



Red Hat Virtualization 4.1

Ruby SDK Guide

Using the Red Hat Virtualization Ruby SDK

Red Hat Virtualization 4.1 Ruby SDK Guide

Using the Red Hat Virtualization Ruby SDK

Red Hat Virtualization Documentation Team

Red Hat Customer Content Services

rhev-docs@redhat.com

Legal Notice

Copyright © 2017 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution-Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide describes how to install and work with the Red Hat Virtualization Ruby software development kit.

Table of Contents

| | |
|--|-----------|
| CHAPTER 1. OVERVIEW | 3 |
| 1.1. PREREQUISITES | 3 |
| 1.2. INSTALLING THE RUBY SOFTWARE DEVELOPMENT KIT | 3 |
| 1.3. DEPENDENCIES | 3 |
| CHAPTER 2. USING THE SOFTWARE DEVELOPMENT KIT | 5 |
| 2.1. CLASSES | 5 |
| 2.2. TYPES | 6 |
| 2.2.1. Creating and Modifying Instances of Types | 6 |
| 2.2.2. Retrieving Instance Attributes | 7 |
| 2.3. SERVICES | 8 |
| 2.3.1. Retrieving Services | 8 |
| 2.3.2. Service Methods | 9 |
| 2.3.3. Get | 9 |
| 2.3.4. List | 10 |
| 2.3.5. Add | 11 |
| 2.3.6. Update | 12 |
| 2.3.7. Remove | 14 |
| 2.3.8. Additional Actions | 14 |
| CHAPTER 3. RUBY EXAMPLES | 16 |
| 3.1. CONNECTING TO THE RED HAT VIRTUALIZATION MANAGER | 16 |
| 3.2. LISTING DATA CENTERS | 16 |
| 3.3. LISTING CLUSTERS | 17 |
| 3.4. LISTING LOGICAL NETWORKS | 17 |
| 3.5. LISTING HOSTS | 18 |
| 3.6. LISTING ISO FILES IN THE ISO STORAGE DOMAIN | 18 |
| 3.7. CREATING AN NFS STORAGE DOMAIN | 19 |
| 3.8. ATTACHING A STORAGE DOMAIN TO A DATA CENTER | 20 |
| 3.9. CREATING A VIRTUAL MACHINE | 21 |
| 3.10. CREATING A VNIC PROFILE | 21 |
| 3.11. CREATING A VNIC | 22 |
| 3.12. CREATING A VIRTUAL DISK | 23 |
| 3.13. ATTACHING AN ISO IMAGE TO A VIRTUAL MACHINE | 24 |
| 3.14. DETACHING A VIRTUAL DISK | 24 |
| 3.15. STARTING A VIRTUAL MACHINE | 25 |
| 3.16. STARTING A VIRTUAL MACHINE WITH SPECIFIC BOOT DEVICES AND BOOT ORDER | 25 |
| 3.17. STARTING A VIRTUAL MACHINE WITH CLOUD-INIT | 26 |
| 3.18. CHECKING SYSTEM EVENTS | 27 |

CHAPTER 1. OVERVIEW

The Ruby software development kit is a Ruby gem that allows you to interact with the Red Hat Virtualization Manager in Ruby 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.

1.1. PREREQUISITES

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

- A system with Red Hat Enterprise Linux 7 installed. Both the Server and Workstation variants are supported.
- A subscription to Red Hat Virtualization entitlements.

1.2. INSTALLING THE RUBY SOFTWARE DEVELOPMENT KIT

1. Enable the required repositories:

```
# subscription-manager repos --enable=rhel-7-server-rpms
# subscription-manager repos --enable=rhel-7-server-rhv-4.1-rpms
```

2. Install the Ruby Software Development Kit:

```
# yum install rubygem-ovirt-engine-sdk4
```



NOTE

You can also install the Ruby Software Development Kit using **gem**:

```
# gem install ovirt-engine-sdk
```

1.3. DEPENDENCIES

The Ruby Software Development Kit has the following dependencies, which you must install manually if you are using **gem**:

- **libxml2** for parsing and rendering XML
- **libcurl** for HTTP transfers
- C compiler
- Required header and library files



NOTE

You do not need to install the dependency files if you installed the RPM.

Install the dependency files:

```
# yum install gcc libcurl-devel libxml2-devel ruby-devel
```



NOTE

If you are using Fedora or CentOS, use **dnf**:

```
# dnf install gcc libcurl-devel libxml2-devel ruby-devel
```

If you are using Debian or Ubuntu, use **apt-get**:

```
# apt-get install gcc libxml2-dev libcurl-dev ruby-dev
```


CHAPTER 2. USING THE SOFTWARE DEVELOPMENT KIT

This chapter defines the modules and classes of the Ruby Software Development Kit and describes their usage.

2.1. CLASSES

The [OvirtSDK4](#) module contains the following software development kit classes:

Connection

The [Connection](#) class is the mechanism for connecting to the server and obtaining the reference to the root of the services tree. See [Section 3.1, “Connecting to the Red Hat Virtualization Manager”](#) for details.

Types

The Type classes implement the types supported by the API. For example, the [Vm](#) class is the implementation of the virtual machine type. The classes are data containers and do not contain any logic. You will be working with instances of types.

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 software development kit.

Services

The Service classes implement the services supported by the API. For example, the [VmsService](#) class is the implementation of the service that manages the collection of virtual machines in the system.

Instances of these classes are automatically created by the SDK when a service is referenced. For example, a new instance of the [VmsService](#) class is created automatically by the SDK when you call the `vms_service` method of the [SystemService](#) class:

```
vms_service = connection.system_service.vms_service
```



WARNING

Do not create instances of these classes manually. The constructor parameters and methods may change in the future.

Error

The [Error](#) class is the base exception class that the software development kit raises when it reports an error.

Certain specific error classes extend the base error class:

- [AuthError](#) - Authentication or authorization failure
- [ConnectionError](#) - Server name cannot be resolved or server is unreachable
- [NotFoundError](#) - Requested object does not exist

- `TimeoutError` - Operation time-out

Other Classes

Other classes (for example, HTTP client classes, readers, and writers) are used for HTTP communication and for XML parsing and rendering. Using these classes is not recommended, because they comprise internal implementation details that may change in the future. Their backwards-compatibility cannot be relied upon.

2.2. TYPES

2.2.1. Creating and Modifying Instances of Types

Creating or modifying an instance of a type does not have any effect on the server side, unless the changes are explicitly sent to the server calling a service method, as described below. Changes on the server side are not automatically reflected in instances that already exist in memory.

The constructors of these classes have multiple optional arguments, one for each attribute of the type. This simplifies the creation of objects, by using nested calls to multiple constructors.

In the following example, an instance of a virtual machine is created and its attributes (cluster, template, and memory) are set:

Creating a Virtual Machine Instance with Attributes

```
vm = OvirtSDK4::Vm.new(  
  name: 'myvm',  
  cluster: OvirtSDK4::Cluster.new(  
    name: 'mycluster'  
  ),  
  template: OvirtSDK4::Template.new(  
    name: 'mytemplate'  
  ),  
  memory: 1073741824  
)
```

The hashes (for example, `cluster: OvirtSDK4::Cluster.new`) passed to these constructors are processed recursively.

In the following example, plain hashes are used instead of explicitly calling the constructors for the `Cluster` and `Template` classes. The SDK internally converts the hashes to the required classes.

Creating a Virtual Machine Instance with Attributes Expressed as Plain Hashes

```
vm = OvirtSDK4::Vm.new(  
  name: 'myvm',  
  cluster: {  
    name: 'mycluster'  
  },  
  template: {  
    name: 'mytemplate'  
  },  
  memory: 1073741824  
)
```

Using constructors in this way is recommended, but not mandatory.

In the following example, a virtual machine instance is created with no arguments in the call to the constructor. You can add the virtual machine instance's attributes one by one, using setters, or by using a combination of setters and constructors.

Creating a Virtual Machine Instance and Adding Attributes Individually

```
vm = OvirtSDK4::Vm.new
vm.name = 'myvm'
vm.cluster = OvirtSDK4::Cluster.new(name: 'mycluster')
vm.template = OvirtSDK4::Template.new(name: 'mytemplate')
vm.memory = 1073741824
```

Attributes that are defined as lists of objects in the API specification are implemented as Ruby arrays. For example, the `custom_properties` attributes of the `Vm` type are defined as a list of objects of type `CustomProperty`.

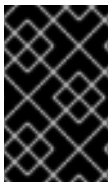
Adding a List of Attributes as an Array

```
vm = OvirtSDK4::Vm.new(
  name: 'myvm',
  custom_properties: [
    OvirtSDK4::CustomProperty.new(...),
    OvirtSDK4::CustomProperty.new(...),
    ...
  ]
)
```

Attributes that are defined as enumerated values in the API are implemented as constants in a module with the same name as the enumerated type.

The following example shows how the status attribute of the `Vm` type is defined using the `VmStatus` enumerated value.

```
case vm.status
when OvirtSDK4::VmStatus::DOWN
  ...
when OvirtSDK4::VmStatus::IMAGE_LOCKED
  ...
end
```



IMPORTANT

In the API specification, the values of `enum` types are lower case, because that is the convention for XML and JSON. In Ruby, however, the convention is to use **upper case** for these constants.

2.2.2. Retrieving Instance Attributes

You can retrieve instance attributes using the corresponding attribute readers.

The following example retrieves the name and memory of the virtual machine instance:

Retrieving Virtual Machine Instance Attributes

```
puts "vm.name: #{vm.name}"
puts "vm.memory: #{vm.memory}"
vm.custom_properties.each do |custom_property|
  ...
end
```

Retrieving Instance Attributes as Links

Some instance attributes are returned as links and require the `follow_link` method to retrieve the data. In the following example, the response to a request for a virtual machine's attributes is formatted as XML with a link:

Retrieving Virtual Machine Attributes as a Link

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

The link, `vm.disk_attachments`, does not contain the actual disk attachments. To retrieve the data, the [Connection](#) class provides a [follow_link](#) method that uses the value of the `href` XML attribute to retrieve the actual data.

In the following example, `follow_link` enables you to go to the disk attachments, and then to each disk, to retrieve the `alias`:

Retrieving Virtual Machine Service

```
vm = vm_service.get
```

Using `follow_link` to Retrieve Disk Attachment and Disk Alias

```
attachments = connection.follow_link(vm.disk_attachments)
attachments.each do |attachment|
  disk = connection.follow_link(attachment.disk)
  puts "disk.alias: #{disk.alias}"
end
```

2.3. SERVICES

2.3.1. Retrieving Services

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

In the Ruby software development kit, the root of that tree of services is implemented by the `system service`. It is retrieved by calling the [system_service](#) method of the connection:

Retrieving System Service

```
system_service = connection.system_service
```

Once you have the reference to the `system_service`, you can use it to retrieve references to other services, using the `*_service` methods (called **service locators**).

For example, to retrieve a reference to the service that manages the collection of virtual machines in the system, you can use the `vms_service` service locator:

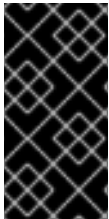
Retrieving Other Services

```
vms_service = system_service.vms_service
```

To retrieve a reference to the service that manages the virtual machine with identifier **123**, use the service locator of the `vm_service` service. The service locator uses the virtual machine identifier as a parameter:

Retrieving Virtual Machine Service Using Identifier

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



IMPORTANT

The objects returned by the service locator calls are pure services, and do not contain data. For example, the `vm_service` Ruby object retrieved in the previous example is not the representation of a virtual machine. It is the service that is used to retrieve, update, delete, start, and stop a virtual machine.

2.3.2. Service Methods

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

Services that manage collections of objects usually have the `list` and `add` methods.

Services that manage a single object usually have the `get`, `update`, and `remove` methods.

Services may have additional action methods, which perform actions other than retrieving, creating, updating, or removing. These methods are most commonly found in services that manage a single object.

2.3.3. Get

The `get` method retrieves the representation of a single object.

The following example locates and 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 result will be an instance of the corresponding type. In this case, the result is an instance of the Ruby class [Vm](#).

The `get` method of some services supports additional parameters that control how to retrieve the representation of the object, or which representation to retrieve, if there is more than one.

For example, you may want to retrieve a virtual machine's future state, after boot-up. The `get` method of the service that manages a virtual machine supports a `next_run` Boolean parameter:

Retrieving a Virtual Machine's `next_run` State

```
# 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 software development kit for details.

If the object cannot be retrieved, the software development kit will raise an [Error](#) exception, containing details of the failure. This will occur if you try to retrieve a non-existent object.



NOTE

The call to the `service locator` method never fails, even if the object does not exist, because the `service locator` method does not send a request to the server.

In the following examples, the `service locator` method will succeed, while the `get` method will raise an exception:

Locating the Service of Non-existent Virtual Machine: No Error

```
# Find the service that manages a virtual machine that does
# not exist. This will succeed.
vm_service = vms_service.vm_service('non_existent_VM')
```

Retrieving a Non-existent Virtual Machine Service: Error

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

2.3.4. List

The `list` method retrieves the representations of multiple objects in a collection.

Listing a Collection of Virtual Machines

```
# Find the service that manages the collection of virtual
# machines:
```

```
vms_service = system_service.vms_service
vms = vms_service.list
```

The result is a Ruby array containing the instances of the corresponding types. In the above example, the response is a list of instances of the Ruby class [Vm](#).

The `list` method of some services supports additional parameters.

For example, almost all of the top-level collections support a `search` parameter to filter the results, and a `max` parameter to limit the number of results returned by the server.

Listing Ten Virtual Machines Called "my*"

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



NOTE

Not all the list methods support the `search` or `max` parameters. Some list methods may support other parameters. See the [reference](#) documentation for details.

If the list of results is empty, the returned value will be an empty Ruby array. It will never be `nil`.

If the list of results cannot be retrieved, the SDK will raise an [Error](#) exception containing the details of the failure.

2.3.5. Add

Add methods add new elements to collections. 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.

Adding a New Virtual Machine

```
# Add the virtual machine:
vm = vms_service.add(
  OvirtSDK4::Vm.new(
    name: 'myvm',
    cluster: {
      name: 'mycluster'
    },
    template: {
      name: 'mytemplate'
    }
  )
)
```



IMPORTANT

The Ruby object returned by the `add` method is an instance of the relevant type. It is not a service, just a container of data. In the above example, the returned object is an instance of the [Vm](#) class.

If you need to perform an action on the virtual machine you just added, you must locate the service that manages it and call the service locator:

Starting a New Virtual Machine

```
# 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
```

The creation of most objects is an asynchronous task. For example, if you create a new virtual machine, the `add` method will return the virtual machine before the virtual machine is completely created and ready to be used. You should poll the status of the object until it is completely created. For a virtual machine that means checking until the status is **DOWN**.

The recommended approach is to create a virtual machine, locate the service that manages the new virtual machine, and retrieve the status repeatedly until the virtual machine status is **DOWN**, indicating that all the disks have been created.

Adding a Virtual Machine, Locating Its Service, and Retrieving Its Status

```
# 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 all the
# disks have been created:
loop do
  sleep(5)
  vm = vm_service.get
  break if vm.status == OvirtSDK4::VmStatus::DOWN
end
```

If the object cannot be created, the SDK will raise an [Error](#) exception containing the details of the failure. It will never return `nil`.

2.3.6. Update

Update 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.

**NOTE**

The Ruby object returned by this update method is an instance of the relevant type. It is not a service, just a container of data. In this particular example the returned object will be an instance of the [Vm](#) class.

In the following example, the `service_locator` method locates the service managing the virtual machine and the `update` method updates its name:

Updating a Virtual Machine Name

```
# Find the virtual machine and the service that
# manages it:
vm = vms_service.list(search: 'name=myvm').first
vm_service = vms_service.vm_service(vm.id)

# Update the name:
updated_vm = vms_service.update(
  OvirtSDK4::Vm.new(
    name: 'newvm'
  )
)
```

When you update an object, update only the attributes you want to update:

Updating a Selected Attribute of a Virtual Machine (Recommended)

```
vm = vm_service.get
vm.name = 'newvm'
```

Do not update the entire object:

Updating All Attributes of a Virtual Machine (Not Recommended)

```
# Retrieve the current representation:
vms_service.update(vm)
```

Updating all attributes of the virtual machine is a waste of resources and can introduce unexpected bugs on the server side.

Update methods of some services support additional parameters that can be used to control how or what to update. For example, you may want to update the memory of a virtual machine, not in its current state, but the next time it is started. The `update` method of the service that manages a virtual machine supports a [next_run](#) Boolean parameter:

Updating the Memory of a Virtual Machine at Next Run

```
vm = vm_service.update(
  OvirtSDK4::Vm.new(
    memory: 1073741824
  ),
  next_run: true
)
```

If the update cannot be performed, the SDK will raise an [Error](#) exception containing the details of the failure. It will never return `nil`.

2.3.7. Remove

Remove methods remove existing objects. They normally do not support parameters because they are methods of services that manage single objects, and the service already knows what object to remove.

Removing a Virtual Machine with Identifier 123

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

Some **remove** methods support 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:

Removing a Virtual Machine while Preserving Disks

```
vm_service.remove(detach_only: true)
```

The **remove** method returns `nil` if the object is removed successfully. It does not return the removed object.

If the object cannot be removed, the SDK will raise an [Error](#) exception containing the details of the failure.

2.3.8. Additional Actions

There are additional action methods, apart from the methods described above. The service that manages a virtual machine has methods to start and stop it.

Starting a Virtual Machine

```
vm_service.start
```

Some action methods include parameters that modify the operation. For example, the **start** method supports a [use_cloud_init](#) parameter.

Starting a Virtual Machine with Cloud-Init

```
vm_service.start(use_cloud_init: true)
```

Most action methods return `nil` when they succeed, and raise an [Error](#) when they fail. Some action methods, however, return values. For example, the service that manages storage domains has an **is_attached** action method that checks whether the storage domain is already attached to a data center. The **is_attached** action method returns a Boolean value:

Checking for Attached Storage Domain

```
sds_service = system_service.storage_domains_service
sd_service = sds_service.storage_domain_service('123')
```

```
if sd_service.is_attached
  ...
end
```

See the [reference documentation](#) of the software development kit to see the action methods supported by each service, their parameters, and return values.

CHAPTER 3. RUBY EXAMPLES

3.1. CONNECTING TO THE RED HAT VIRTUALIZATION MANAGER

The `Connection` class is the entry point of the software development kit. It provides access to the services of the Red Hat Virtualization Manager's REST API.

The parameters of the `Connection` class are:

- `url` - Base URL of the Red Hat Virtualization Manager API
- `username`
- `password`
- `ca_file` - PEM file containing the trusted CA certificates. The `ca.pem` file is required when connecting to a server protected by TLS. If you do not specify the `ca_file`, the system-wide CA certificate store is used.

Connecting to the Red Hat Virtualization Manager

```
connection = OvirtSDK4::Connection.new(
  url: 'https://engine.example.com/ovirt-engine/api',
  username: 'admin@internal',
  password: '...',
  ca_file: 'ca.pem',
)
```

IMPORTANT

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

```
connection.close
```

The connection, and all the services obtained from it, cannot be used after the connection has been closed.

If the connection fails, the software development kit will raise an `Error` exception, containing details of the failure.

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/Connection:initialize>.

3.2. LISTING DATA CENTERS

This Ruby example lists the data centers.

```
# Get the reference to the root of the services tree:
system_service = connection.system_service
```

```
# Get the reference to the service that manages the
# collection of data centers:
dcs_service = system_service.data_centers_service

# Retrieve the list of data centers and for each one
# print its name:
dcs = dcs_service.list
dcs.each do |dc|
  puts dc.name
end
```

In an environment with only the Default data center, the example outputs:

```
Default
```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/DataCentersService:list>.

3.3. LISTING CLUSTERS

This Ruby example lists the clusters.

```
# Get the reference to the root of the services tree:
system_service = connection.system_service

# Get the reference to the service that manages the
# collection of clusters:
cls_service = system_service.clusters_service

# Retrieve the list of clusters and for each one
# print its name:
cls = cls_service.list
cls.each do |cl|
  puts cl.name
end
```

In an environment with only the Default cluster, the example outputs:

```
Default
```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/ClustersService:list>.

3.4. LISTING LOGICAL NETWORKS

This Ruby example lists the logical networks.

```
# Get the reference to the root of the services tree:
system_service = connection.system_service

# Get the reference to the service that manages the
# collection of networks:
nws_service = system_service.networks_service
```

```
# Retrieve the list of clusters and for each one
# print its name:
nws = nws_service.list
nws.each do |nw|
  puts nw.name
end
```

In an environment with only the default management network, the example outputs:

```
ovirtmgmt
```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/NetworksService:list>.

3.5. LISTING HOSTS

This Ruby example lists the hosts.

```
# Get the reference to the root of the services tree:
system_service = connection.system_service

# Get the reference to the service that manages the
# collection of hosts:
host_service = system_service.hosts_service

# Retrieve the list of hosts and for each one
# print its name:
host = host_service.list
host.each do |host|
  puts host.name
end
```

In an environment with only one attached host (**Atlantic**) the example outputs:

```
Atlantic
```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/HostsService:list>.

3.6. LISTING ISO FILES IN THE ISO STORAGE DOMAIN

This Ruby example lists the ISO files in the ISO storage domain, **myiso**.

```
# Get the reference to the root of the services tree:
system_service = connection.system_service

# Find the service that manages the collection of storage domains:
sds_service = system_service.storage_domains_service

# Find the ISO storage domain:
sd = sds_service.list(search: 'name=myiso').first

# Find the service that manages the ISO storage domain:
```

```
sd_service = sds_service.storage_domain_service(sd.id)

# Find the service that manages the collection of files available in the
# storage domain:
files_service = sd_service.files_service

# List the names of the files. Note that the name of the .iso file is
# contained
# in the id attribute.
files = files_service.list
files.each do |file|
  puts file.id
end
```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4%2FFilesService:list>.

If you do not know the name of the ISO storage domain, this Ruby example retrieves all the ISO storage domains.

```
# Find the ISO storage domain:
iso_sds = sds_service.list.select { |sd| sd.type ==
OvirtSDK4::StorageDomainType::ISO }
```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/StorageDomainsService:list>.

3.7. CREATING AN NFS STORAGE DOMAIN

This Ruby example adds an NFS storage domain.

```
# Get the reference to the root of the services tree:
system_service = connection.system_service

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

# Create a new NFS data storage domain:
sd = sds_service.add(
  OvirtSDK4::StorageDomain.new(
    name: 'mydata',
    description: 'My data',
    type: OvirtSDK4::StorageDomainType::DATA,
    host: {
      name: 'myhost'
    },
    storage: {
      type: OvirtSDK4::StorageType::NFS,
      address: 'server0.example.com',
      path: '/nfs/ovirt/40/mydata'
    }
  )
)

# Wait until the storage domain is unattached:
```

```
sd_service = sds_service.storage_domain_service(sd.id)
loop do
  sleep(5)
  sd = sd_service.get
  break if sd.status == OvirtSDK4::StorageDomainStatus::UNATTACHED
end
```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/StorageDomainsService:add>.

3.8. ATTACHING A STORAGE DOMAIN TO A DATA CENTER

This Ruby example attaches an existing NFS storage domain, `mydata`, to an existing data center, `mydc`. This example is used to attach both data and ISO storage domains.

```
# Get the reference to the root of the services tree:
system_service = connection.system_service

# Locate the service that manages the storage domains and use it to
# search for the storage domain:
sds_service = 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 = system_service.data_centers_service
dc = dcs_service.list(search: 'name=mydc')[0]

# Locate the service that manages the data center where you 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(
  OvirtSDK4::StorageDomain.new(
    id: sd.id
  )
)

# Wait until the storage domain is active:
attached_sd_service = attached_sds_service.storage_domain_service(sd.id)
loop do
  sleep(5)
  sd = attached_sd_service.get
  break if sd.status == OvirtSDK4::StorageDomainStatus::ACTIVE
end
```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/StorageDomainsService:add>.

3.9. CREATING A VIRTUAL MACHINE

This Ruby example creates a virtual machine. This example uses a hash with symbols and nested hashes as their values. Another method, more verbose, is to use the constructors of the corresponding objects directly. See [Creating a Virtual Machine Instance with Attributes](#) for more information.

```
# 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(
  OvirtSDK4::Vm.new(
    name: 'myvm',
    cluster: {
      name: 'mycluster'
    },
    template: {
      name: 'Blank'
    }
  )
)
```

After creating a virtual machine, it is recommended to [poll the virtual machine's status](#), to ensure that all the disks have been created.

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/VmsService:add>.

3.10. CREATING A VNIC PROFILE

This Ruby example creates a vNIC profile.

```
# Find the root of the tree of services:
system_service = connection.system_service

# Find the network where you want to add the profile. There may be
# multiple
# networks with the same name (in different data centers, for example).
# Therefore, you must look up a specific network by name, in a specific
# data center.
dcs_service = system_service.data_centers_service
dc = dcs_service.list(search: 'name=mydc').first
networks = connection.follow_link(dc.networks)
network = networks.detect { |n| n.name == 'mynetwork' }

# Create the vNIC profile, with pass-through and port mirroring disabled:
profiles_service = system_service.vnic_profiles_service
profiles_service.add(
  OvirtSDK4::VnicProfile.new(
    name: 'myprofile',
    pass_through: {
      mode: OvirtSDK4::VnicPassThroughMode::DISABLED,
    },
    port_mirroring: false,
    network: {
```

```

        id: network.id
      }
    )
  )

```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/VnicProfilesService:add>.

3.11. CREATING A VNIC

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

This Ruby example creates a vNIC and attaches it to an existing virtual machine, `myvm`.

```

# Find the root of the tree of services:
system_service = connection.system_service

# Find the virtual machine:
vms_service = system_service.vms_service
vm = vms_service.list(search: 'name=myvm').first

# In order to specify the network that the new NIC will be connected to,
# you must
# specify the identifier of the vNIC profile. However, there may be
# multiple
# profiles with the same name (for different data centers, for example),
# so first
# you must find the networks that are available in the cluster that the
# virtual machine belongs to.
cluster = connection.follow_link(vm.cluster)
networks = connection.follow_link(cluster.networks)
network_ids = networks.map(&:id)

# Now that you know what networks are available in the cluster, you can
# select a
# vNIC profile that corresponds to one of those networks, and has the
# name that you want to use. The system automatically creates a vNIC
# profile for each network, with the same name as the network.
profiles_service = system_service.vnic_profiles_service
profiles = profiles_service.list
profile = profiles.detect { |p| network_ids.include?(p.network.id) &&
  p.name == 'myprofile' }

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

# Add the new network interface card:
nics_service.add(
  OvirtSDK4::Nic.new(
    name: 'mynic',
    description: 'My network interface card',
    vnic_profile: {
      id: profile.id
    }
  )
)

```

```

    }
  )
)

```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/VmsService:add>.

3.12. CREATING A VIRTUAL DISK

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

This Ruby example creates and attaches a virtual storage disk to a virtual machine.

```

# 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=myvm')[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(
  OvirtSDK4::DiskAttachment.new(
    disk: {
      name: 'mydisk',
      description: 'My disk',
      format: OvirtSDK4::DiskFormat::COW,
      provisioned_size: 10 * 2**30,
      storage_domains: [{
        name: 'mydata'
      }]
    },
    interface: OvirtSDK4::DiskInterface::VIRTIO,
    bootable: false,
    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)
loop do
  sleep(5)
  disk = disk_service.get
  break if disk.status == OvirtSDK4::DiskStatus::OK
end

```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/DiskAttachmentsService:add>.

3.13. ATTACHING AN ISO IMAGE TO A VIRTUAL MACHINE

This Ruby example attaches a CD-ROM to a virtual machine and changes it to an ISO image in order to install the guest operating system.

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

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

# List the first CDRom device:
cdrom = cdroms_service.list[0]

# Locate the service that manages the CDRom device you just found:
cdrom_service = cdroms_service.cdrom_service(cdrom.id)

# Change the CD of the VM to 'my_iso_file.iso'. By default this
# operation permanently changes the disk that is visible to the
# virtual machine after the next boot, but it does not have any 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(
  OvirtSDK4::Cdrom.new(
    file: {
      id: 'CentOS-7-x86_64-DVD-1511.iso'
    }
  ),
  current: false
)
```

For more information, see http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4%2FVmService:cdroms_service.

3.14. DETACHING A VIRTUAL DISK

This Ruby example detaches a disk from a virtual machine.

```
# Find the virtual machine:
vm = vms_service.list(search: 'name=myvm').first

# Detach the first disk from the virtual machine:
vm_service = vms_service.vm_service(vm.id)
attachments_service = vm_service.disk_attachments_service
```

```
attachment = attachments_service.list.first

# 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)
```

For more information, see http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/VmService:disk_attachments_service.

3.15. STARTING A VIRTUAL MACHINE

This Ruby example starts a virtual machine.

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

# Find the virtual machine:
vm = vms_service.list(search: 'name=myvm')[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 status is UP:
loop do
  sleep(5)
  vm = vm_service.get
  break if vm.status == OvirtSDK4::VmStatus::UP
end
```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/VmService:start>.

3.16. STARTING A VIRTUAL MACHINE WITH SPECIFIC BOOT DEVICES AND BOOT ORDER

This Ruby example starts a virtual machine specifying the boot devices and boot order.

```
# Find the root of the tree of services:
system_service = connection.system_service

# Find the virtual machine:
vms_service = system_service.vms_service
vm = vms_service.list(search: 'name=myvm').first

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

# Start the virtual machine explicitly indicating the boot devices and
```

```

order:
vm_service.start(
  vm: {
    os: {
      boot: {
        devices: [
          OvirtSDK4::BootDevice::NETWORK,
          OvirtSDK4::BootDevice::CDROM
        ]
      }
    }
  }
)

# Wait until the virtual machine is up:
loop do
  sleep(5)
  vm = vm_service.get
  break if vm.status == OvirtSDK4::VmStatus::UP
end

```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/BootDevice>.

3.17. STARTING A VIRTUAL MACHINE WITH CLOUD-INIT

This Ruby example starts a virtual machine using the Cloud-Init tool to set the root password and network configuration.

```

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

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

# Create a cloud-init script to execute in the
# deployed virtual machine. The script must be correctly
# formatted and indented because it uses YAML.
my_script = "
write_files:
  - content: |
      Hello, world!
    path: /tmp/greeting.txt
    permissions: '0644'
"

# 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: {
    initialization: {
      user_name: 'root',
      root_password: 'redhat123',
      host_name: 'myvm.example.com',

```

```

    nic_configurations: [
      {
        name: 'eth0',
        on_boot: true,
        boot_protocol: OvirtSDK4::BootProtocol::STATIC,
        ip: {
          version: OvirtSDK4::IpVersion::V4,
          address: '192.168.0.100',
          netmask: '255.255.255.0',
          gateway: '192.168.0.1'
        }
      }
    ],
    dns_servers: '192.168.0.1 192.168.0.2 192.168.0.3',
    dns_search: 'example.com',
    custom_script: my_script
  }
}
)

```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4/VmService:start>.

3.18. CHECKING SYSTEM EVENTS

This Ruby example retrieves logged system events.

```

# In order to ensure that no events are lost, it is recommended to write
# the index of the last processed event, in persistent storage.
# Here, it is stored in a file called index.txt. In a production
# environment,
# it will likely be stored in a database.
INDEX_TXT = 'index.txt'.freeze

def write_index(index)
  File.open(INDEX_TXT, 'w') { |f| f.write(index.to_s) }
end

def read_index
  return File.read(INDEX_TXT).to_i if File.exist?(INDEX_TXT)
  nil
end

# This is the function that is called to process the events. It prints
# the identifier and description of each event.
def process_event(event)
  puts("#{event.id} - #{event.description}")
end

# Find the root of the tree of services:
system_service = connection.system_service

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

```

```
# If no index is stored yet, retrieve the last event and start with it.
# Events are ordered by index, in ascending order. max=1 retrieves only
# one event,
# the last event.
unless read_index
  events = events_service.list(max: 1)
  unless events.empty?
    first = events.first
    process_event(first)
    write_index(first.id.to_i)
  end
end

# This loop retrieves the events, always starting from the last index. It
# waits
# before repeating. The from parameter specifies that you want to retrieve
# events that are newer than the last index that was processed. Note: the
# max
# parameter is not used, so that all pending events will be retrieved.
loop do
  sleep(5)
  events = events_service.list(from: read_index)
  events.each do |event|
    process_event(event)
    write_index(event.id.to_i)
  end
end
```

For more information, see <http://www.rubydoc.info/gems/ovirt-engine-sdk/OvirtSDK4%2FEventsService:list>.