CHAPTER 3

Tutorials / How-Tos

3.1 Importing a Context from the GENI Portal

In order to communicate with any federation resource using `geni-lib` you need to construct a `Context` object that contains information about the framework you are using (for example ProtoGENI, Emulab, GENI Clearinghouse, etc.), as well as your user information (SSH keys, login username, federation urn, etc.). This simple tutorial will walk you through the easiest way to create a `Context` if you have an account at the GENI Portal.

3.1.1 Download The omni.bundle

First we need a file called `omni.bundle` which is available from the GENI Portal web interface. Once you log into the GENI Portal you can use the following steps to locate your `omni.bundle` download:

- At the top of the Portal home page click on the tab labeled **Profile**
- In the tabs on the Profile page click on the one labeled **Configure omni**
- Embedded in the text under the **Option 1: Automatic omni configuration** header, there is a button labeled **Download your omni data**. Click this button.

Note: If you see a warning that no SSH keys have been uploaded you can still use the bundle, but you will need to specify an SSH public key path in a later step.

- Click the **Download your omni data** button at the bottom of the next page and it should start downloading immediately in your browser.

3.1.2 Run Context Import Tool

A script called `context-from-bundle` was installed as part of your `geni-lib` installation, which can convert your `omni.bundle` into the data necessary for `geni-lib` to create a `Context` object for you. The instructions for using this tool are below - choose the section appropriate for your OS.

MacOS X / Linux

In most installations your path should already include the import tool and it should run cleanly without any additional configuration:

```
$ context-from-bundle --bundle /path/to/omni.bundle
```

If no arguments are supplied the bundle is assumed to be in the current directory. If your bundle does not contain an SSH public key you will be required to supply a path to one using the `--pubkey` argument at the command line.

Windows

Unfortunately the default Python installation on Windows does not add the site Scripts directory to your path, so you need to invoke it directly. If you are using Python 2.8 you will need to replace `Python27` with `Python28` below:

```
C:\> python C:\Python27\Scripts\context-from-bundle --bundle path\to\omni.bundle
```

If no arguments are supplied the bundle is assumed to be in the current directory. If your bundle does not contain an SSH public key you will be required to supply a path to one using the `--pubkey` argument at the command line.

3.1.3 Test It Out!

Now we can take your newly imported information, instantiate our context, and query an aggregate:

```
$ python
>>> import geni.util
>>> context = geni.util.loadContext()
>>> import geni.aggregate.instageni as IG
>>> import pprint
>>> pprint.pprint(IG.GPO.getversion(context))
{'code': {'am_code': 0,
          'am_type': 'protogeni',
          'geni_code': 0,
          'protogeni_error_log': 'urn:publicid:IDN+instageni.gpolab.bbn.com+log+abedbcc20e6defe716eb8
          'protogeni_error_url': 'https://boss.instageni.gpolab.bbn.com/spewlogfile.php3?logfile=abed
...snip...
```

You should get a large structure of formatted output telling you version and configuration information about the GPO InstaGENI aggregate. If you get any errors read them thorougly and review what they may be telling you about any mistakes you may have made. You can also ask your instructor if at an in-person tutorial.

3.1.4 Finished!

Assuming you have experienced no errors, your `geni-lib` installation is now set up and can communicate with all aggregates in the federation. If you have any issues you can send a message to the geni-users google group for help.

3.3 Querying the Federation

Before we can reserve resources, it is useful to know what resources are available across the federation. This tutorial will walk you through using the `Context` object you created in the previous tutorial to communicate with aggregates known to `geni-lib`.

3.3.1 Finding Aggregate Locations

`geni-lib` contains a set of package files which have pre-built objects representing known aggregates that are ready for you to use, contained within the following Python modules:

```
geni.aggregate.exogeni
geni.aggregate.instageni
geni.aggregate.instageni_openflow
geni.aggregate.opengeni
geni.aggregate.protogeni
geni.aggregate.vts
```

While these aggregates objects will likely cover your needs, `geni-lib` may of course not be updated as frequently as new aggregates come online. You can find a list of the current set of aggregates on the GENI Wiki.

3.3.2 Getting Aggregate Information

Given that we have our previously created `Context` object, and a wealth of aggregate objects available to us, the GENI federation provides the ability to request two blocks of information from each aggregate - the version information (which you may have seen briefly in a previous tutorial), and a list of the advertised resources.

The result from `getversion`, as we saw in the previous tutorial, is reasonably concise and human readable (but also contains information about API versions and supported request formats that you may need to extract in your tools). The list of advertised resources is acquired using the `listresources` call, and returns a large XML document describing the available resources, which is relatively difficult to work with without a tool.

Note: We will be using GENI AM API version 2 throughout this tutorial. Some API call names will be different if you elect to interact with aggregates using AM API version 3 in the future.

- Lets start by getting an advertisement from a single aggregate. If you built a custom context using Python code you will need to replace the code below to load your custom context:

```
$ python
>>> import geni.util
>>> context = geni.util.loadContext()
>>> import geni.aggregate.instageni as IGAM
>>> ad = IGAM.Illinois.listresources(context)
```

  Now of course we have an advertisement (assuming everything went well) stored into a Python object, which is reasonably boring!

Note: If you get timeouts or failures, you may want to try a different InstaGENI aggregate (this one may be particularly busy). You can get a list of (mostly) aggregate objects by using the `dir()` command on the IGAM module - `dir(IGAM)`.

- We can simply print out the advertisement raw text to see what the aggregate sent us:

```
>>> print ad.text
<rspec xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ......
...
```

As you can see, even with this relatively small rack (5 hosts) the amount of data is significant.

- As `geni-lib` has parsed this advertisement into a more functional object, we have access to data objects instead of just raw xml. For example, we can inspect the routable address space available at a site:

```
>>> ad.routable_addresses.available
167
>>> ad.routable_addresses.capacity
190
```

- You may have noticed that if you just print the `routable_addresses` attribute, you get nothing useful:

```
>>> ad.routable_addresses
<geni.rspec.pgad.RoutableAddresses object at 0x1717f10>
```

While we are adding online documentation for `geni-lib` objects, there are many objects that are undocumented. However, you can still gain some insight by using the `dir()` built-in to see what attributes are available:

```
>>> dir(ad.routable_addresses)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__'
'__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'available', 'capacity', 'configured
```

In general attributes starting with underscores are not useful to us, so we can see 3 attributes of value - `available`, `capacity`, and `configured`. In most cases their meanings should be obvious, so just knowing they exist even without documentation is quite helpful.

- There are also 3 iterators that are provided with `Advertisement` objects - `nodes`, `links`, and `shared_vlans`:

```
>>> for svlan in ad.shared_vlans:
...   print svlan
...
mesoscale-openflow
exclusive-openflow-1755
exclusive-openflow-1756
exclusive-openflow-1757
...snip...
```

- While `shared_vlans` just iterates over a set of strings, `node` objects are much more complex and have many more attributes and nested data structures to allow you to fully inspect their state:

```
>>> print dir(ad.nodes[0])
[..., 'available', 'component_id', 'component_manager_id', 'exclusive', 'hardware_types', 'image
'interfaces', 'location', 'name', 'shared', 'sliver_types']
```

- Particularly useful for the puposes of binding requests to certain nodes at a given site is the component_id:

```
>>> for node in ad.nodes:
...     print node.component_id
...
urn:publicid:IDN+instageni.illinois.edu+node+procurve2
urn:publicid:IDN+instageni.illinois.edu+node+pc3
urn:publicid:IDN+instageni.illinois.edu+node+pc5
urn:publicid:IDN+instageni.illinois.edu+node+interconnect-ion
```

```
    urn:publicid:IDN+instageni.illinois.edu+node+pc1
    urn:publicid:IDN+instageni.illinois.edu+node+interconnect-campus
    urn:publicid:IDN+instageni.illinois.edu+node+pc2
    urn:publicid:IDN+instageni.illinois.edu+node+interconnect-geni-core
    urn:publicid:IDN+instageni.illinois.edu+node+pc4
    urn:publicid:IDN+instageni.illinois.edu+node+internet
```

- Spend some time inspecting the other attributes of each node. You can get a specific node by using Python indexing on the `nodes` iterator:

```
    >>> node = ad.nodes[1]
    >>> node.component_id
    'urn:publicid:IDN+instageni.illinois.edu+node+pc3'
```

3.3.3 Iterating Over Aggregates

Often you will want to inspect a large number of aggregates (particularly if there are of an idential or similar type) in order to find those that have availability in the resources that you require. The aggregate modules in `geni-lib` provide some convenience methods for assisting in this task:

```
>>> import geni.aggregate.instageni as IGAM
>>> for am in IGAM.aggregates():
...     print am.name
...
ig-cenic
ig-cwru
ig-clemson
ig-cornell
ig-ohmetrodc
ig-gatech
ig-gpo
ig-illinois
...snip...
```

Using this iterator you can act on each aggregate in a given module with the same snippet of code.

- Lets try getting (and saving) the `getversion` output from each InstaGENI site:

```
    >>> import json
    >>> for am in IGAM.aggregates():
    ...     print am.name
    ...     verdata = am.getversion(context)
    ...     ver_file = open("%s-version.json" % (am.name), "w+")
    ...     json.dump(verdata, ver_file)
    ...
    ig-cenic
    ig-cwru
    ig-clemson
    ...snip...
```

  This will write out a file for every aggregate (barring any exceptions) to the current directory.

Note: `verdata` in the above case is a Python `dict` object, so we need to pick a way to write it (in a human readable form) to a file. In the above example we pick serializing to JSON (which is reasonably readable), but you could also use the `pprint` module to format it nicely to a file as a nice string.

3.3.4 Exercises

We can now combine all of the above pieces, plus some Python knowledge, into some useful scripts.

1. Move the `getversion` code fragment above into a standalone script, and improve it to continue to the next aggregate if any exceptions are thrown by the current aggregate (unreachable, busy, etc.).
2. Write a script that prints out the number of availble routable IPs for each InstaGENI aggregate.

3.5 Creating a Request for a Single VM

This example walks through the basic of creating an RSpec (xml file) requesting a single VM from a compute aggregate. This example does not require that *geni-lib* is configured with user credentials or keys - it will create an XML file that you can feed into another tool such as *Jacks* or *Omni* (other examples cover how to make this request using *geni-lib* itself).

Note: You can find the complete source code for this example in a single file in the *geni-lib* distribution in *samples/onevm.py*.

3.5.1 Walk-through

- Since we only want to output the XML of the request, we need very few imports:

```
import geni.rspec.pg as PG
import geni.rspec.egext as EGX
import geni.rspec.igext as IGX
```

Note: While the first module is named 'pg' (after ProtoGENI), the base rspec format is common across compute aggregates and all will use the same *Request* container, although the resources in that container will differ based on what is available at a given site.

- Now we need to create the basic *Request* container:

```
r = PG.Request()
```

- Unfortunately there is no unified VM object for all compute aggregates, so you will need to know which "flavor" of compute aggregate you intend to use (most commonly either InstaGENI or ExoGENI).

Note: In later examples you will see how, if you are using *geni-lib* to make your reservations directly with the aggregates, you can indeed create a single VM request that can be used across aggregate "flavors".

- Now we will allocate a VM object that can be added to our request (examples shown here for both ExoGENI and InstaGENI):

```
# ExoGENI
exovm = EGX.XOSmall("vm1")

# InstaGENI
igvm = IGX.XenVM("vm1")
```

Note: The only required configuration for each resource is the *name* argument that is passed to the constructor. These names must be unique within a single site, but can be reused at different sites.

- For the purposes of this example we will only add the InstaGENI VM to the actual request that we will produce:

```
r.addResource(igvm)
```

- Now that we have a request that contains a resource, we can write the XML to disk that represents this request:

```
r.writeXML("onevm-request.xml")
```