CLI Reference Guide

Overview

The Vantiq Command Line Interface (CLI) is a tool for managing development resources for a Vantiq project. To download the CLI, click *Help* -> *Developer Resources* and look for the section on the CLI in the *Developer Resources* pane that opens or click here.

The Vantiq CLI manages system defined resources such as:

- types
- rules
- procedures
- services
- sources
- documents
- semanticindexes
- llms
- namespaces
- users
- tests
- testsuites
- topics
- tokens
- profiles
- projects
- systemmodels
- scheduledevents
- subscriptions
- aicomponents
- collaborationtypes - apps are exported as collaborationtypes, too
- configurations
- catalogs
- debugconfigs
- deployconfigs
- environments

The CLI will also manage user defined resources (though not all commands apply to these resources).

The major commands available in the CLI include:

- **help**
- **version**
- **list** <resource>
- **find** <resource> <resourceId>
- **dump** <types | semanticindexes>
- **load** <resource> <filename>
- **delete** <resource> <resourceId>
- **deleteMatching** <resource> <query>
- **execute** <procedureName> <p1> … <pN>
- **run** <test | testsuite | procedure> <testName | testSuiteName | procedureName> [<p1> … <pN>]
- **stop** <testReportId>
- **select** <resource> [<resourceId>] [-qual <filename> | <query>] [-props <filename> | <propertyList>] [-chunk <size>]
- **insert** <resource> <fileName>
- **upsert** <resource> <fileName>
- **checkedInsert** <resource> <fileName>
- **checkedUpsert** <resource> <fileName>
- **recommend** [<recommendationProcedure>] <productId>
- **deploy** <configurationName> | <deploymentName>
- **undeploy** <configurationName> | <deploymentName>
- **pull** <pullDescriptorFileName>
- **export** [data | metadata | project <projectName> | projectdata <projectName> | hidden] [-d <directory>] [-chunk <size>] [-include <typeName(s)>] [-exclude <typeName(s)>] [-until <DateTime>] [-ignoreErrors]
- **import** [data | metadata] [-d <directory>] [-chunk <size>]

Command line options available include:

- -s <profileName>
- -b <baseURL>
- -u <username>
- -p <password>

- -t <token>
- -f <profileFile>
- -v

Installation

Prerequisites

The Vantiq CLI is a Java (Groovy) application and requires an installation of Java 11.

Download

The Vantiq CLI is available as a downloadable zip file accessible from the Resources pane of the developer UI.

Once downloaded, the zip file can be expanded in any directory and will create a sub-directory with the name `vantiq-x.x.x`, where `x.x.x` is the version of the CLI. It is recommended that the directory `./vantiq-x.x.x/bin` be added to your path. Once installed and the PATH set, the CLI can be invoked on the command line. On Mac/Linux:

```
vantiq <command>
```

On Windows operating systems:

```
vantiq.bat <command>
```

The following examples in this document use the Mac/Linux form, `vantiq`. If you are on Windows, use the `vantiq.bat` command.

Profile

The command line can obtain credentials from a profile file. On Mac/Linux, the profile file is `~/.vantiq/profile`. On Windows, the profile file is `%UserProfile%\.vantiq\profile`. The profile filename can be changed using the '*-f*' flag.

Entries in the profile file take the following form:

```
base
{
    url = 'https://dev.vantiq.com'
    username = 'myUsername'
    password = 'myPassword'
}
oauth1 {
    url = 'https://dev.vantiq.com'
    // this is my token for namespace ABC
    token = 'rTTbtHd8Z7gFPEQPE32137HfYNDg8YA84zmOWtVbdYg='
}
personal {
    username = 'myPersonalName'
    password = 'myPersonalPassword'
}
```

Specifying the 'url' is optional. The url will default to the standard values (depicted in the example). If both token *and* password are specified, the password is used instead of the token.

The profile file may contain multiple profiles with each profile given a name. These profiles can then be easily referenced when invoking the vantiq CLI via the '*-s*' flag.

Notes:

- public clouds and any server using keycloak access require use of the *token* option (as in *oauth1* above). The token needs to be a long-lived access token created on the server.
- username/password can only be used for Edge servers. That form will log you into your default namespace (or home namespace if no default is set). To log into a different namespace, use the *token* option or the *namespace* option.
- the *namespace* option can only be used with username/password; it cannot be used with long-lived access tokens.

HttpClient options can also be specified in the profile. Below is an example:

```
withProxy {
    username = 'myPersonalName'
    password = 'myPersonalPassword'
    clientOptions {             # New (any attribute from HttpClientOptions)
        trustAll = true
        verifyHost = false
        forceSni = true
        proxyOptions {
            host = 'proxyhost'
            port = 8888
            username = 'pname'
            password = 'pname_pwd'
        }
    }
}
```

A target namespace can also be specified. For example:

```
personal2 {
    username = 'myPersonalName'
    password = 'myPersonalPassword'
    // my default namespace is ABC,
    // but use namespace DEF instead of default
    namespace = 'DEF'
}
```

If a namespace is specified both in profile and as a command line option, the command line option prevails.

Command Line Options

-s profileName

Specify the name of a profile, stored in: *~/.vantiq/profile*, that supplies the URL of the service implementation to access and the credentials used to access the service. Default: *base*

-b baseURL

Specifies the base URL that identifies the Vantiq system. Default: *https://dev.vantiq.com*

-u username

Specifies the user’s assigned username to connect to the Vantiq system.

-p password

Specifies the user’s assigned password to connect to the Vantiq system.

-t token

Specifies the user’s access token to connect to the Vantiq system. If a password is specified, it is used instead of the token.

If the -s option is used to specify a profile and any of the -b, -u, -p, or -t options are also specified, then the command-line options override the values that are provided in the *~/.vantiq/profile* file.
Note that the profile file should be in *ASCII* or *UTF-8* file-encoding.

-n namespace

Execute the command using the specified target namespace instead of the user’s default namespace. This option will not work when using a long-lived access token. It only works with username/password credentials.

-trust

Force SSL options to trust remote server certificate and host name. Same as setting clientOptions *trustAll=true* plus *verifyHost=false*.

-v

Prints the CLI version and the URL for the connected Vantiq service.

Other Options

Options that are unique to specific CLI commands are described in the documentation for each command.

Supported Commands

Help

The help command displays a short summary of the commands available in the CLI.

```
vantiq help
```

List

The list command displays a list of all resources of the type specified in the command.

```
vantiq list <resource>
```

There are no variations of the command. This command applies to both system and user-defined resources. For system resources it will display the resource's primary identifier (typically "name"). For user defined resources it prints the id property.

Find

The find command finds an individual instance of a resource by name or query.

```
vantiq find <resource> <resourceId>
```

When performing a find for system defined resources you can use the <resourceName> variant, since all system resources are known to have this property. For user defined resources use the <resourceId> option instead. In this case you provide a query which selects the desired instance(s). The query is expressed as a JSON document in the same form as the `where` parameter in the REST API.

The output is the "external" form of the given resource instance which can be used for either the **load** or **import** commands. Typically this will be a single JSON file named "<resourceId>.json", with the following exceptions:

- *rule* The output for a rule is VAIL code. The find command places the code in a file in the current directory named `<resourceName>.vail`
- *procedure* The output for a procedure is VAIL code. The find command places the code in a file in the current directory named `<resourceName>.vail`
- *document* A document contains either textual or binary data (depending on its content type). The find command places the contents of the document in a file in the current directory named `<resourceName>`
- *user defined resources* The file created is called `<resource>.json`. It contains the JSON representation of the selected instance(s).

Dump

The dump command provides an efficient means to export resource definitions and data for types and semantic indexes. The result of the command is a `.dmp` file which is a self-contained and compressed data file that can be used with subsequent `load` commands. This is particularly useful for exporting large types and semantic indexes for either backup or migration purposes. The command format is:

```
vantiq dump semanticindexes <index name>
vantiq dump types <type name>
```

Load

The load command will load a resource instance effectively defining or redefining the named resource. The load command accepts the "external" form of the resource as produced by **find** and **export**. For most resources this consists of a single file containing the JSON representation of the artifact. For these resources the command format is:

```
vantiq load <resource> <filename>
```

The exceptions to this are:

- *rule* The input file contains the VAIL code which defines the rule.
- *procedure* The input file contains the the VAIL code which defines the procedure.
- *document* The command accepts either a single file or a directory as its input. If a file is provided, it will be loaded using its simple name (ignoring any directories). If a directory is provided, the load document operation will recursively inspect the specified directory and all sub-directories and load all files found in the inspected directories. The name of the document is set to the name of the file, relative to the root directory.
- *user defined resources* The input file contains a JSON array containing one or more resource instances. For load to be successful the resource must support the `upsert` operation (meaning that it must have a natural key defined or a `name` property).
- *types* The input is a `.dmp` file created by the **dump** command which contains both the resource definition and a binary file containing a snapshot of the data for the type.
- *semanticindexes* The input can be either a `.json` containing the index definition or a `.dmp` file created by the **dump** command which contains both the resource definition and a snapshot of the entries with associated vector data.
- *semanticindexentries* The input is the name of the index into which the entries should be loaded followed by one of the following:
  - a file containing the content for the index entry
  - a directory containing potentially many content files for the index entry (these will be combined in a ZIP file)

- a URI pointing to a file containing the index entry content

Delete

The delete command is used to delete an resource instance.

```
vantiq delete <resource> <resourceId>
```

Any artifact may be deleted although namespaces and users can only be deleted by sufficiently privileged administrators. For system resources the `resourceId` is the name of the instance while for user defined resources it is the `id` property.

Select

The select command is a convenience to allow you to retrieve data from the Vantiq database without having to develop an app. Any data type for which the user has read privileges may be queried.

```
vantiq select <resource> [<resourceId>]
```

The **resource** parameter specifies the name of the resource from which to select.

The **resourceId** parameter is optional. If specified then it will be used to query for the resource instance with the given id. If resourceId is not specified, all instances of the resource will be returned.

The **-qual <fileName>** parameter is optional. If specified, the fileName must be a file which contains a qualifier to restrict the data being selected. For example, the contents could be: `{"property1": {"$lt" : 100}}`.

The **-props <fileName> | <propertyList>** parameter is optional. If specified, the value must be either the name of a file containing the property list or a string directly specifying the property list. The property list must be given as a JSON array containing the property names as strings (using the same format as the “PROPS” parameter of the REST over HTTP binding).

Insert

The insert command is a convenience for loading relatively small quantities of data into the Vantiq database (recommended maximum 1000 objects).

The data is placed in a file in JSON format as an array of JSON objects. The insert command parses the JSON and sends the resulting objects to the Vantiq services for insert into the database.

The operation performed is a standard insert action. If an object with the same values already exists and the type allows duplicates. Each invocation of the insert command using the same file will add duplicate objects to the database. See **Upsert** for an alternative that will not insert duplicate objects.

type is the data type in which to insert the objects.

file is the name of the file containing the array of JSON objects to upsert.

Upsert

The upsert command is a convenience for loading relatively small quantities of data into the Vantiq database (recommended maximum 1000 objects).

The data is placed in a file in JSON format as an array of JSON objects. The upsert command parses the JSON and sends the resulting objects to the Vantiq services for insert or update into the database.

An update is performed if the naturalKey for the object references an object already in the database. An insert is performed if the naturalKey for the object does not reference an object already in the database. The naturalKey consists of those properties specified as the natural key for the type. If no naturalKey is specified, **name** is assumed to be the natural key.

```
vantiq upsert <type> <file>
```

type is the data type in which to upsert the objects.

file is the name of the file containing the array of JSON objects to upsert.

CheckedInsert/CheckedUpsert

When loading larger quantities of data it is not always convenient to construct the data as a single array of JSON objects in a text file. *CheckedInsert* and *CheckUpdate* support a file format in which each object is represented as a separate JSON object with the JSON object definitions arranged sequentially within the file.

At runtime each object is parsed separately and added to the database. If a JSON object fails to parse, an error is reported with the index number of the object within the file and an attempt is made to continue the operation with the next JSON object in the file.

To ensure parsing errors can easily and uniquely identify the cause, the file should be formatted in the following manner:

- Each object must start with the opening brace on a separate line.
- Each object must end with the closing brace on a separate line.
- Objects must be comma-separated.

- If object definition contains nested objects, do not place a nested opening brace on a line by itself as it will be confused with a opening brace indicating a new object.
- All objects must be enclosed by opening and closing square brackets.

Example:

```
[{
  "name": "a sample name",
  "address": "a sample address"
},
{
  "name": "second sample name",
  "address": "second sample address"
}]
```

Example of illegal JSON file:

```
[{
  "name": "a sample name",
  "address":
  {
  "street": "a sample street",
  "city": "a sample city"
  }
},
{
  "name": "second sample name",
  "address": "second sample address"
}]
```

This example is illegal because the brace opening the definition of the nested `address` object is the first character on the line. The example could be converted to a legal form by placing the opening brace directly following the `address` property name as follows:

```
 [{
  "name": "a sample name",
  "address": {
  "street": "a sample street",
  "city": "a sample city"
  }
},
{
  "name": "second sample name",
  "address": "second sample address"
}]
```

Insert and upsert behaviors are the same as the **insert** and **upsert** commands.

Execute

The execute command can be used to execute a VAIL procedure.

```
execute <procedureName> <p1Name>:<p1Value> ... <pNName>:<pNValue>
```

procedureName must be the name of a previously defined procedure.

Procedure parameters are specified as <name>:<value> pairs using a colon (:) as the separator character. Spaces cannot be embedded in the <name>:<value> pair

The result is the JSON response produced by the executed procedure.

Note that the execute command is deprecated in favor of the run procedure command as of release 1.37. It may be removed in a future release.

Run

The run command can be used to run a test, test suite, or VAIL procedure. In the case of a test you must supply the test name:

```
vantiq run test <testName>
```

The result of the command indicates the test report ID for the test that was started. This ID can be used to check on the status of the test via the **find** command (with the testreports resource), as well as to stop the test prior to completion. See the **stop** command.

When running a test suite you must supply the test suite name and optionally specify a test name as the first test to run:

```
vantiq run testsuite <testSuiteName> [ testName ]
```

When a test name is given, any tests in the test suite before the named test will be skipped. If no test name is given, all tests in the test suite will be run.

Similar to the **run test** command, the result of the command indicates the test report ID for the test suite that was started. This ID can be used to check on the status of the running test suite via the **find** command (with the testreports resource), as well as to stop the in-progress run of the test suite. See the **stop** command.

Lastly, you can run a VAIL procedure by supplying the procedure name and any parameters:

```
vantiq run procedure <procedureName> [ <p1Name>:<p1Value> ... <pNName>:<pNValue> ]
```

The result is the JSON response produced by the invoked procedure. This is equivalent to the **execute** command, and is the supported mechanism for invoking procedures going forward. The **execute** command is now deprecated and may be removed in a future release. Unlike the **run test** and **run testsuite** commands, the in-progress run of a procedure cannot be stopped.

Stop

The stop command can be used to stop a test or test suite that was started using the **run** command. The command takes the test report ID of the test or test suite to stop:

```
vantiq stop <testReportId>
```

The **stop** command halts the execution of the in-progress test or test suite and initiates the clean-up of testing resources. The specified testReport ID corresponds to an instance of the testreports resource returned by the **run test** or **run testsuite** command.

The result of the **stop** command is a boolean value indicating whether the command was successful. The subsequent status of the test or test suite run can be checked using the **find** command (with the testreports resource).

Recommend

The recommend command can be used to submit requests for recommendations to the server if you have created the corresponding rules to process the request.

The recommend command takes two parameters:

- recommendationProcedure - the procedure to invoke to begin the recommendation process. If the recommendationProcedure is NOT supplied,the recommendationProcedure defaults to **recommendProduct**. The named procedure must be defined at the time the recommend command is issued.
- productId - the identity of the product for which recommendations are requested. The value of productId must match the productId of an object in the type in which the products are stored. The type storing the products and catalog of recommendations is user defined. The productId parameter MUST be supplied.

The response is a list of recommendations.

Export

The **export** command writes either the resource meta-data or data stored in user defined types into files stored in a directory on the local machine. The command takes an optional parameter used to indicate what should be exported:

- metadata – export the resource definitions (e.g. types, sources, rules, etc…)
- data – export the data contained in user defined types and the documents resource.
- project – export the resource definitions within a project.
- projectdata – export the data contained in user defined types within a project, and the documents used by clients within a project.

If neither option is provided the default is to export the metadata.

The export command supports the following options to customize the export behavior:

- **-d <directoryName>** the directoryName is the name of the directory in which the export should be placed. If the -d option is not specified, the export is placed in the current working directory.If the -d option is not specified, the export is placed in the current working directory.
- **-exclude <resourceType>** - Suppresses export of the named resource type during *export data*. It can be used to suppress the export of any resource type.
- **-chunk <integer>** - The chunk option can be used to export the objects in smaller sets; specifically, the **chunk** size.
- **-until <DateTime>** - Limit *export data* to instances created before the specified ISO DateTime string (example: 2019-09-16T22:19:29.313Z). Specify `NOW` instead of a DateTime string to export all instances created before the current time (when the export begins).
- **-ignoreErrors** - Ignore errors that occur when trying to export data with unusable names, e.g. documents with names containing `|` or `:` on Windows. The errors and related filenames will be printed on the console, but otherwise ignored.

The export/import commands can be used to conveniently migrate a system from one namespace to another namespace by creating an export directory, running the export command and then running the import command on the same directory using credentials for a user in the target namespace.

When the export runs it creates the following sub-directories and places the exported artifacts and data in the equivalently named directory. The formats used are the same as those described earlier for the **find** command.

Meta-Data directories:

- aicomponents - genAI flow definitions
- catalogs - catalog settings and the entries registered
- clients - client definitions
- collaborationtypes - app definitions
- procedures - procedure definitions
- projects - project definitions
- rules - rule definitions
- scheduledevents - scheduled event definitions
- services - service definitions
- sources - source definitions
- subscriptions - subscription definitions
- systemmodels - system model definitions
- topics - topic definitions
- types - type definitions

- configurations - node configuration definitions

- debugconfigs - debug configuration definitions
- deployconfigs - deployment definitions
- environments - environment definitions

Data Directories:

- data - the data stored in each user defined type is placed in this directory with the data stored in one file per defined type.
- documents - any uploaded documents

The output of the export command may be imported into any namespace using the **import** command.

Examples:

1. Export all metadata from a namespace
   vantiq -s oauth1 export -d /my/directory

2. Export all types from a namespace
   vantiq -s oauth1 export data -d /my/directory

3. Export all types from a namespace except for typeName1 and typeName2
   vantiq -s personal export data -exclude typeName1 -exclude typeName2 -d directory2

4. Export all types from a namespace for types, but only rows created before the current time (or a specified ISO time)
   vantiq -s oauth1 export data -until NOW
   vantiq -s oauth1 export data -until 2019-09-16T22:19:29Z

5. Run a procedure. This system procedure is a good way to check your credentials.
   vantiq -s personal execute Utils.getNamespaceAndProfiles

Import

The **import** command reads all artifact definitions stored in a directory and loads them into the current namespace. The command takes an optional parameter used to indicate what should be imported:

- metadata – import the resource definitions (e.g. types, sources, rules, etc…)
- data – import the data contained in user defined types and the documents resource.

The target directory must be structured as documented for the export command. In addition, files stored in the data sub-directory will have their contents loaded into the type with the same name as the file.

The import command supports the following options to customize the import behavior:

- **-d <directoryName>** - the name of the directory that serves as the root directory from which the previously exported data is read.
- **-ignore <resourceType>** - suppresses loading of the named resource type. It can be used to suppress the loading of any resource type.
- **-chunk <integer>** - if a user defined type exported a large number of object instances, the chunk option can be used to insert or upsert the array of objects in smaller sets; specifically, the **chunk** size. Sets of instances smaller than 5,000 most likely can be loaded without chunking. For larger sets it is recommended to set the initial chunk size to 5000.
- **-include <typeName>** - the name of a type to be included in the export (all other types will be ignored). This option can be specified more than once to include multiple types.
- **-exclude <typeName>** - the name of a type to be excluded from the export (all other types will be included). This option can be specified more than once to exclude multiple types.

The export/import command pair can be used to conveniently migrate a system from one namespace to another namespace by creating an export directory, running the export command and then running the import command on the same directory using credentials for a user in the target namespace.