FortiManager Ansible Generator
Software Architecture Document

Version 1.0

Revision History

Date	Version	Description	Author
11/28/2019	1.0	Created	Link Zheng

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

The document describes the architectural design of the Ansible Generator for FortiManager. The user-case view is given to show users the cases to utilize generated Ansible modules and design details are given to show developers the architectural considerations for the generator.

This document aims to help software developers to understand how thousands of FortiManager Ansible modules are generated and maintain the generator in case of any module functional defect.

1.2 Scope

The software architecture applies to the system Phase1A's development, and is the basis of the subsequent phase's software architecture.

1.3 Definitions, Acronyms and Abbreviations

N/A

1.4 References

Ansible Httpapi Plugin introduction:
https://docs.ansible.com/ansible/latest/network/dev_guide/developing_plugins_network.html
FortiManager API reference: https://fndn.fortinet.net/index.php?/fortiapi/5-fortimanager/
FortiManager Httpapi Plugin:
https://github.com/ansible/ansible/blob/devel/lib/ansible/plugins/httpapi/fortimanager.py

1.5 Overview

This document consists of 5 sections:

- Section 1 is the overview of the purpose and scope of this document.
- Section 2 is the architectural representation of the generator.
- Section 3 is the goals and constraints of the generator.
- Section 4 presents the use cases by giving the use-case views.
- Section 5 specifically describe all the prerequisites and aspects of how generator is implemented.

2. Architectural Representation

To show the users what the generated modules are for and what the generator is, the use-case view is provided to give users a general introduction of what the generator is. Ansible is an open automation framework that you can have multiple choices when you are implementing your own modules, this document explains the considerations and compromises which are made to better adapt FortiManager to Ansible. Specifically, there are several parts to illustrate the design: The FortiManager API calling conventions, structure of the FortiManager API schemas, and the fortimanager httpapi plugin and the generator which does schema to module conversion.

3. Architectural Goals and Constraints

Design Objectives:

- To implement a comprehensive generator to convert FortiManger API to Ansible modules.

The major design and implementation constraints for the system are:

- The modules are generated on a per-API basis, therefore the module is named as the canonical name of the API, so it's less user friendly.
- The schema doesn't enforce attribute ***required*** or ***optional*** for an argument, user have no idea which argument is required until the FortiManager reports error message. Fortunately, most of the arguments are optional.

4. Use-Case View

The FortiManager is a centralized resource manager which manages devices like Fortigate and other resource profiles or templates, with FortiManager, you are able to configure attached devices at one go. Ansible enables you to automate the management and configuration of various resources. However, FortiManager is not a standard device which is already fully supported in Ansible repository[1].

The FortiManager Ansible Modules have the use cases in Figure 1. The Admin, which is often a user, initiates a task to configure the FortiManager via an Ansible module. These modules include not not limited to ***devices & groups***, ***policy & packages***, ***policy objects*** and ***system administration***, all the modules include one dedicated plugin which encodes the user parameters in a FortiManager recognizable format and decodes the response.

[1] There are several modules for FortiManager since Ansible 2.8. but very limited.

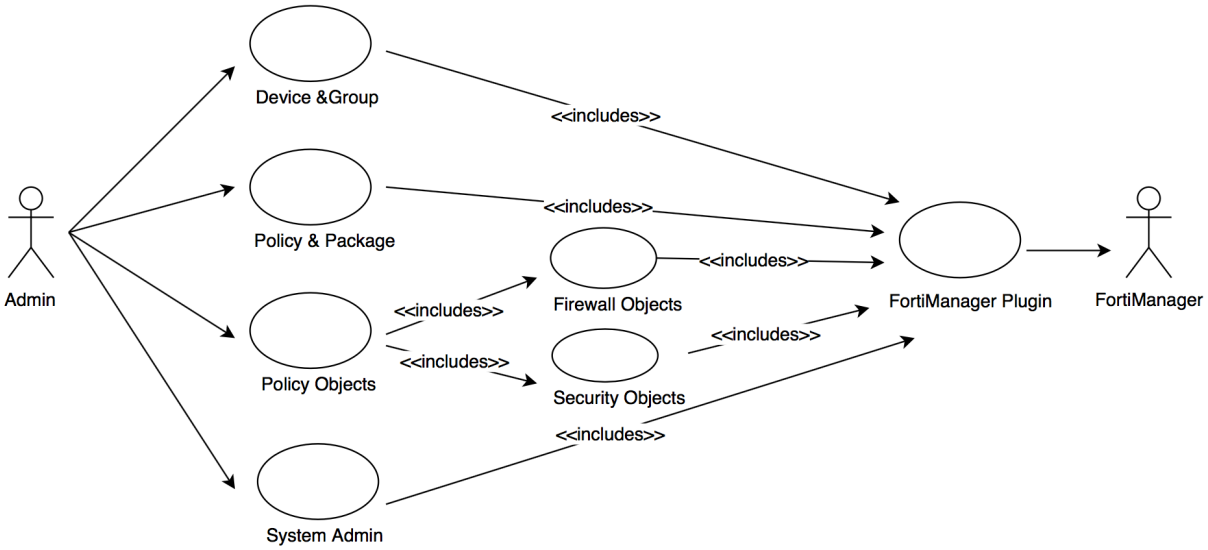

Figure 1 The use cases of FortiManager Ansible modules

As stated earlier, there are so many functions in FortiManager, to realize all the modules manually is error-prone and it's also hard to maintain for future versions as FortiManager evolves. Fortunately, we have API schema which is feasible to be converted to corresponding Ansible modules, we call the entity which converts the API schema to Ansible modules the generator. Figure 2 provides the use case view for the FortiManager Ansible generator.

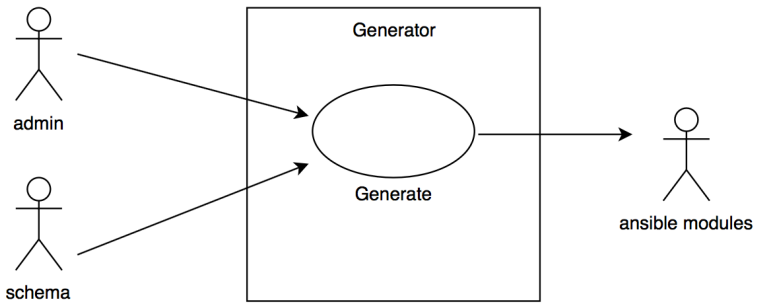

Figure 2 The use case of FortiManager Ansible generator

Actually there is only once use case: to generate Ansible modules. To generate the modules, it's required to provide API schema as input as well.

5. Architectural Design

This whole section describes API calling convention and schema structure as well as the problems encountered during doing schema conversion and generating modules.

5.1 FortiManager API calling convention overview

The FortiManager API is based on JSON-RPC which is a remote procedure call protocol. To request the FortiManager API, one has to send a HTTP POST request to the url *http(s)://fortimanager_address/json_rpc* with JSON-RPC encoded payload.

The http payload in the request has the following format:

```
{
    "id": 1,
    "method": "...",
    "params": [ ... ],
    "session": "..."
}
```

- **id** is to identify the recently issued task and is better incremented every time a request has completed, the sender may also choose random one.
- **method** is confined in the list [***get, add, set, update, delete, move, clone, exec***], each method has its own semantics.
- **params** is always an array, and it varies among APIs.
- **session** is the authentication string which is obtained via the following rpc call ***sys/login/user***, this is the only API which requires no session string in request:

the following json-rpc request is to obtain the session token.

```
{
   "id":1,
   "method":"exec",
   "params":[
      {
         "url":"sys/login/user",
         "data":[
            {
               "user":"APIUser",
               "passwd":"Fortinet1!"
            }
         ]
      }
   ],
   "session":null,
}
```

Here is an example to obtain the session token:

```
#curl -s --insecure -X POST https://192.168.190.102/jsonrpc -d @login |python3 -m
```

```
json.tool
{
    "id": 1,
    "result": [
        {
            "status": {
                "code": 0,
                "message": "OK"
            },
            "url": "sys/login/user"
        }
    ],
    "session":
"4CKN1Q7Acdd8Xd2rLnWddNGBCpmKc16/qbsHgzFHQgnsia3VDKTQ4iaWqz6uWka1xBK/BvcPziBUt1hoODbeH0
y8e0Wux7IC"
}
```

The session token then can be reused for subsequent API calls.

The response of the API call also has fixed format. as you can see from the following table:

```
{
    "id": 1,
    "result": [
        "data": [ ... ],
        "status": {
            "code": 0,
            "message": "OK"
        },
        "url": "..."
    ],
    "session": "..."
}
```

- **id** is the same one you put in the request, it helps you identify which API call you just issued.
- **session** is always returned.
- **result** has the fixed format as well, among them the **url** is the the full API url.
- **result**.**status.code** is indicator of the execution result, code 0 indicates OK
- **result.status.message** contains very useful message if something goes wrong.
- **result.data** is the only optional field in the response, it's usually present in GET semantics API call.

5.2 FortiManager API Schema overview

The API schema defines the pattern that the sender must follow so that the FortManager can resolve them correctly. Specifically, the schema defines the following aspects for a parameter:

- What the parameter's name is.
- What the type of a parameter can be: [***integer***, ***string***, ***array***, ***dictionary***].

- What the enumerated values are if there are.
- What the default value is if there is.
- What the descriptive text is if there is.
- Whether a top-level parameter is required.
- What place a top-level parameter belongs to: [***path, body***]

The document references the schema exported from FNDN, it's formatted into JSON. ***definitions*** and ***paths*** in schema JSON data are of our concern. An example is provided to illustrate the very top-level schema structure.

```
{
        "definitions": {
                "common.filter.object": {……},
                "common.range.object": {……}……
        },
        "paths": {
                "\/dvmdb\/adom\/{adom}\/script (add)": {……},
                "\/dvmdb\/adom\/{adom}\/script (get)": {……},
                "\/dvmdb\/adom\/{adom}\/script (set)": {……},
                "\/dvmdb\/adom\/{adom}\/script (update)": {……},
            "\/dvmdb\/adom\/{adom}\/script\/{script} (delete)": {……}……
        }……
}
```

In the document we refer to an URL plus a method as an **API endpoint**.
Each API endpoint is a human readable item in paths, and the schemas for [***add, set, update***] usually share same definition. the semantics of **set** and **update** are identical: both of them are to incrementally update the configurable object, while the semantics of **add** is to create a new configurable object, it reports error message: "object already exists" if the object has already been created.

5.2.1 Parameter Definition Linkage

there are many parameters in different paths which share one definition fragment, to put one fragment as one item in **definitions** and link other duplicated parameters to the definition item saves quite a lot of space. And it turns out to be very effective.

One example of definition linkage is:

```
{
    "definitions":{
        "dvmdb.script":{
            "type":"object",
            "properties":{
                "content":{
                    "type":"string",
                    "format":"string"
                },
                "desc":{
                    "type":"string",
```

```
                    "format":"string"
                },
                "filter_build":{
                    "type":"integer",
                    "format":"int32"
                }……
            }
        }
    },
    "paths":{
        "/dvmdb/adom/{adom}/script (add)":{
            "data":{
                "type":"array",
                "items":{
                    "$ref":"#/definitions/dvmdb.script"
                }
            }
        },
        "/dvmdb/adom/{adom}/script/{script} (update)":{
            "data":{
                "$ref":"#/definitions/dvmdb.script"
            }
        }……
    }
}
```

As you can see, more than one API endpoints refer to the same definition, to get the full schema definition, we have to replace all the links with their genuine definition.

CAVEAT: there is only one definition that contains circular definition which may cause trouble when resolving it, the definition is as below:

```
"pm.pkg": {
    "type":"object",
    "properties":{
        ……
        "package setting":{
            "$ref":"#/definitions/pm.pkg.package.setting"
        },
        "scope member":{
            "$ref":"#/definitions/common.scope.object"
        },
        "subobj":{
            "type":"array",
            "items":{
                "$ref":"#/definitions/pm.pkg"
            }
        }……
    }
}
```

The ***subobj*** in definition ***pm.pkg*** is also another ***pm.pkg***, the problem we encounter is it's not easy to express nested schema in Ansible. To work this around, we remove the attribute ***subobj*** since this is the only exception in FortiManager API schemas and we don't have to define a ***subobj*** when we are creating or updating a profile package.

5.2.2 Parameter specification

There are two kinds of parameters: parameters in url path and parameters in api body.
for an API endpoint, there might be several parameters which are used to form the full URL, for example, we have the url: **/pm/pkg/adom/{adom}/{pkg_path}**. there are two parameters for url construction:

```
"parameters": [
    {
        "name":"adom",
        "in":"path",
        "type":"string",
        "required":true
    },
    {
        "name":"pkg_path",
        "in":"path",
        "type":"string",
        "required":true
    }
]
```

Note that all these parameters contain a key ***in*** and their values are the ***path***, when we are making the full url, we extract arguments from user input, and replace **{adom}** and **{pkg_path}** with them.

The remaining parameters are all for API body, the parameters have the key ***in*** with value ***body***. the format is also fixed: the top-level parameters in body are [***method, session, id, params***][2], and the param.url is also fixed, i.e. it must be the full url we constructed using url skeleton plus the parameters in path. A typical top-level body parameter is as below:

```
{
    "name":"body",
    "in":"body",
    "required":true,
    "schema":{
        "type":"object",
        "properties":{
            "method":{
                "type":"string",
                "example":"add"
            },
            "params":{
```

[2] The parameters definition is in accordance with the calling convention as stated in section 5.1

```
                "type":"array",
                "items":{
                    "type":"object",
                    "properties":{
                        "data":{
                            "type":"array",
                            "items":{
                                "$ref":"#/definitions/dvmdb.script"
                            }
                        },
                        "url":{
                            "type":"string",
                            "format":"string",
                            "example":"/dvmdb/adom/{adom}/script"
                        }
                    }
                }
            },
            "session":{
                "type":"string"
            },
            "id":{
                "type":"integer",
                "format":"int32",
                "example":1
            }
        }
    }
}
```

The only variable part of the example is marked in bold.
Notes:

- The ***params*** of the top-level parameters is always defined as an array.
- No ***required*** key in low-level parameters, it's allowed to omit all of them but probably you will receive a response with error code complaining you missed some options.

5.2.3 Response specification

As with input parameters, the response almost has fixed format, there are three top-level parameters: [***id, method, result***], a typical schema for response message is as follow:

```
{
    "method":{
        "type":"string",
        "example":"get"
    },
    "result":{
        "type":"array",
        "items":{
            "type":"object",
            "properties":{
                "data":{
```

```
                    "type":"array",
                    "items":{
                        "$ref":"#/definitions/dvmdb.script"
                    }
                },
                "status":{
                    "$ref":"#/definitions/common.result.object"
                },
                "url":{
                    "type":"string",
                    "format":"string",
                    "example":"/dvmdb/adom/{adom}/script"
                }
            }
        }
    },
    "id":{
        "type":"integer",
        "format":"int32",
        "example":1
    }
}
```

The ***result.status*** and ***result.url*** in top-level parameter ***result*** is always returned while the data is optionally returned in GET semantics API calls. Usually the ***result.status*** is linked to a separate definition: ***common.result.object***, it contains so regular content that we are able to enumerate all of them.
The ***common.result.object*** has the following definition:

```
"common.result.object":{
    "type":"object",
    "properties":{
        "code":{
            "type":"integer",
            "format":"int32"
        },
        "message":{
            "type":"string"
        }
    }
}
```

The ***code*** and ***message*** is one-one mapped. Here is part of the mapping dictionary:

```
{
    "0":{
        "msg":"OK"
    },
    "-100000":{
        "msg":"Module returned without actually running anything. "
    },
```

```
    "-2":{
        "msg":"Object already exists."
    },
    "-6":{
        "msg":"Invalid Url. Sometimes this can happen because the path is mapped to a
hostname or object that doesn't exist. Double check your input object parameters."
    },
    "-3":{
        "msg":"Object doesn't exist."
    },
    "-10131":{
        "msg":"Object dependency failed. Do all named objects in parameters exist?"
    },
    "-9998":{
        "msg":"Duplicate object. Try using mode='set', if using add. STOPPING. Use
'ignore_errors=yes' in playbook to override and mark successful."
    },
    "-20042":{
        "msg":"Device Unreachable."
    },
    "-10033":{
        "msg":"Duplicate object. Try using mode='set', if using add."
    },
    "-10000":{
        "msg":"Duplicate object. Try using mode='set', if using add."
    },
    "-20010":{
        "msg":"Device already added to FortiManager. Serial number already in use."
    },
    "-20002":{
        "msg":"Invalid Argument -- Does this Device exist on FortiManager?"
    }
}
```

The status code is resolved in Ansible output format, and further the users know whether the task succeeds or not, the rules to resolve API status code will be covered in future sections.

5.2.4 API endpoint tagging

For every API endpoint, there is one tag. Through the tag, we are able to find the descriptive string in the global tags list if the tag exists in the list.

5.3 FortiManager Ansible Httpapi Plugin

Ansible Httpapi plugin is an Ansible plugin which utilizes HTTP as the transport.

Ansible defines a base interface ***HttpApiBase***, any httpapi plugin have to inherit the base class and realize the predefined methods. The FotiManager httpapi plugin also extends several fields and methods as shown in Figure 3.

It's stated in previous chapter 5.1 that all API calls require a session token and the session token is obtained via url ***sys/login/user*** with which the session_id is allowed to be empty and remembered for the subsequent calls.

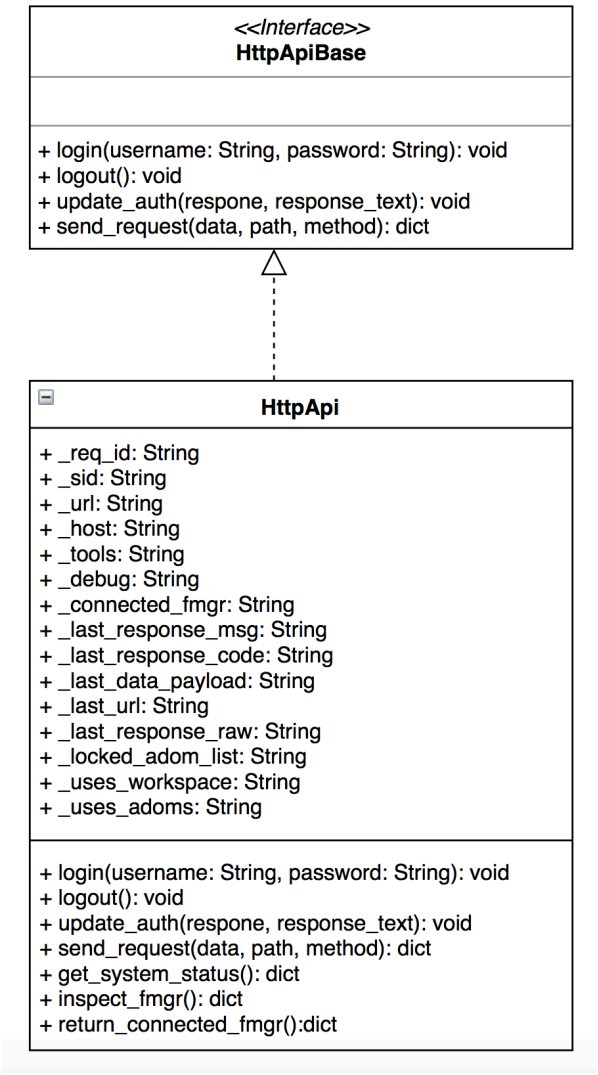

Figure 3 HttpApi Class Diagram

5.3.1 Plugin Procedure: send_request

The process to send request is quite straightforward: it encodes all input data into the format as introduced in chapter 5.1, and then calls into urllib to send the request to the web service.

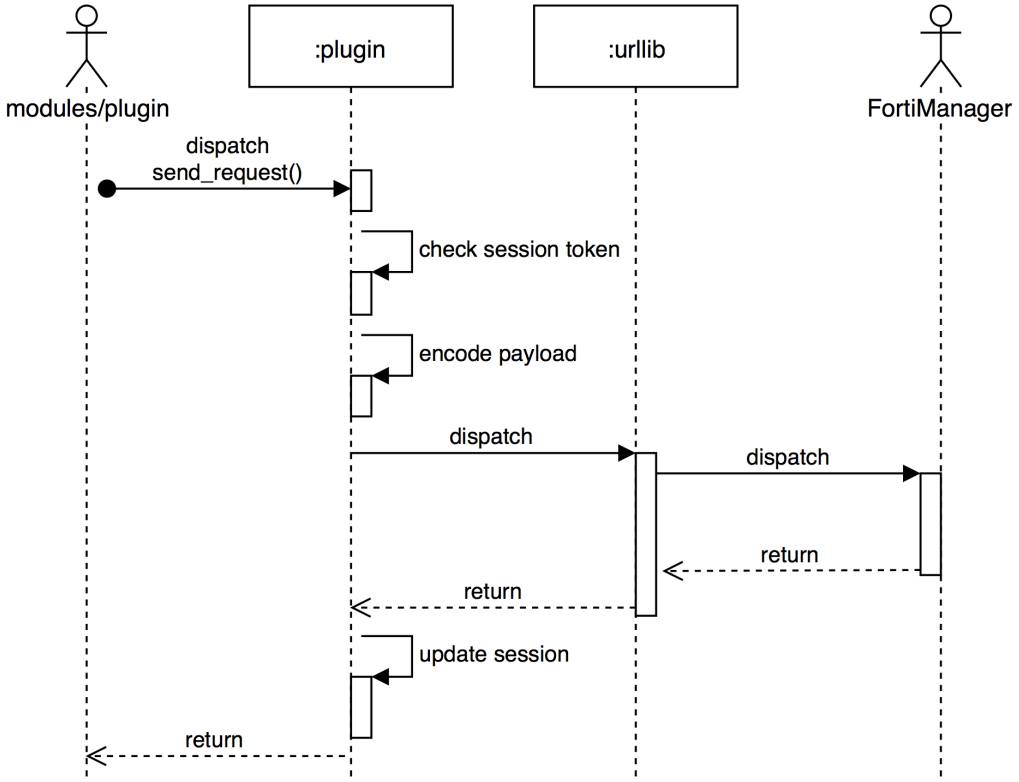

Figure 4 the flow to send request of FortiManager Httpapi Plugin

Here we always update session token at the end of each api call, as a matter of fact, it's only required to remember the session token string at the first time to log in. next section we are going to introduce the specific steps to login a user.

5.3.2 Plugin Procedure: login

To login is more than to authenticate the plugin, some extra work needs to be done during the login stage, the whole process is depicted in Figure 5.

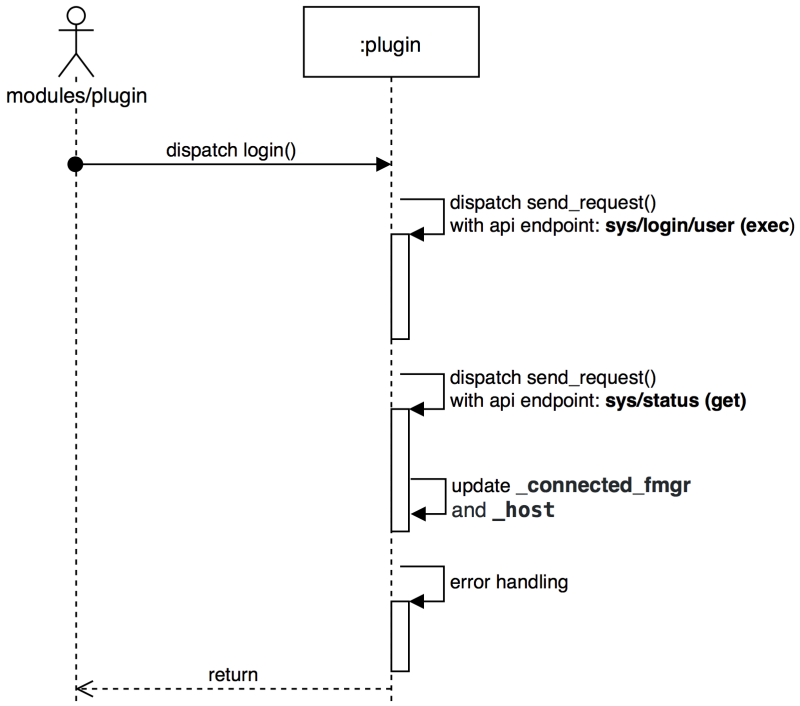

Figure 5 the flow to log in the FortiManager plugin

There are two APIs involved in the procedures to login the FortiManager plugin: ***sys/login/user*** and ***sys/status***. The first one is to retrieve the session token string, and the latter one is to retrieve the device system information, the order to invoke these two APIs can not be reversed.

the credentials for login are managed by Ansible uniformly, we don't have to input them except putting them into Ansible inventory.

By requesting ***sys/status***, we are able to construct the facts later and return all of them to users.

5.3.3 Plugin Procedure: logout

To Log out, requesting ***sys/logout (exec)*** with no extra parameters is all right.

5.3.4 Module/Plugin Instantiation

This section describes how module utilizes httpapi plugin.

After the module is instantiated, the module immediately builds a connection. To build the connection here is more than to build a pure TCP or HTTP connection, moreover, above the HTTP connection the initializer calls ***login*** as show in Figure 6. In essence, the connection here is an instance of FortiManager httpapi plugin.

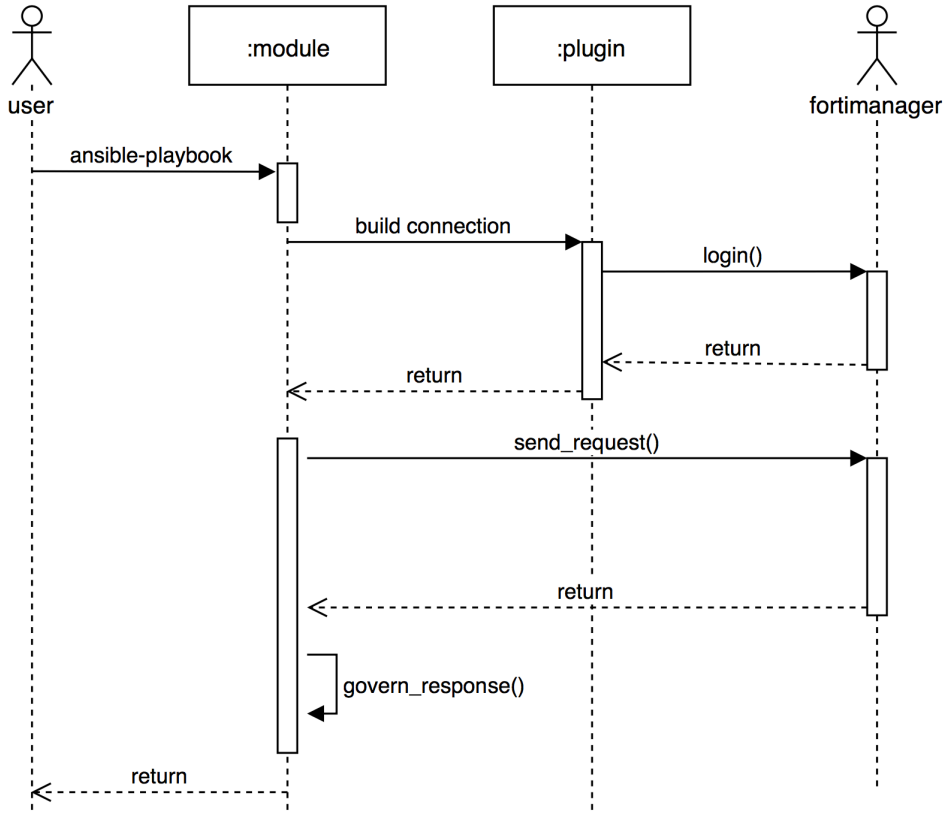

Figure 6 Interaction between FortiManager Ansible Module and httpapi plugin

After the connection is built successfully, the module is responsible for organizing the data to send to FortiManager, how to organize the data will be introduced in Section 5.4. To request the API service, the module directly calls into connection and invokes ***send_request*** which is overwritten in FortiManager plugin (see Section 5.3.1).

The most interesting part of the module processing is to create the mapping from the response to Ansible readable output.

The Ansible readable output list is [***changed, unreachable, failed, skipped, rescued, ignored***]. the response from FortiManager is explained and converted to Ansible readable output list using the rules in Table 1:

Table 1 FortiManager API Response status code and Ansible status mapping table

	Success (default:False)	**Failed** (default:false)	**Changed** (default:False)	**Unreachable** (default: False)	**Skipped** (default:False)
0	True		True		
-100000		True			
-2	True				True
-3	True				True

-6		True			
-10131		True			True
-9998		True			
-20042		True			True
-10033		True			True
-10000		True			True
-20010	True				True
-20002		True			True

***** Table 1 does not try to enumerate all possible status codes that FortiManager can return, most of the APIs return 0 indicating the requests don't go wrong. Several cases with non-zero return code are also considered as success and but changed is marked as false, for example, the object already exists.

5.4 FortiManager Ansible Generator

This chapter is going to give an overview of the generator and tell how to generate Ansible modules for all FortiManager APIs.

5.4.1 Ansible Parameters Arrangement

As shown in section 5.1, a good request must have several parts in its payload: [***id, method, params.url, session***], among them, ***id*** and ***session*** are managed by httpapi plugin so we don't have to care for them: ***id*** is incremented after composing a request and ***session*** is obtained after building the httpapi based connection, thus guaranteeing we have a valid authentication token string to compose our subsequent request.

Method is always a mandatory parameter obviously.

we also need parameters to compose full url, for example, with api url template: ***/pm/config/adom/{adom}/obj/application/list/{list}***, two named parameters are required in this case: [***adom, list***]. In generator, the parameters in api url are defined as a dictionary with name **url_params**.

Besides above parameters, there are other parameters which are actually the main body of the API request, they vary much depending on what the APIs are. To accommodate them all, the generator defines a dictionary named **params**.

In general, we have the following arguments specification for all generated modules.

```
    module_arg_spec = {
        'params': {
            'type': 'list',
            'required': False
        },
        'method': {
            'type': 'str',
            'required': True,
            'choices': [
```

```
                'clone',
                'delete',
                'get',
                'set',
                'update'
            ]
        },
        'url_params': {
            'type': 'dict',
            'required': False
        }
    }
```

* All the top-level parameters are validated by Ansible itself in a very coarse granularity. It's allowed for you to put whatever in ***params*** and ***url_params*** as long as they are a JSON-converted list and dictionary. This is definitely not what we want: because it's error-prone, and hard to debug incorrect parameters. it's required to validate within module or module utility.

5.4.2 Custom Parameters Definition

In last section it's stated to validate parameters ourselves, but why not put all parameters definition in Ansible argument specification?

There are several reasons:

1. For an api url, there are multiple methods and each method has its own schema, to reduce the number of generated modules, one url which has multiple methods results in one module. Defining per-method schema will be generating lots of duplicated definition items.
2. There is no field in FortiManager API schema that can be mapped to 'required' in Ansible module specification except for the top-level options which are of no use.
3. There is the case that parameter name itself is unknown, to define and validate this type of parameter is not possible with Ansible argument specification. One example of it:

```
 {
    'name': 'sortings',
    'type': 'dict',
    'dict': {
        'type': 'array',
        'items': {
            '{attr_name}': {
                'type': 'integer',
                'enum': [
                    1,
                    -1
                ]
            }
        }
    },
}
```

   The ***{attr_name}*** can be any string.
4. There is also another situation that Ansible cannot address: for an api url and its one

method, there are more than one associated API endpoint. Every schema of the endpoint is legal and should be taken into consideration. One example of it:

```
/dvmdb/adom/{adom}/script/{script}
    delete ['/dvmdb/adom/{adom}/script/{script} (delete)',
            '/dvmdb/adom/{adom}/script/{script}/object member (delete)']
    get ['/dvmdb/adom/{adom}/script/{script} (get)'
    set ['/dvmdb/adom/{adom}/script/{script} (set)',
         '/dvmdb/adom/{adom}/script/{script}/object member (set)']
    update ['/dvmdb/adom/{adom}/script/{script} (update)',
            '/dvmdb/adom/{adom}/script/{script}/object member (update)']
```

5. the circulation in schema can not be applied to Ansible argument definition either. See more in section 5.2.1 for definition of ***pm.pkg***.

To validate the parameters in module/module_util itself requires the schema to be separately defined as well.

The schema for parameters in API url is named ***url_schema*** and defined as an array, the array is empty for those API urls which contain no variadic parameters. An example is given as below:

```
jrpc_url = '/pm/config/adom/{adom}/obj/dnsfilter/profile'
url_schema = [
        {
            'name': 'adom',
            'type': 'string'
        }
]
```

For parameters in body, the schema is compressed because the schemas for some methods of the url are duplicated, to define only one schema object is enough and it greatly saves the size of a generated file.

The scheme for parameters in body are defined as ***body_schema***, it has the structure shown in the following example:

```
    body_schema =  {
        'schema_objects': {
            'object0': [...],
            'object1': [...],
            'object2': [...]
        },
        'method_mapping': {
            'clone': 'object0',
            'delete': 'object1',
            'get': 'object2',
            'set': 'object0',
            'update': 'object0'
        }
    }
```

For every method in ***method_mapping***, the corresponding schema object can be found in **schema_objects**.

The method ***clone***, ***set*** and ***update*** share the same one schema object ***object0*** in the example.

* for the method which has more than one API endpoint, the parameters definitions are distinguished by ***api_tag*** in each parameter definition, we will tell more of it in next section.

5.4.3 Custom Parameters Validation

For those methods which have more than one API endpoint. the schema for the same API endpoint is distinguished by a field named: ***api_tag***, thus the schemas are selected and categorized into groups, any schema definitions in the same group have the same api_tag.

The policy for parameters to pass the validation for multiple schemas is intuitive: the parameters pass the validation as long as they fit any one grouped schema of them. In other words, they fail the validation only if they fail all the grouped schemas. Next we will introduce how the process to match parameters with schemas goes like.

Parameters are organized to be an irregular tree[3], each parameter name is a tree node. Schemas is also regarded as a tree which has full parameters definitions. To match the parameter with the schema is a **depth-first-search** of the parameter tree to see whether every parameter matches the definition in schema tree at the same position during the traversal.

There are five types of schema definition, four of which have key ***type*** in it, the values of ***type*** are in [***string***, ***integer***, ***array***, ***dict***]. The fifth type of schema definition has no key named ***type***, in this document, we call them discrete schema definition.

a. For schema with ***type interger***, check whether the value is an integer and whether the value is in the enumerated array if ***enum*** is in schema definition, search is terminated.

b. for schema with ***type string***, check whether the value is a string and whether the value is in the enumerated array if ***enum*** is in schema definition, search is terminated.

c. for schema with ***type array***, check whether the given value is a list, then travel deeper for every value in the list with its subordinate schema definition.

d. for schema with ***type dict***, check whether the given value is a dict, then travel deeper for the dictionary value and its subordinate schema definition.

e. for schema without ***type*** keyword, traverse all the the named parameters, for one parameter name, try to find a subordinate schema definition whose key is the name of the parameter, if found, go deeper, otherwise, report error.

During the traversal, there are some exceptions that have to be handled:

1) if the parameter name starts with character '{' and end with character '}', then ignore the name check and go on doing value check.
2) The name ***type*** is confusing in schema definition, it's not reserved as a keyword, most of the time it's specifying a parameter type, while it also used a parameter name.

[3] There is only one exception with parameter definition ***pm.pkg*** which contains a circle. by removing the ***subopt*** in it, the circular is broken.

5.4.4 Issue request and handle response

See section 5.3.4.

6. Size and Performance

Some data quantifying the generator is provided as below:

1) Number of top-level schema files: 118
2) Number of json-rpc url: 2099
3) Number of generated modules: 2089 (about 10 less then the #url)
4) Number of supported API requests: > 8456 (here multiple API endpoints counted as one)

7. Quality

N/A