whether to allow or deny the automatic execution of function calls (*functions* parameter above). If functionAuthorizer is not provided, function execution is automatic. If provided, functionAuthorizer must return *true* to grant execution and *false* to deny it. The functionAuthorizer procedure must have the following parameters:
        - **name** - the name of the function to be executed
        - **arguments** - the function invocation parameter values

      See LLMs Reference Guide for more details.
    - **tools** - an optional Array of AI tools descriptors. These descriptors can be either an instance of the `io.vantiq.ai.FunctionDescriptor` schema type or a ResourceReference instance referring to a VAIL service or procedure. When specified, they are provided to the LLM along with the provided prompt. The LLM may then choose to request an invocation of a tool in order to obtain the context needed to respond to the given prompt. When referencing a service or procedure, this invocation will be handled automatically. When describing a tool as a schema (e.g., using `io.vantiq.ai.FunctionDescriptor`) the tool cannot be called and the `submitPrompt` returns an instance of `io.vantiq.ai.ChatMessage` representing a request to invoke the tool. If a *description* exists for a procedure, the LLM will use that description to help decide which procedure to invoke. For a ResourceReference, if both the service interface and the procedure have a description defined for the procedure, the one from the procedure will be used. See LLMs Reference Guide for examples.
- **LLM.submitPromptAsSequence(llmName String, prompt, conversationId String, responseProperties Object, functions, runtimeConfig, functionAuthorizer, tools)** - submits the specified prompt to the named LLM and returns the generated response as a sequence. This is used to stream output to the caller as it is produced. The parameter definitions are identical to those for *submitPrompt*.

Since this is a built-in procedure, it cannot be used as part of an EXECUTE statement. Therefore its results must be processed using an iteration statement.

LocationTracking

The **LocationTracking** service contains a collection of procedures used to implement location tracking on the Vantiq Mobile Apps. It contains the following procedures:

- **LocationTracking.trackCollaborator(activity, user, level, distanceFilter, desiredAccuracy, destination, destinationRadius, arrivalTopic, reportingTopic)** - begin tracking the *user* to the specified *destination* for a specific *activity* in a collaboration type. The *distanceFilter*, *desiredAccuracy*, and *destinationRadius* are all `Real` numbers which can be tuned to adjust the reporting frequency, the accuracy of the measurements, and how close to the destination the user needs to be to trigger an arrival event. The *reportingTopic* is where location updates will be published by the Vantiq Mobile App, and the *arrivalTopic* is where the location arrival event will be published.
- **LocationTracking.untrackCollaborator(active, user)** - remove the **ArsActiveTrack** record *active* and stop tracking the *user* for that activity. There may be other activities tracking the user, in which case this may not turn off location tracking entirely.

Math

The Math service contains a collection of basic math functions available for use in rules and procedures. It contains the following procedures:

- **Math.abs(val)** - returns the absolute value of val.
- **Math.ceil(val)** - returns val, rounded up to the nearest `Integer`.
- **Math.exp(val)** - returns E^val (where E is the base of the natural logarithm).
- **Math.floor(val)** - returns val, rounded down to the nearest `Integer`.
- **Math.log(val)** - returns the natural logarithm of val.
- **Math.log2(val)** - returns the base 2 logarithm of val.
- **Math.log10(val)** - returns the base 10 logarithm of val.
- **Math.max(x,y)** - returns the max of numbers x and y. Note that x and y must be the same type of number, so if x is a `Real`, y must also be a `Real`.
- **Math.min(x,y)** - returns the min of numbers x and y. Note that x and y must be the same type of number, so if x is a `Real`, y must also be a `Real`.
- **Math.round(val)** - returns val rounded to the nearest `Integer`. The returned value is a `Real`.
- **Math.sqrt(val)** - returns the square root of val.
- **Math.random(min,max)** - returns a random `Integer` between min (inclusive) and max (inclusive).
- **Math.randomReal()** - returns a random `Real` between 0 (inclusive) and 1 (exclusive).

Available Trig functions (NOTE: all input values must be in radians):

- **Math.cos(val)** - returns the cosine of val.
- **Math.sin(val)** - returns the sine of val.
- **Math.tan(val)** - returns the tangent of an angle val.
- **Math.acos(val)** - returns the arccosine of val.
- **Math.asin(val)** - returns the arcsine of val.
- **Math.atan(val)** - returns the arctangent of val. Note: result will always be in the range [-PI/2,PI/2].
- **Math.atan2(y,x)** - returns the arctangent of y/x. Note: The y coordinate is the first argument and the x coordinate is the second argument.

A set of standard constants have been implemented as parameterless procedures:

- **Math.E()** - returns Euler's number, the base of the natural logarithm (approx. 2.718).
- **Math.LN2()** - returns the natural log of 2 (approx. 0.693).
- **Math.LN10()** - returns the natural log of 10 (approx. 2.302).
- **Math.LOG2E()** - returns the base-2 log of E (approx. 1.442).
- **Math.LOG10E()** - returns the base-10 log of E (approx. 0.434).
- **Math.PI()** - returns Pi (approx. 3.14).
- **Math.SQRT2()** - returns the square root of 2 (approx. 1.414).
- **Math.SQRT1_2()** - returns the square root of 1/2 (approx. 0.707).

Motion Tracking

The **MotionTracking** service contains a collection of procedures used to track and understand the motion of entities in images. It contains the following procedures:

- **MotionTracking.trackMotion(state, newObjects, algorithm, qualfier, maxAbsent, timeOfObservation, coordinateProperty, labelProperty)** – returns an `Object` with the following properties: *trackedObjects* containing the new object positions and *droppedObjects* containing the list of objects dropped after this call.
  - **state** - the current set of tracked objects. A `null` value indicates no current state.
  - **newObjects** - the set of new objects with positions.
  - **algorithm** - algorithm to use to determine motion.
  - **qualifier** - value used to determine if two positions *could* be movement of the same entity.
  - **maxAbsent** - an interval after which an entity is considered missing. Missing objects are dropped from the set of known objects.
  - **timeOfObservation** – time to assign to new positions. If unspecified, use the current time.
  - **coordinateProperty** – (optional) the name of the property from which to get the coordinates. The default value is `location`. This can be used if the input stores location information under a different property name.

  - **labelProperty –** (optional) the name of the property from which to extract the label. The default value is `label`. This can be used of the input labels things using a different property name.
- **MotionTracking.buildAndPredictPath(state, newObjects, maxSize, pathProperty, coordinateProperty, doPredictions, timeOfPrediction)** - returns an `Object` with the following properties: *trackedPaths* containing the list of tracked objects with their paths and *droppedPaths* property containing the list of paths dropped with their predicted locations.
  - **state** - the current set of tracked paths. A `null` value indicates no current state.
  - **newObjects** - the set of new objects with positions.
  - **maxSize** - (optional) the maximum path length. Default value is 10.
  - **pathProperty** - (optional) property name to use to store the path within the location object. Default is `trackedPath`.
  - **coordinateProperty** (optional) The name of the property from which to get the location information. Default is `location`.
  - **doPredictions** - (optional) `Boolean` indicating whether to predict the positions of objects dropped from tracking.
  - **timeOfPrediction** - (optional) time to assign to predicted positions. If unspecified, use the current time.
- **MotionTracking.findRegionsForCoordinate(regionList, coordinate)** - returns the list of region names in which the coordinate is located. The list can be empty.
  - **regionList** - an `Array` of the **TrackingRegions** from which to choose.
  - **coordinate** - an `Object` containing the location information.
- **MotionTracking.findRegionsForBoundingBox(regionList, boundingBox, overlapPercentage)** - returns the list of region names in which the bounding box is located. The list can be empty.
  - **regionList** - an `Array` of the **TrackingRegions** from which to choose.
  - **boundingBox** - an `Object` containing the bounding box in question.
  - **overlapPercentage** – a number (`Integer` or `Real`) containing the percentage of the bounding box that must be contained in a region for the region to qualify. If the number given is > 1, then it is divided by 100 to get a fractional percentage. If the number given is < 1, it is taken as the fractional percentage. The number given must be greater than zero. For example, if “85” is provided as the `overlapPercentage`, then the 85% (0.85) of the bounding box must be contained in a region for that region to qualify as containing the bounding box.
- **MotionTracking.computeVelocity(regionList, path, lastLegOnly)** - returns a velocity object with the properties: *speed* in units/second (units the same as those in the regions), and *direction* as compass direction (number of degrees). Either or both properties can be missing of no *distance* or *direction* components are found in the *regionList*.
  - **regionList** - an `Array` of **TrackingRegions** from which to gather distance and direction
  - **path** - an object’s path (a list of positions)
  - **lastLegOnly** - `Boolean` indicating whether to use the last two positions. If `false`, use the first and last positions in the path.
- **MotionTracking.predictPositions(pathsToPredict, timeOfPrediction, pathProperty)** – returns the list of paths with their predicted locations.
  - **pathsToPredict** - the current set of tracked paths for which to predict next positions.
  - **timeOfPrediction** - (optional) time to assign to predicted positions. If unspecified, use the current time.
  - **pathProperty** - (optional) property name to use to store the path within the location object. Default is `trackedPath`.
- **MotionTracking.predictPositionsBasedOnAge(candidatePaths, expirationTime, timeOfPrediction, pathProperty)** – returns the list of expired paths with their predicted locations. This procedure evaluates the *candidatePaths* against the *expirationTime*. For any paths whose last *timeOfObservation* is at or before the *expirationTime*, we predict then next position (based on *timeOfPrediction*) and return that list. Paths after the *expirationTime* are ignored.
  - **candidatePaths** - the current set of tracked paths for which to predict next positions.
  - **expirationTime** - the time representing the latest time considered expired.
  - **timeOfPrediction** - (optional) time to assign to predicted positions. If unspecified, use the current time.
  - **pathProperty –** (optional) property name to use to store the path within the location object. Default is `trackedPath`.
- **MotionTracking.dbscanCluster(distance, minPoints, points)** – run the density based clustering algorithm DBSCAN using the specified distance, minPoints parameters and a list of points expressed as [[x1,y1],[x2,y2],…]. Return the list of clusters found, where one cluster is the list of all points belonging to the cluster. If no cluster is found, returns an empty list.
  - **distance** - distance that defines the ε-neighborhood of a point.
  - **minPoints** - minimum number of density-connected points required to form a cluster.
  - **points** - list of points expressed as an array of [x,y] coordinates. Coordinates can be specified either as integer or double values.

MotionTracking.trackMotion() and **MotionTracking.buildAndPredictPath()** are procedures that maintain state. As such, their return value (from any given call) should be the *state* parameter for the next call. **MotionTracking.findRegionsForBoundingBox()**, **MotionTracking.findRegionsForCoordinate()**, and **MotionTracking.computeVelocity()** are simply informational and maintain no state.

The list of paths provided by **MotionTracking.buildAndPredictPath()** may be used to call **MotionTracking.predictPositions()** or **MotionTracking.predictPositionsBasedOnAge()** as deemed appropriate by the application.

Notification

The **Notification** service is used to send notification to the Vantiq Mobile Apps. It contains the following procedures:

- **Notification.deletePayloadMessages(msgIds)** - delete all messages with ids in the *msgIds* array. This is used by the mobile apps when a user deletes messages from the mobile app.
- **Notification.retractPayload(msgId, excludeList)** - retract a notification (**ArsPayloadMessage**) specified by *msgId* that was sent out to the Vantiq Mobile Apps. The *excludeList* can be used to specify a list of usernames to not retract the notification from, and if an empty list or null is specified the notification will be retracted for all users.

Open Inference

The **io.vantiq.ai.OpenInference** service provides methods related to the Open Inference Protocol support in Vantiq.

- **OpenInference.reshape(data, shape)** - reshapes the input data tensor into the specified shape. The input data must be a 1-dimensional array, and the shape must be an array of integers specifying the desired dimensions. The total number of elements in the input data must match the product of the dimensions in the shape.
    - `data` - an `Array` containing the data to reshape.
    - `shape` - an `Array` of integers specifying the desired shape.
    - Returns a multidimensional `Array` representing the reshaped data.
- **OpenInference.flatten(data)** - flattens the input data into a 1-dimensional array. The input data can be a multi-dimensional array, and the output will be a single array containing all elements in row-major order.
    - `data` - an `Array` containing the data to flatten.
    - Returns a 1-dimensional `Array` representing the flattened data.

Recommend

The **Recommend** Service contains the built in recommenders. They all follow a similar pattern, in which they take 4 parameters:

- **matchDirectives** An `Object` containing tunable parameters which can be used to influence the output of the recommender. All recommenders should respect *maxRecommendations*, which limits the number of results returned by the recommender, and *excludeProperties*, which specifies a list of properties to avoid matching on.
- **pattern** The input `Object` to match against.
- **candidateType** The name of the type to pull the candidate set from.
- **matchType** The type of the input pattern. In some recommenders like **nearbyRecommendations** the matchType must match the candidateType, in others like **defaultRecommendations** they do not need to match.

Currently there are 2 built in recommenders:

- **Recommend.defaultRecommendations(matchDirectives, pattern, candidateType, matchType)** - finds the most similar instances of *candidateType* to the input *pattern* using simple distance measurements on all properties with the same name in the pattern and candidate type.
- **Recommend.nearbyRecommendations(matchDirectives, pattern, candidateType, matchType)** - finds the closest instances of candidateType to pattern by calculating and summing distances across all non-excluded GeoJSON fields in the pattern. The matchType must match the candidateType. In addition to excludeProperties and maxRecommendations, nearbyRecommendations also supports the maxDistance matchDirective which prevents candidates that are more than the maxDistance away from the pattern from appearing in the result list.

Resource

The **Resource** service is used to find specific information about Resources. It contains the following procedures:

- **Resource.buildInfo(resourceType)** - returns build information about the resource type. This information may be used when the resource type in question makes use of other facilities. If there is no relevant build information, the value returned will be empty.

ResourceAPI

The **ResourceAPI** service is used to access the full Vantiq Resource API. Whenever possible we encourage the use of more direct VAIL syntax to access most Vantiq resources. However, there are some resource operations which have not yet (and may never be) bound directly to the VAIL language. In that case the **ResourceAPI** service can be used to execute those operations. It contains the following procedures:

- **ResourceAPI.executeOp(message)** - executes the resource operation described by the supplied operation message. See the API Reference Guide for more details on how to use the API and the section on the VAIL Binding for the structure of the request messages and responses.

> The *asSequence* parameter has been deprecated and will be removed in a subsequent release. Please use `ResourceAPI.executeOpAsSequence(message)` instead in any situation where the operation is known to produce a sequence.

- **ResourceAPI.executeOpAsSequence(message)** - executes the resource operation described by the supplied operation message. The response is returned as a sequence. This is useful for processing potentially large responses and for streaming results from an LLM.
- **ResourceAPI.patch(resource, resourceId, patchCmds)** - executes the given PATCH commands on the specified resource instance. The `patchCmds` are given as an array of JSON objects conforming to the JSON Patch format.

ResourceConfig

The **ResourceConfig** Service allows you to access Assembly Configuration Properties in VAIL Rules and Procedures.

- *ResourceConfig.get(property)* - returns the configuration value for the property passed as a parameter. This property name must be defined in the Usage section for an Assembly Configuration Property that references the calling Rule or Procedure.

SemanticSearch

The **io.vantiq.ai.SemanticSearch** service supports the answering of questions using the context stored in a specified semantic index.

- **SemanticSearch.answerQuestion(indexName String Required, question Required, qaLLM String, conversationId String, minSimilarity Real, contextPrompt String, answerProperties Object, runtimeConfig Object): Object** - answer a question using the context contained in the specified semantic index. The expected parameters are:
    - `indexName` - the name of the SemanticIndex to use when answering the question.
    - `question` - the question to be answered. This is either a `String` or an `io.vantiq.ai.ChatMessage` instance. The latter form is typically used when the application wants to attach properties to the question when it is recorded in a conversation.

- `qaLLM` - the generative LLM used to synthesize the answer. If no `qaLLM` value is provided, then it will use the index's default Q&A LLM (if there is one).
- `conversationId` - associates the question with a previously established "conversation" (see ConversationMemory).
- `minSimilarity` - a number from 0 to 1, used to filter the relevant documents returned from the semantic index. When no documents qualify, the answer defaults to "I don't know." If no similarity threshold is provided, the index returns the most relevant documents regardless of their score.
- `contextPrompt` - a prompt used to help the LLM identify the context within the ongoing conversation that is most relevant for answering the follow-up question.
- `answerProperties` - optional properties that will be attached to the `ChatMessage` instance representing the LLM's response when it is added to the active conversation (the value is ignored if there is no conversation).
- `runtimeConfig` - an optional Object that specifies LLM configuration settings for the `answerQuestion` invocation. When specified, *runtimeConfig* is merged with the LLM configuration, overriding its configuration properties. The merged configuration is then used for the `answerQuestion` call; the LLM configuration itself is not changed.

The returned object will have the following properties:

- **answer** (String) – the answer synthesized by the generative LLM.
- **metadata** (Object Array) – an array of metadata objects derived from the index entries used to synthesize the answer. The exact metadata present is determined by the index entries. The semantic index may be configured to exclude this property.
- **rephrasedQuestion** (String) – the question rephrased by the LLM using the relevant context. If **rephraseQuestion** is set to `false` this will still contain the rephrased question, but it will not be the question asked to the LLM. This property is only included if the semantic index is configured to include it.

- **SemanticSearch.answerQuestionAsSequence(indexName String Required, question Required, qaLLM String, conversationId String, minSimilarity Real, contextPrompt String, answerProperties Object, runtimeConfig Object): Object** - answer a question using the context contained in the specified semantic index and return the answer as a sequence. The parameter definitions are the same as those for *answerQuestion*.

Since this is a built-in procedure, it cannot be used as part of an EXECUTE statement. Therefore its results must be processed using an iteration statement.

- **SemanticSearch.similaritySearch(indexName String Required, query String Required, minSimilarity Real, metadataFilter Object, limit Integer): Object Array** - perform a similarity search of the target semantic index using the given query. Returns up to `limit` (default 4) most similar records from the index. Providing a `minSimilarity` value (a number from 0 to 1) limits the result to records with at least that score. The `metadataFilter` value can be used to return only records with matching metadata values. It can either be an Object containing the expected values e.g. `{"key1": "val1", "key2": "val2"}`, or use the Qdrant filter format e.g. `{"must": [{"key": "key1", "match": {"value": "val1"}}, {"key": "key2", "match": {"value": "val2"}}]}`. The return value is a list of `Object` instances with the following properties:
  - **content** (String) – the content of the associated record.
  - **metadata** (Object Array) – the record's metadata.
  - **score** (Real) – the record's similarity score.
- **SemanticSearch.retrieveChunks(indexName String Required, entryId String Required, limit Integer, offset String): Object** - Retrieve the contents of the entry.
  - `indexName` - the name of the SemanticIndex to retrieve the content from.
  - `entryId` - the id of the entry whose content will be retrieved.
  - `limit` - the maximum number of chunks to retrieve at a time.
  - `offset` - the id of the chunk to start retrieving from. Starts at the first chunk if `null`. This should be taken from the *offset* field of the returned Object.

  The returned object will have the following properties.

  - **offset** (String) – the offset to be used in future *retrieveChunks()* calls to continue where this call left off. `null` if *points* contains the last chunk of this entry.
  - **points** (Object Array) - the points stored in the database. They will be in the following format:
    - **content** (String) - the content of this chunk.
    - **metadata** (Object) - the metadata for this chunk.

StorageManager

The **io.vantiq.StorageManager** service provides procedures to manage database transactions when working with a pluggable storage manager that supports them. For more information about pluggable storage managers, see the Storage Manager Guide and the section on transaction support.

- **StorageManager.startTransaction(storageManager String, options Object): String** - starts a transaction on the specified storage manager and returns the transaction ID.
- **StorageManager.commitTransaction(storageManager String, transaction String, options Object)** - commits the specified transaction in the given storage manager.
- **StorageManager.abortTransaction(storageManager String, transaction String, options Object)** - aborts the specified transaction in the given storage manager.

Template

The **io.vantiq.text.Template** service provides procedures to generate text from a template.

- **Template.format(template, input)** - returns the text generated by applying the input to the template. The *input* is an `Object` of key/value pairs where the keys are the names of the variables in the *template* to be substituted with the key values. Variables are of the form `${keyName}` or `$keyName`. To include a literal `$` in your template use the escape sequence `\$`. The code that processes the *template* value will fully interpret any escape sequences (this is in addition to the normal interpretation of those sequences when used in a VAIL `String` constant). For example, if you want your template to contain the `\` character, your template string value would need to contain `\\\\` (and not simply `\\`).
- **Template.documentReference(docName)** - returns a reference to the document resource named *docName*. This reference can be used as a parameter of **Template.format()**.

In its simplest form, a template is just a string containing variables replaced by the key values from the input. For example,

```
var input = {name: "alice"}
io.vantiq.text.Template.format("Hello ${name}", input) //  returns "Hello alice"
```

Instead of specifying a string, a template can be stored in a document. For example, assuming that we have a document named “greetingTemplate” containing the following,

```
Hello ${guest}, this is ${host}.
```

We can use this template as follows,

```
import service io.vantiq.text.Template

var input = {guest: "alice", host: "bob"}
var ref = Template.documentReference("greetingTemplate")
Template.format(ref, input) // returns "Hello alice, this is bob."
```

The template parameter can also be a list or an `Object`. In this case, the list/`Object` is recursively parsed and all document references are resolved to their text values. Then, all strings contained within the list/`Object` - except keys - are considered to be templates and are processed as described above. In that case, the **Template.format()** call returns the same structure as the template parameter but with all variables replaced by their values.

Templates may contain “repeat” blocks of the form:

```
@repeat(<variable>)<template to be repeated>@endrepeat
```

The `@repeat` command refers to an input variable based on the template it is part of. The variable must refer to an “iterable” value (typically a List). The text found between `@repeat` and `@endrepeat` is treated as a template and is applied to each item in the iterable value. The input to the repeat template depends on the type of the values found in the iterable. Any value of type `Object` is used directly as the repeat template input. For all other values, the input will be an `Object` with the single property *value*, pointing at the value from the iterable. For example:

```
import service io.vantiq.text.Template

var input = {items: ['one', 'two', 'three']}
var template = "Here is a list of items:
@repeat(items)    - ${value}
@endrepeat
All done"
Template.format(template, input)
```

The result of the formatting is:

```
Here is a list of items:
    - one
    - two
    - three
All done
```

Repeat blocks can be nested like so:

```
var template = "The following applicants are listed by name and job skills:

@repeat(name) ${name} : @repeat(jobSkill)${value}, @endrepeat\n@endrepeat

Choose who'd best fit into your department."
```

Remember that when evaluating the repeated template, the template’s “input” is taken by iterating over the referenced property name. This means that the input must be similarly nested like so:

```
var input = {
    name: [
        { name: "Mario", jobSkill: ["coin-flipping", "navel-gazing"]},
        { name: "Anna", jobSkill: ["navel-gazing", "underwater basket-weaving"]},
        { name: "Frederick", jobSkill: ["underwater basket-weaving", "coin-flipping"]},
    ]
}
```

The repeat template can itself be a document reference by using the following as the template text `io.vantiq.text.Template.documentReference("<documentName>")`. This must be the only text in the repeat template (including whitespace). Here is the previous template using a reference instead:

```
import service io.vantiq.text.Template

var input = {items: ["one", "two", "three"]}
var template = "Here is a list of items:
@repeat(items)io.vantiq.text.Template.documentReference(\"listEntry.txt\")@endrepeat
All done"
Template.format(template, input)
```

If the document “listEntry.txt” contained the text:

```
    - ${value}
```

The output produced would be identical to the previous example.

Finally, it is also possible to use document references for the input values. For example,

```
# assuming document "prompt.txt"
As a personal assistant who likes to be ${personality}, you answer the user question in the style of ${author}."

# assuming document "author.txt"
Mark Twain

# assuming document "question.txt"
why did the cow jump over the moon?
```

We can use these documents as follows,

```
import service io.vantiq.text.Template

var input = {
    personality: "funny",
    author: Template.documentReference("author.txt"),
    question: Template.documentReference("question.txt")
}

var template = {
    prompt: Template.documentReference("prompt.txt"),
    ask: "The question is: ${question}"
}

var templateResult = Template.format(template, input)
```

And the templateResult value will be,

```
{
    prompt: "As a personal assistant who likes to be funny, you answer the user question in the style of Mark Twain.",
    ask: "The question is: why did the cow jump over the moon?"
}
```

TensorFlowOperation

TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.

The **TensorFlowOperation** service provides the ability to run the specified neural net model against a document or image. For more information about this capability, please see the Image Processing Guide. It contains the following procedures:

YOLO Models

Use the following procedures to run YOLO models (models of type `tensorflow/yolo`)

- **TensorFlowOperation.processDocument(document, modelName, confidence, colorScaleFactor)**
- **TensorFlowOperation.processImage(image, modelName, confidence, colorScaleFactor)**

- **TensorFlowOperations.processTempImage(tempImage, modelName, confidence, colorScaleFactor)**
    - **document**, **image**, or **tempImage** - the document or image to be processed
        - When using `TensorFlowOperations.processTempImage()`, the `tempImage` parameter should be the entity returned by the SELECT statement against the video source or the event contents.
    - **modelName** - the name of the TensorFlow model to be used
    - **confidence** - a `Real` number between 0 & 1 specifying the confidence required to identify something in the document or image. 0.75 means 75% confidence.
    - **colorScaleFactor** - Optional and very rarely used. This is an `Integer` between 0 and 255 used to convert the color values in the image into floating point numbers. The default value is 255 which is usually what is desired.
    - **Result** - a list of Objects where each object describes an object detected by the model. This is described in detail here.

Plain Tensorflow Models

Use the following to run other models (models of type `tensorflow/plain`)

- **TensorFlowOperation.executeTFModelOnDocument(document, modelName, inputs)**
- **TensorFlowOperation.executeTFModelOnImage(image, modelName, inputs)**
- **TensorFlowOperation.executeTFModelOnTempImage(tempImage, modelName, inputs)**
    - **document**, **image**, or **tempImage** - the document or image to be processed
        - When using `TensorFlowOperations.executeTFModelOnTempImage()`, the `tempImage` parameter should be the entity returned by the SELECT statement against the video source or the event contents.
    - **modelName** - the name of the tensorflowmodel to be used
    - **inputs** - an `Object` containing the input specifications for execution
- **TensorFlowOperation.executeTFModelOnTensors(modelName, inputs)**
    - **modelName** - the name of the tensorflowmodel to be used
    - **inputs** - an `Object` containing the input specifications for execution

The result is an `Object` consisting of the output tensors, identified by name.

The *inputs* parameter should be an `Object` containing the following properties:

- **targetTensorName** - this is the name of the tensor into which to place the image or document.
    - This value should not be passed to **TensorFlowOperation.executeTFModelOnTensors()**.
- **inputTensors** - (optional) if the model requires other input, then this contains an `Object` where each input tensor is provided via a property named for that tensor where the property value is an `Object` containing two properties:
    - **tensorType** - a `String` identifying the type of the tensor,
    - **value** - the value to be provided.

Working with these types of TensorFlow models is complex. More detail is available here. Note also that there may be restrictions on the use of Plain Tensorflow Models in some installations. Please see Operational Restrictions for details.

For a simple example, assume we wish to process an image named *mycar.jpg* using a model named *identifyCars*. Further assume *identifyCars* supports three (3) input tensors:

- 'carPic' - the image to analyse
- 'year' - (optional) the year in which the car was manufactured
- 'country' - (optional) the country of origin for the car.

and that *identifyCars* returns two (2) tensors:

- `modelName` - a String, the model of car
- `manufacturer` - a String, the car maker

We could then execute the simple version (leaving out the optional parameters) as follows:

```
var tfResult = TensorFlowOperations.executeTFModelOnImage(
                "mycar.jpg",
                "identifyCars",
                { targetTensorName: "carPic" })
```

The more complex version including all input parameters might look like this:

```
var tfResult = TensorFlowOperations.executeTFModelOnImage(
                "mycar.jpg",
                "identifyCars",
                { targetTensorName: "carPic",
                  inputTensors: {
                        year: { tensorType: "int", value: 1980 },
                        country: { tensorType: "string", value: "USA"}
                    }
                 })
```

After execution of this code snippet, `tfResults` will be an object whose values might be (depending on the image in question)

```
{
    modelName: { tensorType: "string", value: "Fusion" },
    manufactuer: { tensorType: "string", value: "Ford" }
}
```

In the previous example, input and output tensors (with the exception of the input image) are scalars. That is, they are simple numbers or strings. Input or output tensors can, of course, be arrays. Note that TensorFlow tensors are always single, simple types (listed above), and regular, meaning that all rows in an array have the same number of columns (and, of course, extending to any number of dimensions).

To see how this might be represented, we will extend this example a little. Assume that the *identifyCars* also returns (in a tensor named *colors*) a list of colors in which the car was originally available. Using our same calling example above, `tfResults` will be an object whose values might be (depending on the image in question)

```
{
    modelName: { tensorType: "string", value: "Fusion" },
    manufactuer: { tensorType: "string", value: "Ford" },
    colors: { tensorType: "string", value: [ "red", "midnight pearl", "chartreuse", "taupe" ] }
}
```

Test

The **Test** service allows developers to control aspects of the application testing environment such as source “mocking”. For more information about this capability, please see the Testing Guide. It contains the following procedures:

- **Test.sendMockSourceEvent(sourceName REQUIRED, message REQUIRED, topic)** - generates an event which appears as if it arrived from the specified source. The event data will be the provided `message` value. An optional `topic` can be specified for sources that receive their data from topics (e.g. MQTT or Kafka).
- **Test.startSourceMocking(sourceName, mockQueryProcedure, mockPublishProcedure)** - turns on mocking for the **source** identified by *sourceName*. The optional *mockQueryProcedure* and *mockPublishProcedure* provide the ability to override the default mocking behavior for the respective operations. If unspecified (or `null`), the default mocking behavior remains unchanged.
- **Test.stopSourceMocking(sourceName)** - turns off mocking for the **source** identified by *sourceName*.

Utils

The **Utils** service contains a collection of useful procedures, including:

- **Utils.getHttpHeaders()** - for procedures invoked through a REST call, return the request’s HTTP headers excluding the Authorization header.
- **Utils.initPushSource(pushSourceName)** - validate that the push source with the name *pushSourceName* exists, and if it does not, create a push source with the specified name. If no name is specified, the name will be “VantiqPushNotification”.
- **Utils.stripSystemProperties(object)** - removes all system properties that a user should not directly modify from an object and returns the modified object. Example properties are *ars_createdAt* and *_id*. Useful when selecting a resource instance from the persistent data model, modifying it, and then using that instance in an `UPDATE` statement, so that the `UPDATE` does not try to update read-only system properties.
- **Utils.deepMerge(target, source)** - recursively merges the source object into the target object. If the target object has a property that is an object, the source object’s property is merged into the target object’s property. If the target object has a property that is an Array, the source object’s property is merged to the target object’s property by array index. If the target object has a property that is a scalar, the source object’s property replaces the target object’s property. The target object is modified in place and returned, while the source object is not modified. Here is an example:

```
    var target = {
        'a': [{ 'b': 2 }, { 'd': 4 }]
    }

    var source = {
        'a': [{ 'c': 3 }, { 'e': 5 }]
    }

    Utils.deepMerge(target, source)
    // target becomes => { 'a': [{ 'b': 2, 'c': 3 }, { 'd': 4, 'e': 5 }] }

    target = {
        'a': 1
    }

    source = {
        'a': { 'b': 2 }
    }

    Utils.deepMerge(target, source)
    // target becomes => { 'a': { 'b': 2 }}
```

When the source value for a particular key is not an Object or an Array, the source value replaces the target value. For example:

```
var target = {
    'a': { 'b': 1 }
}

var source = {
    'a': 2
}

Utils.deepMerge(target, source)
// target becomes => { 'a': 2 }
```