API Reference Guide

Overview

The Vantiq API provides access to the Vantiq resource model in a programmatic fashion. The core concepts of the API are:

- **resources** – resources define a class of object all of which have a common set of attributes. The resource defines what properties will exist and the behavior that instances of the resource may exhibit.
- **resource instances** – an object which represents a specific incarnation of the more general “resource”. All resource instances can be uniquely identified and will respond to specific operations.
- **operations** – actions that you can perform on a resource and/or its instances via the API.

In this document we start by presenting the general resource model and the operations that all resources support. We then present the supported API bindings which allow the API to be used from different contexts. These bindings are:

- REST over HTTP – provides access to the API via the HTTP protocol. In this binding resource instances are represented as URIs and operations are mapped to HTTP verbs.
- JSON over WebSockets – provides access to the API over a persistent WebSocket connection. Each operation is represented by a JSON document which specifies the resource, resource instance, operation, and any parameters.
- VAIL – provides access to the API from VAIL code (see the Rule and Procedure Reference Guide for more details). As with WebSockets, each operation is represented by a JSON document which specifies the resource, resource instance, operation, and any parameters.

Vantiq also provides several SDK’s which express the API in various languages such as Java or Python (these SDKs are covers over the REST binding presented here, designed to make it easier for programmers familiar with a specific language to use).

Accessing the API

The Vantiq Platform is a cloud-based system, so in most cases use of the API means interacting with a remote system over the Internet. To use the API you must know which Vantiq server you are accessing, you must be provisioned for access to that server, and you must authenticate your identity to the server so it knows which resources/resource instances you should have access to.

Service Endpoints

The Vantiq API is accessible through specific service endpoints:

- Cloud production accounts are accessible at https://api.vantiq.com.
- Cloud developer accounts are accessible at https://dev.vantiq.com.
- If you have custom deployment, the services are at an endpoint selected during the deployment process. Consult your system administrator for the specific address.

The remainder of this document assumes a developer endpoint, https://dev.vantiq.com.

Provisioning

A customer may request Vantiq to provision access to either the developer cloud or production systems. Vantiq will provision an organization with an initial namespace and organization administrator account. The credentials for the organization administrator will be provided to the customer’s authorized contact. Using these credentials the customer can self-provision additional namespaces and associated user accounts, subject to licensing constraints. See the User and Namespace Administration Guide for more details.

Authentication

Vantiq resources are secure and the requesting client must authenticate before issuing requests against the services. The details of this process are specific to each binding and are covered in those sections.

API Resource Types

The API provides access to both the **system** resources provided by the Vantiq server and to **custom** resources defined and deployed as part of a Vantiq “smart system”. Details on the Vantiq system resources can be found in the Resource Reference Guide. The details for any custom resources are obviously outside the scope of this documentation; however, in general any *type* which is defined in a namespace will appears as a custom resource to users with access to that namespace.

API Operations

Resources can be acted upon using the operations described in this section.

Examples of the use of these operations are contained in the REST API Examples and WebSockets API Examples sections. See also Client Examples. Some of the system types also support type-specific operations, which are described in the Resource Reference Guide

SELECT

The SELECT operation performs a query to retrieve matching instances of a given resource type. SELECT supports the following parameters:

Parameter	Type	Description	Example
where	JSON Object	Criteria used to filter the results (see where parameter)	`where={ "age":{"$gt":10}}`
sort	JSON Object	Order in which the results are returned (see sort parameter)	`sort={"lastName":-1}`
props	JSON Array	List of the properties that should be included in each resource record (see props parameter)	`props=["firstName","lastName"]`
page	Integer	Page of results to return, starting with 1	`page=2`
limit	Integer	Number of results to return in a single page	`limit=10`
count	“true”	If set to *true*, the count of selected instances will be returned (how is binding specific)	`count=true`

INSERT

The INSERT operation creates one or more new instances of a given resource type.

UPDATE

The UPDATE operation updates a single preexisting instance of a given resource type. The instance must be identified using its resource id.

UPSERT

The UPSERT operation either creates a new instance or updates an existing instance of a given resource type. The instance to be inserted/updated must be identified by its resource id.

DELETE

The DELETE operation removes existing instances of a given resource type. DELETE supports the following parameters:

Parameter	Type	Description	Example
where	JSON Object	Criteria used to identify which instances to remove (see where parameter)	`where={ "value":{"$gt":10}}`
count	“true”	If set to *true*, the count of deleted instances will be returned (how is binding specific)	`count=true`

PATCH

The PATCH operation performs operations on an identified resource instance according to a modification commands defined in JSON Patch.

AGGREGATE

DEPRECATED as of 1.26 – this operation will be removed in a future version of the system so you should update your applications to remove its use.

The AGGREGATE operation executes an aggregation pipeline involving the specified resource. AGGREGATE supports the following parameters:

Parameter	Type	Description	Example
pipeline	JSON Array	The aggregate pipeline defined in the aggregate pipeline format.	`pipeline=[{"$match": {}}]`

Standard Operation Parameters

`where` Parameter

SELECT and DELETE accept a *where* parameter which is a JSON object used to filter retrieved objects (SELECT) or delete objects according to the filter (DELETE). The *where* parameter accepts the following operators:

Operator	Symbol	Example
Equals		`{ prop: 42 }`

Operator	Symbol	Example
Greater Than	`$gt`	`{ prop: { "$gt":  42 } }`
Greater Than or Equal	`$gte`	`{ prop: { "$gte": 42 } }`
Less Than	`$lt`	`{ prop: { "$lt":  42 } }`
Less Than or Equal	`$lte`	`{ prop: { "$lte": 42 } }`
Not Equal	`$ne`	`{ prop: { "$ne":  42 } }`
Match one in list	`$in`	`{ prop: { "$in":  [ 13, 26, 42 ] } }`
Match all in list	`$all`	`{ prop: { "$all": [ 13, 26, 42 ] } }`
Do not match any in list	`$nin`	`{ prop: { "$nin": [ 13, 26, 42 ] } }`
Match on all operators	`$elemMatch`	`{ prop: { "$elemMatch": { "$gt": 13, "$lt": 26 } } }`
Regular Expression	`$regex`	`{ prop: { "$regex": "acme.*comp" } }`
Match on array size	`$size`	`{ prop: { "$size": 10 } }`

Operators can be combined using the following compound operators:

Logical AND (`$and`)

A logical AND operation uses the `$and` operator, e.g.,

```
{
    "$and": [
        { prop1: 10 },
        { prop2: { "$gt": 5 } }
    ]
}
```

Logical OR (`$or`)

A logical OR operation uses the `$or` operator, e.g.,

```
{
    "$or": [
        { prop1: 10 },
        { prop2: { "$gt": 5 } }
    ]
}
```

Logical NOT (`$not`)

A logical NOT operation uses the `$not` operator, e.g.,

```
{
    prop: { "$not": { "$gt": 5 } }
}
```

Logical NOR (`$nor`)

A logical NOR operation is one that does not satify any of the given expressions. The logical NOR uses the `$nor` operator, e.g.,

```
{
    "$nor": [
        { prop1: { "$gt": 42 } },
        { prop2: { "$eq": "abc" } }
    ]
}
```

GeoJSON

For GeoJSON data types, the following operators are supported:

Intersection (`$geoIntersects`)

To determine if the given location intersects with a given shape, use the `$geoIntersects` operator. E.g,

```
{
    prop: {
        "$geoIntersects": {
            "$geometry": {
                type:        "Polygon",
                coordinates: [ [0,0], [1,0], [1,1], [0,1] ]
            }
        }
    }
}
```

Within (`$geoWithin`)

To determine if the given location resides within a given shape, use the `$geoWithin` operator. E.g.,

```
{
    prop: {
        "$geoWithin": {
            "$geometry": {
                type:        "Polygon",
                coordinates: [ [0,0], [1,0], [1,1], [0,1] ]
            }
        }
    }
}
```

Proximity (`$near` and `$nearSphere`)

To determine if the given location is a given distance away from a given point, use the `$near` operator. E.g.,

```
{
    prop: {
        "$near": {
            "$geometry": {
                type:        "Point",
                coordinates: [ 13, 42 ]
            },
            "$maxDistance": 4,
            "$minDistance": 3
        }
    }
}
```

To perform a proximity search using spherical geometry, use the `$nearSphere` operators. E.g.,

```
{
    prop: {
        "$nearSphere": {
            "$geometry": {
                type:        "Point",
                coordinates: [ 13, 42 ]
            },
            "$maxDistance": 4,
            "$minDistance": 3
        }
    }
}
```

`props` Parameter

In the SELECT operation, the `props` parameter defines the list of properties to return within each resource instance. The `props` parameter should be a JSON array where each element is the name of the desired property to include. The following `props` parameter requests that the SELECT operation only return the `name`, `ts`, and `value` properties in each resource instance returned.

```
[ "name", "ts", "value" ]
```

`sort` Parameter

In the SELECT operation, the `sort` parameter defines the order of the results. The `sort` parameter is a JSON object where the property indicates the field to sort on and the value indicates if the sort is ascending or descending. A value of *1* means ascending order and *-1* means descending order. For example, the following requests to sort the results by the name field in descending order:

```
{
    "name": -1
}
```

API Responses

Every response to an API request has an indication of success or failure, the result of the operation, and possibly meta-data which helps further describe the result. Each of the bindings communicates these differently as will be described in the next section.

API Bindings

REST Over HTTP Binding

Authentication

Authentication is performed via the following HTTP request:

```
GET https://dev.vantiq.com/authenticate
```

The request must include an `Authorization` HTTP header. For installations using OAuth logins, the header must contain a token created in the target namespace, in the form:

```
Authorization: Bearer <access token>
```

For an installation that is not using OAuth, the request may use either the token form shown above OR standard HTTP basic authentication credentials, consisting of:

```
Authorization: Basic <encoded credentials>
```

The `<encoded credentials>` are created by Base64 encoding the concatenation of the username and password with a colon separator (i.e. `username:password`).

Upon successful authentication, a JSON object is returned that includes an access token. This access token is valid for 24 hours from the authentication request and must accompany each subsequent HTTP API request. The preferred mechanism for providing the token is via the Authorization header, using the Bearer realm:

```
Authorization: Bearer <access token>
```

The token can also be provided on the request URI via the `token` URI parameter:

```
GET https://dev.vantiq.com/docs/rcs/image/3F00D4D1-C2CB-45BF-A976-3C9C553C8651.png?token=<access token>
```

This approach is most often used to provide access to a 3rd party application or system that can only consume URIs and cannot set the HTTP header.

The access token identifies the caller and authorizes the request. Therefore, the token should be kept confidential by the client to which it was issued. When the access token expires, the caller must re-authenticate to retrieve a new token.

In addition to short term, user specific tokens, authentication can be performed using long lived access tokens. These tokens typically represent a specific application or 3rd party system that has been granted access to Vantiq for some purpose. For more details see the Tokens section of the Resource Reference Guide.

Namespace

A request can target a specific namespace by either including the HTTP header `X-Target-Namespace` or using the `targetNamespace` URI parameter:

```
GET http://localhost:8080/api/v1/resources/types
X-Target-Namespace: MySampleNS
```

or,

```
GET http://localhost:8080/api/v1/resources/types?targetNamespace=MySampleNS
```

To succeed, the authenticated request must have the proper privileges on the targeted namespace.

Resource URIs

Vantiq supports REST-based access to the resources using the following resource URI structure:

```
/api/v<version>/resources/[custom/]<resource name>[/<resource identifier>]
```

The current `version` of the API is 1, (i.e. `/api/v1/...`).

The optional `/custom` sub-path is included when accessing custom resources that correspond to user defined types. It is not used when accessing Vantiq system resources.

The `<resource name>` can be one of the Vantiq resources defined in the Resource Reference Guide or can be the name of a user-defined data type.

For some operations, the `<resource identifier>` is needed to uniquely identify a specific instance of a resource. For example, the *users* endpoint refers to a user resource type, which has a property *username* defined as natural key. The URI:

```
/api/v1/resources/users
```

refers to all user instances whereas:

```
/api/v1/resources/users/joe
```

refers to a specific user, *joe*.

Operation to HTTP Method Mapping

API operations are issued to the HTTP REST API by mapping them to the HTTP method. The following table describes all the supported operations and their corresponding HTTP structure. All URIs are relative to `/api/v#/resources`:

Operation	URL Path	URI Params	Method	Description
SELECT	`/[custom/]<name>`	where, props, sort, count, page, limit	GET	Returns all matching resource instances of type `<name>`.
SELECT	`/[custom/]<name>/select`	props, sort, count, page, limit	POST	Returns all matching resource instances of type `<name>`. The `where` clause is provided as the body of the request.
SELECT	`/[custom/]<name>/<id>`		GET	Returns a single instance of the resource of type `<name>` identified by `<id>`.
INSERT	`/[custom/]<name>`		POST	Inserts a new instance of resources of type `<name>`
UPDATE	`/[custom/]<name>/<id>`		PUT	Updates an existing instance of resources of type `<name>` identified by `<id>`
UPSERT	`/[custom/]<name>?upsert=true`		POST	Upserts a new instance of resources of type `<name>`
DELETE	`/[custom/]<name>`	count	DELETE	Removes all matching resources of type `<name>`.
DELETE	`/[custom/]<name>/<id>`		DELETE	Removes the resource instance of type `<name>` identified by `<id>`
PATCH	`/api/v#/resources/[custom/]<name>/<id>`		PATCH	Patches the resource instance of type `<name>` identified by `<id>`
AGGREGATE	`/api/v#/resources/[custom/]<name>/aggregate`	pipeline	GET	Executes a given aggregate pipeline for the resource of type `<name>`
AGGREGATE	`/api/v#/resources/[custom/]<name>/aggregate`		POST	Executes a given aggregate pipeline for the resource of type `<name>` where the requested pipeline is provided in the request body as a JSON document.

The following are type specific operations that are supported by the HTTP REST API:

Operation	URL Path	Query Params	Method	Description
PUBLISH	`/<name>/<id>`		POST	Publishes the JSON payload in the request body. Only *topics*, *sources* and *services* support the publish operation.

Operation	URL Path	Query Params	Method	Description
EXECUTE	`/services/<id>/<procedureName>`	stream, maxBufferSize, maxFlushInterval	PUT	Executes the specified procedure from the specified service and returns the results. The arguments to the procedure are provided as either a JSON array (positional arguments) or a JSON document (named arguments).
EXECUTE	`/procedures/<id>`	stream, maxBufferSize, maxFlushInterval	POST	Executes the specified procedure and returns the results. The arguments to the procedure are provided as either a JSON array (positional arguments) or a JSON document (named arguments).
QUERY	`/sources/<id>/query`		POST	Performs a query against the specified source. The parameters for the query are provided in the request body as a JSON document.
AUTHORIZED USERS	`/namespaces/<namespace>/authorizedUsers`		GET	Returns the users who are authorized in the specified namespace (must be admin)
NAMESPACE AUTHORIZATION	`/namespaces/<namespace>`		POST	Depending on the command specified, will either authorize a user for the given namespace or revoke authorization from an existing user. The arguments to the command are supplied as a JSON document. See the Namespaces resource for more details.

The extra parameters supported by the `EXECUTE` operation are:

Parameter	Type	Description	Example
stream	Boolean	If `true` and the returned value is a sequence, then the results will be returned incrementally using HTTP chunked encoding.	
maxBufferSize	Integer	The maximum size (in bytes) of the buffer used to hold intermediate results. Once the buffer size is exceeded, it will be flushed to the network. The default value is 64K.	
maxFlushInterval	Integer	The maximum time (in milliseconds) between buffer flushes. A value of `0` (the default) means that the buffer will only be flushed based on size. A value of `-1` indicates that the buffer should be flushed after every write.	

Response Status

Operation responses are communicated as the result of the operation's HTTP request. The overall status of the request is communicated via the HTTP response code. The HTTP status codes are divided into the following classes of responses:

- `2xx` Success responses indicating the requested operation completed without failure
- `4xx` Failure responses indicating that the client submitted an invalid request
- `5xx` Failure responses indicating that the server failed to process a valid request

The following are the supported specific responses

Codes	Type	Reason
200	Success	Success with a response body provided
204	Success	Success with no response body provided
400	Client Failure	Invalid request, usually indicating the request body is incorrectly formatted
401	Client Failure	The requester has not been authorized or the access token has expired
403	Client Failure	The requester is authorized but forbidden to access the requested resource
404	Client Failure	The requested resource cannot be found or the requester is forbidden to access the requested resource
405	Client Failure	The request used an HTTP verb that is not accepted by the requested resource
409	Client Failure	The resource create or update request cannot be completed due to a uniqueness constraint

Codes	Type	Reason
410	Client Failure	Indicates that an event path to which the client is subscribed has been removed (typically due to resource removal).
429	Client Failure	The request cannot be completed due to a lack of resources or because a quota assigned to the requester
500	Server Failure	The server encountered an unexpected condition which the API user should report to Vantiq support

Some of the operations support processing of more than one resource instance in a single request (INSERT, for example). The Vantiq server does not treat these as a single atomic request (or transaction), so it is possible for the operation to succeed for some of the instances and fail for others. In this case, the response produced will have the following characteristics:

- The response code will be set to indicate the overall success or failure of the operations. Since the request was not rejected outright the server reports that there was at least some success.
- If requested, the count will be set to the number of successfully processed instances.
- The response body will consist of a JSON array with the same cardinality as the input array. Each member of this array will be the corresponding result for the same index in the original array and will contain the JSON representation of a standard response (either success or failure).

Response Body

The body of the response will contain any returned data associated with the requested operation. Typically, the body will be transported in JSON format, either a single document or array of documents. In the case of a successful operation, it will return any data requested. In the case of a failure, it will communicate the details of any errors.

Error Body

If an error occurs during the requested operation, Vantiq will return an array of JSON documents containing the details of each of the failures. Each error document contains the following:

Property	Type	Description
code	String	A unique identifier for the error.
message	String	A human readable message that describes the details of the error.
params	JSON Array	A list of values related to the error and used to construct the error message.

As mentioned, the VAIL binding communicates these as exceptions which can be inspected and processed programmatically.

Response Meta-data

The response meta-data provides additional information relating to the requested operation. For example, if the operation requested a count, that information is returned as meta-data. When using the API with REST over HTTP, the meta-data is communicated via HTTP response headers. Currently the following headers are supported:

- `X-Total-Count`: Returned when the request parameter *count* is true (e.g. `/api/v1/resources/types?count=true`)
- `Content-Type`: Indicates the format of the body of the response (e.g. `Content-Type: application/json`)

Public Resources

In all standard HTTP requests an Authorization header is expected to convey the identity and authorizations of the requester. Requests processed under the `/api/v1/resources/public` path ignore any specified authorization headers and instead restrict requests to only resource instances that have been marked with the `ars_public` flag set to `true`.

Unauthenticated requests for public resources must use slightly different url structures to disambiguate the target namespace. While a normal `GET` request to fetch an instance of a system resource would be structured like this:

```
GET https://dev.vantiq.com/api/v1/resources/<resource>/<resourceId>
```

Requests for public resources insert an additional `/public/<namespace>` fragment to the path to disambiguate the target namespace:

```
GET https://dev.vantiq.com/api/v1/resources/public/<namespace>/<resource>/<resourceId>
```

Most system resources only support public `GET` requests. Procedures, however, also support public execution, which is performed by making a `POST` request on the URL targeting the procedure like this:

```
POST https://dev.vantiq.com/api/v1/resources/public/<namespace>/procedures/<procedureName>
```

Requests made to the public API endpoint are restricted to only operate on resources that have been marked as publicly accessible. If a request targets a resource that is not public the result is a generic “Resource Not Found” error. A request for a non-existent resource or a request that specifies a non-existent namespace produce the same error.

Relationships

Vantiq can infer the relationships between resources to create a semantically rich understanding of a Vantiq project or namespace. For more information see the resource reference guide.

To query for the inferred relationships on a particular resource:

```
GET https://dev.vantiq.com/api/v1/resources/<resource>/<resourceId>/_relationships
```

This returns a list of all of the transitive relationships originating from the queried resource where each entry takes the form:

```
{
    "origin": "/<resource>/<resourceId>",
    "to": "/<resource>/<resourceId>",
    "type": "<relationship type>"
}
```

For example, for a given Procedure, *procedureA*, which calls *procedureB* which publishes to */topicA*:

```
GET https://dev.vantiq.com/api/v1/resources/procedures/procedureA/_relationships
```

returns the following:

```
[
    {
        "origin": "/procedures/procedureA",
        "to": "/procedures/procedureB",
        "type": "executes"
    },
    {
        "origin": "/procedures/procedureB",
        "to": "/topics//topicA",
        "type": "publishes"
    }
]
```

The `relationships` operation has two possible options defined:

- **direct** returns only the direct (non-transitive) relationships for the resource. Using the example above:

```
GET https://dev.vantiq.com/api/v1/resources/procedures/procedureA/_relationships?direct=true
```

returns

```
[
    {
        "origin": "/procedures/procedureA",
        "to": "/procedures/procedureB",
        "type": "executes"
    }
]
```

- **explicit** returns only the explicit relationships defined on the resource. See the section below for defining explicit relationships.

In cases where inferencing falls short, it may be useful to create explicit relationships on a resource. To accomplish this, set the `ars_relationships` property on any resource on insert or update. See the VAIL Reference Guide for applying explicit relationships to VAIL resources.

Example: To update an existing client with an explicit relationship between a Client and a Procedure:

```
Method: PUT
URL: /api/v1/resources/system.clients/myClientName
Body: { "ars_relationships": [{"to": "/procedures/myProceudreName", "type": "executes"}] }
```

```
GET https://dev.vantiq.com/api/v1/resources/procedures/procedureA/_relationships?explicit=true
```

returns

```
[
    {
        "origin": "/system.clients/myClientName",
        "to": "/procedures/myProceudreName",
        "type": "executes"
    }
]
```

Any select on this resource will include the `ars_relationships` property enumerating the explicit relationships on the resource.

Batch requests

It is possible to group a set of REST API calls into a single request. This is done by sending a POST request to the `/api/v1/batch` endpoint with a JSON body containing an array of objects. Each object in the array represents a single request and must contain the following properties:

```
{
    "method": <HTTP method>,
    "headers": <request headers>,
    "uri": <request URI>,
    "body": <request body>
}
```

The request `URI` is relative to the `/api/v1/resources` endpoint. The `X-Target-Namespace` header, when set on the `/api/v1/batch` POST request, targets a specific namespace for all batched requests. Setting this header on individual requests within the batch will override the target namespace for that specific request.

Requests are executed in parallel and each GET request within a batch is limited to 1000 rows. All requests must complete before the batch response is returned. The batch response is an array of objects, one for each request, with the following properties:

```
{
    "status": <HTTP status code>,
    "headers": <response headers>,
    "body": <response body>
}
```

Examples Using REST Over HTTP

The following examples refer to a user-defined `TempByZip` type, which is assumed to have three properties:

- `temperature` An Integer representing a temperature reading
- `zipCode` An Integer representing a five-digit US Zip Code
- `timeZone` A String representing the time zone of the Zip Code

When referencing specific resources, the resource's `_id` property, which is automatically assigned by Vantiq when a resource is created must be used. In the examples, `5696c2511de9a702175aadbb` is used as an example `_id`.

Objective: Retrieve a list of all user-created and system-defined database types.

```
Method: GET
URL: /api/v1/resources/types
```

Objective: Insert a new weather reading object of the *TempByZip* type.

```
Method: POST
URL: /api/v1/resources/custom/TempByZip
Body: { "temperature": 32, "zipCode": 96160, "timeZone": "Pacific" }
```

Objective: Update an existing weather reading object of the *TempByZip* type.

```
Method: PUT
URL: /api/v1/resources/custom/TempByZip/5696c2511de9a702175aadbb
Body: { "temperature": 90, "zipCode": 96160, "timeZone": "Pacific" }
```

Objective: Insert or update a weather reading object of the *TempByZip* type, depending on whether the ZIP Code already exists.

```
Method: POST
URL: /api/v1/resources/custom/TempByZip?upsert=true
Body: { "temperature": 32, "zipCode": 95389, "timeZone": "Pacific" }
```

Objective: Delete all the *TempByZip* objects associated with particular time zone.

```
Method: DELETE
URL: /api/v1/resources/custom/TempByZip?where={"timeZone":"Pacific"}
```

Objective: Replace the temperature value with 100 in a specific *TempByZip* object.

```
Method: PATCH
URL: /api/v1/resources/custom/TempByZip/5696c2511de9a702175aadbb
Body: [ { "op": "replace", "path": "/temperature", "value": 100 } ]
```

Objective: Retrieve all *TempByZip* objects with temperatures below 32:

```
Method: GET
URL: /api/v1/resources/custom/TempByZip/aggregate?pipeline=[{"$match":{"temperature":{"$lt":32}}}]
```

WebSockets Binding

WebSockets URI

Vantiq supports WebSocket access to the published resources. A WebSocket connection is created by connecting to the URL:

```
wss://dev.vantiq.com/api/v<version>/wsock/websocket
```

The current version of the API is 1, (i.e. `wss://dev.vantiq.com/api/v1/wsock/websocket`)

Authentication

Before a WebSocket connection can accept requests, the client must establish an authenticated session. Once established, this session is bound for the life of the connection and all requests made over that connection use the previously established identity. When the socket is closed, the session associated session is destroyed.

To establish a session using user credentials, the client must issue an `authenticate` request by sending a request document with the following properties:

- `op` : “authenticate”
- `resourceName` : “system.credentials”
- `object` : a JSON document with the properties:
  - `username` : the name of the user being authenticated
  - `password` : the password for the user being authenticated

For example, to authenticate the user `joe` with password `joe#!&^`, the client would send the following authentication request:

```
{
    "op": "authenticate",
    "resourceName": "system.credentials",
    "object": { "username": "joe", "password": "joe#!&^" }
}
```

To establish a session using an existing access token, the client must issue a `validate` request by sending a request document with the following properties:

- `op` : “validate”
- `resourceName` : “system.credentials”
- `object` : <access token>

For example,

```
{
    "op": "validate",
    "resourceName": "system.credentials",
    "object": "5696c2511de9a702175aadbb"
}
```

It may be necessary to retry either the *authenticate* or the *validate* operation at 100ms intervals since these operations are not guaranteed delivery to the server. The server responds to these operations with a JSON object containing a *status* field, which will carry the HTTP status of the operation. Usually the return status will be 200, which indicates a successful token authentication.

Request Format

Once authenticated, the operations defined in this document may be issued via the WebSocket session. To issue an request, the message sent to the WebSocket may contain the following properties:

Name	Type	Required	Description
`op`	String	Yes	The desired operation
`resourceName`	String	Yes	The type of the resource

Name	Type	Required	Description
`resourceId`	String	Depends on `op`	The unique identifier of the resource instance
`object`	JSON Object	Depends on `op`	The JSON-encoded data associated with the operation, if any
`parameters`	JSON Object	No	The parameters associated with the request (e.g. `count=true`)
`targetNamespace`	String	No	The specific namespace that this request must target

In addition to the standard operation parameters, the WebSockets binding also supports the `requestId` parameter. If used it should be set to a unique, opaque string (such as a UUID). The server will ensure that the same value will be returned in the response associated with this specific request (allows matching of responses to requests when processing multiple, simultaneous requests).

Response Format

The response from the server is communicated as a JSON object with the following properties:

- status – the overall status of the request. These use the same HTTP status code that were described in the previous section.
- headers – any headers associated with the response. The current headers supported are:
  - X-Request-Id – communicates the request id which was associated with the initial request.
- body – the response body
- contentType – the content type of the response body

Data returned from the server will be in BLOB (Binary Large OBject) format which must be converted to JSON. For example, if a TypeScript/JavaScript app is using WebSockets, then the following example shows how to receive response BLOBs:

```
// the WebSocket has received a message from the server
 private wsonmessage(evt:MessageEvent) {
   // the server returns a Blob in evt.data (rather than text)
   var blobReader:FileReader = new FileReader();
   var wsThis = this;
   blobReader.addEventListener("loadend", function() {
     wsThis.processMessage(wsThis.abToString(blobReader.result));
   });
   blobReader.readAsArrayBuffer(evt.data);
 }
```

The preceding example requires two callbacks, one to convert the BLOB to a string and the second to process the contents of the string into JSON:

```
private abToString(ab:ArrayBuffer) {
  var binaryString:string = '';
  var bytes:Uint8Array = new Uint8Array(ab);
  var length:number = bytes.length;
  for (var i = 0; i < length; i++) {
    binaryString += String.fromCharCode(bytes[i]);
  }
  return binaryString;
}
```

```
private processMessage(msg:string) {
  var evtObject:Object = JSON.parse(msg);
  if (evtObject["status"] == 200) {
    // we've gotten a good response from the server
    // evtObject["body"] contains the response data
    // evtObject["errors"] contains any errors associated with the request
  }
}
```

Response Meta-Data

Any response meta-data (such as counts) will be communicated as the first element of the response body and *not* in the headers.

Examples Using WebSockets

The following examples refer to a user-defined `TempByZip` type, which is assumed to have three properties:

- `temperature` An Integer representing a temperature reading
- `zipCode` An Integer representing a five-digit US Zip Code
- `timeZone` A String representing the time zone of the Zip Code

When referencing specific resources, the resource's `_id` property, which is automatically assigned by Vantiq when a resource is created must be used. In the examples, `5696c2511de9a702175aadbb` is used as an example `_id`.

Objective: Retrieve a list of all user-created and system-defined database types.

```
{
    "op": "select",
    "resourceName": "types"
}
```

Objective: Retrieve a list of all user-created and system-defined database types for a specific namespace.

```
{
    "op": "select",
    "resourceName": "types",
    "targetNamespace": "MySampleNS"
}
```

Objective: Insert a new weather reading object of the `TempByZip` type.

```
{
    "op": "insert",
    "resourceName": "TempByZip",
    "object": { "temperature": 32, "zipCode": 96160, "timeZone": "Pacific" },
    "parameters": { "requestId": "94563Reading" }
}
```

Objective: Update an existing weather reading object of the `TempByZip` type.

```
{
    "op": "update",
    "resourceName": "TempByZip",
    "resourceId": "5696c2511de9a702175aadbb",
    "object": { "temperature": 90, "zipCode": 96160, "timeZone": "Pacific" }
}
```

Objective: Insert or update a weather reading object of the `TempByZip` type, depending on whether the ZIP Code already exists.

```
{
    "op": "upsert",
    "resourceName": "TempByZip",
    "object": { "temperature": 32, "zipCode": 95389, "timeZone": "Pacific" }
}
```

Objective: Delete all the `TempByZip` objects associated with particular time zone.

```
{
    "op": "delete",
    "resourceName": "TempByZip",
    "parameters": { "where": { "timeZone": "Pacific" } }
}
```

Objective: Replace the temperature value with 100 in a specific `TempByZip` object.

```
{
    "op": "patch",
    "resourceName": "TempByZip",
    "resourceId": "5696c2511de9a702175aadbb",
    "object": [ { "op": "replace", "path": "temperature", "value": 100 } ]
}
```

Objective: Retrieve all `TempByZip` objects with temperatures below 32:

```
{
    "op": "aggregate",
    "resourceName": "TempByZip",
    "object": [{"$match":{"temperature":{"$lt":32}}}]
}
```

Subscribing Over WebSockets

The above examples demonstrate how to make requests that could be made over the REST API. WebSockets are persistent connections, which means they can support some things REST requests can’t because of their short life cycle. In particular,it’s possible to subscribe to events so all events on a resource on the Vantiq server are forwarded over the WebSocket to the client.

Creating a subscription requires first sending a request from your external application over the WebSocket to the Vantiq server and receiving a response indicating the subscription has been created. After that the Vantiq server will handle forwarding all events on the subscribed resource to your WebSocket connection. Each event sent over the WebSocket for a subscription will contain a status code of 100 to indicate more events may follow.

As an example, consider this request to subscribe to all source insert events:

```
{
    "op": "subscribe",
    "resourceName": "events",
    "resourceId": "/types/system.sources/insert",
    "parameters": {
        "requestId": "SubscriptionXYZ"
    }
}
```

Note that the op is `subscribe` and the resourceName is `events`. This is consistent across all subscription requests.

The `resourceId` property specifies the complete event path, which in this case indicates the events will occur on the `system.sources` type, and the events we're interested in are inserts.

The `parameters` map can optionally include a `requestId` which will be included in every event sent back to your application over the WebSocket. This is an opaque string, which can be used to disambiguate which subscription an event is for if multiple subscriptions were created over the same WebSocket.

After issuing the subscribe request, whenever an event occurs the WebSocket client will receive a message. For example, if the above subscribe request was made and then a source named NewSource was inserted into the namespace, the WebSocket client would receive the following message:

```
{
    "status": 100,
    "contentType": "application/json",
    "headers": {
        "X-Request-Id": "SubscriptionXYZ"
    },
    "body": {
        "path": "/types/system.sources/insert/NewSource",
        "value": {
            "name": "NewSource",
            "type": "MQTT",
            "active": true,
            "activationConstraint": "",
            "config": {
                    "passwordType": "string",
                    "contentType": "application/json",
                    "qos": "AT_LEAST_ONCE",
                    "keepAliveInterval": 15,
                    "connectionTimeout": 15,
                    "cleanSession": true,
                    "automaticReconnect": true,
                    "serverURIs": ["tcp://localhost:1883"],
                    "topics": ["test.topic.XYZ"]
            },
            "ars_namespace": "modelo",
            "ars_version": 1,
            "ars_createdAt": "2019-01-15T02:07:34.914Z",
            "ars_createdBy": "modelo",
            "_id": "5c3d4067a725e9e0ff41c07a"
        },
        "username": "<USERNAME OF SUBSCRIBER>",
        "namespace": "<NAMESPACE OF SUBSCRIBER>"
    }
}
```

The `status` of 100 indicates this is just an event, and more may still come.

The `headers` include the `requestId` (as `X-Request-Id`) that was specified in the initial subscription.

The `body` defines the `path` on which the event occurred and the `value` contains the complete source definition.

The client may receive a response with the status `410` (aka **GONE**). This indicates that the event path to which the client was subscribed has been removed. This most typically happens when the resource associated with the event path was deleted.

Filtering Subscribed Events

Sometimes a client only wants to receive a sub-set of the events sent for a given subscription. This is accomplished by setting the `eventFilter` parameter on the subscription request. The value given must be an `Object`. In order to be delivered, the event *value* must also be an `Object` and it must contain matching values for any property included in the filter.

For example, to receive an event when a new REMOTE source was created, the client would issue the following subscription request:

```
{
    "op": "subscribe",
    "resourceName": "events",
    "resourceId": "/types/system.sources/insert",
    "parameters": {
        "requestId": "SubscriptionXYZ",
        "eventFilter": {
            "type": "REMOTE"
        }
    }
}
```

Subscribing to reliable events

To subscribe to reliable events over a WebSocket, set the `persistent` parameter to true. For example:

```
{
    "op": "subscribe",
    "resourceName": "events",
    "resourceId": "/topics/myTopic",
    "parameters": {
        "persistent": true
    }
}
```

The `subscribe` response contains a subscription name:

```
{
  "status": 200,
  "contentType": "application/json",
  "body": {
    "name": "a5d77f40-edf4-11ed-a21a-469595d141dc",
    "namespace": "MyNamespace",
    "topic": "/topics/myTopic"
  }
}
```

To acknowledge the event, use the subscription name in conjunction with the partitionId and sequenceId.

For example, assuming the following event:

```
{
  "status": 100,
  "contentType": "application/json",
  "headers": {},
  "body": {
    "path": "/topics/myTopic/publish",
    "createdAt": 1683587362415,
    "partitionId": 0,
    "sequenceId": 2,
    "value": {"name": "bob"},
    "redelivered": false,
    "namespace": "MyNamespace",
    "username": "alice"
  }
}
```

Send the following `acknowledge` message:

```
{
    "op": "acknowledge",
    "resourceId": "/topics/myTopic",
    "resourceName": "events",
    "parameters": {
        "partitionId": 0,
        "sequenceId": 2,
        "subscriptionName": "a5d77f40-edf4-11ed-a21a-469595d141dc"
    }
}
```

Note that the reliable event has a `redelivered` attribute, which indicates whether the event is being delivered for the first time (value set to false) or if it is being redelivered due to not being acknowledged yet (value set to true).

Server side Unsubscribe

Under certain circumstances, the server may decide to terminate an active subscription. For example, if a namespace is deleted, then all subscriptions in that namespace will be unsubscribed. In this case, it may be necessary for the client to take some action based on the fact that the subscription no longer exists. Therefore, any time the server decides to unsubscribe from a subscription that was established via WebSockets, it will send the client an "unsubscribe" message, which looks like:

```
{
  "status": 100,
  "contentType": "application/json",
    "headers": {
        "X-Request-Id": "SubscriptionXYZ"
    },
  "body": {
    "op": "unsubscribe",
    "path": "/topics/myTopic/publish",
    "namespace": "MyNamespace",
    "subscriptionName": "a5d77f40-edf4-11ed-a21a-469595d141dc"
  }
}
```

VAIL Binding

The VAIL binding provides API access via the `ResourceAPI` built-in service.

Request Format

In the VAIL binding, API requests are issued using the server's native message format. Similar to WebSockets API requests are represented as JSON objects with the following properties:

Name	Type	Required	Description
`op`	String	Yes	The desired operation
`resourceName`	String	Yes	The type of the resource
`resourceId`	String	Depends on `op`	The unique identifier of the resource instance
`object`	JSON Object	Depends on `op`	The JSON-encoded data associated with the operation, if any
`parameters`	JSON Object	No	The parameters associated with the request (e.g. `count=true`)
`targetNamespace`	String	No	The specific namespace that this request must target

Response Format

The execution of an operation results in a sequence which contains the operation's results. Depending on the output of the operation this sequence may contain zero, one, or many elements and the element types may be any of those supported by VAIL.

Response Errors

Response errors are communicated via VAIL exceptions. These exceptions can be caught and examined just like any other VAIL exception (see `TRY` statement for more details).

Examples Using VAIL

The following examples refer to a user-defined `TempByZip` type, which is assumed to have the following properties:

- `temperature` An Integer representing a temperature reading
- `zipCode` An String representing a five-digit US Zip Code

Objective: Compute the average of all `TempByZip` objects grouped by zip:

```
var avgTmpMsg = {
    "op": "aggregate",
    "resourceName": "TempByZip",
    "object": [{"$group":{"_id": "$zipCode", "avgTmp": {"$avg": "$temperature"}, "count": { "$sum": 1 }}}]
}

var avgByZip = ResourceAPI.executeOp(avgTmpMsg)
... process resulting sequence ...
```

Objective: Find the users who are authorized in the current namespace:

```
var authZUsers = {}
authZUsers.op = "getAuthorizedUsers"
authZUsers.resourceName = "system.namespaces"
authZUsers.resourceId = Context.namespace()

ResourceAPI.executeOp(authZUsers)
```