Hogeschool van Amsterdam
Media, Creatie en Informatie

TECHNICAL DOCUMENTATION
PROJECT NETWORK MANAGEMENT SYSTEM

20-01-2014
Versie 1.0

Remy Bien
Sebastiaan Groot
Wouter Miltenburg
Koen Veelenturf

Table of Contents

1. nms – yet another Network Management System

1.1 Introduction

nms is the namespace for the modules that make up the yaNMS web-application. The list of modules under the nms namespace can be found in table 1.1.

Name	Description
nms.admin	Registration for the built-in Django admin module
nms.commands	Task execution and connection management
nms.middleware	yaNMS-specific middleware hooks
nms.models	Database table definitions
nms.passwordstore	Querying of device passwords
nms.sshconnection	SSH2 connection routines
nms.telnetconnection	Telnet connection routines
nms.urls	URL to view mapping
nms.views	Views which correspond to pages in the application
nms.xmlparser	XML and regex parsing routines

Table 1.1: nms modules

Apart from python modules, the application also uses a small C library shmlib, which can be found under …/nms/c/shmlib.c

1.2. nms.admin - Registration for the built-in Django admin module

1.2.1. Introduction

The admin module is used for registration for the build-in Django admin module. This can be used as an alternative method for manipulating the database.

The nms.admin module contains no variables, functions, or classes, but is used internally by Django to register nms.models database tables to the built-in admin module.

1.2.2. Examples

When a new django.db.models.Model class is defined in nms.models with name 'X', the following line should be added to nms/admin.py:

```
admin.site.register(X)
```

1.3. nms.commands – Task execution and connection management

1.3.1. Introduction

The nms.commands module provides abstractions for sending commands and handling connections to devices.

1.3.2. Reference

interfaces

A dictionary used internally to cache queried interface names from devices.

interfacesLock

A multiprocessing.Lock lock to allow only one thread to access the commands.interfaces dictionary at a time.

connections

A dictionary used internally to keep track of open interactive sessions.

connectionsLock

A multiprocessing.Lock lock to allow only one thread to access the commands.connections dictionary at a time.

removeInterfaces(device)

Removes cached interfaces for the supplied nms.models.Device object.

getInterfaces(command, parser, device, user)

Obtains a list of all interfaces available on the device. The command argument is a list of strings, each containing a cli command. The parser is an nms.xmlparser.RegexWrapper object. The device is an nms.models.Device object. The user is obtained from django.http.HttpRequest.user, where a django.http.HttpRequest object is passed to each request.

This method returns a list of strings, each representing the name of an interface available on the device.

executeTask(taskpath, device, uargs, user)

Executes one or more commands on a device and returns parsed cli output. Taskpath is a string containing the keys needed to traverse the task dictionary to the specified task. This taskpath is generated as part of the nms.xmlparser.getAvailableTaskHtml function, and is passed as a GET argument in the nms.views.send_command view. Device is a nms.models.Device object. Uargs is a list of strings containing user arguments to use in the command. User is obtained from django.http.HttpRequest.user, where a django.http.HttpRequest object is passed to each request.

getConnection(user, device)

Returns a connection object (either an sshconnection.SSHConnection or telnetconnection.TelnetConnection object) to the device for the user. If there already exists an open connection to this device for this user, the open connection is returned. User is obtained from django.http.HttpRequest.user, where a django.http.HttpRequest object is passed to each request. Device is a nms.models.Device object.

removeConnection(user, device)

Closes an open connection for this user – device combination. The user is obtained from django.http.HttpRequest.user, where a django.http.HttpRequest object is passed to each request. Device is a nms.models.Device object.

1.3.3. Examples

Obtaining a list of available interfaces on a device:

```
from nms import commands
from nms import xmlparser
from nms.models import Devices

device = Devices.objects.get(pk=1) #Obtain device 1
xml_root = xmlparser.get_xml_struct(device.gen_dev.file_location.location)
command, parser = xmlparser.getInterfaceQuery(xml_root)
interfaceList = commands.getInterfaces(command, parser, device, request.user)
```

Executing a task using the taskpath and uargs obtained from a request:

```
from nms import commands
from nms import xmlparser
from nms.models import Devices

def my_view(request):
      taskpath = request.GET['taskpath']
      uargs = request.GET['uargs']
      device = Devices.objects.get(pk=int(request.GET['dev_id']))
      messages.info(request, commands.executeTask(taskpath, device,
            uargs, request.user)
      return HttpResponseRedirect(reverse('nms:devices'))
```

1.4. nms.middleware - yaNMS-specific middleware hooks

1.4.1. Introduction

The nms.middleware module contains middleware classes that are used to process each request. In order to use a class defined here, it must be added in the MIDDLEWARE_CLASSES list in settings.py.

1.4.2. Reference

1.4.2.1 class nms.middleware.SessionLogout

The SessionLogout class logs users out if their session timeout has expired. This value must be defined in settings.py as SESSION_EXPIRATION_TIME, which is an integer representing the timeout in seconds.

process_request(request)

Request is a django.http.HttpRequest object, supplied with each incoming request.

1.5. nms.models - Database table definitions

1.5.1. Introduction

The nms.models module defines classes inheriting from django.db.Model. Django uses these definitions to create database tables and allows querying and manipulating these tables.

1.5.2. Reference

app_label

Indicates into which namespace the models in the models.py file fall.

1.5.2.1. class nms.models.Vendor

vendor_id
The primary key for this model.

vendor_name
A CharField for the name of the vendor.

1.5.2.2. class nms.models.OS_type

os_type_id
The primary key for this model.

type
A CharField for the name of this OS type.

1.5.2.3. class nms.models.OS

os_id
The primary key for this model.

vendor_id
A foreign key to nms.models.Vendor with delete protection.

os_type_id
A foreign key to nms.models.OS_type with delete protection.

build
A CharField for the version code of the OS.

short_info
A CharField for a short description for the OS.

name
A CharField for the major name of the OS.

1.5.2.4. class nms.models.Dev_model

model_id
The primary key for this model.

model_name
A CharField for the name of the device model.

version
A CharField for the version code of the device model.

1.5.2.5. class nms.models.Dev_type

dev_type_id
The primary key for this model.

dev_type_name
A CharField for the name of this device type.

1.5.2.6. class nms.models.File_location

file_location_id
The primary key for this model.

location
A CharField for the path to the file.

1.5.2.7. class nms.models.Gen_dev

gen_dev_id
The primary key for this model.

vendor_id
A foreign key to nms.models.Vendor with delete protection.

model_id
A foreign key to nms.models.Dev_model with delete protection.

dev_type_id
A foreign key to nms.models.Dev_type with delete protection.

file_location_id
A foreign key to nms.models.File_location with delete protection.

1.5.2.8. class nms.models.OS_dev

os_dev_id
The primary key for this model.

os_id
A foreign key to nms.models.OS with delete protection.

gen_dev_id
A foreign key to nms.models.Gen_dev with delete protection.

1.5.2.9. class nms.models.Devices

dev_id
The primary key for this model.

gen_dev_id
A foreign key to nms.models.Gen_dev with delete protection.

os_dev_id
A foreign key to nms.models.OS_dev with delete protection.

ip
A django.db.models.GenericIPAddressField.

ip_version
A positive integer field indicating the version of the IP address.

port
A positive integer field defaulting to 22.

login_name
A CharField for the username used to log into the device.

password_remote
A CharField for the cli login password. It starts with a 16-byte salt, followed by a $ and AES encrypted password.

password_enable
A CharField for the password for privileged-mode login. It starts with a 16-byte salt, followed by a $ and AES encrypted password.

pref_remote_prot
A CharField for the preferred protocol. It defaults to ssh.

1.5.2.10. class nms.models.History

history_id
: The primary key for this model.

action
: A CharField for a description of the logged action.

action_type
: An optional CharField for the type of action performed.

dev_id
: An optional foreign key to nms.models.Devices if the action involved a device.

user_id
: An optional foreign key to django.contrib.auth.models.User for the user affected by the action.

user_performed_task
: An optional foreign key to django.contrib.auth.models.User for the user who performed the action.

date_time
: An optional django.db.models.DateTimeField for the time when the action was performed.

group_id
: An optional foreign key to django.contrib.auth.models.Group for the group affected by the action.

1.5.2.11. class nms.models.Settings

settings_id
: The primary key for this model.

known_id
: A positive integer field giving this setting a static id.

known_name
: A CharField for the name of this setting.

known_boolean
: An optional boolean for this setting.

string
: An optional CharField for this setting.

1.5.2.12. class nms.models.Dev_group

dev_group_id
: The primary key for this model.

gid
: A foreign key to django.contrib.auth.models.Group with delete protection.

devid
: A foreign key to django.contrib.auth.models.User with delete protection.

1.6. nms.passwordstore - Querying of device passwords

1.6.1. Introduction

Often, the application needs to communicate with devices for various reasons, such as querying for available interfaces, sending commands and keeping open interactive sessions. In order to prevent the user from supplying credentials each time a connection is made, login and privileged-mode passwords are stored in the database, encrypted with AES encryption.

1.6.2. Reference

lib

A ctypes.CDLL handle to the shmlib.so shared-object file, containing C routines for shared-memory access.

storeMasterPassword(password)

Writes the master password to shared memory, to share it between processes in a multi-process webserver. Password is a string of 16 characters and will constitute half of the AES key. The function returns -1 if the password length wasn't 16 and 0 otherwise.

hasMasterPassword()

Returns True if lib.get_password() returns a 16-character string and False otherwise.

storeEnablePassword(device, password)

Generates a salt for this password, and stores the salt and AES-encrypted password to device.password_enable. Device is a nms.models.Devices object. Password is a string.

storeRemotePassword(device, password)

Generates a salt for this password, and stores the salt and AES-encrypted password to device.password_remote. Device is a nms.models.Devices object. Password is a string.

getEnablePassword(device)

Decrypts and returns the password stored in device.password_enable. Device is a nms.models.Devices object.

getRemotePassword(device)

Decrypts and returns the password stored in device.password_remote. Device is a nms.models.Devices object.

1.6.2.1. class nms.passwordstore.AESCipher(key)

Key is a 32-byte bytestring used in AES encryption and decryption.

encrypt(raw)

Encrypts the bytestring raw and returns base64-encoded, AES-encrypted data.

decrypt(enc)

Decrypts the base64-encoded, AES-encrypted bytestring enc.

1.7. nms.sshconnection - SSH2 connection routines

1.7.1. Introduction

In order to allow different kinds of connection methods make use of the same functions, all connection protocol implementations must supply the same set of functions. The duck-typing system of python then allows to treat multiple connection classes the same way. This module implements an SSHConnection class, creating an interface for paramiko's SSH functions.

1.7.2. Reference

1.7.2.1. class nms.sshconnection.SSHConnection(hostname, username, password, port=22)

Hostname is the address of the endpoint to connect to. Username and password are both strings used for authentication. Port is the network port to connect to.

connect()
Opens the SSH connection.
send_and_receive(command, delay = 0)
Sends a command, waits for the specified delay and receives data. Command is a single string to send over the connection.
receive()
Calls recv(4096) on the connection and returns its return value.
send(text)
Sends the bytestring text over the connection.
close()
Closes the SSH connection.

1.8. nms.telnetconnection - Telnet connection routines

1.8.1. Introduction

In order to allow different kinds of connection methods, make use of the same functions, all connection protocol implementations must supply the same set of functions. The duck-typing system of Python then allows to treat multiple connection classes the same way. This module implements a TelnetConnection class, creating an interface for the standard Python library telnetlib.

1.8.2. Reference

1.8.2.1. class nms.telnetconnection.TelnetConnection(hostname, username, password, port=23)

Hostname is the address of the endpoint to connect to. Username and password are both strings used for authentication. Port is the network port to connect to.
connect()
Opens the Telnet connection.
send_and_receive(command, delay = 0)
Sends a command, waits for the specified delay and receives data. Command is a single string to send over the connection.
receive()
Calls read_some() on the connection and returns its return value.
send(text)
Sends the bytestring text over the connection.
close()
Closes the Telnet connection.

1.9. nms.urls – URL path to view mapping

1.9.1. Introduction

When Django receives a connection, it uses the regex rules in the urls.py files to match the request to a certain views.py method. When a new view gets defined, it can be assigned to a path in this module.

1.9.2. Examples

When new view "def myview(request)" is defined in views.py, it can be matched to a path by adding the following line in urls.py as new argument in the patterns(…) call:

```
url(r'^a/new/path$', views.myview, name='myview'),
```

1.10. nms.xmlparser – XML structure handling functies

1.10.1. Introduction

The XML files defining how to interact with devices contain multiple data structures that are used in multiple different ways within the application. The nms.xmlparser module contains a number of functions to transform the XML structure into different formats.

1.10.2. Reference

cacheLock
: A multiprocessing.Lock lock to allow only one thread to access the xmlparser.cache dictionary at a time.

cache
: A dictionary used as a cache for XML files.

taskcacheLock
: A multiprocessing.Lock lock to allow only one thread to access the xmlparser.taskcache dictionary at a time.

taskcache
: A dictionary used as a cache for the collections.OrderedDict objects that hold the parsed XML information including regex parsers and lists of commands.

removeXmlStruct(device)
: Removes an XML structure from xmlparser.cache given the file_location for this device.

get_xml_struct(filepath)
: Obtains the root of an XML structure of a filepath from xmlparser.cache if available. Otherwise, it adds the XML structure to the cache first.

getDeviceInfo(root)
: Obtains information from the deviceInfo tag in the XML structure.

getSupportedOperatingSystems(root)
: Obtains information from the supportedOperatingSystems tag in the XML structure.

getInterfaceQuery(root)
: Obtains a list of commands and xmlparser.RegexWrapper object from the information under the interfaceQuery tag in the XML structure.

removeTaskCache(root)
: Removes the task OrderedDict from the xmlparser.taskcache dictionary.

getAvailableTasks(root, interfaces = [], privPassword = '')
: Returns the task OrderedDict from the taskcache, or builds it from scratch given the XML structure, list of interface names and privileged-mode password.

getAvailableTasksHtml(root, id, interfaces = [], privPassword = '')
: Given the root of an XML structure, device id, list of interface names and privileged-mode password, returns the HTML used in the manage device webpage.

1.11. shmlib.c

1.11.1. Introduction

In order to avoid storing the master password for device credential encryption on disk, an application administrator is required to enter the master password once when the application restarts. This password is never stored on disk, but gets stored in shared

memory in order to share it between different webserver threads and processes. The shmlib.so library provides shared-memory functions for the yaNMS application.

1.11.2. Reference

void store_password(char *password, int n)

Given n characters at password, copy the characters to the shared memory segment.

char *get_password(void)

If available, returns the char pointer to the shared memory segment containing the password.

2. Adding protocol support

Adding new modules to support different types of communication protocols in yaNMS is done by writing a connection class, and adding it to the nms.commands module.

2.1. Writing a connection class

A connection class always has the same structure. The following shows one such structure implementing the fictional USSH protocol:

```
import usshlib
import time

class USSHConnection:
	def __init__(self, hostname, username, password, port = 42):
		self.hostname = hostname
		self.username = username
		self.password = password
		self.port = port
		self.ussh = usshlib.USSH()

	def connect(self):
		try:
			self.ussh.connect(self.hostname, self.port)
			self.ussh.auth(self.username, self.password)
		except:
			return -1
		return 0

	def send_and_receive(self, command, delay=0):
		if type(command) != type(bytes()):
			try:
				command = command.encode()
			except AttributeError:
				return
		self.ussh.send(command)
		time.sleep(delay)
		return self.ussh.recv()

	def receive(self):
		return self.ussh.recv()

	def send(self, text):
		return self.ussh.send(text)

	def close(self):
		self.ussh.close()
```

2.2. Adding the connection class to nms.commands

Four changes need to be made to the nms.commands file to add your implementation to the application.

1. Import the module containing the new class
2. In nms.commands.executeTask, add an elif clause for the new implementation.
3. In nms.commands.getConnection, add an elif clause for the new implementation.
4. Add an __create[name]Connection__(device) function to create and return a non-blocking connection.

```
from nms import mynewconnection

…

def executeTask(taskpath, device, uargs, user):
      …
      if device.pref_remote_prot == 'SSH2':
            …
      elif device.pref_remote_prot == 'MYNEWPROT':
            s = mynewconnection.MyNewConnection(device.ip, device.login_name,
                  passwordstore.getRemotePassword(device), device.port)
…

def getConnection(user, device):
      …
      if device.pref_remote_prot == 'SSH2':
            …
      elif device.pref_remote_prot == 'MYNEWPROT':
            connections[user][device] = __createMyNewConnection__(device)
…

def __createMyNewConnection__(device):
      connection = mynewconnection.MyNewConnection(device.ip,
            device.login_name, passwordstore.getRemotePassword(device),
            device.port)
      connection.connect()
      connection.socket.setblocking(0)
      return connection
```

3. Supporting devices

Support for devices is defined in XML files that describes how to perform a number of tasks on a device. This chapter describes the different tags present in the XML structure.

3.1. XML Tags

3.1.1. Header

The XML file should start with a valid XML header, such as:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

3.1.2. device

The <device> tag contains all other tags for this device

3.1.3. deviceInfo

The <deviceInfo> tag contains attributes *type*, *vendor* and *model*. This tag is not used inside the application and is used to inform the reader for which device this XML file defines actions.
type: The type of the device, such as "router" or "switch"
vendor: The vendor of the device, such as "CISCO" of "Juniper"
model: The model name of the device, such as "C3660"

3.1.4. supportedOperatingSystems

The <supportedOperatingSystems> tag contains one or more <operatingSystem> tags. While the application contains functions to obtain these, they are currently not used and are only there for informative purposes.

3.1.5. operatingSystem

The <operatingSystem> tag contains the attribute *name*.
name: Name of the operating system, such as "IOS 12.4-15"

3.1.6. supportedProtocols

The <supportedProtocols> tag, like <supportedOperatingSystems>, exists for informative purposes. It contains one or more <protocol> tags.

3.1.7. protocol

The <protocol> tag contains the attribute *name*.
name: Name of the protocol, such as "SSH2"

3.1.8. interfaceQuery

The <interfaceQuery> tag contains a <command> tag and a <returnParsing> tag. The interfaceQuery tag is used to obtain a list of interfaces on the device.

3.1.9. command

The <command> tag contains one or more <argElement> tags, containing commands to send in sequence to obtain raw interface information.

3.1.10. argElement

The argElement tag contains the attributes *position* and *type* and a *text* portion.
position: The number of this element in the sequence of argElements.
type: argElements in the interfaceQuery tag can only be "plain-text", as it should execute without the use of user-supplied arguments.
text: The text portion contains the command to be sent.

3.1.11. returnParsing
The <returnParsing> tag contains the attributes *delimiter* and *type*, as well as a *text* portion.
delimiter: The (set of) character(s) to split the output text on. Python-defined special characters (such as '\n') can be used.
type: The type of parsing text in the *text* segment. Currently, only "regex" is supported.
text: The regex parsing rules used with the python *re* module.

3.1.12. configurationItems
The <configurationItems> tag contains one or more <category> or <item> tags and is the collection of tasks that can be performed with the device. Adding new tasks is as easy as adding <item> tags in this tag.

3.1.13. category
The <category> tag is used to define a set of other <category> or <item> tags. It contains the attribute *name*.
name: The name of the category (displayed on the user interface).

3.1.14. item
The <item> tag defines one task that can be performed with a sequence of commands. Like the <interfaceQuery> tag, it contains a <command> tag and a <returnParsing> tag. It has the attributes *name* and *type*.
name: The name of the task (displayed on the user interface) .
type: Either "single" for single tasks, or "per-interface" for tasks that should be copied for each interface available on the device.

3.1.15. command
The <command> tag inside an <item> tag contains one or more <argElement> tags and can has the optional attribute *userArgs*.
userArgs: A ':'-seperated string with the names of arguments that should be supplied by the user.

3.1.16. argElement
The <argElement> tag inside a <command> tag inside an <item> tag has the attributes *position* and *type* and usually contains a *text* segment (unless the *type* is "privpasswd") with different possible types of text formatting symbols.
position: The number of this element in the sequence of argElements.
type: The type of command. Either "plain-text", which takes the *text* segment as command, or "privpasswd", which takes the stored privileged-mode password as command.
text: The command to send to the device, with two possible types of text formatting symbols. %if% gets replaced for "per-interface" items with an interface name and %arg:<name>% with user-supplied arguments.

3.2. Example XML file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<device>
      <deviceInfo type=”router” vendor=”CISCO” model=”C3660” />
      <supportedOperatingSystems>
            <operatingSystem name=”IOS 12.4-15” />
      </supportedOperatingSystems>
      <supportedProtocols>
            <protocol name=”SSH2” />
      </supportedProtocols>

      <interfaceQuery>
            <command>
                  <argElement position=”0” type=”plaintext”>show ip
interface brief</argElement>
            </command>
            <returnParsing delimited=”\n” type=”regex”>(?=(^[0-9a-zA-
Z/.]+))(?=(?!Interface))(?=(?!^.*?>))</returnParsing>
      </interfaceQuery>

      <configurationItems>
            <category name=”Interface options”>
                  <item name=”Set IP configuration %if%” type=”per-
interface”>
                        <command userArgs=”ip:subnet_mask”>
                              <argElement position="0"
type="plaintext">enable</argElement>
                              <argElement position="1" type="privpasswd"
/>
                              <argElement position="2"
type="plaintext">configure terminal</argElement>
                              <argElement position="3"
type="plaintext">interface %if%</argElement>
                              <argElement position="4"
type="plaintext">ip address %arg:ip% %arg:subnet_mask%</argElement>
                              <argElement position="5"
type="plaintext">end</argElement>
                              <argElement position="6"
type="plaintext">disable</argElement>
                        </command>
                        <returnParsing delimiter="\n"
                  type="regex">(^.*?\(config-
                  [(sub)]*if\)#ip .*$)</returnParsing>
                  </item>
            </category>
      </configurationItems>
</device>
```

4. ACL

ACL is a system module of yaNMS and is integrated in most parts of the application. This chapter describes how ACL can be integrated into custom modules.

4.1. Database

There are multiple tables that are related to the ACL system module:

- auth_user
  - Contains user related information
- auth_group
  - Contains group information
- auth_group_permissions
  - Relations between groups and permissions are maintained here
- auth_user_groups
  - Relations between user and groups are maintained here
- auth_user_user_permissions
  - Relations between users and permissions are maintained here
- auth_permission
  - All permissions are defined here. Most important part is the **codename**, in a sense it is just a label for a specific permission.
- nms_dev_group
  - Relations between devices and groups are maintained here
- nms_devices
  - Contains device related information

For more information about the listed tables please read the Django documentation[1] and Chapter 1.10.2: *"Reference."*

4.2. Authentication

When one wants to add a view where user authentication is required, a decorator is sufficient. The following decorator checks if the user making the request is authenticated:

@login_required

If someone who is not logged in wants to access a page that requires authentication, it will simply redirect the user to the login page. But in the other case where a user is authenticated, the user is allowed to access this page.

[1] Django auth documentation: https://docs.djangoproject.com/en/1.5/topics/auth/

4.3. Authorization

This subchapter will describe how authorization is implemented in yaNMS and two methods are provided to implement authorization in new views.

4.3.1. Codename

Codenames, are in a sense, just labels representing a permission. In yaNMS there are different codenames for different permissions. The following codenames exist within a default yaNMS installation:

- User (auth):
  - auth.add_user
  - auth.change_user
  - auth.delete_user
  - auth.list_user
- Group(auth):
  - auth.add_group
  - auth.change_group
  - auth.delete_group
  - auth.list_group
- Devices(nms):
  - nms.add_devices
  - nms.change_devices
  - nms.delete_devices
  - nms.manage_devices
  - nms.list_devices
- Gen_dev(nms):
  - nms.add_gen_dev
  - nms.change_gen_dev
  - nms.delete_gen_dev

When Django detects a custom model it will automatically create three permissions for that model (add, change and delete) and will also create the codename for a specific permission. The codename will be named after the application name and the permission (i.e., nms.add_devices).

Codenames are used to refer to a specific permission and can therefore be used in yaNMS for authorization. Subsequent chapters will describe how authorization is done within a view and how authorization can be integrated in a custom view.

4.3.2. Simple authorization

When a custom view is created and when one wants to add authorization, it only needs an if statement for that view. For example, when one wants to check if a user has the rights 'nms.add_devices'[2] and 'nms.list_devices' it only needs to include this small if statement in the view:

```
if request.user.has_perm('nms.add_devices') and
request.user.has_perm('nms.list_devices'):
```

[2] In this chapter codenames are mostly used to check for permissions.

4.3.3. A bit more advanced authorization

If there is a custom view that needs to check if a user is already added to a device group and if it has devices, the following code can be used:

```
groups_list = [x for x in groups if x.permissions.filter(codename='list_devices').exists()]
dev_group = [x.dev_group_set.all() for x in groups_list][0]
devices = {x.devid for x in dev_group}
devices = list(devices)
if groups_list != [] and devices != []:
        return render(request, 'nms/devices_manage.html', {'devices': devices,
'request':request})
```

This code will first check if the user is added to a group, which has the 'nms.list_devices' right. If this is the case, it will be added to the 'group_list', which will be later used to form 'dev_group'. The list 'dev_group' will contain all device groups that have the 'nms.list_devices' permission. Because the 'Dev_group' table contains foreign keys containing devices, it is possible to list all those devices in the 'devices' set. The next step is to check if neither of the lists are empty and when this is the case, the user will be granted access to the management page of all the devices that the user is granted access to.