

FMC API Programming with Python

May 20, 2020

Version 0.1

**Cisco Systems, Inc.  
Corporate Headquarters  
170 West Tasman Drive  
San Jose, CA 95134-1706 USA**[**http://www.cisco.com**](http://www.cisco.com) **Tel: 408 526-4000 Toll Free: 800 553-NETS (6387)  
Fax: 408 526-4100**

Contents

[Contents 2](#_Toc43041316)

[Lists of Tables and Figures 4](#_Toc43041317)

[About FMC API Programming with Python 5](#_Toc43041318)

[Approvals 5](#_Toc43041319)

[Revision History 5](#_Toc43041320)

[1 Module 1: Introduction to the FMC REST API 6](#_Toc43041321)

[1.1 Course Introduction 6](#_Toc43041322)

[1.2 Course and Lab Prerequisites 6](#_Toc43041323)

[1.3 Course Format 6](#_Toc43041324)

[1.4 Enabling the FMC REST API 7](#_Toc43041325)

[1.5 Introduction to the FMC API Explorer 7](#_Toc43041326)

[1.6 REST API Basics 7](#_Toc43041327)

[1.6.1 Endpoints 8](#_Toc43041328)

[1.6.2 Method 9](#_Toc43041329)

[1.6.3 Headers 9](#_Toc43041330)

[1.6.4 The Data or Body 9](#_Toc43041331)

[1.6.5 HTTP Status Codes and Error Messages 10](#_Toc43041332)

[1.6.6 REST API Authentication 10](#_Toc43041333)

[2 Module 2: JSON 11](#_Toc43041334)

[2.1 JSON Data Types 11](#_Toc43041335)

[3 Module 3: API Explorer In Depth 13](#_Toc43041336)

[4 Module 4: Postman 16](#_Toc43041337)

[5 Module 5: Introduction to Python 18](#_Toc43041338)

[6 Module 6: API Requests with Python 20](#_Toc43041339)

[6.1 Understanding FMC API Sample Code 20](#_Toc43041340)

[6.2 Configuring the API Python Code 23](#_Toc43041341)

[6.3 Turning Off SSL Verification 23](#_Toc43041342)

[6.4 Executing the Program 23](#_Toc43041343)

[6.5 Executing the Program with Parameters 23](#_Toc43041344)

[6.6 Eliminating the InsecureRequestWarnings 24](#_Toc43041345)

[6.7 Requesting Access Control Rules 24](#_Toc43041346)

[7 Module 7: Building a Code Framework 25](#_Toc43041347)

[7.1 Avoid Duplication of Code through Reusable Modules 25](#_Toc43041348)

[7.1.1 Converting the GET call to a function 25](#_Toc43041349)

[7.2 Parsing the Response 27](#_Toc43041350)

[7.3 Separate Code that Changes from Code that Does Not Change 28](#_Toc43041351)

[8 Module 8: AC Policy and AC Rule GET Operations 29](#_Toc43041352)

[9 Module 9: POSTing an AC Policy 30](#_Toc43041353)

[10 Module 10: GETting and POSTing Objects 31](#_Toc43041354)

[11 Module 11: POSTing AC Rules 32](#_Toc43041355)

[12 Module 12: Converting Rules from Text 33](#_Toc43041356)

[13 Module 13: AC Rule Advanced Operations 34](#_Toc43041357)

[14 Module 14: Deployment 35](#_Toc43041358)

[15 Appendix A: Acronym Listing 36](#_Toc43041359)

[Trademarks and Disclaimers 37](#_Toc43041360)

Lists of Tables and Figures

[Table 1: REST API methods 9](#_Toc43041361)

[Table 2: REST API response status codes 10](#_Toc43041362)

[Table 3: JSON data types 11](#_Toc43041363)

[Table 4: JSON example 12](#_Toc43041364)

[Table 3: Features of a GET request 14](#_Toc43041365)

[Table 3: Python description 18](#_Toc43041366)

[Table 1: Explanation of FMC API Python code that requests access policies. 22](#_Toc43041367)

[Table 1: SSL verification comments and code 23](#_Toc43041368)

[Table 1: Example output of request for access policies 27](#_Toc43041369)

[Figure 2 GET for access policies 13](#_Toc43041370)

[Figure 2 GET response 14](#_Toc43041371)

[Figure 2 Postman GET request for access policies 16](#_Toc43041372)

[Figure 2 Python code snippet for GET request for access policies 17](#_Toc43041373)

[Figure 2 hello\_world.py and its execution 19](#_Toc43041374)

About FMC API Programming with Python

|  |  |
| --- | --- |
| Author | Richard Clendenning |
| Change Authority | Cisco Systems Advanced Services, Security & Collaboration IDT, Implementation Americas |

Approvals

|  |  |
| --- | --- |
| Area of Responsibility | Name or User ID |
| Document Owner | Richard Clendenning |

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | Name or User ID | Comments |
| 0.1 | 05/20/2020 | Richard Clendenning | Document creation |
|  |  |  |  |

# Module 1: Introduction to the FMC REST API

## Course Introduction

The FMC API Programming with Python course teaches participants how to perform operations with Cisco’s Firepower Management Center (FMC) REST API using the Python programming language.

The FMC REST (Representational State Transfer) API (Application Programming Interface) allows FMC operations to be performed by sending HTTP requests to the FMC. It provides an alternative to the FMC GUI in working with the FMC. By using the FMC REST API (hereafter, FMC API), users can issue individual requests using a web client to request information or perform an operation. But the real power of the FMC API is that it allows the user to write scripts or applications that can perform many FMC operations very quickly. It also can minimize configuration errors by avoiding manual input of configurations into the FMC GUI.

The course begins with an introduction to REST APIs and the FMC API Explorer, and continues with the fundamentals of JSON, Postman, programming with Python, and finally using Python to work with the FMC API.

## Course and Lab Prerequisites

This course does not assume any knowledge of programming or of Python. The only prerequisite knowledge is familiarity with the Firepower Management Center GUI and its policies and objects.

To perform all of the lab exercises, the participant will use a client computer with the following installed:

1. A browser
2. Postman
3. Visual Studio Code
4. Python 3.x

## Course Format

This course consists of modules, and each module contains a lecture portion and a lab portion. This document contains the course material for the lectures, and a separate lab manual contains the lab exercises.

## Enabling the FMC REST API

In the FMC, the REST API is enabled by default. However, if you are intending to use the REST API, you should confirm that it is enabled under **System>Configuration>REST API Preferences>Enable REST API**.

## Introduction to the FMC API Explorer

The FMC API Explorer provides a limited interface for the REST API as well as giving a view of the abilities of the REST API.

The FMC REST API Explorer follows the OpenAPI Specification (OAS), which is a broadly adopted industry standard for describing modern APIs. The specification is a JSON file that can be read by text editors and is used to present a readable user interface.

You can log into the API Explorer using any account on the Firepower Management Center, but you will only be able to perform the functions for which the account has permissions.

Note that while you can log into the API Explorer using the FMC GUI credentials, it is recommended (both in this course and in general practice) to use FMC GUI accounts exclusively for FMC GUI access and API Explorer accounts exclusively for API Explorer access. This allows one user to be logged into both at the same time and provides better accountability.

You can log into an FMC’s API Explorer by visiting **Error! Hyperlink reference not valid.**, where *<FmcHostName>* is the host name or IP address of the FMC.

On logging in, you will see the OpenAPI specification. There is an older interface for the FMC API Explorer which you can access via the Legacy Explorer button at the top right. In this course, we will work with the OpenAPI specification. However, the Legacy Explorer can be useful because it provides code snippets in Python and Perl in the interface, whereas the OpenAPI interface does not. To obtain code snippets via the OpenAPI specification, you must download CodeGen software and run it, providing it with the OpenAPI specification file.

## REST API Basics

Generally, a device has one or more interfaces that allow you to configure the device or obtain information from the device. For example, a Cisco ASA firewall provides these operations through a console interface, through an ssh connection to the device, or through use of an ASDM software client that can communicate with the ASA.

But there are complications with each of these methods of interacting with the ASA. The console connection requires close physical proximity. The ssh connection involves typing various commands, and it is not easy to store these commands for re-use or to automate operations. The ASDM similarly does not lend itself to repeating or automating operations.

With the explosion of the World Wide Web and the ubiquity of HTTP in the last three decades, devices (including the ASA, and of course the FMC) have increasingly come with a REST API that allows you to interact with them with in much the same way that you would interact with a web site. That is, you can use a browser and an HTTP GET request to obtain information from devices, or an HTTP POST request to provide configuration information to a device.

More generally, REST (Representational state transfer) is a software architectural style used for web services. An API (Application Programming Interface) defines, for a web service, device, or any piece of code, how a client can interact with that code to get information from it or provide information to it. A device with a REST API lets clients interact with the device like through an interface that is similar to a web server; that is, through HTTP methods.

A REST API request has four parts:

1. An endpoint
2. A method
3. Headers
4. The data (or body)

These are described in the subsections below.

### Endpoints

The endpoint is the url of the request, and consists of the root-endpoint, the path, and optionally, query parameters. For example, the endpoint for a request the list of access policies from an FMC looks like the following:

**https://10.1.2.3**[**/api/fmc\_config/v1/domain/e276abec-e0f2-11e3-8169-6d9ed49b625f/policy/accesspolicies**](https://10.81.127.36/api/api-explorer/legacy-explorer/)

In this url, the root-endpoint is **https://10.1.2.3**, where 10.1.2.3 is the IP address of the FMC. The path is the remainder of the url:

[**/api/fmc\_config/v1/domain/e276abec-e0f2-11e3-8169-6d9ed49b625f/policy/accesspolicies**](https://10.81.127.36/api/api-explorer/legacy-explorer/)

The path specifies that it is the access policies that are being requested.

The request can be tuned with one or more query parameters. A query parameter is a key-value pair. A question mark separates the path from the query parameters. For example, with the request above, we can add two query parameters to limit the results to the first five policies (using the “limit” parameter), and to request more detailed information about the policies that are returned (using the “expanded” parameter):

**https://10.1.2.3**[**/api/fmc\_config/v1/domain/e276abec-e0f2-11e3-8169-6d9ed49b625f/policy/accesspolicies**](https://10.81.127.36/api/api-explorer/legacy-explorer/)**?limit=5&expanded=true**

As you can see in the example above, multiple parameters are joined with an ampersand (&).

### Method

Requests to the FMC API will use one of the four methods shown in the table below:

|  |  |  |
| --- | --- | --- |
| Method | Description | Equivalent Database Operation |
| GET | Used to retrieve information from the FMC. | READ |
| POST | Used to create a new resource or object on the FMC. | CREATE |
| PUT | Used to update a resource or object on the FMC. | UPDATE |
| DELETE | Used to delete a resource or object from the FMC. | DELETE |

Table 1: REST API methods

### Headers

Headers are property-value pairs that are separated by a colon. For example, when sending JSON in the body of a POST request, you should use a header to specify that the content is JSON, as follows:

"Content-Type: application/json"

We will see later that headers are used for other purposes, especially for authentication.

### The Data or Body

When a new object is created, the information for the new object is placed in the body of a POST request. The data will typically be in JSON format.

When an object is modified, the information being modified is placed in the body of a PUT request.

GET requests and DELETE requests typically omit the body. The object being requested, or the object to be deleted, is usually specified by its id in the endpoint.

### HTTP Status Codes and Error Messages

When a request is made, the result of the request is communicated in the response via a status code. The code is a three-digit number and may reflect a successful request, a redirection, a client error or a server error as shown in the table below.

|  |  |
| --- | --- |
| Status Code Range | Description |
| 200+ | The request succeeded. A successful request to the FMC REST API will typically result in a status code of 200. |
| 300+ | The request is redirected to another URL. |
| 400+ | An error that originates from the client has occurred. |
| 500+ | An error that originates from the server has occurred |

Table 2: REST API response status codes

### REST API Authentication

Generally, a client authenticates with a REST API using:

1. Basic authentication (a username and password), or
2. A secret token

The FMC REST API uses token authentication. To obtain the access token requires a request that provides the username and password of a valid FMC account.

The access token expires after 30 minutes. But when requesting the access token, the API provides a refresh token in addition to the access token. The refresh token can be used to request a new access token without having to provide the username and password again. It can be used to do this up to three times.

Authentication is described in detail in the Firepower Management Center REST API Quick Start Guide at <https://www.cisco.com/c/en/us/td/docs/security/firepower/660/api/REST/firepower_management_center_rest_api_quick_start_guide_660/Connecting_With_A_Client.html>.

In this course, we will show how to obtain and use the access token and refresh token, both with Postman and programmatically with Python.

# Module 2: JSON

JSON (JavaScript Object Notation) is a text-based open standard for human-readable data interchange.

Generally, when requesting data from a REST API or sending data to a REST API, it will be formatted in JSON or XML. In this course, we will use JSON exclusively because the FMC REST API sends and accepts JSON only.

Formatting information into a JSON file is called serialization. JSON and XML are data serialization languages.

A file that contains data formatted in JSON should have a file extension of .json. The JSON Internet Media type is **application/json**.

## JSON Data Types

In JSON, values must be one of the following data types:

* a string
* a number
* an object (JSON object)
* an array
* a boolean
* null

The following table describes these data types and provides examples.

|  |  |  |
| --- | --- | --- |
| JSON Data Type | Description | Example |
| String | Text in double quotes | “AC Policy” |
| Number | Can be integer or floating point | 14 |
| JSON object | Set of key-value pairs in curly braces | {  “firstName”: “Susan”,  “lastName”: “Jones” } |
| Array | Ordered list of values in square brackets | [  “green”,  “red”,  “blue” ] |
| Boolean | Lower case true or false | true |
| Null | A value can be null | null |

Table 3: JSON data types

The following example shows a JSON object with two key-value pairs. The first pair has a key of “items”, and its value is an array of two items (in this case, to AC policies). Both items are objects with three key-value pairs.

The second key-value pair in the main JSON object has a key of “count” and a value of 2.

|  |
| --- |
| JSON Example |
| {  "items": [  {  "id": "005056A4-5126-0ed3-0000-025769803780",  "name": "ACPolicy-1",  "sendEventsToFMC":true,  },  {  "id": "005086A4-5126-0ed3-0000-025769803924",  "name": "ACPolicy-2",  "sendEventsToFMC":false,  }  ],  "count": 2  } |

Table 4: JSON example

# Module 3: API Explorer In Depth

The API Explorer is a rich environment in which to explore the API. The following figure shows the GET method and endpoint for retrieving access policies. The table below the figure explains the items in the screen shot.

A screenshot of a cell phone

Description automatically generated

Figure GET for access policies

The following table describes the items shown in the screen shot above.

|  |  |
| --- | --- |
| Item | Description |
| Policy | The API is divided into categories which are expandable in the API Explorer. The category is part of the endpoint, after the domainUUID. |
| GET | The method for this particular API call |
| Endpoint | The endpoint is displayed after the method. In this example, the endpoint ends with “accesspolicies”. |
| Parameters | Parameters are part of the endpoint: domainUUID, offset, limit, and expanded. |
| domainUUID | The API Explorer provides the domainUUID since it is common to all API calls. It is embedded in the path; the other parameters are appended as key-value pairs. |
| offset | Index of the first item to return. This is used when the GET request returns more than one page of data. The offset is then set to begin at the start of the next page. |
| limit | The number of items to return. |
| expanded | If set to true, the GET response displays a list of objects with additional attributes. If set to false, each item has only its name and id. |

Table 5: Features of a GET request

The following screen shot shows a response:

A screenshot of a cell phone

Description automatically generated

Figure GET response

Curl is a tool that is used to easily make a request and receive a response on a Linux server. The Request URL is shown here also. It is useful for making API calls with clients such as Postman. In this example, the response code is 200 (success), and the top portion of the response is shown. The response is a json object, and the data requested is the value of the items key. In the screenshot, the first item in the items array is shown, and it gives the name and id of an access policy.

# Module 4: Postman

Postman is a software development tool that allows us to test calls to APIs. We can easily perform the following steps to issue an API call:

1. Select a method
2. Enter an endpoint with or without parameters
3. Configure parameters, authorization, headers, and the body
4. Send the request

Postman (or another similar client) is important because it helps us become familiar with all parts of the API request, without having to use a programming language like Python. The API Explorer is also useful, but its ease of use hides details from us that we will need when we are programming in Python. In particular, it hides the authentication process (the API call to request the access token), which our code will need to perform.

The following figure shows an example of a Postman request for FMC access policies. The header for X-auth-access-token, which contains the access key, is shown in the Headers tab.

A screenshot of a cell phone

Description automatically generated

Figure Postman GET request for access policies

Postman also provides sample code in many languages for the API requests that you create. The figure below shows the Python code for the API call above.

A screenshot of a cell phone

Description automatically generated

Figure Python code snippet for GET request for access policies

# Module 5: Introduction to Python

Python is an [interpreted](https://en.wikipedia.org/wiki/Interpreted_language), [high-level](https://en.wikipedia.org/wiki/High-level_programming_language), [general-purpose](https://en.wikipedia.org/wiki/General-purpose_programming_language) [programming language](https://en.wikipedia.org/wiki/Programming_language). These terms are described in the table below.

|  |  |
| --- | --- |
| Item | Description |
| Interpreted | Generally, computer languages are either interpreted or compiled when executed. For a compiled language (such as Java), the source code is compiled into an executable format, and then the executable code is run. For an interpreted language, there is no compilation step. One invokes execution of the source code directly, and an interpreter reads the code and performs the operations. |
| High-level | A high-level language is consists of instructions mostly in a human language (such as Python’s print function). By contrast, the instructions of a low-level language are written in the language of the instruction set of the computer that will execute them. |
| General-purpose | A general purpose language has instructions for a wide variety of purposes. By contrast, a domain-specific language has instructions that are specialized for a particular application domain. |

Table 6: Python description

Python was designed to allow programmers to make code as readable as possible.

To begin working with Python, we will use the Visual Studio Code (VS Code) editor. It is possible to use a simple text editor to write Python programs, but an editor provides many features that we will find valuable. It is made by Microsoft but it is free, released under the MIT License.

The lab exercises for this module involve starting VS Code and writing and executing a simple Python program. When you have successfully executed your first Python program, you are ready to begin expanding your knowledge of Python. The screenshot below shows the hello\_world.py program and its execution.

A screenshot of a cell phone

Description automatically generated

Figure hello\_world.py and its execution

In this module, your instructor will use online resources to explain the basics of the Python programming language, and in the lab you will write several Python programs to start working with the Python language.

# Module 6: API Requests with Python

In this module we will obtain sample Python code from the FMC API Explorer that gets a list of the FMC’s access policies. We will configure this code for the lab FMC, turn off SSL verification, and execute this program to get the list of policies. We will then execute the same program again, but this time we will run it with the username and password on the command line. Finally, we will select a policy and modify our program to retrieve the access rules in that policy.

## Understanding FMC API Sample Code

The API Explorer provides sample Python code for all of the FMC API calls. In the lab, you will copy and use the code that is provided for GET requests for access policies. The following table provides an explanation of this Python code.

|  |  |
| --- | --- |
| Item # |  |
| #  # Generated FMC REST API sample script  # | These are comments and are not executed. |
| import json  import sys  import requests | Each of these lines refers to code that Python provides with a default installation.  When you execute this program, the Python interpreter incorporates that code, and we’ll use it in our program. |
| server = "https://10.81.127.36" | Creating a variable named server and assigning it the string that includes our FMC’s IP or hostname. |
| username = "admin"  if len(sys.argv) > 1:  username = sys.argv[1]  password = "sf"  if len(sys.argv) > 2:  password = sys.argv[2] | Assigning variables named username and password to the values needed to authenticate the request.  The if statements allow us to provide the username and password when we run the program. For example, we could run  $python get\_ac\_policies.py admin C1sco.123 |
| r = None | Assigning the variable r to None for the moment. We will use this variable to receive the results of requests later in the program. |
| headers = {'Content-Type': 'application/json'} | The headers variable will be used in the request to tell the FMC that the body (if any) will be in the JSON format. |
| api\_auth\_path = "/api/fmc\_platform/v1/auth/generatetoken"  auth\_url = server + api\_auth\_path | The first line assigns a variable to the endpoint path.  The second line appends this path to the server variable to create the complete request URL. |
| try:  some code we want to run  except Exception as err:  Code that runs if there is an error | The try … except construct in Python lets us run some code that could encounter an error.  If there is no error, then the code in the try block completes and the code in the except block does not execute.  If there is an error (in Python, an Exception) when running the code in the try block, then execution goes immediately to the start of the code in the except block. |
| # 2 ways of making a REST call are provided: | This and the next 7 lines are explained in the section on ssl verification. |
| r = requests.post(auth\_url, headers=headers, auth=requests.auth.HTTPBasicAuth(username,password), verify=False) | This line makes the request to the FMC for the token.  “requests” is called an object, and it provides a method called post(). We are telling the interpreter to execute the post() method, and we are providing:   * 1. URL,   2. Headers,   3. Username and password, and   4. A parameter that says “do not verify the FMC’s SSL certificate”   The post() method returns the FMC’s response, and we assign the variable r to that response. |
| auth\_headers = r.headers | The response r is also an object, and contains a lot of information.  Here we extract the headers from the response and store them in a variable called auth\_headers. |
| auth\_token = auth\_headers.get('X-auth-access-token', default=None) | The headers in the response contain the token. In this line, we extract the token from the headers. |
| if auth\_token == None:  print("auth\_token not found. Exiting...")  sys.exit() | If for some reason the headers do not contain the token, then we print an error message and terminate the program. |
| except Exception as err:  print ("Error in generating auth token --> "+str(err))  sys.exit() | If the API call to the FMC encountered an error, then we print an error message and terminate the program. |
| headers['X-auth-access-token']=auth\_token | This line adds a new header to the set of headers we already have.  This is the X-auth-access-token, and in this line we set it to the auth\_token that we just received from the FMC. |
| api\_path = "/api/fmc\_config/v1/domain/e276abec-e0f2-11e3-8169-6d9ed49b625f/policy/accesspolicies"  url = server + api\_path | Creating the URL for the upcoming request for access control policies. |
| if (url[-1] == '/'):  url = url[:-1] | If the url has a forward slash at the end, then we re-define the url, removing the forward slash at the end. |
| # GET OPERATION | A comment to indicate that we are ready to launch the GET request for access policies. |
| # REST call with SSL verification turned off:  [and the next three lines] | See the section on ssl verification. |
| r = requests.get(url, headers=headers, verify=False) | This line makes the request to the FMC for the access policies.  We are telling the interpreter to execute the get() method, and we are providing:   1. URL, 2. Headers, 3. A parameter that says “do not verify the FMC’s SSL certificate”   The get() method returns the FMC’s response, and we assign the variable r to that response. |
| status\_code = r.status\_code | One part of the response is the status\_code, and we store that in a variable called status\_code. |
| resp = r.text | The body of the response containing the access policies (in JSON) is assigned to resp. |
| if (status\_code == 200):  print("GET successful. Response data --> ")  json\_resp = json.loads(resp)  print(json.dumps(json\_resp,  sort\_keys=True,indent=4,  separators=(',', ': '))) | If the status code is 200, then we print the body that contains the access policies. |
| else:  r.raise\_for\_status()  print("Error occurred in GET --> "+resp) | Otherwise (if the status code is not 200), then we print an error message. |
| except requests.exceptions.HTTPError as err:  print ("Error in connection --> "+str(err)) | If an error occurs when we make the call, then we print an error. |
| finally:  if r : r.close() | Finally, we close the response object r. |

Table 7: Explanation of FMC API Python code that requests access policies.

## Configuring the API Python Code

The code provided by the API Explorer to GET access policies must be configured with the FMC hostname (or IP address), username, and password in order to operate properly. The third and fourth rows of the table above show where these appear in the code, and in the lab, you will change them to reflect the FMC you are working with and the account you are using.

## Turning Off SSL Verification

The code provided by the API Explorer assumes that the request will verify the FMC’s certificate to authenticate the FMC. However, it contains instructions for turning off this verification. The table below shows the instructions and the code.

|  |
| --- |
| Item # |
| # 2 ways of making a REST call are provided:  # One with "SSL verification turned off" and the other with "SSL verification turned on".  # The one with "SSL verification turned off" is commented out. If you like to use that then  # uncomment the line where verify=False and comment the line with =verify='/path/to/ssl\_certificate'  # REST call with SSL verification turned off:  # r = requests.post(auth\_url, headers=headers, auth=requests.auth.HTTPBasicAuth(username,password), verify=False)  # REST call with SSL verification turned on: Download SSL certificates from your FMC first and provide its path for verification. |
|
|

Table 8: SSL verification comments and code

As the comments indicate, in order to turn off SSL verification, the line that assigns the variable r needs to be commented out, and the commented-out line that assigns r with verify=false needs to be uncommented. You will perform this in the lab, because the lab FMC may not be configured with a valid certificate.

## Executing the Program

The program is ready to execute and when executed it should return an InsecureRequestWarning (because SSL verification is turned off) and the list of access policies in JSON format.

## Executing the Program with Parameters

The sample code provides for supplying the username and password as parameters. In the lab, you will test this feature.

## Eliminating the InsecureRequestWarnings

The warning message about making an insecure request becomes tiresome in the output. This warning message can be eliminated for the duration of this course by commenting out the lines in Python that produce it. The lab exercise contains instructions for doing this.

Please remember to uncomment these lines after the course, so that you are warned whenever you attempt an insecure connection to a remote host.

## Requesting Access Control Rules

To create a request for an access policy’s rules, we could use code provided by API Explorer, as we did in the previous section to obtain a list of access policies. The code for this request is very similar to the code for requesting an access policy. The main difference is that the endpoint changes to contain the id of the policy and to reflect that we are requesting the rules of that policy.

Similarly, we could copy each piece of code from API Explorer, for each API call. However, the result would be many files, all with nearly identical code, with mainly the endpoint changing. Instead, we are going to work towards making a generic GET request that can receive any endpoint and issue the request. For the moment, in the lab for this module you will simply change the code we have for requesting access policies to code that requests the rules for an access policy.

# Module 7: Building a Code Framework

At the end of Module 6, we saw how the request for a list of access policies and the request for the list of rules in a policy only differed by the endpoint. This allowed us to easily modify the initial request to make another request.

The goal of this module is to begin constructing a framework to make it easier to perform many functions in the FMC API without duplicating code.

The principles we will follow in constructing this framework are as follows:

1. Avoid duplication of code.
2. Separate code that changes from code that does not change

These principles are described in the sections below.

## Avoid Duplication of Code through Reusable Modules

We want to avoid duplicate code for two primary reasons:

1. To keep the code base smaller and simpler
2. To make it easier to improve/update the code

A primary way to avoid duplicate code is to create a re-usable function.

### Converting the GET call to a function

At this point, we have created two programs:

1. A program that retrieves access policies.
2. A program that retrieves the access rules for a policy.

Both of these programs are complicated, and they are nearly identical.

As we continue to create more programs that GET items, what is there in these programs that could change?

* Server
* Username
* Password
* Endpoint

We should consider a function that does not change, but that receives these items, issues the GET request, and returns the result.

This would allow us to write a simple program that retrieves information by simply defining the endpoint and then calling the reusable GET function. You will do this in the corresponding lab exercises.

The program that we execute is often called a main program. In this case, our main program performs the following operations:

1. Imports the module that contains the get\_object function. Note that the module is named get\_7\_1\_v1 and the file is named get\_7\_1\_v1.py. But as we use it in this main program, we want to refer to this import simply as get.
2. Creates a JSON object that has the FMC information, and assign it to a variable called fmc\_info.
3. Creates an endpoint variable, assign it a value, call the get\_object function, and print the result.
4. Reassigns the endpoint variable so that it points to accessrules for a particular policy, call the get\_object function, and print the result.

The following code illustrates the simplicity of the main program and the reusability of the get\_object function.

import get\_7\_1\_v1 as get

fmc\_info = {

'server': 'https://10.81.127.36',

'username': 'api',

'password': 'superpass'

}

endpoint = 'policy/accesspolicies'

print (get.get\_object(fmc\_info, endpoint))

endpoint = 'policy/accesspolicies/005056A4-5126-0ed3-0000-042949673459/accessrules'

print (get.get\_object(fmc\_info, endpoint))

## Parsing the Response

The JSON returned from the FMC provides a lot of info that is not very useful, as illustrated in the example output below which shows just two policies:

|  |
| --- |
| Example Output of a Request for Access Control Policies |
| RCLENDEN-M-D6UH:Mod7\_1 rclenden$ python3 main\_1.py  GET successful. Response data -->  {  "items": [  {  "id": "005056A4-5126-0ed3-0000-025769803780",  "links": {  "self": "https://10.81.127.36/api/fmc\_config/v1/domain/e276abec-e0f2-11e3-8169-6d9ed49b625f/policy/accesspolicies/005056A4-5126-0ed3-0000-025769803780"  },  "name": "ChildPolicy",  "type": "AccessPolicy"  },  {  "id": "005056A4-5126-0ed3-0000-042949673459",  "links": {  "self": "https://10.81.127.36/api/fmc\_config/v1/domain/e276abec-e0f2-11e3-8169-6d9ed49b625f/policy/accesspolicies/005056A4-5126-0ed3-0000-042949673459"  },  "name": "ParentPolicy",  "type": "AccessPolicy"  }  ],  "links": {  "self": "https://10.81.127.36/api/fmc\_config/v1/domain/e276abec-e0f2-11e3-8169-6d9ed49b625f/policy/accesspolicies?offset=0&limit=3"  },  "paging": {  "count": 3,  "limit": 3,  "offset": 0,  "pages": 1  }  } |
|
|

Table 9: Example output of request for access policies

The only useful information in this output is the name and ID of each policy. We can extract this information from the output to print only the useful information. You will do this in the corresponding lab exercises.

## Separate Code that Changes from Code that Does Not Change

Suppose I have a single program with 100 lines of Python code. In virtually any kind of program like this, one can identify parts of the code that are performing a particular function. The code as a whole is likely receiving some input, processing it, and producing output, but inside the program, parts of it are performing smaller input>process>output operations to produce the overall output. As an example, in requesting access policies, the first part of the code requests the access token, and the second part uses that access token to make the request for access policies.

It is important to identify these “sub-functions” and separate them out as actual Python functions. This is because it will simplify our code. If we are writing a program and part of it involves retrieving all the access policies, we want to have one line of code that calls a function that performs this, and not have all the detailed code inline at that spot.

In the 7.3 lab, you will separate the properties code away from the main program. The properties will go into a properties file and will be read in by a settings module. You will also separate the code that prints items into a utils module, and separate constants into a constants module.

## Module 7.4 has been eliminated

## Module 7.5 has been eliminated

## Logging

As our program grows more complex, it becomes more difficult to troubleshoot issues in execution. Often we insert print statements to assist with debugging, but logging is far more efficient as our program grows larger.

In the lab for this module, you will learn to incorporate logging into our program.

# Module 8: GETting and POSTing Objects

## GETting Objects

In the lab for this module, you will use code we have already written to GET objects to get network objects.

## POSTing an Object

In the lab for this module, you will start with sample code from the API Explorer for POSTING objects. You will convert that code into a function for POSTing objects. The function will receive the fmc\_info dictionary that includes the server, username, and password. It will also receive the endpoint (the type of object we are posting) and the data (the object name, value, and any other information about the object). It will then POST the object.

## POSTing Objects and Groups in Bulk

Often, we may have many objects or groups that we need to POST. It may be convenient to put the object or group information into a CSV file. Then our program can read each line from the CSV and POST the object or group.

### POSTing Network Objects from a CSV File

In the lab for this section, you will create a CSV file with fields of Name, Type, and Value, and populate it with various network, range, and host objects. You will then read in the file and iterate through the objects, POSTing each object.

### POSTing Network Groups in Bulk

Whereas an object has a simple value, the “value” of a group consists of its members. Each member can be a literal or an object. A literal has a type and a value, whereas an object has a type and an id. However, when creating a group, we want to specify group members that are objects by name, not by id. The CSV will specify the name of the member object, and our program must look up the id of that object given its name.

To perform this lookup, you will first request all network objects and groups, and create a list of networks and groups. Each network or group will be a dictionary that contains the name and id of the network or group. Then, as we read each group from the CSV, we can look up each member object to obtain its id, and post the group with its literal and object members. You will perform these operations in the lab for this module.

Note that when we request all network objects and groups, we will likely have multiple pages of results. In the lab for this module, we will also create code to retrieve all of the pages of a response and incorporate the results into one list.

## POSTing Objects and Groups in Bulk

Although the code in our main program works, it may be over 100 lines since it has all the details of how to POST network objects and groups. It can also GET and POST objects. In this module you will abstract much of the main code into the post module. But we will begin by using argparse to specify operations via the command line.

### argparse for GET

Consider that our program can do many things now. How can we specify the single operation we want to do at a given moment? One option is to use command line arguments to specify the operation and any needed parameters for the operation we want to perform.

Using command line arguments, we can specify whether we want to get or post objects. If we want to get them, we can specify the types of object we want to get. If we want to post them, we can specify a file that contains the objects to be posted.

The python.org web site has a good tutorial on how to use argparse to incorporate command line arguments into our program. The tutorial is at: <https://docs.python.org/3.3/howto/argparse.html#id1>

In the lab for this module, you will write code to use command line arguments to specify the GET method and the type of object you want to GET.

### argparse for POST

In the lab for this module, we will build on the previous code to allow the user to specify a method of POST at the command line, followed by the path to the CSV file that contains the objects or groups to be posted.

### Posting from net group file

In the lab for this module, we will move the code for posting network groups into the post module.

### Abstracting argparse and post

In the lab for this module, we will improve the code by abstracting the argparse functionality into its own function, and by breaking down code in the post module into smaller functions.

# Module 9: POSTing AC Rules

In this module, we will POST AC rules. As with other POSTs, we will put the rules into a CSV file.

## POSTing Simple AC Rules

We will start with a very simple rules and need only these fields in the CSV:

1. AC Policy
2. Name
3. Description
4. Enabled
5. Type
6. Action

To POST an AC rule, we have to construct the POST body. We will use the DictReader method from the csv module to provide a dictionary of data items from the CSV file, and modify it as needed.

In the lab for this section, you will POST these very simple rules into a policy of your choice, by hardcoding the ID into the post function.

## POSTing AC Rules with AC Policy Lookup

In the last section, we hard-coded the ID of the AC policy into the endpoint that the post code used. In the lab for this section, you will glean the name of the policy from the CSV and perform a lookup for the ID.

# Module 12: Converting Rules from Text

We will explore reading rules from a text file, creating rule objects from them, and POSTing the rules.

# Module 13: AC Rule Advanced Operations

We will explore POSTing rules in bulk, into specific sections and categories, and before and after another rule.

# Module 14: Deployment

We will deploy changes to the vFTD firewall.

# Appendix A: Acronym Listing

|  |  |
| --- | --- |
| Term | Definition |
| API | Application Programming Interface |
| FMC | Firepower Management Center |
| REST | Representational State Transfer |
|  |  |
|  |  |

Trademarks and Disclaimers

THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR CISCO REPRESENTATIVE FOR A COPY.

NOTWITHSTANDING ANY OTHER WARRANTY HEREIN, ALL DOCUMENT FILES AND SOFTWARE OF THIRD PARTY SUPPLIERS ARE PROVIDED “AS IS” WITH ALL FAULTS. CISCO AND THIRD PARTY SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL CISCO OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF CISCO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Cisco and the Cisco Logo are trademarks of Cisco Systems, Inc. and/or its affiliates in the U.S. and other countries. A listing of Cisco's trademarks can be found at www.cisco.com/go/trademarks. Third-party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company.

Any Internet Protocol (IP) addresses and phone numbers used in this document are not intended to be actual addresses and phone numbers. Any examples, command display output, network topology diagrams, and other figures included in the document are shown for illustrative purposes only. Any use of actual IP addresses or phone numbers in illustrative content is unintentional and coincidental.

©2017 Cisco Systems, Inc. All rights reserved.

End of Document