Red Hat Virtualization 4.4

Python SDK Guide

Using the Red Hat Virtualization Python SDK

Last Updated: 2023-02-07

Red Hat Virtualization 4.4 Python SDK Guide

Using the Red Hat Virtualization Python SDK

Red Hat Virtualization Documentation Team
Red Hat Customer Content Services
rhev-docs@redhat.com

Legal Notice

Abstract

This guide describes how to install and work with version 4 of the Red Hat Virtualization Python software development kit.

Table of Contents

CHAPTER 1. OVERVIEW

Version 4 of the Python software development kit is a collection of classes that allows you to interact with the Red Hat Virtualization Manager in Python-based projects. By downloading these classes and adding them to your project, you can access a range of functionality for high-level automation of administrative tasks.

NOTE

Version 3 of the SDK is no longer supported. For more information, consult the RHV 4.3 version of this guide.

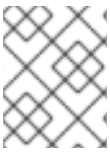
Python 3.7 and async

In Python 3.7 and later versions, **async** is a reserved keyword. You cannot use the **async** parameter in methods of services that previously supported it, as in the following example, because **async=True** will cause an error:

```
dc = dc_service.update(
    types.DataCenter(
        description='Updated description',
    ),
    async=True,
)
```

The solution is to add an underscore to the parameter (**async_**):

```
dc = dc_service.update(
    types.DataCenter(
        description='Updated description',
    ),
    async_=True,
)
```



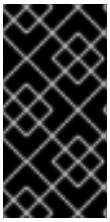
NOTE

This limitation applies only to Python 3.7 and later. Earlier versions of Python do not require this modification.

1.1. PREREQUISITES

To install the Python software development kit, you must have:

- A system where Red Hat Enterprise Linux 8 is installed. Both the Server and Workstation variants are supported.
- A subscription to Red Hat Virtualization entitlements.



IMPORTANT

The software development kit is an interface for the Red Hat Virtualization REST API. Use the version of the software development kit that corresponds to the version of your Red Hat Virtualization environment. For example, if you are using Red Hat Virtualization 4.3, use V4 Python software development kit.

1.2. INSTALLING THE PYTHON SOFTWARE DEVELOPMENT KIT

To install the Python software development kit:

1. Enable the repositories that are appropriate for your hardware platform . For example, for x86-64 hardware, enable:

```
# subscription-manager repos \
    --enable=rhel-8-for-x86_64-baseos-rpms \
    --enable=rhel-8-for-x86_64-appstream-rpms \
    --enable=rhv-4.4-manager-for-rhel-8-x86_64-rpms

# subscription-manager repos \
        --enable=rhel-8-for-x86_64-baseos-eus-rpms \
        --enable=rhel-8-for-x86_64-appstream-eus-rpms

# subscription-manager release --set=8.6
```

2. Install the required packages:

```
# dnf install python3-ovirt-engine-sdk4
```

The Python software development kit is installed into the Python 3 site-packages directory, and the accompanying documentation and example are installed to **/usr/share/doc/python3-ovirt-engine-sdk4**.

CHAPTER 2. USING THE SOFTWARE DEVELOPMENT KIT

This section describes how to use the software development kit for Version 4.

2.1. PACKAGES

The following modules are most frequently used by the Python SDK:

ovirtsdk4

This is the top level module. It most important element is the **Connection** class, which is the mechanism to connect to the server and to obtain the reference to the root of the services tree. The **Error** class is the base exception class that the SDK will raise when it needs to report an error.

For certain kinds of errors, there are specific error classes, which extend the base error class:

- **AuthError** - Raised when authentication or authorization fails.
- **ConnectionError** - Raised when the name of the server cannot be resolved or the server is unreachable.
- **NotFoundError** - Raised when the requested object does not exist.
- **TimeoutError** - Raised when an operation times out.

ovirtsdk4.types

This module contains the classes that implement the types used in the API. For example, the **ovirtsdk4.types.Vm** class is the implementation of the virtual machine type. These classes are data containers and do not contain any logic.
Instances of these classes are used as parameters and return values of service methods. The conversion to or from the underlying representation is handled transparently by the SDK.

ovirtsdk4.services

This module contains the classes that implement the services supported by the API. For example, the **ovirtsdk4.services.VmsService** class is the implementation of the service that manages the collection of virtual machines of the system.
Instances of these classes are automatically created by the SDK when a service is located. For example, a new instance of the **VmsService** class is automatically created by the SDK when doing the following:

```
vms_service = connection.system_service().vms_service()
```

It is best to avoid creating instances of these classes manually, as the parameters of the constructors and, in general, all the methods except the service locators and service methods, may change in the future.

There are other modules, like **ovirtsdk4.http**, **ovirtsdk4.readers**, and **ovirtsdk4.writers**. These are used to implement the HTTP communication and for XML parsing and rendering. Avoid using them, because they are internal implementation details that may change in the future; backwards compatibility is not guaranteed.

2.2. CONNECTING TO THE SERVER

To connect to the server, import the **ovirtsdk4** module, which contains the **Connection** class. This is the entry point of the SDK, and provides access to the root of the tree of services of the API:

```
import ovirtsdk4 as sdk

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)
```

The connection holds critical resources, including a pool of HTTP connections to the server and an authentication token. It is very important to free these resources when they are no longer in use:

```
connection.close()
```

Once a connection is closed, it cannot be reused.

The **ca.pem** file is required when connecting to a server protected with TLS. In a normal installation, it is located in **/etc/pki/ovirt-engine/** on the Manager machine. If you do not specify the **ca_file**, the system-wide CA certificate store will be used. For more information on obtaining the **ca.pem** file, see the *REST API Guide*.

If the connection is not successful, the SDK will raise an **ovirtsdk4.Error** exception containing the details.

2.3. USING TYPES

The classes in the **ovirtsdk4.types** module are pure data containers. They do not have any logic or operations. Instances of types can be created and modified at will.

Creating or modifying an instance does not affect the server side, unless the change is explicitly passed with a call to one of the service methods described below. Changes on the server side are not automatically reflected in the instances that already exist in memory.

The constructors of these classes have multiple optional arguments, one for each attribute of the type. This is intended to simplify creation of objects using nested calls to multiple constructors. This example creates an instance of a virtual machine, specifying its cluster name, template, and memory, in bytes:

```
from ovirtsdk4 import types

vm = types.Vm(
    name='vm1',
    cluster=types.Cluster(
        name='Default'
    ),
    template=types.Template(
        name='mytemplate'
    ),
    memory=1073741824
)
```

Using the constructors in this way is recommended, but not mandatory. You can also create the instance with no arguments in the call to the constructor and populate the object step by step, using the setters, or by using a mix of both approaches:

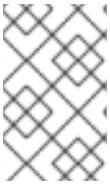
```
vm = types.Vm()
vm.name = 'vm1'
vm.cluster = types.Cluster(name='Default')
vm.template = types.Template(name='mytemplate')
vm.memory=1073741824
```

Attributes that are defined as lists of objects in the specification of the API are implemented as Python lists. For example, the **custom_properties** attributes of the **Vm** type are defined as a list of objects of type **CustomProperty**. When the attributes are used in the SDK, they are a Python list:

```
vm = types.Vm(
    name='vm1',
    custom_properties=[
        types.CustomProperty(...),
        types.CustomProperty(...),
        ...
    ]
)
```

Attributes that are defined as enumerated values in API are implemented as **enum** in Python, using the native support for **enums** in Python 3 and the **enum34** package in Python 2.7. In this example, the status attribute of the **Vm** type is defined using the **VmStatus enum**:

```
if vm.status == types.VmStatus.DOWN:
    ...
elif vm.status == types.VmStatus.IMAGE_LOCKED:
    ....
```



NOTE

In the API specification, the values of **enum** types appear in lower case, because that is what is used for XML and JSON. The Python convention, however, is to capitalize **enum** values.

Reading the attributes of instances of types is done using the corresponding properties:

```
print("vm.name: %s" % vm.name)
print("vm.memory: %s" % vm.memory)
for custom_property in vm.custom_properties:
    ...
```

2.4. USING LINKS

Some attributes of types are defined by the API as links. This convention indicates that the values are not normally populated when retrieving the representation of that object. Rather, a link is returned instead. For example, when retrieving a virtual machine, the XML response from the server includes the **<link>** attribute:

```
<vm id="123" href="/ovirt-engine/api/vms/123">
```

```
    <name>vm1</name>
    <link rel="diskattachments" href="/ovirt-engine/api/vms/123/diskattachments/>
  ...
</vm>
```

The link to **vm.diskattachments** does not contain the actual disk attachments. To obtain the data, the **Connection** class provides a **follow_link** method that uses the value of the **href** XML attribute to retrieve the actual data. For example, to retrieve the details of the disks of the virtual machine, you follow the link to the disk attachments, and then to each of the disks:

```
# Retrieve the virtual machine:
vm = vm_service.get()

# Follow the link to the disk attachments, and then to the disks:
attachments = connection.follow_link(vm.disk_attachments)
for attachment in attachments:
    disk = connection.follow_link(attachment.disk)
    print("disk.alias: " % disk.alias)
```

2.5. LOCATING SERVICES

The API provides a set of services, each associated with a path within the URL space of the server. For example, the service that manages the collection of virtual machines of the system is located in **/vms**, and the service that manages the virtual machine with identifier **123** is located in **/vms/123**.

In the SDK, the root of that tree of services is implemented by the system service. It is obtained calling the **system_service** method of the connection:

```
system_service = connection.system_service()
```

When you have the reference to this system service, you can use it to obtain references to other services, calling the ***_service** methods, called service locators, of the previous service. For example, to obtain a reference to the service that manages the collection of virtual machines of the system, you use the **vms_service** service locator:

```
vms_service = system_service.vms_service()
```

To obtain a reference to the service that manages the virtual machine with identifier **123**, you use the **vm_service** service locator of the service that manages the collection of virtual machines. It uses the identifier of the virtual machine as a parameter:

```
vm_service = vms_service.vm_service('123')
```

IMPORTANT

Calling service locators does not send a request to the server. The Python objects that they return are pure services, which do not contain any data. For example, the **vm_service** Python object called in this example is not the representation of a virtual machine. It is the service that is used to retrieve, update, delete, start and stop that virtual machine.

2.6. USING SERVICES

After you have located a service, you can call its service methods, which send requests to the server and do the real work.

Services that manage a single object usually support the **get**, **update**, and **remove** methods.

Services that manage collections of objects usually support the **list** and **add** methods.

Both kinds of services, especially services that manage a single object, can support additional action methods.

2.6.1. Using get Methods

These service methods are used to retrieve the representation of a single object. The following example retrieves the representation of the virtual machine with identifier **123**:

```
# Find the service that manages the virtual machine:
vms_service = system_service.vms_service()
vm_service = vms_service.vm_service('123')

# Retrieve the representation of the virtual machine:
vm = vm_service.get()
```

The response is an instance of the corresponding type, in this case an instance of the Python class **ovirtsdk4.types.Vm**.

The **get** methods of some services support additional parameters that control how to retrieve the representation of the object or what representation to retrieve if there is more than one. For example, you may want to retrieve either the current state of a virtual machine or its state the next time it is started, as they may be different. The **get** method of the service that manages a virtual machine supports a **next_run** Boolean parameter:

```
# Retrieve the representation of the virtual machine, not the
# current one, but the one that will be used after the next
# boot:
vm = vm_service.get(next_run=True)
```

See the reference documentation of the SDK for details.

If the object cannot be retrieved for any reason, the SDK raises an **ovirtsdk4.Error** exception, with details of the failure. This includes the situation when the object does not actually exist. Note that the exception is raised when calling the **get** service method. The call to the service locator method never fails, even if the object does not exist, because that call does not send a request to the server. For example:

```
# Call the service that manages a non-existent virtual machine.
# This call will succeed.
vm_service = vms_service.vm_service('junk')

# Retrieve the virtual machine. This call will raise an exception.
vm = vm_service.get()
```

2.6.2. Using list Methods

These service methods retrieve the representations of the objects of a collection. This example retrieves the complete collection of virtual machines of the system:

```
# Find the service that manages the collection of virtual
# machines:
vms_service = system_service.vms_service()

# List the virtual machines in the collection
vms = vms_service.list()
```

The result will be a Python list containing the instances of corresponding types. For example, in this case, the result will be a list of instances of the class **ovirtsdk4.types.Vm**.

The **list** methods of some services support additional parameters. For example, almost all top-level collections support a **search** parameter to filter the results or a **max** parameter to limit the number of results returned by the server. This example retrieves the names of virtual machines starting with **my**, with an upper limit of 10 results:

```
vms = vms_service.list(search='name=my*', max=10)
```

NOTE

Not all **list** methods support these parameters. Some **list** methods support other parameters. See the reference documentation of the SDK for details.

If a list of returned results is empty for any reason, the returned value will be an empty list. It will never be **None**.

If there is an error while trying to retrieve the result, the SDK will raise an **ovirtsdk4.Error** exception containing the details of the failure.

2.6.3. Using add Methods

These service methods add new elements to a collection. They receive an instance of the relevant type describing the object to add, send the request to add it, and return an instance of the type describing the added object.

This example adds a new virtual machine called **vm1**:

```
from ovirtsdk4 import types

# Add the virtual machine:
vm = vms_service.add(
    vm=types.Vm(
        name='vm1',
        cluster=types.Cluster(
            name='Default'
        ),
        template=types.Template(
            name='mytemplate'
        )
    )
)
```

If the object cannot be created for any reason, the SDK will raise an **ovirtsdk4.Error** exception containing the details of the failure. It will never return **None**.

IMPORTANT

The Python object returned by this **add** method is an instance of the relevant type. It is not a service but a container of data. In this particular example, the returned object is an instance of the **ovirtsdk4.types.Vm** class. If, after creating the virtual machine, you need to perform an operation such as retrieving or starting it, you will first need to find the service that manages it, and call the corresponding service locator:

```
# Add the virtual machine:
vm = vms_service.add(
  ...
)

# Find the service that manages the virtual machine:
vm_service = vms_service.vm_service(vm.id)

# Start the virtual machine
vm_service.start()
```

Objects are created asynchronously. When you create a new virtual machine, the **add** method will return a response before the virtual machine is completely created and ready to be used. It is good practice to poll the status of the object to ensure that it is completely created. For a virtual machine, you should check until its status is **DOWN**:

```
# Add the virtual machine:
vm = vms_service.add(
  ...
)

# Find the service that manages the virtual machine:
vm_service = vms_service.vm_service(vm.id)

# Wait until the virtual machine is down, indicating that it is
# completely created:
while True:
    time.sleep(5)
    vm = vm_service.get()
    if vm.status == types.VmStatus.DOWN:
        break
```

Using a loop to retrieve the object status, with the **get** method, ensures that the status attribute is updated.

2.6.4. Using update Methods

These service methods update existing objects. They receive an instance of the relevant type describing the update to perform, send the request to update it, and return an instance of the type describing the updated object.

This example updates the name of a virtual machine from **vm1** to **newvm**:

```
from ovirtsdk4 import types

# Find the virtual machine, and then the service that
# manages it:
vm = vms_service.list(search='name=vm1')[0]
vm_service = vm_service.vm_service(vm.id)

# Update the name:
updated_vm = vm_service.update(
    vm=types.Vm(
        name='newvm'
    )
)
```

When performing updates, avoid sending the **complete** representation of the object. Send only the attributes that you want to update. Do not do this:

```
# Retrieve the complete representation:
vm = vm_service.get()

# Update the representation, in memory, without sending a request
# to the server:
vm.name = 'newvm'

# Send the update. Do *not* do this.
vms_service.update(vm)
```

Sending the complete representation causes two problems:

- You are sending much more information than the server needs, thus wasting resources.
- The server will try to update all the attributes of the object, even those that you did not intend to change. This may cause bugs on the server side.

The **update** methods of some services support additional parameters that control how or what to update. For example, you may want to update either the current state of a virtual machine or the state that will be used the next time the virtual machine is started. The **update** method of the service that manages a virtual machine supports a **next_run** Boolean parameter:

```
# Update the memory of the virtual machine to 1 GiB,
# not during the current run, but after next boot:
vm = vm_service.update(
    vm=types.Vm(
        memory=1073741824
    ),
    next_run=True
)
```

If the update cannot be performed for any reason, the SDK will raise an **ovirtsdk4.Error** exception containing the details of the failure. It will never return **None**.

The Python object returned by this update method is an instance of the relevant type. It is not a service, but a container for data. In this particular example, the returned object will be an instance of the **ovirtsdk4.types.Vm** class.

2.6.5. Using remove Methods

These service methods remove existing objects. They usually do not take parameters, because they are methods of services that manage single objects. Therefore, the service already knows what object to remove.

This example removes the virtual machine with identifier **123**:

```
# Find the virtual machine by name:
vm = vms_service.list(search='name=123')[0]

# Find the service that manages the virtual machine using the ID:
vm_service = vms_service.vm_service(vm.id)

# Remove the virtual machine:
vm_service.remove()
```

The **remove** methods of some services support additional parameters that control how or what to remove. For example, it is possible to remove a virtual machine while preserving its disks, using the **detach_only** Boolean parameter:

```
# Remove the virtual machine while preserving the disks:
vm_service.remove(detach_only=True)
```

The **remove** method returns **None** if the object is removed successfully. It does not return the removed object. If the object cannot be removed for any reason, the SDK raises an **ovirtsdk4.Error** exception containing the details of the failure.

2.6.6. Using Other Action Methods

There are other service methods that perform miscellaneous operations, such as stopping and starting a virtual machine:

```
# Start the virtual machine:
vm_service.start()
```

Many of these methods include parameters that modify the operation. For example, the method that starts a virtual machine supports a **use_cloud_init** parameter, if you want to start it using **cloud-init**:

```
# Start the virtual machine:
vm_service.start(cloud_init=True)
```

Most action methods return **None** when they succeed and raise an **ovirtsdk4.Error** when they fail. A few action methods return values. For example, the service that manages a storage domain has an **is_attached** action method that checks whether the storage domain is already attached to a data center and returns a Boolean value:

```
# Check if the storage domain is attached to a data center:
sds_service = system_service.storage_domains_service()
sd_service = sds_service.storage_domain_service('123')
if sd_service.is_attached():
    ...
```

Check the reference documentation of the SDK to see the action methods supported by each service, the parameters that they take, and the values that they return.

2.7. ADDITIONAL RESOURCES

For detailed information and examples, see the following resources:

- V4 REST API Guide
- Python SDK reference documentation
- Python SDK examples

2.7.1. Generating documentation for modules

You can generate documentation using pydoc for the following modules:

- ovirtsdk.api
- ovirtsdk.infrastructure.brokers
- ovirtsdk.infrastructure.errors

The documentation is provided by the **ovirt-engine-sdk-python** package. Run the following command on the Manager machine to view the latest version of these documents:

```
$ pydoc [MODULE]
```

CHAPTER 3. PYTHON EXAMPLES

3.1. OVERVIEW

This section provides examples demonstrating the steps to create a virtual machine within a basic Red Hat Virtualization environment, using the Python SDK.

These examples use the **ovirtsdk** Python library provided by the **ovirt-engine-sdk-python** package. This package is available to systems attached to a **Red Hat Virtualization** subscription pool in Red Hat Subscription Manager. See Installing the Software Development Kit for more information on subscribing your system(s) to download the software.

You will also need:

- A networked installation of Red Hat Virtualization Manager.
- A networked and configured Red Hat Virtualization Host.
- An ISO image file containing an operating system for installation on a virtual machine.
- A working understanding of both the logical and physical objects that make up a Red Hat Virtualization environment.
- A working understanding of the Python programming language.

The examples include placeholders for authentication details (**admin@internal** for user name, and **password** for password). Replace the placeholders with the authentication requirements of your environment.

Red Hat Virtualization Manager generates a globally unique identifier (GUID) for the **id** attribute for each resource. Identifier codes in these examples differ from the identifier codes in your Red Hat Virtualization environment.

The examples contain only basic exception and error handling logic. For more information on the exception handling specific to the SDK, see the pydoc for the **ovirtsdk.infrastructure.errors** module:

```
$ pydoc ovirtsdk.infrastructure.errors
```

3.2. CONNECTING TO THE RED HAT VIRTUALIZATION MANAGER IN VERSION 4

To connect to the Red Hat Virtualization Manager, you must create an instance of the **Connection** class from the **ovirtsdk4.sdk** module by importing the class at the start of the script:

```
import ovirtsdk4 as sdk
```

The constructor of the **Connection** class takes a number of arguments. Supported arguments are:

url

A string containing the base URL of the Manager, such as **https://server.example.com/ovirt-engine/api**.

username

Specifies the user name to connect, such as **admin@internal**. This parameter is mandatory.

password

Specifies the password for the user name provided by the **username** parameter. This parameter is mandatory.

token

An optional token to access the API, instead of a user name and password. If the **token** parameter is not specified, the SDK will create one automatically.

insecure

A Boolean flag that indicates whether the server's TLS certificate and host name should be checked.

ca_file

A PEM file containing the trusted CA certificates. The certificate presented by the server will be verified using these CA certificates. If **ca_file** parameter is not set, the system-wide CA certificate store is used.

debug

A Boolean flag indicating whether debug output should be generated. If the value is **True** and the **log** parameter is not **None**, the data sent to and received from the server will be written to the log.

NOTE

User names and passwords are written to the debug log, so handle it with care.

Compression is disabled in debug mode, which means that debug messages are sent as plain text.

log

The logger where the log messages will be written.

kerberos

A Boolean flag indicating whether Kerberos authentication should be used instead of the default basic authentication.

timeout

The maximum total time to wait for the response, in seconds. A value of **0** (default) means to wait forever. If the timeout expires before the response is received, an exception is raised.

compress

A Boolean flag indicating whether the SDK should ask the server to send compressed responses. The default is **True**. This is a hint for the server, which may return uncompressed data even when this parameter is set to **True**. Compression is disabled in debug mode, which means that debug messages are sent as plain text.

sso_url

A string containing the base SSO URL of the server. The default SSO URL is computed from the **url** if no **sso_url** is provided.

sso_revoke_url

A string containing the base URL of the SSO revoke service. This needs to be specified only when using an external authentication service. By default, this URL is automatically calculated from the value of the **url** parameter, so that SSO token revoke will be performed using the SSO service, which is part of the Manager.

sso_token_name

The token name in the JSON SSO response returned from the SSO server. Default value is **access_token**.

headers

A dictionary with headers, which should be sent with every request.

connections

The maximum number of connections to open to the host. If the value is **0** (default), the number of connections is unlimited.

pipeline

The maximum number of requests to put in an HTTP pipeline without waiting for the response. If the value is **0** (default), pipelining is disabled.

```
import ovirtsdk4 as sdk

# Create a connection to the server:
connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

connection.test()

print("Connected successfully!")

connection.close()
```

For a full list of supported methods, you can generate the documentation for the **ovirtsdk.api** module on the Manager machine:

```
$ pydoc ovirtsdk.api
```

3.3. LISTING DATA CENTERS

The **datacenters** collection contains all the data centers in the environment.

Example 3.1. Listing data centers

This example lists the data centers in the **datacenters** collection and output some basic information about each data center in the collection.

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

dcs_service = connection.system_service().dcs_service()
```

```
dcs = dcs_service.list()

for dc in dcs:
    print("%s (%s)" % (dc.name, dc.id))

connection.close()
```

In an environment where only the **Default** data center exists, and it is not activated, the examples output the text:

```
Default (00000000-0000-0000-0000-000000000000)
```

3.4. LISTING CLUSTERS

The **clusters** collection contains all clusters in the environment.

Example 3.2. Listing clusters

This example lists the clusters in the **clusters** collection and output some basic information about each cluster in the collection.

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

cls_service = connection.system_service().clusters_service()

cls = cls_service.list()

for cl in cls:
    print("%s (%s)" % (cl.name, cl.id))

connection.close()
```

In an environment where only the **Default** cluster exists, the examples output the text:

```
Default (00000000-0000-0000-0000-000000000000)
```

3.5. LISTING HOSTS

The **hosts** collection contains all hosts in the environment.

Example 3.3. Listing hosts

This example lists the hosts in the **hosts** collection and their IDs.

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

host_service = connection.system_service().hosts_service()

hosts = host_service.list()

for host in hosts:
    print("%s (%s)" % (host.name, host.id))

connection.close()
```

In an environment where only one host, **MyHost**, has been attached, the examples output the text:

```
MyHost (00000000-0000-0000-0000-000000000000)
```

3.6. LISTING LOGICAL NETWORKS

The **networks** collection contains all logical networks in the environment.

Example 3.4. Listing logical networks

This example lists the logical networks in the **networks** collection and outputs some basic information about each network in the collection.

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

nws_service = connection.system_service().networks_service()

nws = nws_service.list()
```

```
for nw in nws:
    print("%s (%s)" % (nw.name, nw.id))

connection.close()
```

In an environment where only the default management network exists, the examples output the text:

```
ovirtmgmt (00000000-0000-0000-0000-000000000000)
```

3.7. LISTING VIRTUAL MACHINES AND TOTAL DISK SIZE

The **vms** collection contains a **disks** collection that describes the details of each disk attached to a virtual machine.

Example 3.5. Listing virtual machines and total disk size

This example prints a list of virtual machines and their total disk size in bytes:

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

vms_service = connection.system_service().vms_service()

virtual_machines = vms_service.list()

if len(virtual_machines) > 0:

    print("%-30s  %s" % ("Name", "Disk Size"))
    print("================================================")

    for virtual_machine in virtual_machines:
        vm_service = vms_service.vm_service(virtual_machine.id)
        disk_attachments = vm_service.disk_attachments_service().list()
        disk_size = 0
        for disk_attachment in disk_attachments:
            disk = connection.follow_link(disk_attachment.disk)
            disk_size += disk.provisioned_size

            print("%-30s: %d" % (virtual_machine.name, disk_size))
```

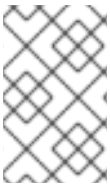
The examples output the virtual machine names and their disk sizes:

```
Name                 Disk Size
=================================================
vm1                  50000000000
```

3.8. CREATING NFS DATA STORAGE

When a Red Hat Virtualization environment is first created, it is necessary to define at least a data storage domain and an ISO storage domain. The data storage domain stores virtual disks while the ISO storage domain stores the installation media for guest operating systems.

The **storagedomains** collection contains all the storage domains in the environment and can be used to add and remove storage domains.



NOTE

The code provided in this example assumes that the remote NFS share has been pre-configured for use with Red Hat Virtualization. See the *Administration Guide* for more information on preparing NFS shares.

Example 3.6. Creating NFS data storage

This example adds an NFS data domain to the **storagedomains** collection.

V4

For V4, the **add** method is used to add the new storage domain and the **types** class is used to pass the following parameters:

- A name for the storage domain.
- The data center object that was retrieved from the **datacenters** collection.
- The host object that was retrieved from the **hosts** collection.
- The type of storage domain being added (**data**, **iso**, or **export**).
- The storage format to use (**v1**, **v2**, or **v3**).

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

# Create the connection to the server:
connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Get the reference to the storage domains service:
sds_service = connection.system_service().storage_domains_service()

# Create a new NFS storage domain:
```

```
sd = sds_service.add(
    types.StorageDomain(
        name='mydata',
        description='My data',
        type=types.StorageDomainType.DATA,
        host=types.Host(
            name='myhost',
        ),
        storage=types.HostStorage(
            type=types.StorageType.NFS,
            address='_FQDN_',
            path='/nfs/ovirt/path/to/mydata',
        ),
    ),
)

# Wait until the storage domain is unattached:
sd_service = sds_service.storage_domain_service(sd.id)
while True:
    time.sleep(5)
    sd = sd_service.get()
    if sd.status == types.StorageDomainStatus.UNATTACHED:
        break

print("Storage Domain '%s' added (%s)." % (sd.name(), sd.id()))

connection.close()
```

If the **add** method call is successful, the examples output the text:

```
Storage Domain 'mydata' added (00000000-0000-0000-0000-000000000000).
```

3.9. CREATING NFS ISO STORAGE

To create a virtual machine, you need installation media for the guest operating system. The installation media are stored in an ISO storage domain.

NOTE

The code provided in this example assumes that the remote NFS share has been pre-configured for use with Red Hat Virtualization. See the *Administration Guide* for more information on preparing NFS shares.

Example 3.7. Creating NFS ISO storage

This example adds an NFS ISO domain to the **storagedomains** collection.

V4

For V4, the **add** method is used to add the new storage domain and the **types** class is used to pass the following parameters:

- A name for the storage domain.

- The data center object that was retrieved from the **datacenters** collection.
- The host object that was retrieved from the **hosts** collection.
- The type of storage domain being added (**data**, **iso**, or **export**).
- The storage format to use (**v1**, **v2**, or **v3**).

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Get the reference to the storage domains service:
sds_service = connection.system_service().storage_domains_service()

# Use the "add" method to create a new NFS storage domain:
sd = sds_service.add(
    types.StorageDomain(
        name='myiso',
        description='My ISO',
        type=types.StorageDomainType.ISO,
        host=types.Host(
            name='myhost',
        ),
        storage=types.HostStorage(
            type=types.StorageType.NFS,
            address='FQDN',
            path='/nfs/ovirt/path/to/myiso',
        ),
    ),
)

# Wait until the storage domain is unattached:
sd_service = sds_service.storage_domain_service(sd.id)
while True:
    time.sleep(5)
    sd = sd_service.get()
    if sd.status == types.StorageDomainStatus.UNATTACHED:
        break

print("Storage Domain '%s' added (%s)." % (sd.name(), sd.id()))

# Close the connection to the server:
connection.close()
```

If the **add** method call is successful, the examples output the text:

```
Storage Domain 'myiso' added (00000000-0000-0000-0000-000000000000).
```

3.10. ATTACHING A STORAGE DOMAIN TO A DATA CENTER

Once you have added a storage domain to Red Hat Virtualization, you must attach it to a data center and activate it before it will be ready for use.

Example 3.8. Attaching a storage domain to a data center

This example attaches an existing NFS storage domain, **mydata**, to the an existing data center, **Default**. The attach action is facilitated by the **add** method of the data center's **storagedomains** collection. These examples may be used to attach both data and ISO storage domains.

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

# Create the connection to the server:
connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Locate the service that manages the storage domains and use it to
# search for the storage domain:
sds_service = connection.system_service().storage_domains_service()
sd = sds_service.list(search='name=mydata')[0]

# Locate the service that manages the data centers and use it to
# search for the data center:
dcs_service = connection.system_service().data_centers_service()
dc = dcs_service.list(search='name=Default')[0]

# Locate the service that manages the data center where we want to
# attach the storage domain:
dc_service = dcs_service.data_center_service(dc.id)

# Locate the service that manages the storage domains that are attached
# to the data centers:
attached_sds_service = dc_service.storage_domains_service()

# Use the "add" method of service that manages the attached storage
# domains to attach it:
attached_sds_service.add(
    types.StorageDomain(
        id=sd.id,
    ),
)

# Wait until the storage domain is active:
attached_sd_service = attached_sds_service.storage_domain_service(sd.id)
while True:
    time.sleep(5)
    sd = attached_sd_service.get()
    if sd.status == types.StorageDomainStatus.ACTIVE:
```

```
        break

print("Attached data storage domain '%s' to data center '%s' (Status: %s)." %
  (sd.name(), dc.name(), sd.status.state()))

# Close the connection to the server:
connection.close()
```

If the calls to the **add** methods are successful, the examples output the following text:

```
Attached data storage domain 'data1' to data center 'Default' (Status: maintenance).
```

Status: maintenance indicates that the storage domains still need to be activated.

3.11. ACTIVATING A STORAGE DOMAIN

Once you have added a storage domain to Red Hat Virtualization and attached it to a data center, you must activate it before it will be ready for use.

Example 3.9. Activating a storage domain

This example activates an NFS storage domain, **mydata**, attached to the data center, **Default**. The **activate** action is facilitated by the **activate** method of the storage domain.

V4

```
import ovirtsdk4 as sdk

connection = sdk.Connection
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Locate the service that manages the storage domains and use it to
# search for the storage domain:
sds_service = connection.system_service().storage_domains_service()
sd = sds_service.list(search='name=mydata')[0]

# Locate the service that manages the data centers and use it to
# search for the data center:
dcs_service = connection.system_service().data_centers_service()
dc = dcs_service.list(search='name=Default')[0]

# Locate the service that manages the data center where we want to
# attach the storage domain:
dc_service = dcs_service.data_center_service(dc.id)

# Locate the service that manages the storage domains that are attached
# to the data centers:
attached_sds_service = dc_service.storage_domains_service()
```

```
# Activate storage domain:
attached_sd_service = attached_sds_service.storage_domain_service(sd.id)
attached_sd_service.activate()

# Wait until the storage domain is active:
while True:
    time.sleep(5)
    sd = attached_sd_service.get()
    if sd.status == types.StorageDomainStatus.ACTIVE:
        break

print("Attached data storage domain '%s' to data center '%s' (Status: %s)." %
  (sd.name(), dc.name(), sd.status.state()))

# Close the connection to the server:
connection.close()
```

If the **activate** requests are successful, the examples output the text:

```
Activated storage domain 'mydata' in data center 'Default' (Status: active).
```

Status: active indicates that the storage domains have been activated.

3.12. LISTING FILES IN AN ISO STORAGE DOMAIN

The **storagedomains** collection contains a **files** collection that describes the files in a storage domain.

Example 3.10. Listing Files in an ISO Storage Domain

This example prints a list of the ISO files in each ISO storage domain:

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

storage_domains_service = connection.system_service().storage_domains_service()

storage_domains = storage_domains_service.list()

for storage_domain in storage_domains:
    if(storage_domain.type == types.StorageDomainType.ISO):
        print(storage_domain.name + ":\n")
        files = storage_domain.files_service().list()

        for file in files:
```

```
        print("%s" % file.name + "\n")

connection.close()
```

The examples output the text:

```
ISO_storage_domain:
file1
file2
```

3.13. CREATING A VIRTUAL MACHINE

Virtual machine creation is performed in several steps. The first step, covered here, is to create the virtual machine object itself.

Example 3.11. Creating a virtual machine

This example creates a virtual machine, **vm1**, with the following requirements:

- 512 MB of memory, expressed in bytes.
- Attached to the **Default** cluster, and therefore the **Default** data center.
- Based on the default **Blank** template.
- Boots from the virtual hard disk drive.

V4

In V4, the options are added as **types**, using the **add** method.

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Get the reference to the "vms" service:
vms_service = connection.system_service().vms_service()

# Use the "add" method to create a new virtual machine:
vms_service.add(
    types.Vm(
        name='vm1',
        memory = 512*1024*1024
        cluster=types.Cluster(
            name='Default',
        ),
        template=types.Template(
            name='Blank',
```

```
        ),
        os=types.OperatingSystem(boot=types.Boot(devices=[types.BootDevice.HD)]
    ),
)

print("Virtual machine '%s' added." % vm.name)

# Close the connection to the server:
connection.close()
```

If the **add** request is successful, the examples output the text:

```
Virtual machine 'vm1' added.
```

3.14. CREATING A VIRTUAL NIC

To ensure that a newly created virtual machine has network access, you must create and attach a virtual NIC.

Example 3.12. Creating a virtual NIC

This example creates a NIC, **nic1**, and attach it to a virtual machine, **vm1**. The NIC in this example is a **virtio** network device and attached to the **ovirtmgmt** management network.

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Locate the virtual machines service and use it to find the virtual
# machine:
vms_service = connection.system_service().vms_service()
vm = vms_service.list(search='name=vm1')[0]

# Locate the service that manages the network interface cards of the
# virtual machine:
nics_service = vms_service.vm_service(vm.id).nics_service()

# Locate the vnic profiles service and use it to find the ovirmgmt
# network's profile id:
profiles_service = connection.system_service().vnic_profiles_service()
profile_id = None
for profile in profiles_service.list():
    if profile.name == 'ovirtmgmt':
        profile_id = profile.id
        break
```

```
# Use the "add" method of the network interface cards service to add the
# new network interface card:

nic = nics_service.add(
    types.Nic(
        name='nic1',
        interface=types.NicInterface.VIRTIO,
        vnic_profile=types.VnicProfile(id=profile_id),
    ),
)

print("Network interface '%s' added to '%s'." % (nic.name, vm.name))

connection.close()
```

If the **add** request is successful, the examples output the text:

```
Network interface 'nic1' added to 'vm1'.
```

3.15. CREATING A VIRTUAL MACHINE DISK

To ensure that a newly created virtual machine has access to persistent storage, you must create and attach a disk.

Example 3.13. Creating a virtual machine disk

This example creates an 8 GB **virtio** disk and attach it to a virtual machine, **vm1**. The disk has the following requirements:

- Stored on the storage domain named **data1**.
- 8 GB in size.
- **system** type disk (as opposed to **data**).
- **virtio** storage device.
- **COW** format.
- Marked as a usable boot device.

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)
```

```
# Locate the virtual machines service and use it to find the virtual
# machine:
vms_service = connection.system_service().vms_service()
vm = vms_service.list(search='name=vm1')[0]

# Locate the service that manages the disk attachments of the virtual
# machine:
disk_attachments_service = vms_service.vm_service(vm.id).disk_attachments_service()

# Use the "add" method of the disk attachments service to add the disk.
# Note that the size of the disk, the `provisioned_size` attribute, is
# specified in bytes, so to create a disk of 10 GiB the value should
# be 10 * 2^30.
disk_attachment = disk_attachments_service.add(
    types.DiskAttachment(
        disk=types.Disk(
            format=types.DiskFormat.COW,
            provisioned_size=8*1024*1024,
            storage_domains=[
                types.StorageDomain(
                    name='data1',
                ),
            ],
        ),
        interface=types.DiskInterface.VIRTIO,
        bootable=True,
        active=True,
    ),
)

# Wait until the disk status is OK:
disks_service = connection.system_service().disks_service()
disk_service = disks_service.disk_service(disk_attachment.disk.id)
while True:
    time.sleep(5)
    disk = disk_service.get()
    if disk.status == types.DiskStatus.OK:
        break

print("Disk '%s' added to '%s'." % (disk.name(), vm.name()))

# Close the connection to the server:
connection.close()
```

If the **add** request is successful, the examples output the text:

```
Disk 'vm1_Disk1' added to 'vm1'.
```

3.16. ATTACHING AN ISO IMAGE TO A VIRTUAL MACHINE

To install a guest operating system on a newly created virtual machine, you must attach an ISO file containing the operating system installation media. To locate the ISO file, see Listing Files in an ISO Storage Domain.

-

Example 3.14. Attaching an ISO image to a virtual machine

This example attaches **my_iso_file.iso** to the **vm1** virtual machine, using the **add** method of the virtual machine's **cdroms** collection.

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Get the reference to the "vms" service:
vms_service = connection.system_service().vms_service()

# Find the virtual machine:
vm = vms_service.list(search='name=vm1')[0]

# Locate the service that manages the virtual machine:
vm_service = vms_service.vm_service(vm.id)

# Locate the service that manages the CDROM devices of the virtual machine:
cdroms_service = vm_service.cdroms_service()

# Get the first CDROM:
cdrom = cdroms_service.list()[0]

# Locate the service that manages the CDROM device found in previous step:
cdrom_service = cdroms_service.cdrom_service(cdrom.id)

# Change the CD of the VM to 'my_iso_file.iso'. By default the
# operation permanently changes the disk that is visible to the
# virtual machine after the next boot, but has no effect
# on the currently running virtual machine. If you want to change the
# disk that is visible to the current running virtual machine, change
# the `current` parameter's value to `True`.
cdrom_service.update(
    cdrom=types.Cdrom(
        file=types.File(
            id='my_iso_file.iso'
        ),
    ),
    current=False,
)

print("Attached CD to '%s'." % vm.name())

# Close the connection to the server:
connection.close()
```

If the **add** request is successful, the examples output the text:

```
Attached CD to 'vm1'.
```

Example 3.15. Ejecting a cdrom from a virtual machine

This example ejects an ISO image from a virtual machine's **cdrom** collection.

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Get the reference to the "vms" service:
vms_service = connection.system_service().vms_service()

# Find the virtual machine:
vm = vms_service.list(search='name=vm1')[0]

# Locate the service that manages the virtual machine:
vm_service = vms_service.vm_service(vm.id)

# Locate the service that manages the CDROM devices of the VM:
cdroms_service = vm_service.cdroms_service()

# Get the first found CDROM:
cdrom = cdroms_service.list()[0]

# Locate the service that manages the CDROM device found in previous step
# of the VM:
cdrom_service = cdroms_service.cdrom_service(cdrom.id)

cdrom_service.remove()

print("Removed CD from '%s'." % vm.name())

connection.close()
```

If the **delete** or **remove** request is successful, the examples output the text:

```
Removed CD from 'vm1'.
```

3.17. DETACHING A DISK

You can detach a disk from a virtual machine.

Detaching a disk

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Get the reference to the "vms" service:
vms_service = connection.system_service().vms_service()

# Find the virtual machine:
vm = vms_service.list(search='name=vm1')[0]

# Locate the service that manages the virtual machine:
vm_service = vms_service.vm_service(vm.id)

attachments_service = vm_service.disk_attachments_service()
attachment = next(
    (a for a in disk_attachments if a.disk.id == disk.id), None
)

# Remove the attachment. The default behavior is that the disk is detached
# from the virtual machine, but not deleted from the system. If you wish to
# delete the disk, change the detach_only parameter to "False".
attachment.remove(detach_only=True)

print("Detached disk %s successfully!" % attachment)

# Close the connection to the server:
connection.close()
```

If the **delete** or **remove** request is successful, the examples output the text:

```
Detached disk vm1_disk1 successfully!
```

3.18. STARTING A VIRTUAL MACHINE

You can start a virtual machine.

Example 3.16. Starting a virtual machine

This example starts the virtual machine using the **start** method.

V4

```
import time
import ovirtsdk4 as sdk
import ovirtsdk4.types as types
```

```
connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Get the reference to the "vms" service:
vms_service = connection.system_service().vms_service()

# Find the virtual machine:
vm = vms_service.list(search='name=vm1')[0]

# Locate the service that manages the virtual machine, as that is where
# the action methods are defined:
vm_service = vms_service.vm_service(vm.id)

# Call the "start" method of the service to start it:
vm_service.start()

# Wait until the virtual machine is up:
while True:
    time.sleep(5)
    vm = vm_service.get()
    if vm.status == types.VmStatus.UP:
        break

print("Started '%s'." % vm.name())

# Close the connection to the server:
connection.close()
```

If the **start** request is successful, the examples output the text:

```
Started 'vm1'.
```

The **UP** status indicates that the virtual machine is running.

3.19. STARTING A VIRTUAL MACHINE WITH OVERRIDDEN PARAMETERS

You can start a virtual machine, overriding its default parameters.

Example 3.17. Starting a virtual machine with overridden parameters

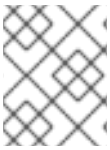
This example boots a virtual machine with a Windows ISO and attach the **virtio-win_x86.vfd** floppy disk, which contains Windows drivers. This action is equivalent to using the **Run Once** window in the Administration Portal to start a virtual machine.

V4

```
import time
```

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Get the reference to the "vms" service:
vms_service = connection.system_service().vms_service()

# Find the virtual machine:
vm = vms_service.list(search='name=vm1')[0]

# Locate the service that manages the virtual machine:
vm_service = vms_service.vm_service(vm.id)

# Locate the service that manages the CDROM devices of the virtual machine:
cdroms_service = vm_service.cdroms_service()

# Get the first CDROM:
cdrom = cdroms_service.list()[0]

# Locate the service that manages the CDROM device found in previous step:
cdrom_service = cdroms_service.cdrom_service(cdrom.id)

# Change the CD of the VM to 'windows_example.iso':
cdrom_service.update(
    cdrom=types.Cdrom(
        file=types.File(
            id='windows_example.iso'
        ),
    ),
    current=False,
)

# Call the "start" method of the service to start it:
vm_service.start(
    vm=types.Vm(
        os=types.OperatingSystem(
            boot=types.Boot(
                devices=[
                    types.BootDevice.CDROM,
                ]
            )
        ),
    )
)

# Wait until the virtual machine's status is "UP":
while True:
    time.sleep(5)
    vm = vm_service.get()
    if vm.status == types.VmStatus.UP:
```

```
        break

print("Started '%s'." % vm.name())

# Close the connection to the server:
connection.close()
```



NOTE

The CD image and floppy disk file must be available to the virtual machine. See Uploading Images to a Data Storage Domain for details.

3.20. STARTING A VIRTUAL MACHINE WITH CLOUD-INIT

You can start a virtual machine with a specific configuration, using the **Cloud-Init** tool.

Example 3.18. Starting a virtual machine with Cloud-Init

This example shows you how to start a virtual machine using the Cloud-Init tool to set a host name and a static IP for the **eth0** interface.

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Find the virtual machine:
vms_service = connection.system_service().vms_service()
vm = vms_service.list(search = 'name=vm1')[0]

# Find the service that manages the virtual machine:
vm_service = vms_service.vm_service(vm.id)

# Start the virtual machine enabling cloud-init and providing the
# password for the `root` user and the network configuration:
vm_service.start(
    use_cloud_init=True,
    vm=types.Vm(
        initialization=types.Initialization(
        user_name='root',
        root_password='password',
        host_name='MyHost.example.com',
        nic_configurations=[
            types.NicConfiguration(
                name='eth0',
                on_boot=True,
```

```
            boot_protocol=types.BootProtocol.STATIC,
            ip=types.Ip(
                version=types.IpVersion.V4,
                address='10.10.10.1',
                netmask='255.255.255.0',
                gateway='10.10.10.1'
            )
        )
    )
)

# Close the connection to the server:
connection.close()
```

3.21. CHECKING SYSTEM EVENTS

Red Hat Virtualization Manager records and logs many system events. These event logs are accessible through the user interface, the system log files, and using the API. The **ovirtsdk** library exposes events using the **events** collection.

Example 3.19. Checking system events

In this example the **events** collection is listed.

The **query** parameter of the **list** method is used to ensure that all available pages of results are returned. By default the **list** method returns only the first page of results, which is **100** records in length.

The returned list is sorted in reverse chronological order, to display the events in the order in which they occurred.

V4

```
import ovirtsdk4 as sdk
import ovirtsdk4.types as types

connection = sdk.Connection(
    url='https://engine.example.com/ovirt-engine/api',
    username='admin@internal',
    password='password',
    ca_file='ca.pem',
)

# Find the service that manages the collection of events:
events_service = connection.system_service().events_service()

page_number = 1
events = events_service.list(search='page %s' % page_number)
while events:
    for event in events:
        print(
            "%s %s CODE %s - %s" % (
                event.time,
```

```
                event.severity,
                event.code,
                event.description,
            )
        )
    page_number = page_number + 1
    events = events_service.list(search='page %s' % page_number)

# Close the connection to the server:
connection.close()
```

These examples output events in the following format:

```
YYYY-MM-DD_T_HH:MM:SS NORMAL CODE 30 - User admin@internal logged in.
YYYY-MM-DD_T_HH:MM:SS NORMAL CODE 153 - VM vm1 was started by admin@internal
(Host: MyHost).
YYYY-MM-DD_T_HH:MM:SS NORMAL CODE 30 - User admin@internal logged in.
```

APPENDIX A. LEGAL NOTICE