ONAP Command-Line Interface (CLI)

One command to command whole ONAP !

Kanagaraj.Manickam@Huawei.com

License

ONAP Command-Line Interface (CLI)

ONAP cli community developed a model based Command-Line interface (CLI) framework, which facilitates vendors to implement the CLI for any given RESTful services by writing YAML based template and mostly relives vendor from implementing and maintaining the CLI using any programming language. It also provides a auto-discoverable plug-in architecture, which helps vendor to implement CLI command as a plug-in, when CLI model schema is not adequate for implementing a given command.

ONAP Command-Line Interface (CLI)

Both in Telco and enterprise industries customer would prefer the commands over GUI on many situations such as automation, CI, etc. And ONAP spans across both kinds of these customers, so this CLI project provides required commands for ONAP.

onap CLI is an console based application helps to operate ONAP from Microsoft windows or Linux Operating system. It's ***one command to command the whole ONAP***

Framework Features

1. Auto-discover any commands which are plugged in the CLI framework.
2. Provides YAML based model schema to create CLI commands without writing code.
3. Provides man page for each command automatically with option --help.
4. It is on-par with other existing commands in terms of exit code. I.e. on success it set exit code 0 otherwise 1.
5. Supports multiple output format such as tabular format, csv with option –format.
6. Provides --debug option to find the HTTP request and response executed as part of the command.
7. Every command can print the results in portrait mode or landscape mode, which is completely controlled by the vendor, who develops the given command.
8. Provides --long option to control the set of result attributes. By default every command will provide the only mandatory result attributes marked by vendor.
9. Provides –no-title option to skip of the title from the result. This would be useful in case of automation where the format option is given with csv.
10. Each command is provide with –version to report the service name and version it uses.
11. In case of error, it would print the specific error code, error message and corresponding http status code reported from the service.
12. After installation, user can add additional commands by following instructions provided here.

For any help, please mailto:onap-discuss@list.onap.org

Developer guide for implementing a command

CLI framework provides following option to implement a command

1. Command as a plug-in
2. Command as a YAML based model schema

Command as a plug-in

To implement a command as a plug-in:

1. Create a new maven jar project
2. Add dependency to the maven onap cli framework as below

```
        <dependency>
            <groupId>org.onap.cli</groupId>
            <artifactId>cli-framework</artifactId>
            <version>1.1.0-SNAPSHOT</version>
        </dependency>
```

3. Refer schema section and create the schema **without http section** in it. And place it under resources folder. (it should be in class path). Say its named as *onap-sample-test-schema.yaml*
4. Implement the command by using `org.onap.cli.fw.OnapCommand` as base class and add command name and schema file location using the annotation given below. Also Implement the abstract method run() in the new plug-in.

```
import org.onap.cli.fw.OnapCommandSchema;
import org.onap.cli.fw.OnapCommand;

@OnapCommandSchema(name = "user-create", schema = "onap-sample-test-schema.yaml")
public class OnapSampleTestCommand extends OnapCommand {
     @Override
     protected void run() throws OnapCommandException {
           //TODO: implement the command
     }
```

5. Add this project under *cli/plugins* project as maven module.

NOTE:

In ONAP, all common services commands under cli-plugin-common-service projects, are implemented in this approach, and can be used as reference.

This approach is very useful if the given command is not possible to implement using the YAML model schema explained in next section. Some commands like *csar upload* requires custom code to handle it and this approach would be more useful.

Command as a YAML based model schema

To implement a command using YAML model schema:

1. Refer schema section and add the new schema file for a command under the folder onap-cli-schema.

NOTE:

In ONAP most of the commands are provided using this approach. Please refer the commands provided for catalog services in cli-plugins project.

How to build and run CLI?

Follow the steps given below for building the CLI and run it:

1. Git clone the CLI project from ONAP gerrit
2. Run maven build and go to deployment/target/deployunzip directory
3. Set the following environment variables
   a. ONAP_CLI_HOME to above extracted folder say deployment/target/deployunzip.
   b. ONAP_USERNAME to username of ONAP. (currently set to some junk value)
   c. ONAP_PASSWORD to password of the user provided in ONAP_USERNAME. (currently set to some junk value)
   d. ONAP_MSB_URL to MSB url (currently set to some junk value)
4. Run the command under $ONAP_CLI_HOME/bin/onap.sh . (currently only schema related commands are supported and micro-service commands are provided as samples)

ONAP Command model Schema 1.0

ONAP release provides the 1.0 version of schema for writing the CLI and has following sections.

Schema version

Every command written in YAML template should have the following entry at the first line

```
onap_cmd_schema_version: 1.0
```

This schema version supports to define CLI inputs, results, name, description and realization of the command using http.

Name

Every command should be provided with unique name, which would be self explanatory like *vim-create* which means this command will be used to create a VIM. So the corresponding YAML entry would look like as below:

```
name: vim-create
```

Description

It helps to provide more details about the command for which the YAML is written. It could have details like pre-requisites, description about the command, limitations ,etc. So the corresponding YAML entry would look like as below:

```
description: Helps to register a VIM in ONAP ESR service.
```

Service

Every ONAP command would use the corresponding the ONAP service REST API for realizing it. In ONAP, every service is auto-registered in the Micro-service Bus (MSB) and user could query the MSB for finding endpoints for a given service and its version. This concept is modeled in the YAML schema, where every command could mention the required service name and version. For example, vim-create command would mention the service entry as below:

```
service:

  name: extsys

  version: v1

 no-auth: true
```

In ONAP, every services other than, while no-auth option provided above would help to by-pass the authentication in following scenarios:

1. Some features like micro-service discovery commands do not require authentication
2. During the command schema development, this option would help to disable the authentication on need basis.

Parameters

Every CLI command required set of parameters for proving the inputs for running it. Each input is modeled as Parameter and is having following properties. And there are two kinds of input parameters. One is provided with short/long option while another is provided with option is called positional argument. CLI model schema supports both the types.

S.No	Property Name	Details	By Default
1	name	Name of the attribute such as vim-name	NA
2	description	Details of the property such as help message, example, pre-requisites.	NA
3	type	Property type. Followings are supported 1. long	String.

		2. string 3. json 4. yaml 5. array 6. map 7. url 8. bool For JSON type, the input value could be provided either directly or using local file path. Array and map types helps to provide the same option multiple times as some commands required it.	
4	short_option	Short option name and is always with only one letter	NA
5	long_option	Long option name and should be similar to the name.	NA
6	is_optional	To mark the given property as mandatory or optional.	false
7	is_secured	To mark the given property as secured such as password. These kind of propertys will be not reported with *****	false

NOTE: when both short_option and long_option is absent, its considered as 'positional argument' and user could provide these kind of parameter values without preceding option name.

Example parameter used for command vim-create would be as below:

```
- name: password

  description: ONAP VIM password

  scope: short

  type: string

  is_secured: true

  short_option: j

  long_option: password

  is_optional: false

  is_secured: true
```

NOTE: In case of url type, if user wants to provide in URI format, it is recommended to skip the beginning forward slash (/). Its take care by framework as there is some issue.

Results

Command usually reports the outputs in tabular format in portrait or landscape mode. For example, list kind of commands would print the outputs in landscape mode while create kind of commands would print the results in portrait mode. To support this feature, CLI model schema provides below attribute:

```
results:

  direction: portrait
```

Attributes

The output results are capture using set of attributes and each attribute is having the following properties defined in CLI model schema.

S.No	Attribute Name	Details	By Default
1	name	Name of the attribute such as vim-name	NA
2	description	Details of the attribute such as help message, example, pre-requisites.	NA
3	type	Attribute type. Followings are supported 1. long 2. string 3. json 4. yaml 5. url 6. bool	String.
4	Scope	Every attribute is included in the command output based on the defined scope. There are two types 1. Short 2. Long. By default all attributes defined with short scope will be reported in the command output. To print the long scoped attributes, --long option should be provided while executing the command	short

Example result used for command vim-create would be as below:

```
results:

  direction: portrait

  attributes:

   - name: id

     description: ONAP VIM ID

     scope: short
```

```
    type: string
```

http

This section captures the required details for executing the command by communicating with given service mentioned in the service section. And is applicable only for the http command.

Request

Request section helps to add following details.

S.No	Attribute Name	Details	Optional
1	url	Service URI. It just holds the corresponding URI mentioned in the swagger definitions. NOTE: Don't add basepath in this url. Its automatically discovered by framework using the service name and version provided as part of CLI model schema.	NO
2	method	HTTP method like POST, GET, etc	NO
3	headers	List of headers name and values.	YES
4	queries	List of queries name and values	YES
5	body	Body json.	YES (for DELETE its optional)

Parameter Macro

Here a macro in the form of ***${parameter-name}*** and when the command is executed, it is used to feed the values given in the input parameters into any of the http request attributes mentioned below. And this macro could be used in any of the attributes mentioned in above table.

Following example shows the sample http request section for vim-create command with marco used in body section.

```
   uri: /vims

   method: POST

   headers:

   queries:

   body: '{"name":"${name}","vendor":"${vendor}","version":"${vim-
version}","description":"${description}","type":"${type}","url":"${url}","userName":"${username}","pass
word":"${password}","domain":"${domain}","tenant":"${tenant}"}'
```

success_codes

This section captures the list of http status code, which is defined in the service swagger definitions. Framework will check for this status code from http response and will fail the command otherwise.

For example, vim-create command would have the following success codes:

```
success_codes:

  - 201

  - 200
```

result_map

CLI model schema provides this section to fetch the data from the http response returned from the service and assign the attributes defined in the results section. It is an map of attribute name and its corresponding value.

CLI framework also provides below macros for fetching value from response dynamically

Body Marco

Body marco is provided in the form of **$b{json-path}** , Here the json-path is the JPATH used to pointer to the given attribute in the response body. More details on json-path is available here.

For example, vim-create command would have the following sample attributes, for a sample response provided in the sample_response section:

```
result_map:

 id: $b{$.vimId}

 name: $b{$.name}

 vendor: $b{$.vendor}
```

Header Macro

CLI framework also provide another header macro in the form of **$h{header-name}** to fetch the data from the given response header-name in service response.

sample_response:

This section helps to capture any sample data for a given http aspects like sample response body. It will act as reference to understand the the result_map easily and is more optional.

Below is the sample response for vim-create command:

```
sample_response:

body:{"serviceName":"test","version":"v1","url":"/api/test/v1","protocol":"REST","visualRange":"1","lb_
policy":"hash","nodes":[{"ip":"127.0.0.1","port":"8012","ttl":0,"nodeId":"test_127.0.0.1_8012","expirati
on":"2017-02-10T05:33:25Z","created_at":"2017-02-10T05:33:25Z","updated_at":"2017-02-
10T05:33:25Z"}],"status":"1"}
```

Reference

- wiki https://wiki.onap.org/display/DW/Command+Line+Interface+Project
- gerrit http://gerrit.onap.org/r/cli