API Programming Exercises for PyCon UK 2014

Interoute Virtual Data Centre

Document webpage: `http://cloudstore.interoute.com/main/PyConUK`

Published: 2014-09-19

Table of Contents

Download the code and exercises

https://github.com/Interoute/python-rules-the-cloud

The program `vdc_api_call.py` provides the base class for performing API calls (building the URL string and calculating the credential signature for each call). That program does need to be present for any of the other programs to work. For simplicity, you can put all 'py' files in one folder, and run Python from that folder.

Get started with the API

Let's start by working interactively in Python. Type:

```
python -i vdc_starter.py
```

This program sets up everything required for API communication, then returns control to the Python interpreter (indicated by the '>>>'), ready for you to type something. Let's try the API command 'listZones':

```
>>>api.listZones({})
```

You should have got a response that starts:

```
{u'count': 7, u'zone': [{u'localstorageenabled': False, u'domain':
 u'guest.vdc', u'name': u'Paris (ESX)' ...
```

It's more readable if you pretty-print it:

```
>>>pprint(api.listZones({}))
```

So there are the essential parts of the API communication; the *command*, which will usually have *parameters* (the content of the '{}') which in this case is empty. And, if the commmunication is successful, you get back a *response*, which is a JSON-format object, which conveniently can be handled directly as a Python dictionary object.

It's worth trying to break the API call and see how an error looks:

```
>>>pprint(api.listzones({}))
```

Normally, you will want to store the response as a Python object, and then use keys to extract parts of the response:

```
>>> result=api.listZones({})
>>> result.keys()
[u'count', u'zone']
>>> result['count']
7
>>> result['zone'][1]
{u'localstorageenabled': False, u'domain': u'guest.vdc', u'name':
 u'Milan (ESX)', u'tags': [], u'zonetoken': u'700452f2-bd55-3a75-96ca-
ba0ead5ab438', u'securitygroupsenabled': False, u'allocationstate':
 u'Enabled', u'dhcpprovider': u'VirtualRouter', u'networktype':
 u'Advanced', u'id': u'58848a37-db49-4518-946a-88911db0ee2b'}
```

A very common thing you will want to do is to loop through all the contents of a response, and print or otherwise use only certain values. Here's a 'for' loop that will do that:

```
>>> for z in result['zone']:
...     print("Zone: %s, zoneid: %s" % (z['name'],z['id']))
...
Zone: Paris (ESX), zoneid: 374b937d-2051-4440-b02c-a314dd9cb27e
Zone: Milan (ESX), zoneid: 58848a37-db49-4518-946a-88911db0ee2b
Zone: Berlin (ESX), zoneid: fc129b38-d490-4cd9-acf8-838cf7eb168d
Zone: Amsterdam 2 (ESX), zoneid: 3c43b32b-fadf-4629-b8e9-61fb7a5b9bb8
Zone: London 2 (ESX), zoneid: f6b0d029-8e53-413b-99f3-e0a2a543ee1d
Zone: Slough (ESX), zoneid: 5343ddc2-919f-4d1b-a8e6-59f91d901f8e
Zone: Geneva 2 (ESX), zoneid: 1ef96ec0-9e51-4502-9a81-045bc37ecc0a
```

Of course, there are multiple other ways of iterating on dictionary contents in Python, which you might want to explore.

Those long, random-looking ID strings are called UUIDs ('universally unique identifier' - a very interesting concept, by the way, if you have not seen it before; VDC always uses 'version 5' UUIDs which are sequences of 32 hexadecimal digits grouped 8-4-4-4-12 with hyphens). You need to use them all the time to refer to the particular VDC resources that you want to operate with.

So far, we have not used parameters at all. For API calls, parameters play the role like inputs to functions. A common behaviour of VDC API commands that list different types of resources is that, if you put no parameter you will get a list of all resources of that type, or if you specify a resource by its UUID then you get a response only for that resource. For example, to get only the zone details for Paris, I would type:

```
>>> pprint(api.listZones({'id':'374b937d-2051-4440-b02c-a314dd9cb27e'}))
```

So parameters are always presented as key : value pairs, and separated by commas. The keys are always Python strings and must be quoted.

Before moving on, take a look at some of the other 'list' commands in the API by using the *API Command Reference* (**http://cloudstore.interoute.com/main/knowledge-centre/library/api-command-reference**) and try building API calls with different parameters. 'listVirtualMachines' and 'listNetworks' will be put to use later.

Deploying and configuring virtual machines

The VDC API has commands equivalent to all of the functions of the graphical user interface, and more. (Actually, this is not surprising because the GUI uses only API calls to communicate with the cloud servers; if you turn on the developer console in your browser you can inspect the API calls being used.)

API commands which can take a long time to run (for example, creating virtual machines and storage drives) are setup as 'asynchronous' (in the Command Reference you will see '(A)' next to the command name). This means that the command is passed into a separate 'job' by the VDC server, and the response to the command contains a 'jobid' which allows you to follow the progress of job completion. This is a useful feature because otherwise you would have to write multi-threaded Python programs to cope with long-running commands. The `vdc_api_call` class has a simple function to deal with asynchronous commands, which we will see in a moment.

A common task for the API is 'deploying' (creating) virtual machines. A minimum of three parameters need to be specified: the template (type of operating system), the service offering (CPU and RAM required for the VM), and the VDC zone where the VM should run.

Run the program `vm_deploy_info.py` to print out the three sets of parameter information that you need:

```
python vm_deploy_info.py
```

This simply runs three API calls and three loops to extract the information.

Now run the program `vm_deploy.py` and enter the required information to deploy a VM.

Notice this is an asynchronous command, so the program sets up a loop to query job status until job completion is signalled.

At this point you might like to find out about the commands to start ('startVirtualMachine'), stop ('stopVirtualMachine') and scale ('scaleVirtualMachine') a VM, and test these on your new VM(s).

Getting and displaying state information

When you are running a set of VMs in the cloud, a key piece of monitoring information is to know if the VMs are running (if they should be running) or have gone into a different state. The command 'listVirtualMachines' returns that information, but not in a very readable form.

The following program, `check-vm-state.py` takes the VM state information and presents it in a more accessible way using colour coding (green for running, red for stopped, and blue for anything else).

```
python check-vm-state.py
```

What other state information could be displayed in an enhanced form like this?

Dashboard widgets with Tkinter

A 'widget' (GUI object) is a good way to present state information because widgets can sit quietly in the corner of your screen and automatically update themselves with VDC state information. This sounds complicated but is actually easy to do with standard Python, because a GUI builder module called Tkinter is one of the standard modules. So an interesting project for Python and VDC is to create a suite of dashboard widgets that present information about the different aspects of VDC (VMs, networks, storage drives).

The following program, `widget-check-vm-state.py` merges together the command-line VM state program with the structure of the minimal 'Hello world' program for Tkinter. You should first try running the program `tkinter-helloworld.py` to see how that works.

The result is in no way pretty but surprising useful for a simple program.

To run a GUI program, which by design does not terminate unless you intentionally quit it, it's best to run the program 'in the background', by using the ampersand symbol:

```
python widget-check-vm-state.py &
```

Something you might want to try now is create more widgets which display other information, for example about zones, networks, or detailed information about only virtual machine. You could also dig into Tkinter and improve the aesthetics and GUI functionality of the widgets.