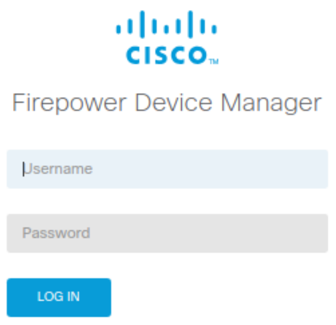
Discovery 13: Synchronize Firepower Device Configuration

Task 1: Explore Firepower Device Manager

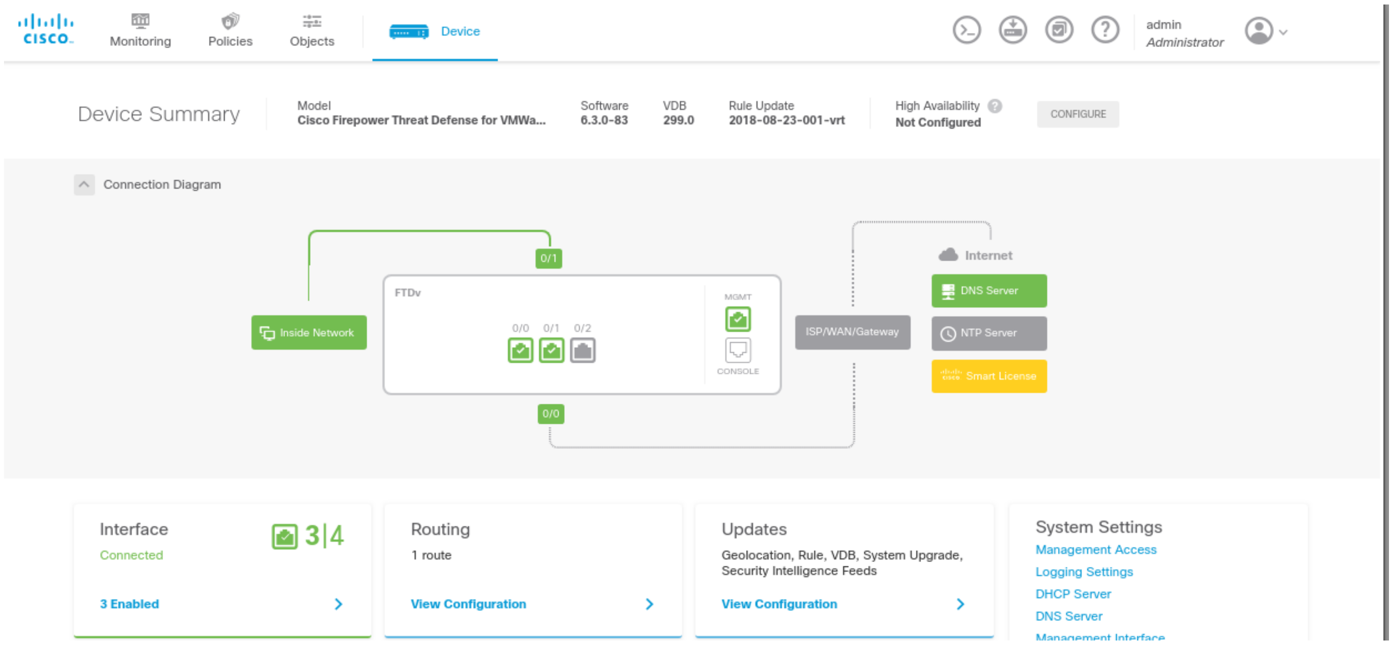
In this task, you will connect to Firepower Device Manager and explore the current configuration of the URL filtering.

Activity

Step 1: Open the web browser from the desktop and connect to the *https://192.168.0.40*. Use credentials *admin/1234QWer*. Accept the security warning.



Note
If you see the Service Unavailable message, wait for couple of minutes for FDM to be fully functional.



Device Summary

Model: Cisco Firepower Threat Defense for VMWa...
Software: 6.3.0-83
VDB: 299.0
Rule Update: 2018-08-23-001-vrt
High Availability: Not Configured

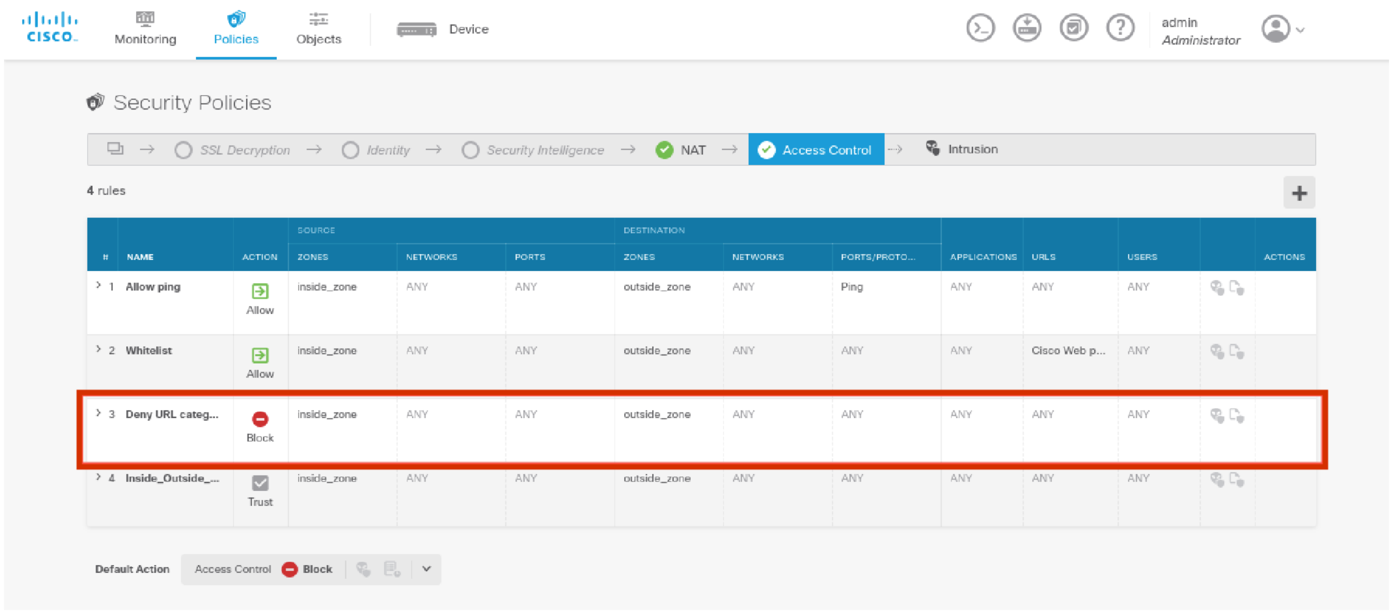
Interface — Connected — 3|4 — 3 Enabled

Routing — 1 route — View Configuration

Updates — Geolocation, Rule, VDB, System Upgrade, Security Intelligence Feeds — View Configuration

System Settings — Management Access, Logging Settings, DHCP Server, DNS Server, Management Interface

Step 2: In the top-left corner, click the *Policies* icon. Observe the access rules.



4 rules

#	NAME	ACTION	SOURCE ZONES	SOURCE NETWORKS	SOURCE PORTS	DESTINATION ZONES	DESTINATION NETWORKS	DESTINATION PORTS/PROTO...	APPLICATIONS	URLS	USERS		ACTIONS
1	Allow ping	Allow	inside_zone	ANY	ANY	outside_zone	ANY	Ping	ANY	ANY	ANY		
2	Whitelist	Allow	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	Cisco Web p...	ANY		
3	Deny URL categ...	Block	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	ANY	ANY		
4	Inside_Outside_...	Trust	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	ANY	ANY		

Default Action Access Control Block

You should see that a few rules are already created. There is also a rule with the name *Deny URL categories*, which you will update through the API.

Task 2: Construct Script Skeleton

In this task, you will prepare the skeleton for your code.

You will create three files:

- *fdm.py*: File will implement the FDM client. This code will handle all communication with FDM.
- *config_sync.py*: File will implement main logic for configuration synchronization. The code will read the configuration file and apply the configuration to FDM through the FDM client.
- *fdm.cfg*: File will include basic FDM parameters and URL categories that you want to filter.

The *config_sync.py* will be the file that you will run when you will deploy the configuration to FDM. Your script will be implemented in such a way that the script will accept a couple of command-line parameters to control the script execution. Apart from that, you will also implement logging, so that you will have basic information about what is going on during the script execution.

Activity

Step 1: Open Visual Studio Code (VSC) on the desktop. Open folder *working_directory* and create three required files in the folder.

Step 2: In the *config_sync.py* file, create a function with the name *parse_arguments* that will parse command-line arguments and will return parsed arguments with values. Use the *argparse* built-in Python module for implementation.

Add the following two arguments:

- *--config*: The argument accepts a configuration file. Default value should be *fdm.cfg*.
- *--debug*: The argument is used to display debug logs when executing the script.

```
import argparse

def parse_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument("--config", "-c",
                                    help="Path to the configuration file.",
                                    default='fdm.cfg')
    parser.add_argument("--debug", "-d",
                                    help="Display debug logs.",
                                    action="store_true")
    return parser.parse_args()
```

To parse command-line arguments with the *argparse* module, you first need to initialize the *ArgumentParser* class. Then you can add arguments as needed. In the example, you will add two arguments: *--config* and *--debug*. You will also add shorter versions: *-c* and *-d*. It is a good practice to add some help text. As per requirements, you need to specify the default value for the *--config* argument: *fdm.cfg*. The *--debug* argument will be stored as *True* or *False*.

Step 3: Add the *init_logger* function to the *config_sync.py* file. The function will implement the initialization for logging functionality. Use the Python built-in *logging* module for logging implementation. The function should accept the *log_level* parameter, which specifies the logging level that will be used in script execution. The default value should be *info*. The function should initialize logger, set the correct logging level and initialize the console logs handler. The format for the logs should be: "*log_date log_level: log_message*". The function should return an initialized logger.

```
import logging

def init_logger(log_level=logging.INFO):
    log = logging.getLogger(__file__)
    log.setLevel(log_level)
    console_handler = logging.StreamHandler()
    formatter = logging.Formatter('%(asctime)s %(levelname)s: %(message)s')
    console_handler.setFormatter(formatter)
    log.addHandler(console_handler)
    return log
```

The logger is initialized with *getLogger* function. You can set the log level with the *setLevel* method, where you pass the log level from logging module.

To define that logs are displayed in the console, you need to add a handler to the logger. You can add multiple handlers, like console handler, file handler, and others. In our example, we will add only console handler to the logger.

Handler accepts the initialized *Formater* class where you can specify the log format. You should add the *Formatter* class to the handler with the *setFormatter* method. Finally, you should add the handler to the logger with the *addHandler* method.

Step 4: Add both function calls to the main execution body of the *config_sync.py* script. Those functions should be called when you execute your script. If *config_sync.py* file is executed with the *--debug* parameter, you should initialize logger with the debug logging level.

```
if __name__ == "__main__":
    args = parse_arguments()

    if args.debug:
        log = init_logger(logging.DEBUG)
    else:
        log = init_logger()
```

The code stores the command-line parameters to *args* variable. If the *--debug* parameter is passed when executing the *config_sync.py* file, the logger is initialized with the *debug* logging level. Otherwise, the logger is initialized with the default level, which is *info*.

You have now prepared the basic skeleton for your script.

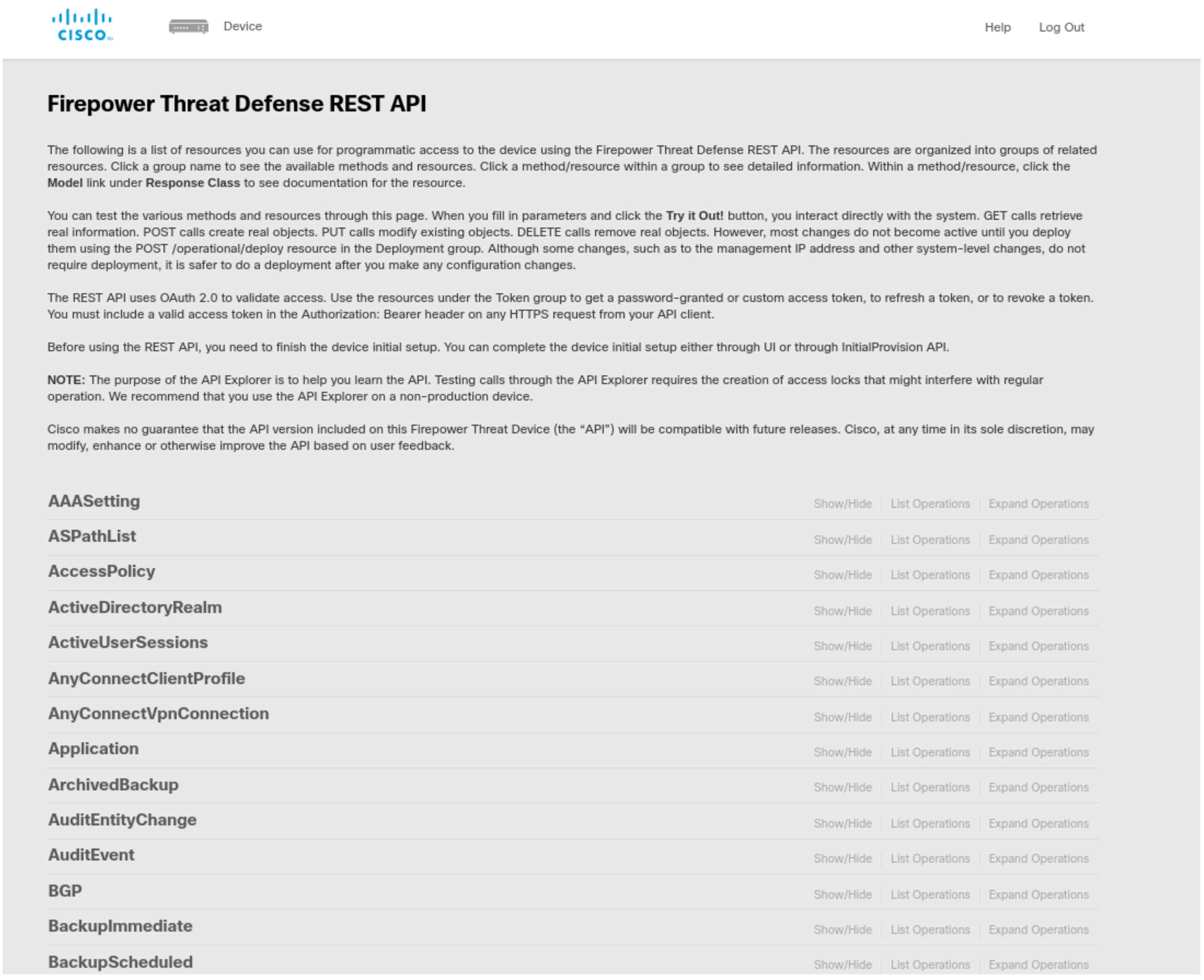
Task 3: Examine API Documentation

Firepower Device Manager has a documentation page where you can examine API calls to FDM. In this task, you will examine the documentation to get a better understanding of how you can automate different aspects of FDM.

You can use the documentation in other tasks to resolve the correct paths that you need throughout your lab activity.

Activity

Step 1: Open a web browser and enter the URL *https://192.168.0.40/#/api-explorer*. If needed, log in with the *admin*/*1234QWer* credentials.



Firepower Threat Defense REST API

The following is a list of resources you can use for programmatic access to the device using the Firepower Threat Defense REST API. The resources are organized into groups of related resources. Click a group name to see the available methods and resources. Click a method/resource within a group to see detailed information. Within a method/resource, click the **Model** link under **Response Class** to see documentation for the resource.

You can test the various methods and resources through this page. When you fill in parameters and click the **Try it Out!** button, you interact directly with the system. GET calls retrieve real information. POST calls create real objects. PUT calls modify existing objects. DELETE calls remove real objects. However, most changes do not become active until you deploy them using the POST /operational/deploy resource in the Deployment group. Although some changes, such as to the management IP address and other system-level changes, do not require deployment, it is safer to do a deployment after you make any configuration changes.

The REST API uses OAuth 2.0 to validate access. Use the resources under the Token group to get a password-granted or custom access token, to refresh a token, or to revoke a token. You must include a valid access token in the Authorization: Bearer header on any HTTPS request from your API client.

Before using the REST API, you need to finish the device initial setup. You can complete the device initial setup either through UI or through InitialProvision API.

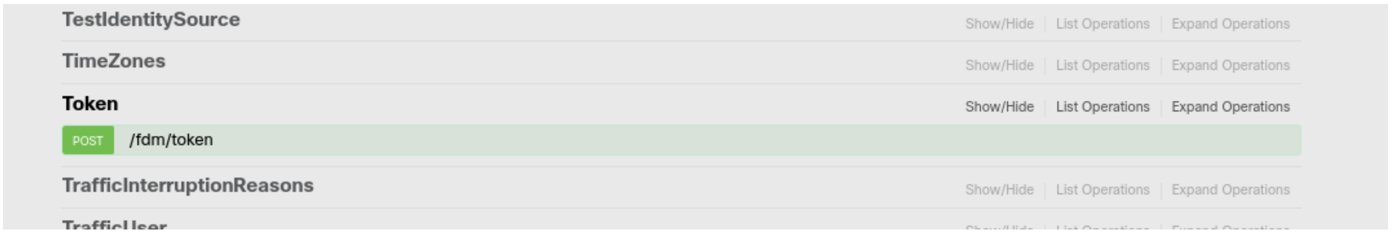
NOTE: The purpose of the API Explorer is to help you learn the API. Testing calls through the API Explorer requires the creation of access locks that might interfere with regular operation. We recommend that you use the API Explorer on a non-production device.

Cisco makes no guarantee that the API version included on this Firepower Threat Device (the "API") will be compatible with future releases. Cisco, at any time in its sole discretion, may modify, enhance or otherwise improve the API based on user feedback.

AAASetting	Show/Hide	List Operations	Expand Operations
ASPathList	Show/Hide	List Operations	Expand Operations
AccessPolicy	Show/Hide	List Operations	Expand Operations
ActiveDirectoryRealm	Show/Hide	List Operations	Expand Operations
ActiveUserSessions	Show/Hide	List Operations	Expand Operations
AnyConnectClientProfile	Show/Hide	List Operations	Expand Operations
AnyConnectVpnConnection	Show/Hide	List Operations	Expand Operations
Application	Show/Hide	List Operations	Expand Operations
ArchivedBackup	Show/Hide	List Operations	Expand Operations
AuditEntityChange	Show/Hide	List Operations	Expand Operations
AuditEvent	Show/Hide	List Operations	Expand Operations
BGP	Show/Hide	List Operations	Expand Operations
BackupImmediate	Show/Hide	List Operations	Expand Operations
BackupScheduled	Show/Hide	List Operations	Expand Operations

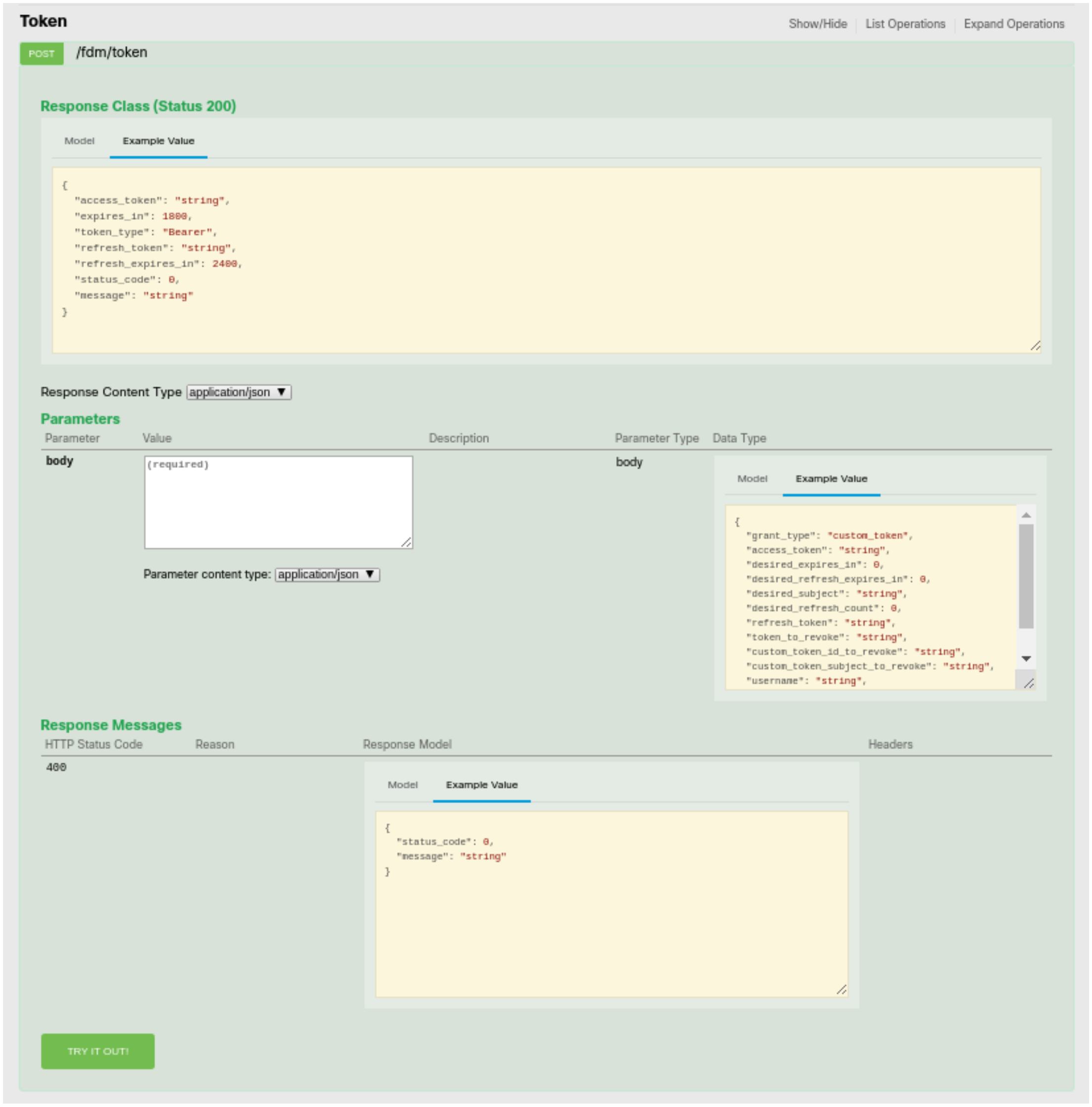
You should see the list of allowed modules that you can use to configure and verify by using API.

Step 2: Navigate to the *Token* module and click *Show/Hide*.


TestIdentitySource
Show/Hide | List Operations | Expand Operations
TimeZones
Show/Hide | List Operations | Expand Operations
Token
Show/Hide | List Operations | Expand Operations
POST /fdm/token
TrafficInterruptionReasons
Show/Hide | List Operations | Expand Operations
TrafficUser

FDM uses token base authentication when you perform subsequent calls to API. It means that you first need to retrieve the token, which is then used to authenticate your session. To retrieve the token, you need to use the POST method and the */fdm/token* path.

Step 3: Click the */fdm/token* to expand the description.


Token
Show/Hide | List Operations | Expand Operations
POST /fdm/token
Response Class (Status 200)
Model
Example Value

```
{
  "access_token": "string",
  "expires_in": 1800,
  "token_type": "Bearer",
  "refresh_token": "string",
  "refresh_expires_in": 2400,
  "status_code": 0,
  "message": "string"
}
```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
body	(required) Parameter content type: application/json		body	Model / Example Value

```
{
  "grant_type": "custom_token",
  "access_token": "string",
  "desired_expires_in": 0,
  "desired_refresh_expires_in": 0,
  "desired_subject": "string",
  "desired_refresh_count": 0,
  "refresh_token": "string",
  "token_to_revoke": "string",
  "custom_token_id_to_revoke": "string",
  "custom_token_subject_to_revoke": "string",
  "username": "string",
```

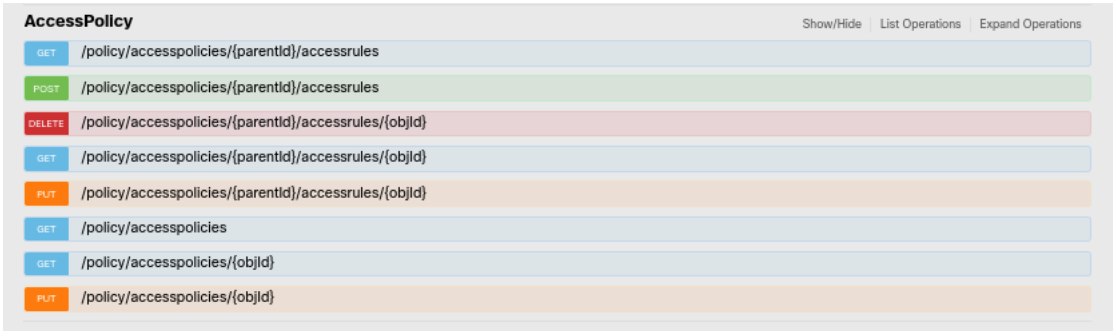
Response Messages

HTTP Status Code	Reason	Response Model	Headers
400		Model / Example Value	

```
{
  "status_code": 0,
  "message": "string"
}
```

You can get the description of all parameters, responses, and response messages. You can also use the "*TRY IT OUT!*" button to try the API call directly from the documentation.

Step 4: Navigate to the *AccessPolicy* module and click *Show/Hide*. Explore the API calls that you can use to configure access policy on FDM.



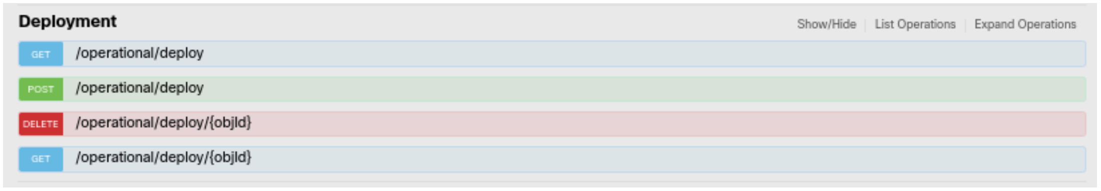
AccessPolicy

Show/Hide | List Operations | Expand Operations

```
GET     /policy/accesspolicies/{parentId}/accessrules
POST    /policy/accesspolicies/{parentId}/accessrules
DELETE  /policy/accesspolicies/{parentId}/accessrules/{objId}
GET     /policy/accesspolicies/{parentId}/accessrules/{objId}
PUT     /policy/accesspolicies/{parentId}/accessrules/{objId}
GET     /policy/accesspolicies
GET     /policy/accesspolicies/{objId}
PUT     /policy/accesspolicies/{objId}
```

You will use these methods to manipulate the access rules on FDM.

Step 5: Finally, navigate to the *Deployment* module, and click *Show/Hide*. Explore the API calls that you can see.



Deployment

Show/Hide | List Operations | Expand Operations

```
GET     /operational/deploy
POST    /operational/deploy
DELETE  /operational/deploy/{objId}
GET     /operational/deploy/{objId}
```

The *Deployment* module is used to deploy the configuration after you push the configuration to FDM. With these calls, you can request deployment or request the status of the deployment.

Task 4: Implement Basic Firepower Device Manager

In this task, you will construct the basic FDM client in Python. You will implement a basic class constructor, some support methods, login, and logout methods. Where appropriate, you will also add logging messages throughout the code.

Activity

Step 1: In the *fdm.py* file create the *FDMClient* class.

Store the variables in the class instance variables. If a log variable was not set, throw an exception.

The class should accept the following variables with the default values:

host: Specifies FDM IP address.
port: Specifies FDM port. Set the default value to *443*.
username: Specifies FDM username. Set the default value to *admin*.
password: Specifies FDM password. Set the default value to *1234QWer*.
log: Specifies the logger object. Set the default value to *None*.

```
class FDMClient:

    def __init__(self, host, port=443, username='admin', password='1234QWer', log=None):
        self.host = host
        self.port = port
        self.username = username
        self.password = password
        self.log = log
        if not log:
            raise Exception('The logger should not be None.')
```

Step 2: In the **FDMClient** constructor method, initialize the token variable and set the value to *None*. You need to send the *Content-Type* and *Accept* headers of every HTTP call. Add the base header dictionary to the **FDMClient** class constructor. The keys should be header types and the value should be set to application/json for both headers. Also add the base URL variable that will be used in every API call.

```
    def __init__(self, host, port=443, username='admin', password='1234QWer', log=None):
        <... output omitted ...>

        self.token = None

        self.base_headers = {
            'Content-Type': 'application/json',
            'Accept': 'application/json',
        }
        self.base_url = f'https://{self.host}:{self.port}/api/fdm/v2'

        requests.packages.urllib3.disable_warnings()

        self.log.debug('FDMClient class initialization finished.')
```

The class constructor sets the *self.token* to *None*. This variable will be used to store the token once the client is successfully authenticated. When you communicate with FDM via API, you need to send the *Content-Type* and *Accept* HTTP headers. Both headers should have the value *application/json*. So, that you don't repeat yourself in the code, you can specify the base header dictionary. You should also specify the base URL, which is used in all API calls. The URL is constructed from the IP address, port, and some suffix.

The line *requests.packages.urllib3.disable_warnings()* is an optional line. It is used to suppress the warnings when you are connecting to hosts via HTTPS with untrusted certificates. In a real production environment, you should add the trusted certificate to FDM. In that case, the line would not be needed.

At the end, there is a simple debug log that writes the debug message to the log. It is a good practice to equip your code with debug logs, which simplify troubleshooting.

Step 3: Next, implement the general send request method. The method should accept as paramaters URL, HTTP method, and optional headers, body, and query string dictionaries. The method should construct the HTTP request and analyze the response. If the error code is different to 200, throw an exception. Otherwise, return a response in dictionary syntax. Use Python *requests* module.

```
import requests


    def _send_request(self, url, method='get', headers=None, body=None, params=None):
        self.log.debug('Sending request to FDM.')
        request_method = getattr(requests, method)
        if not headers:
            headers = self.base_headers

        self.log.debug(f'Using URL: {url}')
        self.log.debug(f'Using method: {method}')
        self.log.debug(f'Using headers: {str(headers)}')
        self.log.debug(f'Using body: {str(body)}')
        self.log.debug(f'Using query strings: {str(params)}')

        response = request_method(url, verify=False, headers=headers, json=body, params=params)
        status_code = response.status_code
        response_body = response.json()
        self.log.debug(f'Got status code: {str(status_code)}')
        self.log.debug(f'Got response body: {str(response_body)}')
        if status_code != 200:
            msg = response_body.get('message', 'Request to FDM unsuccessful.')
            raise Exception(msg)
        return response_body
```

The method *_send_requests* accepts five parameters. The first, mandatory parameter is *url*, which specifies the path to the API resource. The next parameter is *method*, which specifies the method that is used for requesting the resource. The default value for the parameter is set to *get*. Other three parameters are optional and specify headers, body, and query string dictionaries.

The method uses the Python built-in function *getattr* to get the correct function from the *requests* module, based on the HTTP method that was passed to the method. So, if you pass the *get* method, the *getattr* function will return the *requests.get* function, which you can call later to send a request to FDM.

If the *header* parameter is not set, the method will take the *base_header* dictionary that you have created in the class constructor.

The method will then send a request to FDM and extract the HTTP status code and response body from the response. If HTTP status code is different from 200, the method will throw an exception. Finally, the method will return response body dictionary.

Step 4: Implement the *login* method. The *login* method should be used to request a token from FDM. Use the POST method with the */fdm/token* path to the token resource.

In body, you should send three parameters

- *grant_type*, which should be set to string '*password*'.
- *username*, which should be set to defined username.
- *password*, which should be set to defined password.

```
    def login(self):
        self.log.debug('Login to FDM.')

        url = self.base_url + '/fdm/token'
        body = {
            'grant_type': 'password',
            'username': f'{self.username}',
            'password': f'{self.password}',
        }

        self.log.debug('Sending the login request to FDM.')
        response = self._send_request(url, method='post', body=body)
        self.token = response.get('access_token')
        self.log.debug(f'Access token: {self.token}')
```

To construct the URL, you will use the *base_url* variable that you specified in the class constructor and the path to the token resource. The body is a dictionary, which has three variables as specified in code snippet.

The *login* method then uses implemented *_send_request* method to send the HTTP POST request to FDM.

Since the *_send_request* method already checks that HTTP status code was 200, you can assume that response is as expected. You can simply search for the *access_token* parameter in the response and store the value to *self.token* variable. This token will be used for all subsequent API calls.

Step 5: Implement the *logout* method. The *logout* method is similar to *login* method. Only the body of the API call is different.

Use the following parameters in request body:

- *grant_type*, which should be set to the string '*revoke_token*'.
- *access_token*, which should be set to the token that was received during *login* method.
- *token_to_revoke*, which should be set to the token that was received during *login* method.

```
    def logout(self):
        self.log.debug('Logout from FDM.')

        url = self.base_url + '/fdm/token'
        body = {
            'grant_type': 'revoke_token',
            'access_token': self.token,
            'token_to_revoke': self.token,
        }

        self.log.debug('Sending the logout request to FDM.')
        self._send_request(url, method='post', body=body)
        self.log.debug('Logout successful.')
```

The *logout* method is very similar to the *login* method. The request body is different.

Step 6: Finally, implement another helper method *_get_auth_headers*, which will be used by subsequent API calls to get the correct header dictionary. The method should return the base header dictionary, with an additional *Authorization* header where you will add the token.

```
    def _get_auth_headers(self):
        headers = self.base_headers.copy()
        if self.token:
            headers['Authorization'] = f'Bearer {self.token}'
        else:
            msg = 'No token exists, use login method to get the token.'
            raise Exception(msg)
        return headers
```

The method first copies the base headers dictionary. If the *self.token* variable is set, the method adds the *Authorization* header to the dictionary. The value is constructed from the *Bearer* string and token. If no token is set, the method throws an exception.

Task 5: Get Current Configuration

In this task, you will implement basic logic to retrieve the configuration from FDM. You will create a separate class for the configuration synchronization. This class will use the FDM client that you have implemented. You will also create a basic configuration file in YAML syntax to store basic FDM parameters, like host, username, and password. The goal of the task is to get the current configuration of the access rule from FDM by using a script.

Activity

Step 1: First, add basic FDM parameters to the *fdm.cfg* configuration file. Use YAML syntax.

Add the following variables:

The *fdm_host* variable with the value *192.168.0.40*.
The *fdm_username* variable with the value *admin*.
The *fdm_password* variable with the value *1234QWer*.

```
---
fdm_host: '192.168.0.40'
fdm_username: 'admin'
fdm_password: '1234QWer'
```

Step 2: In the config_sync.py file, create a *ConfigSync* class.

The class should accept two parameters:

config: Specifies the path to the configuration file.
log: Specifies the logging object that you have initialized.

```
class ConfigSync:

    def __init__(self, config, log):
        self.log = log
        self.config_file = config
        self.log.info('Initializing ConfigSync class.')
```

Step 3: Create a method that loads the configuration file, reads the configuration file, creates a dictionary from the YAML file, and returns the dictionary. Use the Python *yaml* module.

```
import yaml

class ConfigSync:

<...output omitted ...>

    def _parse_config(self, config_file):
        self.log.info('Parsing the configuration file.')
        with open(config_file, 'r') as f:
            config = yaml.safe_load(f)
        self.log.debug(f'The following parameters were received: {config}')
        return config
```

The best practice to open the file is to use Python *with open* syntax. This syntax makes sure that file is closed after reading. Once the file is opened, you should use the *safe_load* function in the *yaml* Python module to read the content of the file and create a dictionary from the YAML file. The parsed configuration is returned.

Step 4: Create a method that extracts the host, username, and password from the configuration dictionary. After the values are extracted, the method should initialize the *FDMClient* class with the required parameters and log in to FDM.

```
from fdm import FDMClient

class ConfigSync:

<...output omitted ...>

    def _init_fdm_client(self, config):
        self.log.info('Initializing FDMClient class.')
        host = config.get('fdm_host')
        username = config.get('fdm_username')
        password = config.get('fdm_password')
        fdm = FDMClient(host, username=username, password=password, log=self.log)
        self.log.info('Login to FDM.')
        fdm.login()
        return fdm
```

The code above extracts the host, username, and password from the configuration dictionary. Once you have those parameters, you can initialize the *FDMClient* class. The method then uses the *login* method to log in to FDM. Finally, the method returns the *FDMClient* class instance.

Step 5: Add the *_parse_config* and *_init_fdm_client* method calls to the *ConfigSync* class constructor.

```
    def __init__(self, config, log):
        self.log = log
        self.config_file = config
        self.log.info('Initializing ConfigSync class.')
        self.config = self._parse_config(config)
        self.fdm = self._init_fdm_client(self.config)
        self.log.debug('ConfigSync class initialization finished.')
```

Step 6: At the bottom of the file, add *ConfigSync* class initialization code.

```
if __name__ == "__main__":
    args = parse_arguments()

    if args.debug:
        log = init_logger(logging.DEBUG)
    else:
        log = init_logger()

    cs = ConfigSync(config=args.config, log=log)
```

You should initialize the *ConfigSync* class with the path to the configuration file and the logger instance.

Step 7: Now, that you have a basic implementation of the *ConfigSync* class, you can run a basic test. Add an extra line after the *ConfigSync* class initialization to print the received token to verify that log in was successful. After the test, remove the print statement.

```
if __name__ == "__main__":
    args = parse_arguments()

    if args.debug:
        log = init_logger(logging.DEBUG)
    else:
        log = init_logger()

    cs = ConfigSync(config=args.config, log=log)
    print(cs.fdm.token)
```

The initialized *FDMClient* is stored to the *fdm* variable in a *ConfigSync* class instance. You can get the token by accessing the token variable in *FDMClient* instance.

```
(working_directory) student@student-workstation:~/working_directory$ python config_sync.py
2019-10-18 07:23:24,829 INFO: Initializing ConfigSync class.
2019-10-18 07:23:24,829 INFO: Parsing the configuration file.
2019-10-18 07:23:24,830 INFO: Initializing FDMClient class.
2019-10-18 07:23:24,830 INFO: Login to FDM.
eyJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE1NzEzODM0MDEsInN1YiI6ImFkbWluIiwianRpIjoiMjljYjc1ZTItZjE3OC0xMWU5LThlMzEtOWJl
eHAiOjE1NzEzODUyMDEsInJlZnJlc2hUb2tlbkV4cGlyZXNBdCI6MTU3MTM4NTgwMTEwOCwidG9rZW5UeXBlIjoiSldUX0FjY2VzcyIsInVzZ
2QtMTMxZWM4Y2I4ODBhIiwidXNlclJvbGUiOiJST0xFX0FETUlOIiwib3JpZ2luIjoicGFzc3dvcmQiLCJ1c2VybmFtZSI6ImFkbWluIn0.K0
```

Since you can see the token, you know that log in was successful. Now remove the print statement from the code.

```
if __name__ == "__main__":
    args = parse_arguments()

    if args.debug:
        log = init_logger(logging.DEBUG)
    else:
        log = init_logger()

    cs = ConfigSync(config=args.config, log=log)
```

Note
If login is not successful, run the script in debug mode to troubleshoot issues. Use the **Python config_sync.py --debug** command.

Now you are ready to implement the logic to retrieve the access rule that you are going to update with script.

Step 8: First update the *fdm.py* client. You should implement the method that retrieves the access rule by name. Based on the documentation, you need to get the *policy_id* parameter from FDM. Implement two methods. The first method *get_access_policy_id* should search for the *policy_id* parameter on FDM and it should return the parsed ID. The second method *get_access_rule_by_name* should use that *policy_id* parameter to search for the correct access rule that is based on the name that you have passed as a method parameter. The method should extract the current access rule configuration and it should return the current access rule configuration.

```
    def get_access_policy_id(self):

        url = self.base_url + '/policy/accesspolicies'
        headers = self._get_auth_headers()

        self.log.debug('Requesting access policies from FDM.')
        response = self._send_request(url, headers=headers)
        policy_id = response['items'][0]['id']
        self.log.debug(f'Policy ID is: {policy_id}')
        return policy_id
```

The method *get_access_policy_id* connects to FDM and retrieves policies from FDM. Basically, there is only one policy in the list, therefore you can take the first element from the list and search for the *id* parameter. This parameter is the policy ID that you can use later to search for access rules.

```
    def get_access_rule_by_name(self, name):
        self.log.debug('Searching for access rule.')
        policy_id = self.get_access_policy_id()

        url = self.base_url + f'/policy/accesspolicies/{policy_id}/accessrules'
        headers = self._get_auth_headers()

        self.log.debug('Requesting access rules from FDM.')
        response = self._send_request(url, headers=headers)
        access_rules = response.get('items')

        rule_data = None
        for rule in access_rules:
            if name == rule.get('name'):
                rule_data = rule
                break
        if rule_data is None:
            raise Exception('Unable to find requested rule.')
        return rule_data
```

The method first calls the *get_access_policy_id* method to get the *policy_id* parameter. Once the method has the policy_id parameter, it can construct the URL to get all access rules. When the method receives the access rules, it uses the *for* loop to search for the correct access rule that is based on the *name* parameter that you have passed to the method call. The method returns the current configuration for the access rule.

Step 9: Update the configuration file with the *url_filtering* dictionary key. Under the *url_filtering*, add the *rule_name* variable and set the value to '*Deny URL categories*'.

```
---
fdm_host: '192.168.0.40'
fdm_username: 'admin'
fdm_password: '1234QWer'
url_filtering:
  rule_name: 'Deny URL categories'
```

This part of the configuration will specify the rule that you are going to update during the configuration synchronization. You have retrieved the name of the rule in the first task.

Step 10: Create the *get_config* method in the *ConfigSync* class. The *get_config* method calls the *get_access_rule_by_name* method in the *FDMClient* class to retrieve the current configuration of the access rule that is specified in the configuration file.

```
    def get_config(self):
        access_rule_name = self.config['url_filtering']['rule_name']
        self.log.info('Requesting access rule for URL filtering from FDM.')
        self.access_rule = self.fdm.get_access_rule_by_name(access_rule_name)
```

To get the specific access rule, you need to pass the access rule name to the method call. The rule name is extracted from the configuration.

Step 11: Add the *get_config* method call to the script. Add the call after the *ConfigSync* class is initialized.

```
if __name__ == '__main__':
    args = parse_arguments()

    if args.debug:
        log = init_logger(logging.DEBUG)
    else:
        log = init_logger()

    cs = ConfigSync(config=args.config, log=log)
    cs.get_config()
```

Step 12: Test that you receive the access rule configuration from FDM. Add an extra line after the *cs.get_config()* line to print the received access rule. Since the output will be a dictionary, use the Python *json* module for printing to get a better view of the configuration. After the test, remove the test code.

```
if __name__ == '__main__':
    args = parse_arguments()

    if args.debug:
        log = init_logger(logging.DEBUG)
        else:
            log = init_logger()

        cs = ConfigSync(config=args.config, log=log)
        cs.get_config()

        import json
        print(json.dumps(cs.access_rule, indent=4))
(working-directory) student@student-workstation:~/working-directory$ python config_sync.py
2019-10-18 07:59:24,295 INFO: Initializing ConfigSync class.
2019-10-18 07:59:24,295 INFO: Parsing the configuration file.
2019-10-18 07:59:24,296 INFO: Initializing FDMClient class.
2019-10-18 07:59:24,296 INFO: Login to FDM.
2019-10-18 07:59:24,546 INFO: Requesting access rule for URL filtering from FDM.
{
    "version": "jtfqlayuyppwk",
    "name": "Deny URL categories",
    "ruleId": 268435460,
    "sourceZones": [
        {
            "version": "phi2yu6trrl4z",
            "name": "inside_zone",
            "id": "90c377e0-b3e5-11e5-8db8-651556da7898",
            "type": "securityzone"
        }
    ],
    "destinationZones": [
        {
            "version": "dppnkgbo7v57z",
            "name": "outside_zone",
            "id": "b1af33e1-b3e5-11e5-8db8-afdc0be5453e",
            "type": "securityzone"
        }
    ],
    "sourceNetworks": [],
    "destinationNetworks": [],
    "sourcePorts": [],
    "destinationPorts": [],
    "ruleAction": "DENY",
    "eventLogAction": "LOG_BOTH",
    "users": [],
    "embeddedAppFilter": null,
    "urlFilter": {
        "urlObjects": [],
        "urlCategories": [],
        "type": "embeddedurlfilter"
    },
    "intrusionPolicy": null,
    "filePolicy": null,
    "logFiles": false,
    "syslogServer": null,
    "id": "ea9c0121-ee69-11e9-8e31-0d53a9497ea3",
    "type": "accessrule",
    "links": {
        "self": "https://192.168.0.40/api/fdm/v2/policy/accesspolicies/c78e66bc-cb57-43fe-bcbf-96b79b3475b3/a
    }
}
```

You can see that test was successful. When you will edit the access rule, you will have to append the required URL categories into the *urlFilter/urlCategorie*s section. Remove the test code.

```
if __name__ == '__main__':
    args = parse_arguments()

    if args.debug:
        log = init_logger(logging.DEBUG)
    else:
        log = init_logger()

    cs = ConfigSync(config=args.config, log=log)
    cs.get_config()
```

Task 6: Add and Remove Updated Items

In this task, you will implement the code that updates the URL categories in the access rule. You will add the URL categories that you want to filter into the configuration file. To construct the correct API call, you will need the URL category ID. So, to get the correct URL category IDs, you will implement the code in the *FDMClient* class to retrieve all available URL categories with all parameters. You will then update the *ConfigSync* class to read the URL categories from the configuration file. Next, search for correct URL category IDs and update the access rule according to that piece of information. Once the access rule is updated, you will implement the logic in *FDMClient* class to push the rule to FDM.

Activity

Step 1: First, implement the *get_url_categories* method in the *FDMClient* class.

```
    def get_url_categories(self):
        self.log.debug('Searching for URL categories on FDM.')
        url = self.base_url + '/object/urlcategories'
        headers = self._get_auth_headers()
        params = {'limit': '100'}

        self.log.debug('Sending request for getting URL categories from FDM.')
        response = self._send_request(url, headers=headers, params=params)
        return response.get('items')
```

To get all available categories, you should use the */object/urlcategories* path. You should use the *_get_auth_headers* method to get all required HTTP headers. You should also use the query strings in this call. By default, FDM returns only 10 items. You would need to do additional calls to retrieve more items. Alternatively, you can use the query string parameter *limit* to specify the number of objects that you want to receive in single call. If you use 100, you will get all available URL categories.

Step 2: Add some test URL categories to the *fdm.cfg* configuration file. Use the *url_categories* dictionary key.

```
---
fdm_host: '192.168.0.40'
fdm_username: 'admin'
fdm_password: '1234QWer'
url_filtering:
  rule_name: 'Deny URL categories'
  url_categories:
    - 'Hate and Racism'
    - 'Violence'
    - 'Training and Tools'
```

Step 3: Now implement the *sync* method in the *ConfigSync* class. The method should first call the *get_url_categories* method from the *FDMClient* class and store the received URL categories to the variable.

```
    def sync(self):
        self.log.info('Starting the configuration synchronization.')
        self.log.info('Requesting URL categories from FDM.')
        self.url_categories = self.fdm.get_url_categories()
```

Step 4: Since you will need to search for the required URL category among all available URL categories, implement the *_get_url_category* method that accepts a URL category name. Next, prepare the data that you can append to the list of URL categories in the access rule. The following is the correct dictionary structure for the URL category in request:

```
{
    'urlCategory': {
        'name': CATEGORY_NAME,
        'id': CATEGORY_ID,
        'type': CATEGORY_TYPE,
    },
    'type': 'urlcategorymatcher'
}
```

You can get all pieces of information from the list of available URL categories.

```
    def _get_url_category(self, name):
        category_dict = None
        for category in self.url_categories:
            category_name = category['name']
            if category_name == name:
                category_dict = {
                    'urlCategory': {
                        'name': category_name,
                        'id': category['id'],
                        'type': category['type']
                    },
                    'type': 'urlcategorymatcher'
                }
                break
        return category_dict
```

The method performs the *for* loop over all available URL categories. Once the correct URL category is found, the method prepares the required data and returns the dictionary.

Step 5: Now update the *sync* method. The *sync* method should first clean the *urlCategories* list in the access rule configuration. Then it should loop through the required URL categories in the configuration file and update the *urlCategories* list with the dictionary for each individual URL category.

```
    def sync(self):
        self.log.info('Starting the configuration synchronization.')
        self.log.info('Requesting URL categories from FDM.')
        self.url_categories = self.fdm.get_url_categories()
        self.access_rule['urlFilter']['urlCategories'] = []
        self.log.info('Updating the access rule.')
        for category in self.config['url_filtering']['url_categories']:
            cat_dict = self._get_url_category(category)
            if cat_dict:
                self.access_rule['urlFilter']['urlCategories'].append(cat_dict)
```

After you have updated the access rule, you can now send this updated rule to FDM.

Step 6: In the *FDMClient* class, create the *put_access_rule* method that accepts the access rule dictionary. To update the access rule, you need use the HTTP PUT method.

```
    def put_access_rule(self, data):
        self.log.debug('Updating access rule on FDM.')
        url = data['links']['self']
        headers = self._get_auth_headers()

        self.log.debug('Sending the request to update the access rule on FDM.')
        response = self._send_request(url, method='put', headers=headers, body=data)
        return response
```

The *put_access_rule* method accepts the access rule dictionary. You can find the correct URL to update the access rule under the access rule dictionary data. It is located under *links/self*. You then need to use the put method on this URL with the data that you have passed to the method call.

Step 7: Now, update the *sync* method to use the *put_access_rule* method to send the constructed access rule to FDM.

```
    def sync(self):
        self.log.info('Starting the configuration synchronization.')
        self.log.info('Requesting URL categories from FDM.')
        self.url_categories = self.fdm.get_url_categories()
        self.access_rule['urlFilter']['urlCategories'] = []
        self.log.info('Updating the access rule.')
        for category in self.config['url_filtering']['url_categories']:
            cat_dict = self._get_url_category(category)
            if cat_dict:
                self.access_rule['urlFilter']['urlCategories'].append(cat_dict)
        self.log.info('Adding the configuration to FDM.')
        self.fdm.put_access_rule(self.access_rule)
```

The *sync* method calls the *put_access_rule* with the updated access rule to push the configuration to FDM.

Step 8: Add the *sync* method call to the script. Add the call after the *get_config* method call.

```
if __name__ == "__main__":
    args = parse_arguments()

    if args.debug:
        log = init_logger(logging.DEBUG)
    else:
        log = init_logger()

    cs = ConfigSync(config=args.config, log=log)
    cs.get_config()
    cs.sync()
```

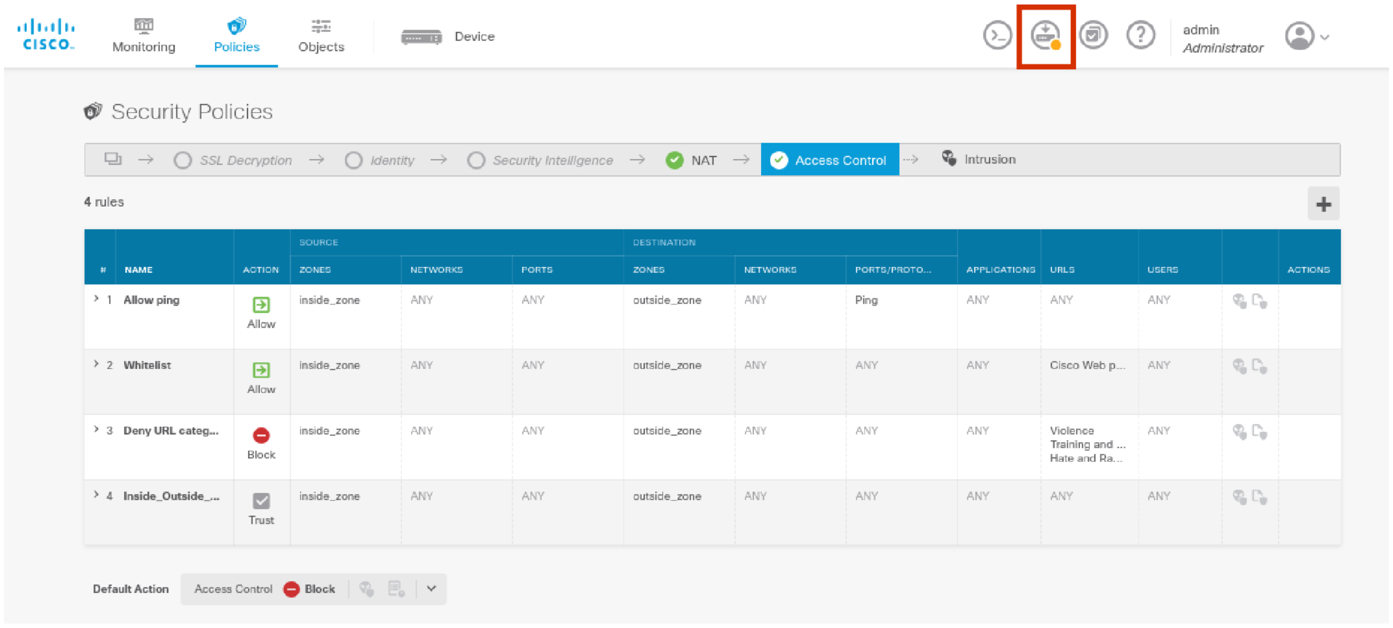
Now you can perform the test to check if your code is working correctly.

Step 9: Run the *config_sync.py* script in the terminal.

```
(working-directory) student@student-workstation:~/working-directory$ python config_sync.py
2019-10-18 10:02:31,336 INFO: Initializing ConfigSync class.
2019-10-18 10:02:31,336 INFO: Parsing the configuration file.
2019-10-18 10:02:31,338 INFO: Initializing FDMClient class.
2019-10-18 10:02:31,338 INFO: Login to FDM.
2019-10-18 10:02:31,577 INFO: Requesting access rule for URL filtering from FDM.
2019-10-18 10:02:31,732 INFO: Starting the configuration synchronization.
2019-10-18 10:02:31,732 INFO: Requesting URL categories from FDM.
2019-10-18 10:02:31,846 INFO: Updating the access rule.
2019-10-18 10:02:31,846 INFO: Adding the configuration to FDM.
```

You can see that the script was executed correctly.

Step 10: Log in to FDM by using the web browser. Navigate to the *Policies* tab. Observe the deploy configuration button.


Monitoring
Policies
Objects
Device
admin
Administrator
Security Policies
SSL Decryption → Identity → Security Intelligence → NAT → Access Control → Intrusion

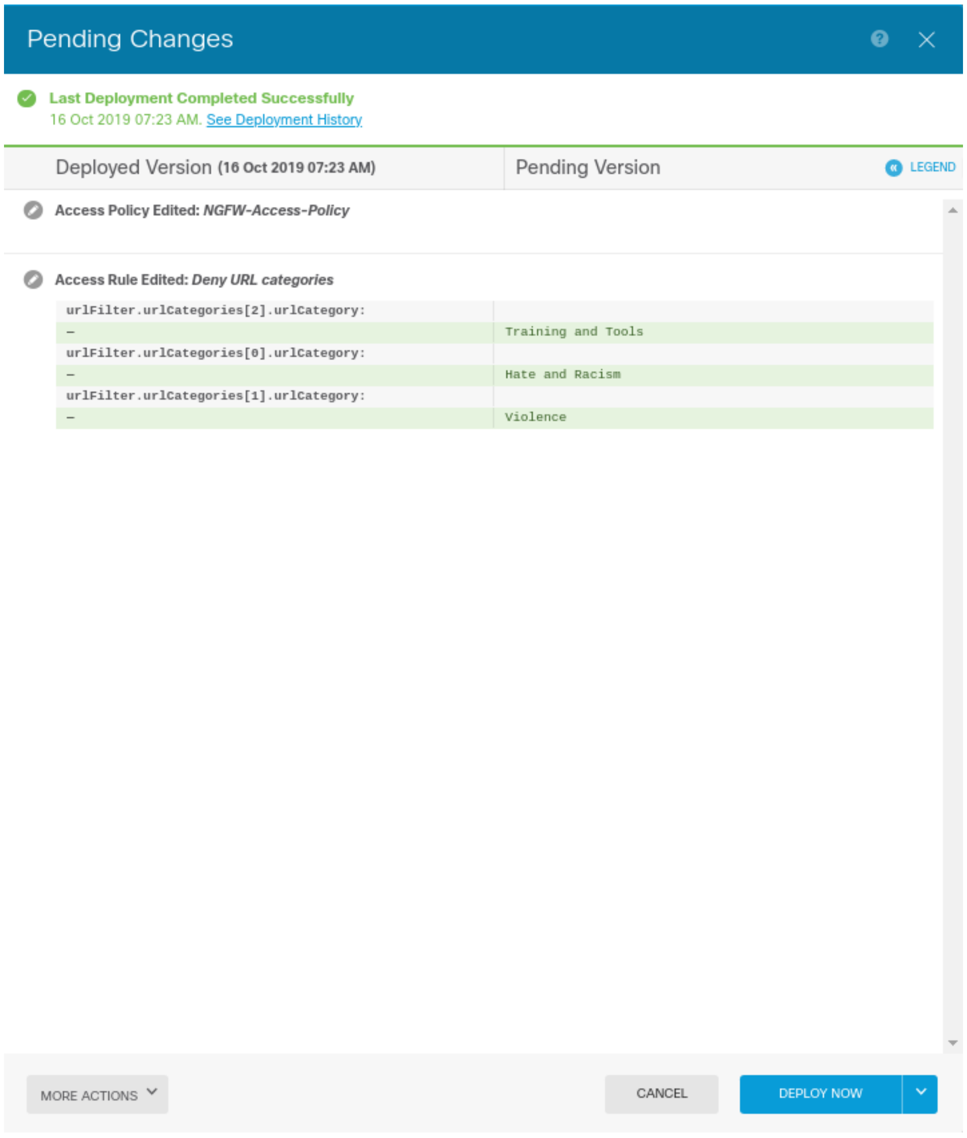
4 rules

#	NAME	ACTION	SOURCE ZONES	SOURCE NETWORKS	SOURCE PORTS	DESTINATION ZONES	DESTINATION NETWORKS	DESTINATION PORTS/PROTO...	APPLICATIONS	URLS	USERS		ACTIONS
1	Allow ping	Allow	inside_zone	ANY	ANY	outside_zone	ANY	Ping	ANY	ANY	ANY		
2	Whitelist	Allow	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	Cisco Web p...	ANY		
3	Deny URL categ...	Block	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	Violence Training and ... Hate and Ra...	ANY		
4	Inside_Outside_...	Trust	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	ANY	ANY		

Default Action Access Control Block

You should see a yellow mark on the button. It means that you have some configuration to deploy.

Step 11: Click the deploy configuration button and observe the window that will open.


Pending Changes
Last Deployment Completed Successfully
16 Oct 2019 07:23 AM. See Deployment History
Deployed Version (16 Oct 2019 07:23 AM)
Pending Version
LEGEND
Access Policy Edited: NGFW-Access-Policy
Access Rule Edited: Deny URL categories
MORE ACTIONS
CANCEL
DEPLOY NOW

```
urlFilter.urlCategories[2].urlCategory:
-                                        Training and Tools
urlFilter.urlCategories[0].urlCategory:
-                                        Hate and Racism
urlFilter.urlCategories[1].urlCategory:
-                                        Violence
```

You should see all three URL categories in the candidate configuration. The configuration is waiting to be deployed. It means that you have successful send a request to FDM.

Step 12: Discard the changes by clicking the *More Actions* button and selecting *Discard All*. You should confirm to discard the configuration.

Task 7: Deploy Configuration

Now that you have successfully pushed the configuration to FDM, you need to implement the logic for deploying the configuration. In this task, you will add the deploy method to the *FDMClient* class. You will then use the deploy method in the *ConfigSync* class.

Activity

Step 1: Implement the *deploy* method in the *FDMClient* class. Add the code that sends the deploy request to FDM. The method should accept the *timeout* as a parameter. Use the value 180 as a default value for the timeout. The *timeout* parameter will be used to define the time that code will wait for deployment to finish.

```
    def deploy(self, timeout=180):
        self.log.debug('Deploying the configuration.')

        url = self.base_url + '/operational/deploy'
        headers = self._get_auth_headers()

        self.log.debug('Sending the request to deploy the configuration.')
        response = self._send_request(url, method='post', headers=headers)
```

You should use the */operational/deploy* path and HTTP POST method for requesting the configuration deployment.

Step 2: Add additional logic that verifies the state of the response. The state in the response should be *QUEUED*. If not, raise an exception, otherwise, periodically check if the configuration was already deployed. You should wait for a maximum time as defined with the *timeout* parameter.

```
import time
import datetime

    def deploy(self, timeout=180):
        <... output omitted ...>
        self.log.debug('Waiting for deploy job to finish.')
        state = response['state']
        if state == 'QUEUED':
            deploy_url = response['links']['self']
            current_time = datetime.datetime.now()
            end_time = current_time + datetime.timedelta(seconds=timeout)
            deployed = False
            while datetime.datetime.now() < end_time:
                self.log.debug('Checking the status of the deploy job.')
                response = self._send_request(deploy_url, headers=headers)
                state = response['state']
                self.log.debug(f'The state of the deploy job is {state}.')
                if state == 'DEPLOYED':
                    deployed = True
                    break
                time.sleep(5)
            if not deployed:
                raise Exception('Error while deploying the configuration.')
        else:
            raise Exception('Error occured when requesting the configuration deployment.')
```

The code first checks the state in the response. If the state is not *QUEUED*, then code raises an exception. If the state is *QUEUED*, then the code periodically checks for the status of the deploy job. You can get the URL to check the status from the response that you have received in the first request. The URL is returned under *links/self* in the dictionary. The code performs the verification in the *while* loop. In each loop, the condition is checked whether timeout was reached. When the deployment job is finished, the state should be *DEPLOYED*.

Step 3: Add the *deploy* method to the *ConfigSync* class. The method should call *deploy* method in *FDMClient* and finally *logout* method in *FDMClient* to revoke the token.

```
    def deploy(self):
        self.log.info('Starting with the configuration deployment.')
        self.fdm.deploy()
        self.log.info('Configuration deployment successful.')
        self.log.info('Logging out from FDM.')
        self.fdm.logout()
```

Step 4: Add the *deploy* method call to the script. Add the call after the *sync* method call.

```
if __name__ == "__main__":
    args = parse_arguments()

    if args.debug:
        log = init_logger(logging.DEBUG)
    else:
        log = init_logger()

    cs = ConfigSync(config=args.config, log=log)
    cs.get_config()
    cs.sync()
    cs.deploy()
```

You are now ready to test your final code.

Task 8: Test the Code

In this task, you will perform some tests to verify that code is working as expected.

Activity

Step 1: Before executing the tests, take a moment and observe the final code and configuration file.

fdm.py

```
import requests
import time
import datetime


class FDMClient:

    def __init__(self, host, port=443, username='admin', password='1234QWer', log=None):
        self.host = host
        self.port = port
        self.username = username
        self.password = password
        self.log = log
        if not log:
            raise Exception('The logger should not be None.')

        self.token = None
        self.base_headers = {
            'Content-Type': 'application/json',
            'Accept': 'application/json',
        }
        self.base_url = f'https://{self.host}:{self.port}/api/fdm/v2'

        requests.packages.urllib3.disable_warnings()

        self.log.debug('FDMClient class initialization finished.')

    def _send_request(self, url, method='get', headers=None, body=None, params=None):
        self.log.debug('Sending request to FDM')
        request_method = getattr(requests, method)
        if not headers:
            headers = self.base_headers

        self.log.debug(f'Using URL: {url}')
        self.log.debug(f'Using method: {method}')
        self.log.debug(f'Using headers: {str(headers)}')
        self.log.debug(f'Using body: {str(body)}')
        self.log.debug(f'Using query strings: {str(params)}')

        response = request_method(url, verify=False, headers=headers, json=body, params=params)
        status_code = response.status_code
        response_body = response.json()
        self.log.debug(f'Got status code: {str(status_code)}')
        self.log.debug(f'Got response body: {str(response_body)}')
        if status_code != 200:
            msg = response_body.get('message', 'Request to FDM unsuccessful.')
        return response_body

    def login(self):
        self.log.debug('Login to FDM.')

        url = self.base_url + '/fdm/token'

        body = {
            'grant_type': 'password',
            'username': f'{self.username}',
            'password': f'{self.password}',
        }

        self.log.debug('Senfing the login request to FDM.')
        response = self._send_request(url, method='post', body=body)
        self.token = response.get('access_token')
        self.log.debug(f'Access token: {self.token}')

    def logout(self):
        self.log.debug('Logout from FDM.')
        url = self.base_url + '/fdm/token'
        body = {
            'grant_type': 'revoke_token',
            'access_token': self.token,
            'token_to_revoke': self.token,
        }

        self.log.debug('Sending the logout request to FDM.')
        self._send_request(url, method='post', body=body)
        self.log.debug('Logout successful.')

    def _get_auth_headers(self):
        headers = self.base_headers.copy()
        if self.token:
            headers['Authorization'] = f'Bearer {self.token}'
        else:
            msg = 'No token exists, use login method to get the token.'
            raise Exception(msg)
        return headers

    def get_access_policy_id(self):
        url = self.base_url + '/policy/accesspolicies'
        headers = self._get_auth_headers()

        self.log.debug('Requesting access policies from FDM.')
        response = self._send_request(url, headers=headers)
        policy_id = response['items'][0]['id']
        self.log.debug(f'Policy ID is: {policy_id}')
        return policy_id

    def get_access_rule_by_name(self, name):
        self.log.debug('Searching for access rule.')
        policy_id = self.get_access_policy_id()

        url = self.base_url + f'/policy/accesspolicies/{policy_id}/accessrules'
        headers = self._get_auth_headers()

        self.log.debug('Requesting access rules from FDM.')
        response = self._send_request(url, headers=headers)
        access_rules = response.get('items')

        rule_data = None
        for rule in access_rules:
            if name == rule.get('name'):
                rule_data = rule
                break
        if rule_data is None:
            raise Exception('Unable to find requested rule.')
        return rule_data

    def get_url_categories(self):
        self.log.debug('Searching for URL categories on FDM.')
        url = self.base_url + '/object/urlcategories'
        headers = self._get_auth_headers()
        params = { 'limit': '100' }

        self.log.debug('Sending request for getting URL categories from FDM.')
        response = self._send_request(url, headers=headers, params=params)
        return response.get('items')

    def put_access_rule(self, data):
        self.log.debug('Updating access rule on FDM.')
        url = data['links']['self']
        headers = self._get_auth_headers()

        self.log.debug('Sending the request to update the access rule on FDM.')
        response = self._send_request(url, method='put', headers=headers, body=data)
        return response

    def deploy(self, timeout=180):
        self.log.debug('Deploying the configuration.')

        url = self.base_url + '/operational/deploy'
        headers = self._get_auth_headers()

        self.log.debug('Sending the request to deploy the configuration.')
        response = self._send_request(url, method='post', headers=headers)

        self.log.debug('Waiting for deploy job to finish.')
        state = response['state']
        if state == 'QUEUED':
            deploy_url = response['links']['self']
            current_time = datetime.datetime.now()
            end_time = current_time + datetime.timedelta(seconds=timeout)
            deployed = False
            while datetime.datetime.now() < end_time:
                self.log.debug('Checking the status of the deploy job.')
                response = self._send_request(deploy_url, headers=headers)
                state = response['state']
                self.log.debug(f'The state of the deploy job is {state}.')
                if state == 'DEPLOYED':
                    deployed = True
                    break
                time.sleep(5)
            if not deployed:
                raise Exception('Error while deploying the configuration.')
        else:
            raise Exception('Error occured when requesting the '
                            'configuration deployment.')
```

config_sync.py

```
import argparse
import logging
import yaml
from fdm import FDMClient


def parse_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument('--config', '-c', help='Path to the configuration file', default='fdm.cfg')
    parser.add_argument('--debug', '-d', help="Display debug logs.", action='store_true')
    return parser.parse_args()

def init_logger(log_level=logging.INFO):
    log = logging.getLogger(__file__)
    log.setLevel(log_level)
    console_handler = logging.StreamHandler()
    formatter = logging.Formatter('%(asctime)s %(levelname)s: %(message)s')
    console_handler.setFormatter(formatter)
    log.addHandler(console_handler)
    return log

class ConfigSync:

    def __init__(self, config, log):
        self.log = log
        self.config_file = config
        self.log.info('Initializing ConfigSync class.')
        self.config = self._parse_config(config)
        self.fdm = self._init_fdm_client(self.config)
        self.log.debug('ConfigSync class initialization finished.')

    def _parse_config(self, config_file):
        self.log.info('Parsing the configuration file.')
        with open(config_file, 'r') as f:
            config = yaml.safe_load(f)
        self.log.debug(f'The following parameters were received: {config}')
        return config

    def _init_fdm_client(self, config):
        self.log.info('Initializing FDMClient class.')
        host = config.get('fdm_host')
        username = config.get('fdm_username')
        password = config.get('fdm_password')
        fdm = FDMClient(host, username=username, password=password, log=self.log)
        self.log.info('Login to FDM.')
        fdm.login()
        return fdm

    def get_config(self):
        access_rule_name = self.config['url_filtering']['rule_name']
        self.log.info('Requesting access rule for URL filtering from FDM.')
        self.access_rule = self.fdm.get_access_rule_by_name(access_rule_name)

    def sync(self):
        self.log.info('Starting the configuration synchronization.')
        self.log.info('Requesting URL categories from FDM.')
        self.url_categories = self.fdm.get_url_categories()
        self.access_rule['urlFilter']['urlCategories'] = []
        self.log.info('Updating the access rule.')
        for category in self.config['url_filtering']['url_categories']:
            cat_dict = self._get_url_category(category)
            if cat_dict:
                self.access_rule['urlFilter']['urlCategories'].append(cat_dict)
        self.log.info('Adding the configuration to FDM.')
        self.fdm.put_access_rule(self.access_rule)

    def _get_url_category(self, name):
        category_dict = None
        for category in self.url_categories:
            category_name = category['name']
            if category_name == name:
                category_dict = {
                    'urlCategory': {
                        'name': category_name,
                        'id': category['id'],
                        'type': category['type'],
                    },
                    'type': 'urlcategorymatcher'
                }
                break
        return category_dict

    def deploy(self):
        self.log.info('Starting with the configuration deployment.')
        self.fdm.deploy()
        self.log.info('Configuration deployment successful.')
        self.log.info('Logging out from FDM.')
        self.fdm.logout()


if __name__ == '__main__':
    args = parse_arguments()

    if args.debug:
        log = init_logger(logging.DEBUG)
    else:
        log = init_logger()

    cs = ConfigSync(config=args.config, log=log)
    cs.get_config()
    cs.sync()
    cs.deploy()
```

fdm.cfg

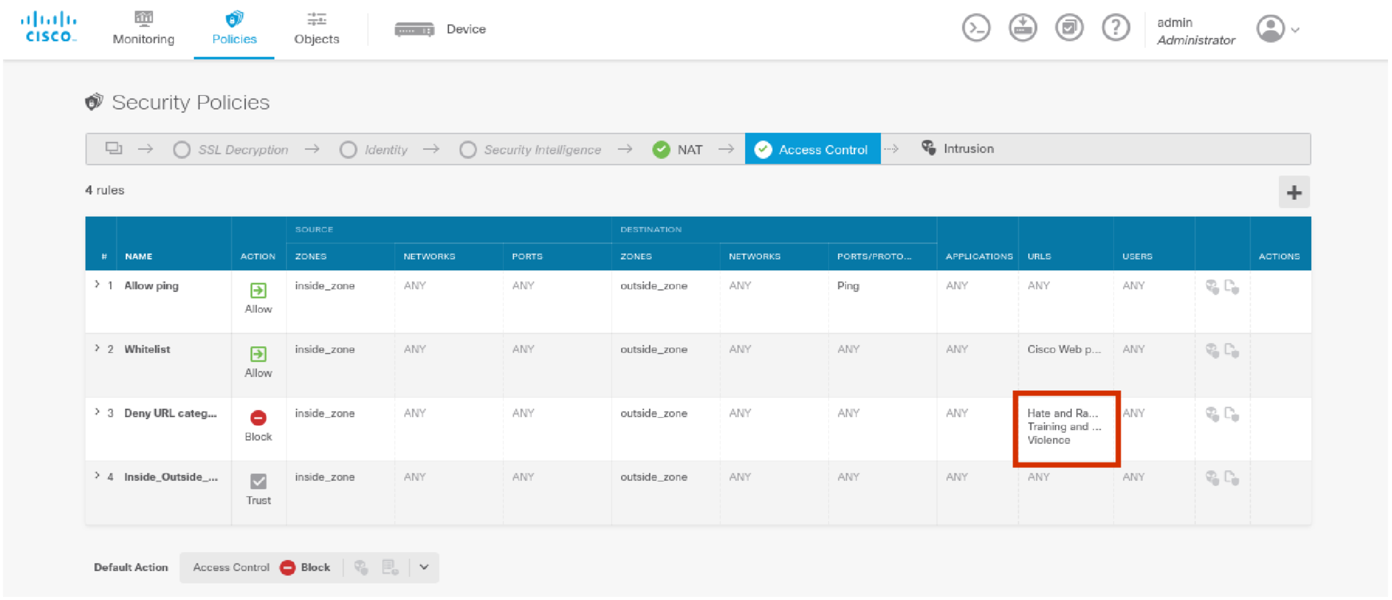
```
---
fdm_host: '192.168.0.40'
fdm_username: 'admin'
fdm_password: '1234QWer'
url_filtering:
  rule_name: 'Deny URL categories'
  url_categories:
    - 'Hate and Racism'
    - 'Violence'
    - 'Training and Tools'
```

Step 2: Run the **config_sync.py** script.

```
(working_directory) student@student-workstation:~/working_directory$ python config_sync.py
2019-10-18 11:12:57,817 INFO: Initializing ConfigSync class.
2019-10-18 11:12:57,817 INFO: Parsing the configuration file.
2019-10-18 11:12:57,819 INFO: Initializing FDMClient class.
2019-10-18 11:12:57,819 INFO: Login to FDM.
2019-10-18 11:12:58,057 INFO: Requesting access rule for URL filtering from FDM.
2019-10-18 11:12:58,204 INFO: Starting the configuration synchronization.
2019-10-18 11:12:58,204 INFO: Requesting URL categories from FDM.
2019-10-18 11:12:58,266 INFO: Updating the access rule.
2019-10-18 11:12:58,266 INFO: Adding the configuration to FDM.
2019-10-18 11:12:58,461 INFO: Starting with the configuration deployment.
2019-10-18 11:14:10,254 INFO: Configuration deployment successful.
2019-10-18 11:14:10,255 INFO: Logging out from FDM.
```

You should see that deployment was successful.

Step 3: Log in to FDM via the web browser and navigate to the *Policies* tab. Observe the *Deny URL categories* rule.



4 rules

#	NAME	ACTION	SOURCE ZONES	SOURCE NETWORKS	SOURCE PORTS	DESTINATION ZONES	DESTINATION NETWORKS	DESTINATION PORTS/PROTO...	APPLICATIONS	URLS	USERS	ACTIONS
1	Allow ping	Allow	inside_zone	ANY	ANY	outside_zone	ANY	Ping	ANY	ANY	ANY	
2	Whitelist	Allow	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	Cisco Web p...	ANY	
3	Deny URL categ...	Block	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	Hate and Ra... Training and ... Violence	ANY	
4	Inside_Outside_...	Trust	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	ANY	ANY	

Default Action Access Control Block

You should see that three required categories appeared in the rule, which means that configuration was deployed correctly.

Step 4: Change the **fdm.cfg** configuration file. Remove the '*Training and Tool*' URL category and add two additional URL categories: '*Social Network*' and '*Web based email*'.

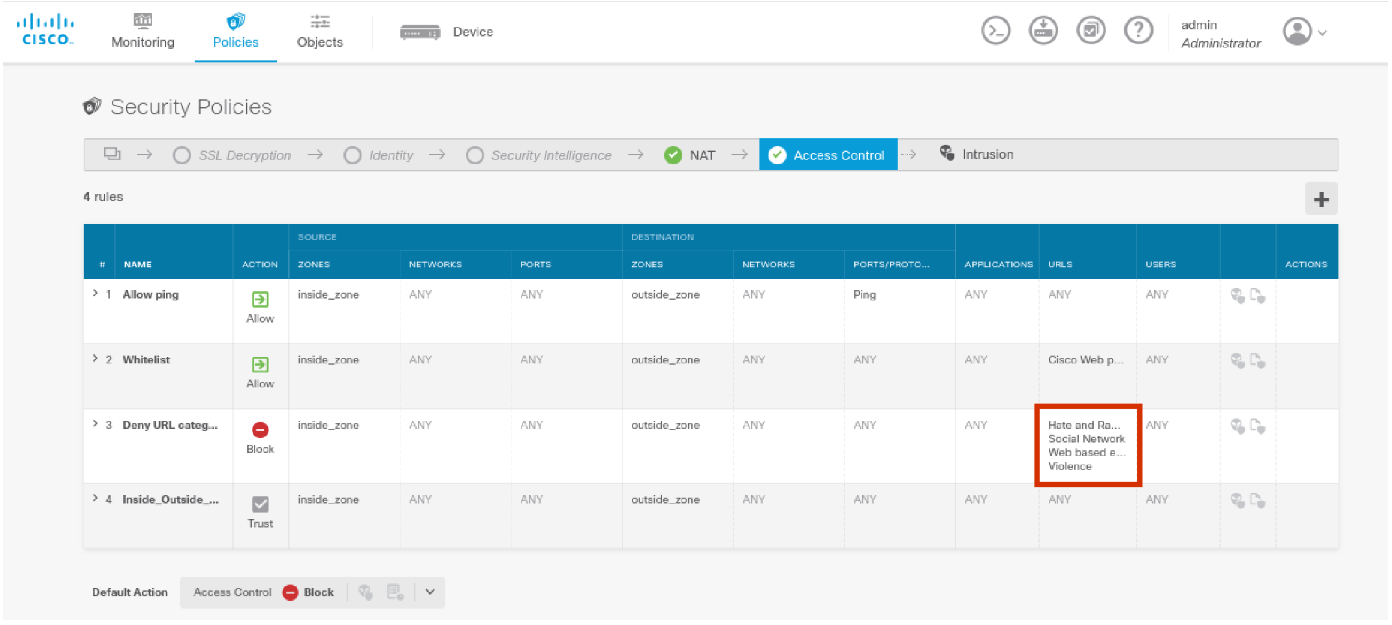
```
---
fdm_host: '192.168.0.40'
fdm_username: 'admin'
fdm_password: '1234QWer'
url_filtering:
  rule_name: 'Deny URL categories'
  url_categories:
    - 'Hate and Racism'
    - 'Violence'
    - 'Social Network'
    - 'Web based email'
```

Step 5: Run your **config_sync.py** script again.

```
(working-directory) student@student-workstation:~/working-directory$ python config_sync.py
2019-10-18 11:23:15,112 INFO: Initializing ConfigSync class.
2019-10-18 11:23:15,112 INFO: Parsing the configuration file.
2019-10-18 11:23:15,113 INFO: Initializing FDMClient class.
2019-10-18 11:23:15,113 INFO: Login to FDM.
2019-10-18 11:23:15,402 INFO: Requesting access rule for URL filtering from FDM.
2019-10-18 11:23:15,550 INFO: Starting the configuration synchronization.
2019-10-18 11:23:15,550 INFO: Requesting URL categories from FDM.
2019-10-18 11:23:15,602 INFO: Updating the access rule.
2019-10-18 11:23:15,602 INFO: Adding the configuration to FDM.
2019-10-18 11:23:15,859 INFO: Starting with the configuration deployment.
2019-10-18 11:24:12,242 INFO: Configuration deployment successful.
2019-10-18 11:24:12,242 INFO: Logging out from FDM.
```

From the script logs, you should see that the configuration was successfully deployed.

Step 6: Refresh the *Policies* tab in FDM web user interface. Observe the '*Deny URL categories*' rule.



4 rules

		SOURCE			DESTINATION						
# NAME	ACTION	ZONES	NETWORKS	PORTS	ZONES	NETWORKS	PORTS/PROTO...	APPLICATIONS	URLS	USERS	ACTIONS
1 Allow ping	Allow	inside_zone	ANY	ANY	outside_zone	ANY	Ping	ANY	ANY	ANY	
2 Whitelist	Allow	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	Cisco Web p...	ANY	
3 Deny URL categ...	Block	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	Hate and Ra... Social Network Web based e... Violence	ANY	
4 Inside_Outside_...	Trust	inside_zone	ANY	ANY	outside_zone	ANY	ANY	ANY	ANY	ANY	

Default Action Access Control Block

You should see that there are now four URL categories in the configuration, as specified in the configuration file. These categories confirm that your code is working as expected.

You can now use this code for deploying the configuration without the need to manually access the FDM web user interface. Of course, for a production environment, this code would probably need some robustness, you will, for example, need to implement more checks.

© 2021 Cisco Systems, Inc.