Python-RHEV Documentation

Release 1.0-rc1.13

Geert Jansen

August 15, 2016

Contents

Contents:

CHAPTER 1

Tutorial

Python-RHEV is a set of bindings for the Python language that facilitate accessing the Red Hat Enterprise Virtualization RESTful API. Normally, RESTful APIs can be accessed via a regular HTTP library, and the Red Hat Enterprise Virtualization API is no exception. Python-RHEV however implements a few features on top of the normal HTTP access that make consumption of the API easier:

- Connection management. Python-RHEV will automatically connect, authenticated, reconnect in case of errors, and follow links.
- Object mapping. Python-RHEV provides Python classes for the resources and collections that are exposed in the API (e.g. Cluster, VM, etc.). This allows you to interact with the API in an object oriented way.
- Exceptions. Errors are raised as exceptions. This means that you don't need to use a lot of in-line code to deal with errors.

This section provides a tutorial on how to use the Red Hat Enterprise Virtualization API for first-time users.

1.1 Connecting to the API

Before you can use the API, you need to connect to it. Three pieces of information are required before a connection can be made:

- The URL of the API entry point
- A user name
- A password

Connections are established via the *rhev.Connection* class. The following example illustrates this:

```
>>> from rhev import Connection
>>> conn = Connection()
>>> conn.connect(url, username, password)
>>> conn.ping()
```

The `connect()` method connects to the API at *url*, and sets the supplied *username* and *password* as the default credentials. The `ping()` method will retrieve the entry point. If no exception is raised in either call, you can be certain that a proper connection is available.

1.2 Resources and Collections

The Red Hat Enterprise Virtualization API is a RESTful API, and the RESTful nature of it permeates through the Python bindings. This is on purpose, both for philosophical reasons (I believe the RESTful paradigm is a particularly useful way to consume this type of functionality), and for practical reasons (the language bindings become a simple shim).

If you are new to RESTful APIs, you need to know the two key concepts of any RESTful API. These are the **Collection** and the **Resource**. Think of a resource as an object with a fixed API (create, read, update and delete) that contains a set of attributes that are relevant for the resource type. Think of a collection as a set of resources of the same type.

The following example shows how to retrieve a list of all virtual machines:

```
>>> from rhev import schema
>>> vms = conn.getall(schema.VM)
>>> type(vms)
<class 'rhev.schema.VMs'>
```

The `getall()` method accepts a class from the `rhev.schema` module that specifies the type of object we want to retrieve. We call these classes *binding classes*. Each binding class corresponds to one type within Red Hat Enterprise Virtualization. Instances of binding classes are called *binding instances*.

Two kinds of binding classes exist, corresponding to resources and collections. Resources have a singular name (like "VM" and "Cluster") and derive from `BaseResource`. Collections have a plural name ("VMs", "Clusters") and derive from `BaseResources`.

If you have the ID of a resource, you can retrieve only that resource using the `get()` call:

```
>>> vm = conn.get(schema.VM, 'xxx')
```

Collections behave like lists. Resources inside collections can be accessed using the Python `[]` operator. Resources behave like Python objects. Attributes inside resources are accessed as regular Python attributes:

```
>>> len(vms)
3
>>> vm = vms[0]
>>> type(vm)
<class 'rhev.schema.VM'>
>>> vm.name
'vm0'
>>> vm.status.state
'running'
```

1.3 Working with Resources

Resources are always created inside a collection. The collection that the resource is created in, depends on the resource type, and is determined automatically by the API. For example, a resource of type `rhev.schema.VM` will be created in the `rhev.schema.VMs` collection. For example:

```
>>> vm = schema.new(schema.VM)
>>> vm.name = 'vm0'
>>> # set other mandatory attributes
>>> conn.create(vm)
```

A resource can be updated by updating its attributes, and then calling the `Connection.update()` method:

```
>>> vm.name = 'newname'
>>> connection.update(vm)
```

A resource is deleted using the `Connnection.delete()` method. Example:

```
>>> connection.delete(vm)
```

Sometimes a resource can be changed by other users of the API while you have a reference to it. The `reload()` call reloads the resource in-place:

```
>>> conn.reload(vm)
```

Many resources in the Red Hat Enterprise Virtualization API contain a `state` sub-resource with a `status` attribute. When a VM is down for example, `state.status` will have the value `'down'`. When starting the VM, the state will go via `'wait for launch'` to `'powering up'` to `'up'`. Sometimes it is necessary to wait for a resource to be in a certain state. Python-RHEV facilitates this with the `wait_for_status()` method:

```
>>> conn.wait_for_status(vm, 'up')
```

Note: This will poll the API repeatedly. They between requests will increase over time up to a certain maximum.

The `update()`, `delete()`, `reload()` and `wait_for_status()` operations are also made available on `BaseResource`. This allows you to use a more object-oriented style of using the API. For example:

```
>>> vm.name = 'newname'
>>> vm.update()
>>> vm.reload()
>>> vm.wait_for_status('up')
>>> vm.delete()
```

The `create()` method is available as well, but it needs to be called on the containing resource. See the section on sub-resources and sub-collections below for more information.

1.4 Sub-resources and Sub-collections

We have already introduced resources and collection. An extension to these concepts are the sub-collection and the sub-resource. A resource can have sub-collections inside it, which can have sub-resources, and so on. Sub-collections and sub-resources are used extensively in the Red Hat Enterprise Virtualization API when modeling dependent objects. For example, a virtual machine will have multiple disks. Those disks related only to the virtual machine, and have no use outside it. The API models this as a sub-collection `schema.Disks` under the resource of type `schema.VM`.

Sub-collections and sub-resources can be accessed and/or modified by supplying the `base` parameter to any of the `get()`, `getall()`, `create()`, `update()` and `delete()` calls. For example:

```
>>> disks = conn.getall(schema.Disk, base=vm)
>>> disk = schema.new(schema.Disk)
>>> # set disk.size, etc.
>>> conn.create(disk, base=vm)
```

Note: Formally, the top-level collections are resources are also considered sub-collections and sub-resources. Their containing resource is the API entry point, can be considered as the only top-level resource. It does not have a corresponding collection, and it can be obtained using:

```
>>> api = conn.get(schema.API)
```

Or its short hand:

```
>>> api = conn.api()
```

By not specifying a `base` parameter to any of the functions mentioned above, you are implicitly specifying a base of `schema.API`.

1.5 Actions

Certain operations in the API do not match cleanly to the 4 standard REST methods `get()`, `update()`, `delete()` and `create()`. These operations are exposed as so-called *actions* by the API. Python-RHEV allows you to call an action via the `Connection.action()` method. For example, to execute the "start" action on a virtual machine:

```
>>> connection.action(vm, 'start')
```

Some actions can take input. In this case, you must pass an instance of a `schema.Action` class. For example:

```
>>> action = schema.new(schema.Action)
>>> action.stateless = true
>>> conn.action(vm, 'start', action)
```

Actions are also available as methods on the `BaseResource` class. So the above is the same as:

```
>>> vm.start(action)
```

1.6 Searching

For certain collections, the RHEV API provides search capabilities. This functionality can be accessed by providing the `search` keyword argument to `get()` and `getall()`:

```
>>> vms = conn.getall(schema.VM, search='name = a* sortby creation_time')
```

Searches that are just simple AND combinations of attribute searches and that do not have a "sortby" or "page" clause can also be provides as keyword arguments:

```
>>> vms = conn.getall(schema.VM, name='a*')
```

Search functionality is particularly useful in two cases. First, to identify an object by its name:

```
>>> vm = conn.get(schema.VM, name='foo')
```

Second, by using the "page" clause for the search syntax, you can retrieve more than the default of 100 objects that is normally returned:

```
>>> vms = []
>>> vms += conn.getall(schema.VM, search='sortby name page 1')
>>> vms += conn.getall(schema.VM, search='sortby name page 2')
```

1.7 Using Relationship Names

Instead of providing a binding type, you can also specify a type by its name. The name needs to correspond to the *relationship name* as it is provided by the API (this is value of the "rel" attribute in <link rel="*name*"/> elements). For example:

```
>>> # These two are equivalent
>>> vm = conn.getall('vms')
>>> vm = conn.getall(schema.VM)
```

Using relationship names is useful to access some non-standards relationship, for example:

```
>>> tag = conn.get('tags/root')
>>> template = conn.get('templates/blank')
```

1.8 Populating Resources

So far we have see that new resources can be populated by setting their attributes simply as Python attributes:

```
>>> vm = schema.new(schema.VM)
>>> vm.name = 'myvm'
```

However, the data types involved in the API are not flat, but nested. In order to set a scheduling policy on a VM for example, you need to do this:

```
>>> vm.placement_policy = schema.new(schema.VmPlacementPolicy)
>>> vm.placement_policy.host = schema.new(schema.Host)
>>> vm.placement_plilcy.host.name = 'myhost'
```

This requires that you need to know the name of all the types involved. Some types (like "Host" in the example above) are also top-level types so you probably already know them. Other types however (like VmPlacementPolicy) are internal types that may be relevant only to one specific resource. Python-RHEV contains a utility function called `subtype()` that can faciliated creating nested attribute where you do not know the exact type:

```
>>> vm.placement_policy = schema.new(schema.subtype(vm.placement_policy))
```

1.9 Error Handling

Error handling is mostly done through exceptions. All exceptions are subclasses of the *`rhev.Error`* class. Generally, any HTTP response that is not in the 2xx or 3xx category is considered an error, and an appropriate exception is raised. Sometimes, a 4xx response contains a valid resource of type `Fault` which describes the error in more detail. If that happens, an exception of type `Fault` is raised that gives you access to the fault itself:

```
>>> vm.name = 'my vm'  # space not allowed
>>> try:
...   vm.update()
... except Fault, f:
...   print 'fault', f.fault.detail
...
Can not edit VM. The given name contains special characters. Only
lower-case and upper-case letters, numbers, '_', '-' allowed.
```

CHAPTER 2

Reference

2.1 rhev – API Access

2.1.1 Exceptions

Most errors in Python-RHEV are reported as exceptions. All exceptions derived from the *Error* class.

exception rhev.**Error**
Base class for all errors.

exception rhev.**ParseError**
Could not parse a response generated by the API.

exception rhev.**ConnectionError**
Could not connect to the API.

exception rhev.**IllegalMethod**
The operation is not available for the specified resource or collection.

exception rhev.**NotFound**
The resource or collection was not found.

exception rhev.**IllegalAction**
The action is not available for the specified resource.

exception rhev.**ResponseError**
The response is not what was excpected.

exception rhev.**Fault**
A fault was returned.

exception rhev.**ActionError**
An action failed.

2.1.2 Logging

Python-RHEV uses the standard Python `logging` facility to output diagnostic messages. To enable debugging output, you can use the following example code:

```
import logging
logging.basicConfig()
logger = logging.getLogger('rhev')
logger.setLevel(logging.DEBUG)
```

2.1.3 The `Connection` class

The connection class is the main class in the Python-RHEV. All interactions with the RESTful API go through instances of this class.

class `rhev.`**`Connection`**

`retries`
Class attribute that defines the number of times a request is retried in case of a transient error. The default value is 5.

`verbosity`
Class attribute that sets the verbosity by which debug messages are logged. The default value is 1. A value of 10 means be maximally verbose.

`url`
Instance attribute that contains the current URL.

`username`
Instance attribute containing the current user name.

`password`
Instance attribute containing the current password.

`scheme`
Instance attribute containing the URL scheme that is currently in use. Possible values are "http" or "https". This value is available after *`connect()`* has been called.

`entrypoint`
Instance attribute containing the relative path of the entry point URL. This value is available after *`connect()`* has been called.

`__init__`(*url=None*, *username=None*, *password=None*)
Create a new connection object. If the *url*, *username* or *password* arguments are provided, this will set the default connection details.

The constructor only prepares the connection. No connection will be made.

`connect`(*url=None*, *username=None*, *password=None*)
Open a TCP/IP connection to the API. This call does not yet authenticate or retrieve a resource.

The *url*, *username* and *password* arguments specify the connection details. If they are not provided, the default connection details are used.

Calling `connect()` is not mandatory. If a connection is needed, a connection will be made automatically.

`close`()
Close the connection to the API.

`ping`()
Test the connection by retrieving the API entry point.

`getall`(*typ*, *base=None*, *detail=None*, *filter=None*, ***query*)
Get resources from a collection. The result is returned as a collection instance.

The *typ* argument specifies which collection to fetch. It must be a resource instance or a string identifying the relationship name.

In case of sub-collections, *base* must be a resource instance that specifies the resource relative to which the collection exists.

The *detail* parameter, if specified, must contain a string specifying the detail level of the response.

If the **query* keyword arguments are provided, then instead of simply returning all resources from the collection, a search operation is performed. The keyword arguments are interpreted as follows. If a keyword argument corresponds to a URL template variable, then it provides the value for that template variable. For example, the query for a search is normally specified by a "search" URL template variable. If, after URL template variable expansion, variables remain, and if no "search" keyword argument was provided, then an AND query is constructed that is passed into the "search" URL template variable, if available. This allows a very tight specification e.g. when searching by name using ‘‘getall(schema.VM, name='myvm').

The *filter* keyword argument specifies an optional filter. This must contain a dictionary of the form {'foo.bar': 'value'}, where "foo.bar" is the full attribute name to filter on and "value" is the value to filter for. Value can contain the "*" and "?" wild cards. The difference between filtering and searching is that search is implemented by the API itself, while filtering is done afterwards on the client. This means that for searching is more efficient than filtering. Filters are still useful because searching is not available for all collections, and even if searching is available, not all attributes can be searched on.

get(*typ*, *id=None*, *base=None*, *detail=None*, *filter=None*, **query*)
Return a single resource from a collection. The *typ* argument specifies the type to return, see *getall()* for a description.

The *id* argument, if specified, contains the ID of the object to retrieve.

The *base*, *detail*, *filter* and **query* arguments are as described for *getall()*.

reload(*resource*, *detail=None*)
Reloads *resource*, which must be a resource instance. If *detail* is specified, the resource is reloaded at that detail level.

This updates the resource in place, in other words, *resource* itself is modified. This call also return the resource.

create(*resource*, *base=None*)
Create a new resource. The *resource* argument specifies the resource to create, and must be a resource instance. The *base* argument, if provided, specifies the resource relative to which the collection exists in which the resource is to be created. By default, resources are created relative to the API entry point.

update(*resource*)
Update the resource specified in *resource*.

delete(*resource*, *base=None*, *data=None*)
Delete the resource specified by *resource*. If *base* is provided, it specifies a resource relative to which the collection exists from which the resource is to be deleted. If *data* is provided, it must be a resource instance of the same type as *resource*, that specifies optional parameters for the delete operation.

action(*resource*, *action*, *data=None*)
Execute an action. The action named *action* is executed on the resource indicated by *resource*. If *data* is provided, it must be an instance of `Action` that is used to provide parameters for the action.

api()
Returns the API entry point. This is a short hand for `get(schema.API)`.

capabilities()
Return the Capabilities resource. This is a short hand for `get(schema.Capabilities)`.

wait_for_status(*resource*, *status*, *timeout=None*)
Wait until *resource* enters a certain state. The state of a resource is the value of the `state.status` attribute. The *status* argument can be a string or a sequence of strings. In the latter case, any of the provides statuses will match. If *timeout* is specified, wait for no longer than this amount of seconds. The default is to wait indefinitely.

As a special case, a *status* of None can be provided to wait until the resource has been deleted.

get_methods(*obj*, *base=None*)
Return a list of methods (HTTP methods) that are available for the resource or collection *obj*. The *obj* parameter can be a resource instance, a resource class, or a collection class. In the latter two cases, *base* can be provided to specify a base.

get_actions(*obj*)
Return a list of actions that are available for the resource or collection *obj*, which must be a resource or collection instance.

get_links(obj):
Return a list of links that are available for the resource or collection *obj*, which must be a resource or collection instance.

2.2 rhev.schema – Binding Types

The `rhev.schema` module contains functionality to query, create and modify binding types. Binding types are Python classes that correspond to the collections or resources in the Red Hat Enterprise Virtualization RESTful API.

The table below lists the available top-level types. Top-level types correspond directly with either a resource or a collection in the RESTful API. Technically these are auto-generated from "complexType" definitions that are defined in the XMLSchema that describes the API.

Resource Type	Collection Type	Description
API	N/A	The API entry point
Capabilities	N/A	The capilities resource
DataCenter	DataCenters	A data center
Cluster	Clusters	A cluster within a data center
StorageDomain	StorageDomains	A storage domain
Network	Networks	A virtual network
Host	Hosts	A physical host
HostNIC	HostNics	A physical NIC on a physical host
Storage	HostStorage	Available block storage on a host
VM	VMs	A virtual machine
NIC	Nics	A virtual NIC of a virtual machine
Disk	Disks	A virtual disk of a virtual machine
CdRom	CdRoms	A virtual CD-ROM of a virtual machine
Floppy	Floppies	A virtual floppy of a virtual machine
Snapshot	Snapshots	A snapshot of a virtual disk
File	Files	A file available from a storage domain
Statistic	Statistics	Statistics
Template	Templates	A virtual machine template
VmPool	VmPools	A pool of virtual machines
User	Users	A user
Role	Roles	A user role
Event	Events	An event
Tag	Tags	A tag
Action	Actions	Used to provide input to actions

All resource types are derived from the BaseResource class, while collections are derived from BaseCollection.

2.2.1 Functions

The following functions are avaialble.

rhev.**type_info**(*key*, *base=None*)
This function returns type information for a type identified by *key*. The *key* argument must be either a resource type, a collection type, or a relatioship name. The return value is tuple containing (resourcetype, collectiontype, singular, plural). The resource and collection types are the types corresponding to the type's resource and collection. The singular and plural elements are strings referring to a singular and a plural name for the type. The plural name is identical to the relationship name.

rhev.**get_xml_schema**()
Return a parsed elementtree of the API's XMLSchema definition. This uses a version of the schema that was included at build time as part of the Python bindings, and may therefore be out of date.

rhev.**new**(*type*, **args*, ***kwargs*)
Create a new instance of a binding type.

rhev.**ref**(*obj*)
Create a reference to the object in *obj*. This creates a copy of the object, and then only sets the "id" attribute. This copy can be used in certain operations that expect only a reference to an object instead of a full representation.

href(obj):
The same as *ref()* but also sets "href".

rhev.**update**(*obj*, *upd*)
Update the object *obj* with all attributes of *upd* that are not None. The parameters *obj* and *upd* need to be instances of the same binding type.

rhev.**copy**(*obj*)
Return a shallow copy of the object *obj*. The attributes are shared between the old and the new instance.

rhev.**create_from_xml**(*s*)
Parse an XML string and return an instance of a binding type. This will raise a ParseError in case the XML cannot be parsed, or in case it corresponds to an unknown or illegal binding type.

rhev.**attributes**(*obj*)
Return an interator that enumerates all attributes of the object *obj*.

rhev.**subtype**(*prop*)
Return the correct binding type for an object attribute. The parameter *prop* must be an attribute of a binding class.

2.2.2 Object Interface

Certain utility methods are provided on the *BaseResource* and *BaseCollection* classes. These methods are available to all resource and collection instances, respectively. This allows the user to work with resources and collections a more object oriented way.

class rhev.**BaseResource**

__getattr__(*name*)
Augments basic attribute access. Users can use the standard dot ('.') attribute access operator to get acccess to actions and related sub-collections, in addition to the data attributes. For example, if a VM instance "vm" has a "start" action, then the following will work:

```
vm.start()
```

reload()
Reload the object. This re-fetches all information from the API.

create(*resource*)
Create a dependent resource to this resource.

update()
Sync back the object to the API. This makes all attribute modifications that were made on the object persistent.

delete(*data=None*)
Deletes the object. If the DELETE operation takes input, then that can be passed in as *data*.

wait_for_status(*status*, *timeout=None*)
Wait for this object to enter *status*. This uses an exponential backoff polling loop. The *timeout* parameter specifies the number of seconds to wait at most. If no timeout is specified, this waits indefinitely.

class rhev.**BaseCollection**

__getitem__(*i*)
Returns the i'th element of the collection.

__len__()
Return the number of resources in the collection.

__iter__()
Iterate over all resources in the collection.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

r

Symbols

A

B

C

D

E

F

G

I

N

P

R

S

T

U

V

W