



Citrix XenServer ® 6.1.0 Software Development Kit

Published Tuesday, 25 September 2012
1.0 Edition



Citrix XenServer ® 6.1.0 Software Development Kit

Copyright © 2012 Citrix Systems, Inc. All Rights Reserved.
Version: 6.1.0

Citrix, Inc.
851 West Cypress Creek Road
Fort Lauderdale, FL 33309
United States of America

Disclaimers

This document is furnished "AS IS." Citrix, Inc. disclaims all warranties regarding the contents of this document, including, but not limited to, implied warranties of merchantability and fitness for any particular purpose. This document may contain technical or other inaccuracies or typographical errors. Citrix, Inc. reserves the right to revise the information in this document at any time without notice. This document and the software described in this document constitute confidential information of Citrix, Inc. and its licensors, and are furnished under a license from Citrix, Inc.

Citrix Systems, Inc., the Citrix logo, Citrix XenServer and Citrix XenCenter, are trademarks of Citrix Systems, Inc. and/or one or more of its subsidiaries, and may be registered in the United States Patent and Trademark Office and in other countries. All other trademarks and registered trademarks are property of their respective owners.

Trademarks

Citrix®
XenServer ®
XenCenter ®



Contents

1. Introduction	1
2. Getting Started	2
2.1. System Requirements and Preparation	2
2.2. Downloading	2
2.3. What's New	2
2.4. Content Map	2
3. Overview of the XenServer API	4
3.1. Getting Started with the API	4
3.1.1. Authentication: acquiring a session reference	4
3.1.2. Acquiring a list of templates to base a new VM installation on	4
3.1.3. Installing the VM based on a template	5
3.1.4. Taking the VM through a start/suspend/resume/stop cycle	5
3.1.5. Logging out	5
3.1.6. Install and start example: summary	6
3.2. Object Model Overview	6
3.3. Working with VIFs and VBDs	8
3.3.1. Creating disks and attaching them to VMs	8
3.3.1.1. Creating a new blank disk image	8
3.3.1.2. Attaching the disk image to a VM	9
3.3.1.3. Hotplugging the VBD	10
3.3.2. Creating and attaching Network Devices to VMs	10
3.3.3. Host configuration for networking and storage	10
3.3.3.1. Host storage configuration: PBDs	10
3.3.3.2. Host networking configuration: PIFs	11
3.4. Exporting and Importing VMs	11
3.4.1. Xen Virtual Appliance (XVA) VM Import Format	12
3.5. XML-RPC notes	15
3.5.1. Datetimes	15
3.6. Where to look next	15

4. Using the API	16
4.1. Anatomy of a typical application	16
4.1.1. Choosing a low-level transport	16
4.1.2. Authentication and session handling	16
4.1.3. Finding references to useful objects	17
4.1.4. Invoking synchronous operations on objects	17
4.1.5. Using Tasks to manage asynchronous operations	18
4.1.6. Subscribing to and listening for events	18
4.2. Language bindings	19
4.2.1. C	19
4.2.2. C#	19
4.2.3. Java	20
4.2.4. PowerShell	20
4.2.5. Python	21
4.2.6. Command Line Interface (CLI)	21
4.3. Complete application examples	22
4.3.1. Simultaneously migrating VMs using XenMotion	22
4.3.2. Cloning a VM using the XE CLI	23
5. Using HTTP to interact with XenServer	25
5.1. VM Import and Export	25
5.2. Getting XenServer Performance Statistics	25
6. XenServer API extensions	27
6.1. VM console forwarding	27
6.1.1. Retrieving VNC consoles using the API	27
6.1.2. Disabling VNC forwarding for Linux VM	28
6.2. Paravirtual Linux installation	29
6.2.1. Red Hat Enterprise Linux 4.1/4.4	29
6.2.2. Red Hat Enterprise Linux 4.5/5.0	29
6.2.3. SUSE Enterprise Linux 10 SP1	29
6.2.4. CentOS 4.5 / 5.0	29



6.3. Adding Xenstore entries to VMs	29
6.4. Security enhancements	30
6.5. Advanced settings for network interfaces	30
6.5.1. ethtool settings	30
6.5.2. Miscellaneous settings	31
6.6. Internationalization for SR names	32
6.7. Hiding objects from XenCenter	32
7. XenCenter API Extensions	33
7.1. Pool	33
7.2. Host	33
7.3. VM	34
7.4. SR	35
7.5. VDI	35
7.6. VBD	36
7.7. Network	36
7.8. VM_guest_metrics	36
7.9. Task	36



Chapter 1. Introduction

Welcome to the developer's guide for XenServer. Here you will find the information you need in order to understand and use the Software Development Kit (SDK) that XenServer provides. This information will provide you with some of the architectural background and thinking that underpins the APIs, the tools that have been provided, and how to quickly get off the ground.



Chapter 2. Getting Started

XenServer includes a XML-RPC based API providing programmatic access to the extensive set of XenServer management features and tools. The XenServer API can be called from a remote system as well as local to the XenServer host. Although it is possible to write applications which use the XenServer Management API directly through raw XML-RPC calls, the task of developing third-party applications is greatly simplified through the use of a language binding which exposes the individual API calls as first-class functions in the target language. The XenServer SDK provides language bindings and example code for the C, C#, Java, Python and PowerShell programming languages.

2.1. System Requirements and Preparation

The first step towards working with the SDK is to install XenServer. A free version, Citrix XenServer, is available for download at <http://www.citrix.com/downloads/xenserver/>. Please refer to the *XenServer Installation Guide* for detailed instructions on how to set up your development host. When the installation is complete, please note the *host IP address* and the *host password*.

2.2. Downloading

The SDK is packaged as a ZIP file and is available as a free download from <http://www.citrix.com/downloads/xenserver/>.

2.3. What's New

The XenServer 6.1.0 SDK provides the API with C, C#, Java, PowerShell and Python language bindings. This release contains several enhancements including:

- **C bindings**
Asynchronous task handling and `get_all_records` calls have been implemented. Several API internal calls that were previously published have now been removed. For more details please see the XenServer Release Notes.

2.4. Content Map

The extracted contents of the SDK ZIP file are in the XenServer-SDK directory. The following is an overview of its structure. Where necessary, subdirectories have their own individual README files. It should be noted that the examples provided are not the same across all the language bindings, so, if you intend to use one binding, it is advisable to also browse the sample code available in the others.

- **XenServer-SDK**
 - **libxenserver**
The XenServer SDK for C.
 - **bin**
libxenserver binaries.
 - **src**
libxenserver source code and examples and a Makefile to build them.
 - **XenServer.NET**
The XenServer SDK for C#.NET.
 - **bin**
XenServer.NET binaries.
 - **samples**
XenServer.NET examples.
 - **src**



XenServer.NET source code.

- **XenServerJava**
The XenServer SDK for Java
 - **bin**
Java binaries.
 - **javadoc**
Java documentation.
 - **samples**
Java examples.
 - **src**
Java source code and a Makefile to build the code and the examples.
- **XenServerPSSnapIn**
The XenServer SDK for PowerShell. This directory contains the XenServer PowerShell Windows installer, *XenServerPSSnapIn.msi*.
 - **src**
XenServer PowerShell source code.
- **XenServerPython**
This directory contains the XenServer Python module *XenAPI.py*. The library *provision.py* is used by the examples.
 - **samples**
XenServer API examples using Python.

Chapter 3. Overview of the XenServer API

In this chapter we introduce the XenServer API (hereafter referred to as just "the API") and its associated object model. The API has the following key features:

- **Management of all aspects of the XenServer Host**
Through the API one can manage VMs, storage, networking, host configuration and pools. Performance and status metrics can also be queried from the API.
- **Persistent Object Model**
The results of all side-effecting operations (e.g. object creation, deletion and parameter modifications) are persisted in a server-side database that is managed by the XenServer installation.
- **An event mechanism**
Through the API, clients can register to be notified when persistent (server-side) objects are modified. This enables applications to keep track of datamodel modifications performed by concurrently executing clients.
- **Synchronous and asynchronous invocation**
All API calls can be invoked synchronously (that is, block until completion); any API call that may be long-running can also be invoked *asynchronously*. Asynchronous calls return immediately with a reference to a *task* object. This task object can be queried (through the API) for progress and status information. When an asynchronously invoked operation completes, the result (or error code) is available from the task object.
- **Remotable and Cross-Platform**
The client issuing the API calls does not have to be resident on the host being managed; nor does it have to be connected to the host over ssh in order to execute the API. API calls make use of the XML-RPC protocol to transmit requests and responses over the network.
- **Secure and Authenticated Access**
The XML-RPC API server executing on the host accepts secure socket connections. This allows a client to execute the APIs over the https protocol. Further, all the API calls execute in the context of a login session generated through username and password validation at the server. This provides secure and authenticated access to the XenServer installation.

3.1. Getting Started with the API

We will start our tour of the API by describing the calls required to create a new VM on a XenServer installation, and take it through a start/suspend/resume/stop cycle. This is done without reference to code in any specific language; at this stage we just describe the informal sequence of RPC invocations that accomplish our "install and start" task.

3.1.1. Authentication: acquiring a session reference

The first step is to call `Session.login_with_password(<username>, <password>, <client_API_version>)`. The API is session based, so before you can make other calls you need to authenticate with the server. Assuming the username and password are authenticated correctly, the result of this call is a *session reference*. Subsequent API calls take the session reference as a parameter. In this way we ensure that only API users who are suitably authorized can perform operations on a XenServer installation.

3.1.2. Acquiring a list of templates to base a new VM installation on

The next step is to query the list of "templates" on the host. Templates are specially-marked VM objects that specify suitable default parameters for a variety of supported guest types. (If you want to see a quick enumeration

of the templates on a XenServer installation for yourself then you can execute the **xe template-list** CLI command.) To get a list of templates from the API, we need to find the VM objects on the server that have their **is_a_template** field set to true. One way to do this by calling **VM.get_all_records(session)** where the session parameter is the reference we acquired from our **Session.login_with_password** call earlier. This call queries the server, returning a snapshot (taken at the time of the call) containing all the VM object references and their field values.

(Remember that at this stage we are not concerned about the particular mechanisms by which the returned object references and field values can be manipulated in any particular client language: that detail is dealt with by our language-specific API bindings and described concretely in the following chapter. For now it suffices just to assume the existence of an abstract mechanism for reading and manipulating objects and field values returned by API calls.)

Now that we have a snapshot of all the VM objects' field values in the memory of our client application we can simply iterate through them and find the ones that have their **"is_a_template"** set to true. At this stage let's assume that our example application further iterates through the template objects and remembers the reference corresponding to the one that has its **"name_label"** set to "Debian Etch 4.0" (one of the default Linux templates supplied with XenServer).

3.1.3. Installing the VM based on a template

Continuing through our example, we must now install a new VM based on the template we selected. The installation process requires 2 API calls:

- First we must now invoke the API call **VM.clone(session, t_ref, "my first VM")**. This tells the server to clone the VM object referenced by **t_ref** in order to make a new VM object. The return value of this call is the VM reference corresponding to the newly-created VM. Let's call this **new_vm_ref**.
- At this stage the object referred to by **new_vm_ref** is still a template (just like the VM object referred to by **t_ref**, from which it was cloned). To make **new_vm_ref** into a VM object we need to call **VM.provision(session, new_vm_ref)**. When this call returns the **new_vm_ref** object will have had its **is_a_template** field set to false, indicating that **new_vm_ref** now refers to a regular VM ready for starting.

Note that the provision operation may take a few minutes, as it is as during this call that the template's disk images are created. In the case of the Debian template, the newly created disks are actually populated with a Debian root filesystem at this stage too.

3.1.4. Taking the VM through a start/suspend/resume/stop cycle

Now we have an object reference representing our newly-installed VM, it is trivial to take it through a few lifecycle operations:

- To start our VM we can just call **VM.start(session, new_vm_ref)**
- After it's running, we can suspend it by calling **VM.suspend(session, new_vm_ref)**,
- and then resume it by calling **VM.resume(session, new_vm_ref)**.
- We can call **VM.shutdown(session, new_vm_ref)** to shutdown the VM cleanly.

3.1.5. Logging out

Once an application is finished interacting with a XenServer Host it is good practice to call **Session.logout(session)**. This invalidates the session reference (so it cannot be used in subsequent API calls) and simultaneously deallocates server-side memory used to store the session object.

Although inactive sessions will timeout eventually, the server has a hardcoded limit of 200 concurrent sessions. Once this limit has been reached fresh logins will evict the oldest session objects, causing their associated session references to become invalid. So if you want your applications to play nice with others accessing the server concurrently, then the best policy is to create a single session at start-of-day, use this throughout the applications

(note that sessions can be used across multiple separate client-server *network connections*) and then explicitly logout when possible.

3.1.6. Install and start example: summary

We have seen how the API can be used to install a VM from a XenServer template and perform a number of lifecycle operations on it. You will note that the number of calls we had to make in order to affect these operations was small:

- One call to acquire a session: **Session.login_with_password()**
- One call to query the VM (and template) objects present on the XenServer installation: **VM.get_all_records()**. Recall that we used the information returned from this call to select a suitable template to install from.
- Two calls to install a VM from our chosen template: **VM.clone()**, followed by **VM.provision()**.
- One call to start the resultant VM: **VM.start()** (and similarly other single calls to suspend, resume and shutdown accordingly)
- And then one call to logout **Session.logout()**

The take-home message here is that, although the API as a whole is complex and fully featured, common tasks (such as creating and performing lifecycle operations on VMs) are very straightforward to perform, requiring only a small number of simple API calls. Keep this in mind while you study the next section which may, on first reading, appear a little daunting!

3.2. Object Model Overview

This section gives a high-level overview of the object model of the API. A more detailed description of the parameters and methods of each class outlined here can be found in the *XenServer API Reference* document.

We start by giving a brief outline of some of the core classes that make up the API. (Don't worry if these definitions seem somewhat abstract in their initial presentation; the textual description in subsequent sections, and the code-sample walk through in the next Chapter will help make these concepts concrete.)

VM	A VM object represents a particular virtual machine instance on a XenServer Host or Resource Pool. Example methods include start , suspend , pool_migrate ; example parameters include <i>power_state</i> , <i>memory_static_max</i> , and <i>name_label</i> . (In the previous section we saw how the VM class is used to represent both templates and regular VMs)
Host	A host object represents a physical host in a XenServer pool. Example methods include reboot and shutdown . Example parameters include <i>software_version</i> , <i>hostname</i> , and [IP] <i>address</i> .
VDI	A VDI object represents a <i>Virtual Disk Image</i> . Virtual Disk Images can be attached to VMs, in which case a block device appears inside the VM through which the bits encapsulated by the Virtual Disk Image can be read and written. Example methods of the VDI class include "resize" and "clone". Example fields include "virtual_size" and "sharable". (When we called VM.provision on the VM template in our previous example, some VDI objects were automatically created to represent the newly created disks, and attached to the VM object.)
SR	An SR (<i>Storage Repository</i>) aggregates a collection of VDIs and encapsulates the properties of physical storage on which the VDIs' bits reside. Example parameters include <i>type</i> (which determines the storage-specific driver a XenServer installation uses to read/write the SR's VDIs) and <i>physical_utilisation</i> ; example methods include scan (which invokes the storage-specific driver to acquire a list of the VDIs contained with the SR and the properties of these VDIs) and create (which initializes a block of physical storage so it is ready to store VDIs).

Network	A network object represents a layer-2 network that exists in the environment in which the XenServer Host instance lives. Since XenServer does not manage networks directly this is a lightweight class that serves merely to model physical and virtual network topology. VM and Host objects that are <i>attached</i> to a particular Network object (by virtue of VIF and PIF instances -- see below) can send network packets to each other.
---------	---

At this point, readers who are finding this enumeration of classes rather terse may wish to skip to the code walk-throughs of the next chapter: there are plenty of useful applications that can be written using only a subset of the classes already described! For those who wish to continue this description of classes in the abstract, read on.

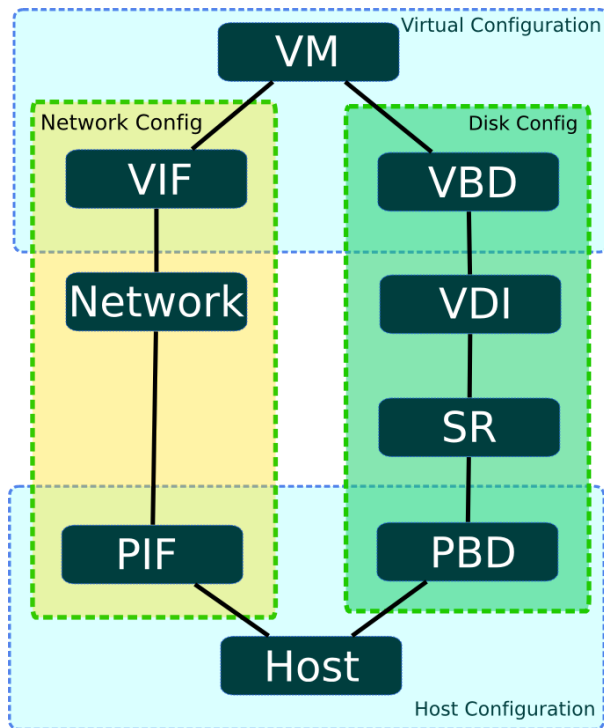
On top of the classes listed above, there are 4 more that act as *connectors*, specifying relationships between VMs and Hosts, and Storage and Networks. The first 2 of these classes that we will consider, *VBD* and *VIF*, determine how VMs are attached to virtual disks and network objects respectively:

VBD	A VBD (<i>Virtual Block Device</i>) object represents an attachment between a VM and a VDI. When a VM is booted its VBD objects are queried to determine which disk images (VDIs) should be attached. Example methods of the VBD class include "plug" (which <i>hot plugs</i> a disk device into a running VM, making the specified VDI accessible therein) and "unplug" (which <i>hot unplugs</i> a disk device from a running guest); example fields include "device" (which determines the device name inside the guest under which the specified VDI will be made accessible).
VIF	A VIF (<i>Virtual network InterFace</i>) object represents an attachment between a VM and a Network object. When a VM is booted its VIF objects are queried to determine which network devices should be created. Example methods of the VIF class include "plug" (which <i>hot plugs</i> a network device into a running VM) and "unplug" (which <i>hot unplugs</i> a network device from a running guest).

The second set of "connector classes" that we will consider determine how Hosts are attached to Networks and Storage.

PIF	A PIF (<i>Physical InterFace</i>) object represents an attachment between a Host and a Network object. If a host is connected to a Network (over a PIF) then packets from the specified host can be transmitted/received by the corresponding host. Example fields of the PIF class include "device" (which specifies the device name to which the PIF corresponds -- e.g. <i>eth0</i>) and "MAC" (which specifies the MAC address of the underlying NIC that a PIF represents). Note that PIFs abstract both physical interfaces and VLANs (the latter distinguished by the existence of a positive integer in the "VLAN" field).
PBD	A PBD (<i>Physical Block Device</i>) object represents an attachment between a Host and a SR (Storage Repository) object. Fields include "currently-attached" (which specifies whether the chunk of storage represented by the specified SR object) is currently available to the host; and "device_config" (which specifies storage-driver specific parameters that determines how the low-level storage devices are configured on the specified host -- e.g. in the case of an SR rendered on an NFS filer, device_config may specify the host-name of the filer and the path on the filer in which the SR files live.)

Figure 3.1. Common API Classes



Graphical overview of API classes for managing VMs, Hosts, Storage and Networking

Figure 3.1, “Common API Classes” presents a graphical overview of the API classes involved in managing VMs, Hosts, Storage and Networking. From this diagram, the symmetry between storage and network configuration, and also the symmetry between virtual machine and host configuration is plain to see.

3.3. Working with VIFs and VBDs

In this section we walk through a few more complex scenarios, describing informally how various tasks involving virtual storage and network devices can be accomplished using the API.

3.3.1. Creating disks and attaching them to VMs

Let's start by considering how to make a new blank disk image and attach it to a running VM. We will assume that we already have ourselves a running VM, and we know its corresponding API object reference (e.g. we may have created this VM using the procedure described in the previous section, and had the server return its reference to us.) We will also assume that we have authenticated with the XenServer installation and have a corresponding **session reference**. Indeed in the rest of this chapter, for the sake of brevity, we will stop mentioning sessions altogether.

3.3.1.1. Creating a new blank disk image

The first step is to instantiate the disk image on physical storage. We do this by calling **VDI.create()**. The **VDI.create** call takes a number of parameters, including:

- **name_label** and **name_description**: a human-readable name/description for the disk (e.g. for convenient display in the UI etc.). These fields can be left blank if desired.
- **SR**: the object reference of the Storage Repository representing the physical storage in which the VDI's bits will be placed.

- **read_only**: setting this field to true indicates that the VDI can *only* be attached to VMs in a read-only fashion. (Attempting to attach a VDI with its **read_only** field set to true in a read/write fashion results in error.)

Invoking the **VDI.create** call causes the XenServer installation to create a blank disk image on physical storage, create an associated VDI object (the datamodel instance that refers to the disk image on physical storage) and return a reference to this newly created VDI object.

The way in which the disk image is represented on physical storage depends on the type of the SR in which the created VDI resides. For example, if the SR is of type "lvm" then the new disk image will be rendered as an LVM volume; if the SR is of type "nfs" then the new disk image will be a sparse VHD file created on an NFS filer. (You can query the SR type through the API using the **SR.get_type()** call.)

Note:

Some SR types might round up the *virtual-size* value to make it divisible by a configured block size.

3.3.1.2. Attaching the disk image to a VM

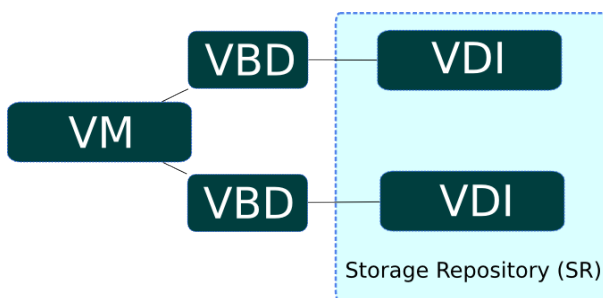
So far we have a running VM (that we assumed the existence of at the start of this example) and a fresh VDI that we just created. Right now, these are both independent objects that exist on the XenServer Host, but there is nothing linking them together. So our next step is to create such a link, associating the VDI with our VM.

The attachment is formed by creating a new "connector" object called a VBD (*Virtual Block Device*). To create our VBD we invoke the **VBD.create()** call. The **VBD.create()** call takes a number of parameters including:

- **VM** - the object reference of the VM to which the VDI is to be attached
- **VDI** - the object reference of the VDI that is to be attached
- **mode** - specifies whether the VDI is to be attached in a read-only or a read-write fashion
- **userdevice** - specifies the block device inside the guest through which applications running inside the VM will be able to read/write the VDI's bits.
- **type** - specifies whether the VDI should be presented inside the VM as a regular disk or as a CD. (Note that this particular field has more meaning for Windows VMs than it does for Linux VMs, but we will not explore this level of detail in this chapter.)

Invoking **VBD.create** makes a VBD object on the XenServer installation and returns its object reference. However, this call in itself does not have any side-effects on the running VM (that is, if you go and look inside the running VM you will see that the block device has not been created). The fact that the VBD object exists but that the block device in the guest is not active, is reflected by the fact that the VBD object's **currently_attached** field is set to false.

Figure 3.2. A VM object with 2 associated VDIs



A VM object with 2 associated VDIs

For expository purposes, [Figure 3.2, "A VM object with 2 associated VDIs"](#) presents a graphical example that shows the relationship between VMs, VBDs, VDIs and SRs. In this instance a VM object has 2 attached VDIs: there



are 2 VBD objects that form the connections between the VM object and its VDIs; and the VDIs reside within the same SR.

3.3.1.3. Hotplugging the VBD

If we rebooted the VM at this stage then, after rebooting, the block device corresponding to the VBD would appear: on boot, XenServer queries all VBDs of a VM and actively attaches each of the corresponding VDIs.

Rebooting the VM is all very well, but recall that we wanted to attach a newly created blank disk to a *running* VM. This can be achieved by invoking the **plug** method on the newly created VBD object. When the **plug** call returns successfully, the block device to which the VBD relates will have appeared inside the running VM -- i.e. from the perspective of the running VM, the guest operating system is led to believe that a new disk device has just been *hot plugged*. Mirroring this fact in the managed world of the API, the **currently_attached** field of the VBD is set to true.

Unsurprisingly, the VBD **plug** method has a dual called "**unplug**". Invoking the **unplug** method on a VBD object causes the associated block device to be *hot unplugged* from a running VM, setting the **currently_attached** field of the VBD object to false accordingly.

3.3.2. Creating and attaching Network Devices to VMs

The API calls involved in configuring virtual network interfaces in VMs are similar in many respects to the calls involved in configuring virtual disk devices. For this reason we will not run through a full example of how one can create network interfaces using the API object-model; instead we will use this section just to outline briefly the symmetry between virtual *networking device* and virtual *storage device* configuration.

The networking analogue of the VBD class is the VIF class. Just as a VBD is the API representation of a block device inside a VM, a VIF (*Virtual network InterFace*) is the API representation of a network device inside a VM. Whereas VBDs associate VM objects with VDI objects, VIFs associate VM objects with Network objects. Just like VBDs, VIFs have a **currently_attached** field that determines whether or not the network device (inside the guest) associated with the VIF is currently active or not. And as we saw with VBDs, at VM boot-time the VIFs of the VM are queried and a corresponding network device for each created inside the booting VM. Similarly, VIFs also have **plug** and **unplug** methods for hot plugging/unplugging network devices in/out of running VMs.

3.3.3. Host configuration for networking and storage

We have seen that the VBD and VIF classes are used to manage configuration of block devices and network devices (respectively) inside VMs. To manage host configuration of storage and networking there are two analogous classes: PBD (*Physical Block Device*) and PIF (*Physical [network] InterFace*).

3.3.3.1. Host storage configuration: PBDs

Let us start by considering the PBD class. A **PBD_create()** call takes a number of parameters including:

Parameter	Description
host	physical machine on which the PBD is available
SR	the Storage Repository that the PBD connects to
device_config	a string-to-string map that is provided to the host's SR-backend-driver, containing the low-level parameters required to configure the physical storage device(s) on which the SR is to be realized. The specific contents of the <i>device_config</i> field depend on the type of the SR to which the PBD is connected. (Executing xe sm-list will show a list of possible SR types; the <i>configuration</i> field in this enumeration specifies the <i>device_config</i> parameters that each SR type expects.)



For example, imagine we have an SR object *s* of type "nfs" (representing a directory on an NFS filer within which VDIs are stored as VHD files); and let's say that we want a host, *h*, to be able to access *s*. In this case we invoke **PBD.create()** specifying host *h*, SR *s*, and a value for the *device_config* parameter that is the following map:

```
( "server", "my_nfs_server.example.com" ), ( "serverpath", "/scratch/mysrs/sr1" )
```

This tells the XenServer Host that SR *s* is accessible on host *h*, and further that to access SR *s*, the host needs to mount the directory `/scratch/mysrs/sr1` on the NFS server named `my_nfs_server.example.com`.

Like VBD objects, PBD objects also have a field called **currently_attached**. Storage repositories can be attached and detached from a given host by invoking **PBD.plug** and **PBD.unplug** methods respectively.

3.3.3.2. Host networking configuration: PIFs

Host network configuration is specified by virtue of PIF objects. If a PIF object connects a network object, *n*, to a host object *h*, then the network corresponding to *n* is bridged onto a physical interface (or a physical interface plus a VLAN tag) specified by the fields of the PIF object.

For example, imagine a PIF object exists connecting host *h* to a network *n*, and that **device** field of the PIF object is set to `eth0`. This means that all packets on network *n* are bridged to the NIC in the host corresponding to host network device `eth0`.

3.4. Exporting and Importing VMs

VMs can be exported to a file and later imported to any XenServer host. The export protocol is a simple HTTP(S) GET, which should be performed on the master if the VM is on a pool member. Authorization is either standard HTTP basic authentication, or if a session has already been obtained, this can be used. The VM to export is specified either by UUID or by reference. To keep track of the export, a task can be created and passed in using its reference. The request might result in a redirect if the VM's disks are only accessible on a pool member.

The following arguments are passed on the command line:

Argument	Description
<code>session_id</code>	the reference of the session being used to authenticate; required only when not using HTTP basic authentication
<code>task_id</code>	the reference of the task object with which to keep track of the operation; optional, required only if you have created a task object to keep track of the export
<code>ref</code>	the reference of the VM; required only if not using the UUID
<code>uuid</code>	the UUID of the VM; required only if not using the reference

For example, using the Linux command line tool `cURL`:

```
curl http://root:foo@myxenserver1/export?uuid=<vm_uuid> -o <exportfile>
```

will export the specified VM to the file `exportfile`.

To export just the metadata, use the URI `http://server/export_metadata`.

The import protocol is similar, using HTTP(S) PUT. The `session_id` and `task_id` arguments are as for the export. The `ref` and `uuid` are not used; a new reference and `uuid` will be generated for the VM. There are some additional parameters:

Argument	Description
restore	if <i>true</i> , the import is treated as replacing the original VM - the implication of this currently is that the MAC addresses on the VIFs are exactly as the export was, which will lead to conflicts if the original VM is still being run.
force	if <i>true</i> , any checksum failures will be ignored (the default is to destroy the VM if a checksum error is detected)
sr_id	the reference of an SR into which the VM should be imported. The default behavior is to import into the <i>Pool.default_SR</i> .

For example, again using cURL:

```
curl -T <exportfile> http://root:foo@myxenserver2/import
```

will import the VM to the default SR on the server.

Note:

Note that if no default SR has been set, and no *sr_uuid* is specified, the error message "DEFAULT_SR_NOT_FOUND" is returned.

Another example:

```
curl -T <exportfile> http://root:foo@myxenserver2/import?sr_id=<opaque_ref_of_sr>
```

will import the VM to the specified SR on the server.

To import just the metadata, use the URI `http://server/import_metadata`

3.4.1. Xen Virtual Appliance (XVA) VM Import Format

XenServer supports a human-readable legacy VM input format called XVA. This section describes the syntax and structure of XVA.

An XVA consists of a directory containing XML metadata and a set of disk images. A VM represented by an XVA is not intended to be directly executable. Data within an XVA package is compressed and intended for either archiving on permanent storage or for being transmitted to a VM server - such as a XenServer host - where it can be decompressed and executed.

XVA is a hypervisor-neutral packaging format; it should be possible to create simple tools to instantiate an XVA VM on any other platform. XVA does not specify any particular runtime format; for example disks may be instantiated as file images, LVM volumes, QCoW images, VMDK or VHD images. An XVA VM may be instantiated any number of times, each instantiation may have a different runtime format.

XVA does not:

- specify any particular serialization or transport format
- provide any mechanism for customizing VMs (or templates) on install
- address how a VM may be upgraded post-install
- define how multiple VMs, acting as an appliance, may communicate

These issues are all addressed by the related Open Virtual Appliance specification.

An XVA is a directory containing, at a minimum, a file called `ova.xml`. This file describes the VM contained within the XVA and is described in Section 3.2. Disks are stored within sub-directories and are referenced from the `ova.xml`. The format of disk data is described later in Section 3.3.



The following terms will be used in the rest of the chapter:

- HVM: a mode in which unmodified OS kernels run with the help of virtualization support in the hardware.
- PV: a mode in which specially modified "paravirtualized" kernels run explicitly on top of a hypervisor without requiring hardware support for virtualization.

The "ova.xml" file contains the following elements:

```
<appliance version="0.1">
```

The number in the attribute "version" indicates the version of this specification to which the XVA is constructed; in this case version 0.1. Inside the <appliance> there is exactly one <vm>: (in the OVA specification, multiple <vm>s are permitted)

```
<vm name="name" >
```

Each <vm> element describes one VM. The "name" attribute is for future internal use only and must be unique within the ova.xml file. The "name" attribute is permitted to be any valid UTF-8 string. Inside each <vm> tag are the following compulsory elements:

```
<label>... text ... </label>
```

A short name for the VM to be displayed in a UI.

```
<shortdesc> ... description ... </shortdesc>
```

A description for the VM to be displayed in the UI. Note that for both <label> and <shortdesc> contents, leading and trailing whitespace will be ignored.

```
<config mem_set="268435456" vcpus="1" />
```

The <config> element has attributes which describe the amount of memory in bytes (mem_set) and number of CPUs (VCPUs) the VM should have.

Each <vm> has zero or more <vbd> elements representing block devices which look like the following:

```
<vbd device="sda" function="root" mode="w" vdi="vdi_sda" />
```

The attributes have the following meanings:

device	name of the physical device to expose to the VM. For linux guests we use "sd[a-z]" and for windows guests we use "hd[a-d]".
function	if marked as "root", this disk will be used to boot the guest. (NB this does not imply the existence of the Linux root i.e. / filesystem) Only one device should be marked as "root". See Section 3.4 describing VM booting. Any other string is ignored.
mode	either "w" or "ro" if the device is to be read/write or read-only
vdi	the name of the disk image (represented by a <vdi> element) to which this block device is connected

Each <vm> may have an optional <hacks> section like the following: <hacks is_hvm="false" kernel_boot_cmdline="root=/dev/sda1 ro"/> The <hacks> element is present in the XVA files generated by XenServer but will be removed in future. The attribute "is_hvm" is either "true" or "false", depending on whether the VM should be booted in HVM or not. The "kernel_boot_cmdline" contains additional kernel commandline arguments when booting a guest using pygrub.

In addition to a <vm> element, the <appliance> will contain zero or more <vdi> elements like the following:

```
<vdi name="vdi_sda" size="5368709120" source="file://sda" type="dir-gzipped-chunks">
```

Each <vdi> corresponds to a disk image. The attributes have the following meanings:



- name: name of the VDI, referenced by the vdi attribute of <vbd> elements. Any valid UTF-8 string is permitted.
- size: size of the required image in bytes
- source: a URI describing where to find the data for the image, only file:// URIs are currently permitted and must describe paths relative to the directory containing the ova.xml
- type: describes the format of the disk data (see Section 3.3)

A single disk image encoding is specified in which has type "dir-gzipped-chunks": Each image is represented by a directory containing a sequence of files as follows:

```
-rw-r--r-- 1 dscott xendev 458286013 Sep 18 09:51 chunk000000000.gz
-rw-r--r-- 1 dscott xendev 422271283 Sep 18 09:52 chunk000000001.gz
-rw-r--r-- 1 dscott xendev 395914244 Sep 18 09:53 chunk000000002.gz
-rw-r--r-- 1 dscott xendev 9452401 Sep 18 09:53 chunk000000003.gz
-rw-r--r-- 1 dscott xendev 1096066 Sep 18 09:53 chunk000000004.gz
-rw-r--r-- 1 dscott xendev 971976 Sep 18 09:53 chunk000000005.gz
-rw-r--r-- 1 dscott xendev 971976 Sep 18 09:53 chunk000000006.gz
-rw-r--r-- 1 dscott xendev 971976 Sep 18 09:53 chunk000000007.gz
-rw-r--r-- 1 dscott xendev 573930 Sep 18 09:53 chunk000000008.gz
```

Each file (named "chunk-XXXXXXXXX.gz") is a gzipped file containing exactly 1e9 bytes (1GB, not 1GiB) of raw block data. The small size was chosen to be safely under the maximum file size limits of several filesystems. If the files are gunzipped and then concatenated together, the original image is recovered.

XenServer provides two mechanisms for booting a VM: (i) using a paravirtualized kernel extracted through pygrub; and (ii) using HVM. The current implementation uses the "is_hvm" flag within the <hacks> section to decide which mechanism to use.

This rest of this section describes a very simple Debian VM packaged as an XVA. The VM has two disks, one with size 5120MiB and used for the root filesystem and used to boot the guest using pygrub and the other of size 512MiB which is used for swap. The VM has 512MiB of memory and uses one virtual CPU.

At the topmost level the simple Debian VM is represented by a single directory:

```
$ ls -l
total 4
drwxr-xr-x 3 dscott xendev 4096 Oct 24 09:42 very simple Debian VM
```

Inside the main XVA directory are two sub-directories - one per disk - and the single file: ova.xml:

```
$ ls -l very\ simple\ Debian\ VM/
total 8
-rw-r--r-- 1 dscott xendev 1016 Oct 24 09:42 ova.xml
drwxr-xr-x 2 dscott xendev 4096 Oct 24 09:42 sda
drwxr-xr-x 2 dscott xendev 4096 Oct 24 09:53 sdb
```

Inside each disk sub-directory are a set of files, each file contains 1GB of raw disk block data compressed using gzip:

```
$ ls -l very\ simple\ Debian\ VM/sda/
total 2053480
-rw-r--r-- 1 dscott xendev 202121645 Oct 24 09:43 chunk-000000000.gz
-rw-r--r-- 1 dscott xendev 332739042 Oct 24 09:45 chunk-000000001.gz
-rw-r--r-- 1 dscott xendev 401299288 Oct 24 09:48 chunk-000000002.gz
-rw-r--r-- 1 dscott xendev 389585534 Oct 24 09:50 chunk-000000003.gz
-rw-r--r-- 1 dscott xendev 624567877 Oct 24 09:53 chunk-000000004.gz
-rw-r--r-- 1 dscott xendev 150351797 Oct 24 09:54 chunk-000000005.gz

$ ls -l very\ simple\ Debian\ VM/sdb
total 516
-rw-r--r-- 1 dscott xendev 521937 Oct 24 09:54 chunk-000000000.gz
```

The example simple Debian VM would have an XVA file like the following:



```
<?xml version="1.0" ?>
<appliance version="0.1">
  <vm name="vm">
    <label>
      very simple Debian VM
    </label>
    <shortdesc>
      the description field can contain any valid UTF-8
    </shortdesc>
    <config mem_set="536870912" vcpus="1"/>
    <hacks is_hvm="false" kernel_boot_cmdline="root=/dev/sda1 ro ">
      <!--This section is temporary and will be ignored in future. Attribute
is_hvm ("true" or "false") indicates whether the VM will be booted in HVM mode. In
future this will be autodetected. Attribute kernel_boot_cmdline contains the kernel
commandline for the case where a proper grub menu.lst is not present. In future
booting shall only use pygrub.-->
    </hacks>
    <vbd device="sda" function="root" mode="w" vdi="vdi_sda"/>
    <vbd device="sdb" function="swap" mode="w" vdi="vdi_sdb"/>
  </vm>
  <vdi name="vdi_sda" size="5368709120" source="file://sda" type="dir-gzippedchunks"/>
  <vdi name="vdi_sdb" size="536870912" source="file://sdb" type="dir-gzippedchunks"/>
</appliance>
```

3.5. XML-RPC notes

3.5.1. Datetimes

The API deviates from the XML-RPC specification in handling of datetimes. The API appends a "Z" to the end of datetime strings, which is meant to indicate that the time is expressed in UTC.

3.6. Where to look next

In this chapter we have presented a brief high-level overview of the API and its object-model. The aim here is not to present the detailed semantics of the API, but just to provide enough background for you to start reading the code samples of the next chapter and to find your way around the more detailed *XenServer API Reference* reference document.

There are a number of places you can find more information:

- The [XenServer Administrators Guide](#) contains an overview of the **xe** CLI. Since a good deal of **xe** commands are a thin veneer over the API, playing with **xe** is a good way to start finding your way around the API object model described in this chapter.
- The code samples in the next chapter provide some concrete instances of API coding in a variety of client languages.
- The [XenServer API Reference](#) reference document provides a more detailed description of the API semantics as well as describing the format of XML/RPC messages on the wire.
- There are a few scripts that use the API in the XenServer Host dom0 itself. For example, `/opt/xensource/libexec/shutdown` is a python program that cleanly shuts VMs down. This script is invoked when the host itself is shut down.

Chapter 4. Using the API

This chapter describes how to use the XenServer Management API from real programs to manage XenServer Hosts and VMs. The chapter begins with a walk-through of a typical client application and demonstrates how the API can be used to perform common tasks. Example code fragments are given in python syntax but equivalent code in the other programming languages would look very similar. The language bindings themselves are discussed afterwards and the chapter finishes with walk-throughs of two complete examples.

4.1. Anatomy of a typical application

This section describes the structure of a typical application using the XenServer Management API. Most client applications begin by connecting to a XenServer Host and authenticating (e.g. with a username and password). Assuming the authentication succeeds, the server will create a "session" object and return a reference to the client. This reference will be passed as an argument to all future API calls. Once authenticated, the client may search for references to other useful objects (e.g. XenServer Hosts, VMs, etc.) and invoke operations on them. Operations may be invoked either synchronously or asynchronously; special task objects represent the state and progress of asynchronous operations. These application elements are all described in detail in the following sections.

4.1.1. Choosing a low-level transport

API calls can be issued over two transports:

- SSL-encrypted TCP on port 443 (https) over an IP network
- plaintext over a local Unix domain socket: `/var/xapi/xapi`

The SSL-encrypted TCP transport is used for all off-host traffic while the Unix domain socket can be used for services running directly on the XenServer Host itself. In the SSL-encrypted TCP transport, all API calls should be directed at the Resource Pool master; failure to do so will result in the error `HOST_IS_SLAVE`, which includes the IP address of the master as an error parameter.

Because the master host of a pool can change, especially if HA is enabled on a pool, clients must implement the following steps to detect a master host change and connect to the new master as required:

Handling pool master changes

1. Subscribe to updates in the list of hosts servers, and maintain a current list of hosts in the pool
2. If the connection to the pool master fails to respond, attempt to connect to all hosts in the list until one responds
3. The first host to respond will return the `HOST_IS_SLAVE` error message, which contains the identity of the new pool master (unless of course the host is the new master)
4. Connect to the new master

Note:

As a special-case, all messages sent through the Unix domain socket are transparently forwarded to the correct node.

4.1.2. Authentication and session handling

The vast majority of API calls take a session reference as their first parameter; failure to supply a valid reference will result in a `SESSION_INVALID` error being returned. Acquire a session reference by supplying a username and password to the `login_with_password` function.

Note:



As a special-case, if this call is executed over the local Unix domain socket then the username and password are ignored and the call always succeeds.

Every session has an associated "last active" timestamp which is updated on every API call. The server software currently has a built-in limit of 200 active sessions and will remove those with the oldest "last active" field if this limit is exceeded. In addition all sessions whose "last active" field is older than 24 hours are also removed. Therefore it is important to:

- Remember to log out of active sessions to avoid leaking them; and
- Be prepared to log in again to the server if a `SESSION_INVALID` error is caught.

In the following fragment a connection is established over the Unix domain socket and a session is created:

```
import XenAPI

session = XenAPI.xapi_local()
try:
    session.xenapi.login_with_password("root", "")
    ...
finally:
    session.xenapi.session.logout()
```

4.1.3. Finding references to useful objects

Once an application has authenticated the next step is to acquire references to objects in order to query their state or invoke operations on them. All objects have a set of "implicit" messages which include the following:

- **get_by_name_label** : return a list of all objects of a particular class with a particular label;
- **get_by_uuid** : return a single object named by its UUID;
- **get_all** : return a set of references to all objects of a particular class; and
- **get_all_records** : return a map of reference to records for each object of a particular class.

For example, to list all hosts:

```
hosts = session.xenapi.host.get_all()
```

To find all VMs with the name "my first VM":

```
vms = session.xenapi.VM.get_by_name_label('my first VM')
```

Note:

Object `name_label` fields are not guaranteed to be unique and so the **get_by_name_label** API call returns a set of references rather than a single reference.

In addition to the methods of finding objects described above, most objects also contain references to other objects within fields. For example it is possible to find the set of VMs running on a particular host by calling:

```
vms = session.xenapi.host.get_resident_VMs(host)
```

4.1.4. Invoking synchronous operations on objects

Once object references have been acquired, operations may be invoked on them. For example to start a VM:

```
session.xenapi.VM.start(vm, False, False)
```

All API calls are by default synchronous and will not return until the operation has completed or failed. For example in the case of **VM.start** the call does not return until the VM has started booting.

Note:



When the **VM.start** call returns the VM will be booting. To determine when the booting has finished, wait for the in-guest agent to report internal statistics through the **VM_guest_metrics** object.

4.1.5. Using Tasks to manage asynchronous operations

To simplify managing operations which take quite a long time (e.g. **VM.clone** and **VM.copy**) functions are available in two forms: synchronous (the default) and asynchronous. Each asynchronous function returns a reference to a task object which contains information about the in-progress operation including:

- whether it is pending
- whether it has succeeded or failed
- progress (in the range 0-1)
- the result or error code returned by the operation

An application which wanted to track the progress of a **VM.clone** operation and display a progress bar would have code like the following:

```
vm = session.xenapi.VM.get_by_name_label('my vm')
task = session.xenapi.Async.VM.clone(vm)
while session.xenapi.task.get_status(task) == "pending":
    progress = session.xenapi.task.get_progress(task)
    update_progress_bar(progress)
    time.sleep(1)
session.xenapi.task.destroy(task)
```

Note:

Note that a well-behaved client should remember to delete tasks created by asynchronous operations when it has finished reading the result or error. If the number of tasks exceeds a built-in threshold then the server will delete the oldest of the completed tasks.

4.1.6. Subscribing to and listening for events

With the exception of the task and metrics classes, whenever an object is modified the server generates an event. Clients can subscribe to this event stream on a per-class basis and receive updates rather than resorting to frequent polling. Events come in three types:

- *add* - generated when an object has been created;
- *del* - generated immediately before an object is destroyed; and
- *mod* - generated when an object's field has changed.

Events also contain a monotonically increasing ID, the name of the class of object and a snapshot of the object state equivalent to the result of a **get_record()**.

Clients register for events by calling **event.register()** with a list of class names or the special string **"*"**. Clients receive events by executing **event.next()** which blocks until events are available and returns the new events.

Note:

Since the queue of generated events on the server is of finite length a very slow client might fail to read the events fast enough; if this happens an **EVENTS_LOST** error is returned. Clients should be prepared to handle this by re-registering for events and checking that the condition they are waiting for hasn't become true while they were unregistered.

The following python code fragment demonstrates how to print a summary of every event generated by a system: (similar code exists in `Xenserver-SDK/XenServerPython/samples/watch-all-events.py`)



```
fmt = "%8s %20s %5s %s"
session.xenapi.event.register(["*"])
while True:
    try:
        for event in session.xenapi.event.next():
            name = "(unknown)"
            if "snapshot" in event.keys():
                snapshot = event["snapshot"]
                if "name_label" in snapshot.keys():
                    name = snapshot["name_label"]
            print fmt % (event['id'], event['class'], event['operation'], name)
    except XenAPI.Failure, e:
        if e.details == [ "EVENTS_LOST" ]:
            print "Caught EVENTS_LOST; should reregister"
```

4.2. Language bindings

4.2.1. C

The SDK includes the source to the C language binding in the directory `XenServer-SDK/libxenserver/src` together with a Makefile which compiles the binding into a library. Every API object is associated with a header file which contains declarations for all that object's API functions; for example the type definitions and functions required to invoke VM operations are all contained in `xen_vm.h`.

C binding dependencies

Platform supported:	Linux
Library:	The language binding is generated as a <code>libxenserver.so</code> that is linked by C programs.
Dependencies:	<ul style="list-style-type: none">• XML library (<code>libxml2.so</code> on GNU Linux)• Curl library (<code>libcurl2.so</code>)

Three simple examples are included with the C bindings:

- `test_vm_ops`: demonstrates how to query the capabilities of a host, create a VM, attach a fresh blank disk image to the VM and then perform various powercycle operations;
- `test_failures`: demonstrates how to translate error strings into `enum_xen_api_failure`, and vice versa;
- `test_event_handling`: demonstrates how to listen for events on a connection.

4.2.2. C#

The C# bindings are contained within the directory `XenServer-SDK/XenServer.NET` and include project files suitable for building under Microsoft Visual Studio. Every API object is associated with one C# file; for example the functions implementing the VM operations are contained within the file `VM.cs`.

C# binding dependencies

Platform supported:	Windows with .NET version 2.0
Library:	The language binding is generated as a Dynamic Link Library <code>XenServer.dll</code> that is linked by C# programs.

Dependencies:	CookComputing.XMLRpcV2.dll is needed for the XenServer.dll to be able to communicate with the xml-rpc server. We test with version 2.1.0.6 and recommend that you use this version, though others may work.
---------------	---

Three examples are included with the C# bindings in the directory `XenServer-SDK/XenServer.NET/samples` as separate projects of the `XenSdkSample.sln` solution:

- `GetVariousRecords`: logs into a XenServer Host and displays information about hosts, storage and virtual machines;
- `GetVmRecords`: logs into a XenServer Host and lists all the VM records;
- `VmPowerStates`: logs into a XenServer Host, finds a VM and takes it through the various power states. Requires a shut-down VM to be already installed.

4.2.3. Java

The Java bindings are contained within the directory `XenServer-SDK/XenServerJava` and include project files suitable for building under Microsoft Visual Studio. Every API object is associated with one Java file; for example the functions implementing the VM operations are contained within the file `VM.java`.

Java binding dependencies

Platform supported:	Linux and Windows
Library:	The language binding is generated as a Java Archive file <code>xenserver-6.1.0.jar</code> that is linked by Java programs.
Dependencies:	<ul style="list-style-type: none"> • <code>xmlrpc-client-3.1.jar</code> is needed for the <code>xenserver.jar</code> to be able to communicate with the xml-rpc server. • <code>ws-commons-util-1.0.2.jar</code> is needed to run the examples.

Running the main file `XenServer-SDK/XenServerJava/samples/RunTests.java` will run a series of examples included in the same directory:

- `AddNetwork`: Adds a new internal network not attached to any NICs;
- `SessionReuse`: Demonstrates how a Session object can be shared between multiple Connections;
- `AsyncVMCreate`: Makes asynchronously a new VM from a built-in template, starts and stops it;
- `VdiAndSrOps`: Performs various SR and VDI tests, including creating a dummy SR;
- `CreateVM`: Creates a VM on the default SR with a network and DVD drive;
- `DeprecatedMethod`: Tests a warning is displayed when a deprecated API method is called;
- `GetAllRecordsOfAllTypes`: Retrieves all the records for all types of objects;
- `SharedStorage`: Creates a shared NFS SR;
- `StartAllVMs`: Connects to a host and tries to start each VM on it.

4.2.4. PowerShell

The PowerShell bindings are contained within the directory `XenServer-SDK/XenPSSnapIn`. We provide the Windows installer `XenServerPSSnapIn.msi` and source code exposing the XenServer API as Windows PowerShell 1.0 cmdlets.

PowerShell binding dependencies

Platform supported:	Windows
---------------------	---------



Library:	XenServerPSSnapIn.dll
Dependencies:	CookComputing.XMLRpcV2.dll is needed to be able to communicate with the xml-rpc server. We test with version 2.1.0.6 and recommend that you use this version, though others may work.

4.2.5. Python

The python bindings are contained within a single file: `XenServer-SDK/XenServerPython/XenAPI.py`.

Python binding dependencies

Platform supported:	Linux
Library:	XenAPI.py
Dependencies:	None

The SDK includes 7 python examples:

- `fixpbds.py` - reconfigures the settings used to access shared storage;
- `install.py` - installs a Debian VM, connects it to a network, starts it up and waits for it to report its IP address;
- `license.py` - uploads a fresh license to a XenServer Host;
- `permute.py` - selects a set of VMs and uses XenMotion to move them simultaneously between hosts;
- `powercycle.py` - selects a set of VMs and powercycles them;
- `shell.py` - a simple interactive shell for testing;
- `vm_start_async.py` - demonstrates how to invoke operations asynchronously;
- `watch-all-events.py` - registers for all events and prints details when they occur.

4.2.6. Command Line Interface (CLI)

Besides using raw XML-RPC or one of the supplied language bindings, third-party software developers may integrate with XenServer Hosts by using the XE command line interface **xe**. The **xe** CLI is installed by default on XenServer hosts; a stand-alone remote CLI is also available for Linux. On Windows, the `xe.exe` CLI executable is installed along with XenCenter.

CLI dependencies

Platform supported:	Linux and Windows
Library:	None
Binary:	xe (xe.exe on Windows)
Dependencies:	None

The CLI allows almost every API call to be directly invoked from a script or other program, silently taking care of the required session management. The XE CLI syntax and capabilities are described in detail in the [XenServer Administrator's Guide](#). For additional resources and examples, visit the [Citrix Knowledge Center](#).



Note:

When running the CLI from a XenServer Host console, tab-completion of both command names and arguments is available.

4.3. Complete application examples

This section describes two complete examples of real programs using the API.

4.3.1. Simultaneously migrating VMs using XenMotion

This python example (contained in `XenServer-SDK/XenServerPython/samples/permute.py`) demonstrates how to use XenMotion to move VMs simultaneously between hosts in a Resource Pool. The example makes use of asynchronous API calls and shows how to wait for a set of tasks to complete.

The program begins with some standard boilerplate and imports the API bindings module

```
import sys, time
import XenAPI
```

Next the commandline arguments containing a server URL, username, password and a number of iterations are parsed. The username and password are used to establish a session which is passed to the function `main`, which is called multiple times in a loop. Note the use of `try: finally:` to make sure the program logs out of its session at the end.

```
if __name__ == "__main__":
    if len(sys.argv) <> 5:
        print "Usage:"
        print sys.argv[0], " <url> <username> <password> <iterations>"
        sys.exit(1)
    url = sys.argv[1]
    username = sys.argv[2]
    password = sys.argv[3]
    iterations = int(sys.argv[4])
    # First acquire a valid session by logging in:
    session = XenAPI.Session(url)
    session.xenapi.login_with_password(username, password)
    try:
        for i in range(iterations):
            main(session, i)
    finally:
        session.xenapi.session.logout()
```

The main function examines each running VM in the system, taking care to filter out *control domains* (which are part of the system and not controllable by the user). A list of running VMs and their current hosts is constructed.

```
def main(session, iteration):
    # Find a non-template VM object
    all = session.xenapi.VM.get_all()
    vms = []
    hosts = []
    for vm in all:
        record = session.xenapi.VM.get_record(vm)
        if not(record["is_a_template"]) and \
            not(record["is_control_domain"]) and \
            record["power_state"] == "Running":
            vms.append(vm)
            hosts.append(record["resident_on"])
    print "%d: Found %d suitable running VMs" % (iteration, len(vms))
```

Next the list of hosts is rotated:



```
# use a rotation as a permutation
hosts = [hosts[-1]] + hosts[:(len(hosts)-1)]
```

Each VM is then moved using XenMotion to the new host under this rotation (i.e. a VM running on host at position 2 in the list will be moved to the host at position 1 in the list etc.) In order to execute each of the movements in parallel, the asynchronous version of the **VM.pool_migrate** is used and a list of task references constructed. Note the *live* flag passed to the **VM.pool_migrate**; this causes the VMs to be moved while they are still running.

```
tasks = []
for i in range(0, len(vms)):
    vm = vms[i]
    host = hosts[i]
    task = session.xenapi.Async.VM.pool_migrate(vm, host, { "live": "true" })
    tasks.append(task)
```

The list of tasks is then polled for completion:

```
finished = False
records = {}
while not(finished):
    finished = True
    for task in tasks:
        record = session.xenapi.task.get_record(task)
        records[task] = record
        if record["status"] == "pending":
            finished = False
    time.sleep(1)
```

Once all tasks have left the *pending* state (i.e. they have successfully completed, failed or been cancelled) the tasks are polled once more to see if they all succeeded:

```
allok = True
for task in tasks:
    record = records[task]
    if record["status"] <> "success":
        allok = False
```

If any one of the tasks failed then details are printed, an exception is raised and the task objects left around for further inspection. If all tasks succeeded then the task objects are destroyed and the function returns.

```
if not(allok):
    print "One of the tasks didn't succeed at", \
          time.strftime("%F:%HT%M:%SZ", time.gmtime())
    idx = 0
    for task in tasks:
        record = records[task]
        vm_name = session.xenapi.VM.get_name_label(vms[idx])
        host_name = session.xenapi.host.get_name_label(hosts[idx])
        print "%s : %12s %s -> %s [ status: %s; result = %s; error = %s ]" % \
              (record["uuid"], record["name_label"], vm_name, host_name,
               record["status"], record["result"], repr(record["error_info"]))
        idx = idx + 1
    raise "Task failed"
else:
    for task in tasks:
        session.xenapi.task.destroy(task)
```

4.3.2. Cloning a VM using the XE CLI

This example is a **bash** script which uses the XE CLI to clone a VM taking care to shut it down first if it is powered on.

The example begins with some boilerplate which first checks if the environment variable *XE* has been set: if it has it assumes that it points to the full path of the CLI, else it is assumed that the XE CLI is on the current path. Next the script prompts the user for a server name, username and password:



```
# Allow the path to the 'xe' binary to be overridden by the XE environment variable
if [ -z "${XE}" ]; then
    XE=xe
fi

if [ ! -e "${HOME}/.xe" ]; then
    read -p "Server name: " SERVER
    read -p "Username: " USERNAME
    read -p "Password: " PASSWORD
    XE="${XE} -s ${SERVER} -u ${USERNAME} -pw ${PASSWORD}"
fi
```

Next the script checks its commandline arguments. It requires exactly one: the UUID of the VM which is to be cloned:

```
# Check if there's a VM by the uuid specified
${XE} vm-list params=uuid | grep -q "${vmuuid}"
if [ $? -ne 0 ]; then
    echo "error: no vm uuid \"${vmuuid}\" found"
    exit 2
fi
```

The script then checks the power state of the VM and if it is running, it attempts a clean shutdown. The event system is used to wait for the VM to enter state "Halted".

Note:

The XE CLI supports a command-line argument `--minimal` which causes it to print its output without excess whitespace or formatting, ideal for use from scripts. If multiple values are returned they are comma-separated.

```
# Check the power state of the vm
name=${${XE} vm-list uuid=${vmuuid} params=name-label --minimal}
state=${${XE} vm-list uuid=${vmuuid} params=power-state --minimal}
wasrunning=0

# If the VM state is running, we shutdown the vm first
if [ "${state}" = "running" ]; then
    ${XE} vm-shutdown uuid=${vmuuid}
    ${XE} event-wait class=vm power-state=halted uuid=${vmuuid}
    wasrunning=1
fi
```

The VM is then cloned and the new VM has its `name_label` set to `cloned_vm`.

```
# Clone the VM
newuuid=${${XE} vm-clone uuid=${vmuuid} new-name-label=cloned_vm}
```

Finally, if the original VM had been running and was shutdown, both it and the new VM are started.

```
# If the VM state was running before cloning, we start it again
# along with the new VM.
if [ "$wasrunning" -eq 1 ]; then
    ${XE} vm-start uuid=${vmuuid}
    ${XE} vm-start uuid=${newuuid}
fi
```

Chapter 5. Using HTTP to interact with XenServer

XenServer exposes an HTTP interface on each host, that can be used to perform various operations. This chapter describes the available mechanisms.

5.1. VM Import and Export

Because the import and export of VMs can take some time to complete, an asynchronous HTTP interface to the import and export operations is provided. To perform an export using the XenServer API, construct an HTTP GET call providing a valid session ID, task ID and VM UUID, as shown in the following pseudo code:

```
task = Task.create()
result = HTTP.get(
    server, 80, "/export?session_id=<session_id>&task_id=<task_id>&ref=<vm_uuid>");
```

For the import operation, use an HTTP PUT call as demonstrated in the following pseudo code:

```
task = Task.create()
result = HTTP.put(
    server, 80, "/import?session_id=<session_id>&task_id=<task_id>&ref=<vm_uuid>");
```

5.2. Getting XenServer Performance Statistics

XenServer records statistics about the performance of various aspects of your XenServer installation. The metrics are stored persistently for long term access and analysis of historical trends. Where storage is available to a VM, the statistics are written to disk when a VM is shut down. Statistics are stored in RRDs (Round Robin Databases), which are maintained for individual VMs (including the control domain) and the server. RRDs are resident on the server on which the VM is running, or the pool master when the VM is not running. The RRDs are also backed up every day.

Warning:

In earlier versions of the XenServer API, instantaneous performance metrics could be obtained using the **VM_metrics**, **VM_guest_metrics**, **host_metrics** methods and associated methods. These methods has been deprecated in favor of using the http handler described in this chapter to download the statistics from the RRDs on the VMs and servers. Note that by default the legacy metrics will return zeroes. To revert to periodic statistical polling as present in earlier versions of XenServer, set the *other-config:rrd_update_interval=<interval>* parameters on your host to one of the following values, and restart your host:

- never This is the default, meaning no periodic polling is performed.
- 1 Polling is performed every 5 seconds.
- 2 Polling is performed every minute.

By default, the older metrics APIs will not return any values, and so this key must be enabled to run monitoring clients which use the legacy monitoring protocol.

Statistics are persisted for a maximum of one year, and are stored at different granularities. The average and most recent values are stored at intervals of:

- 5 seconds for the past 10 minutes
- one minute for the past 2 hours
- one hour for the past week



- one day for the past year

RRDs are saved to disk as uncompressed XML. The size of each RRD when written to disk ranges from 200KiB to approximately 1.2MiB when the RRD stores the full year of statistics.

Warning:

If statistics cannot be written to disk, for example when a disk is full, statistics will be lost and the last saved version of the RRD will be used.

Statistics can be downloaded over HTTP in XML format, for example using **wget**. See <http://oss.oetiker.ch/rrdtool/doc/rrddump.en.html> and <http://oss.oetiker.ch/rrdtool/doc/rrdexport.en.html> for information about the XML format. HTTP authentication can take the form of a username and password or a session token. Parameters are appended to the URL following a question mark (?) and separated by ampersands (&).

To obtain an update of all VM statistics on a host, the URL would be of the form:

```
http://<username>:<password>@<host>/rrd_updates?start=<secondssinceepoch>
```

This request returns data in an rrdtool `xport` style XML format, for every VM resident on the particular host that is being queried. To differentiate which column in the export is associated with which VM, the `legend` field is prefixed with the UUID of the VM.

To obtain host updates too, use the query parameter `host=true`:

```
http://<username>:<password>@<host>/rrd_updates?start=<secondssinceepoch>&host=true
```

The step will decrease as the period decreases, which means that if you request statistics for a shorter time period you will get more detailed statistics.

Additional `rrd_updates` parameters

`cf=<ave|min|max>` the data consolidation mode
`interval=<interval>` the interval between values to be reported

Note:

By default only `ave` statistics are available. To obtain `min` and `max` statistics for a VM, run the following command:

```
xe pool-param-set uuid=<pool_uuid> other-config:create_min_max_in_new_VM_RRDs
```

To obtain all statistics for a host:

```
http://<username:password@host>/host_rrd
```

To obtain all statistics for a VM:

```
http://<username:password@host>/vm_rrd?uuid=<vm_uuid>
```

Chapter 6. XenServer API extensions

The XenAPI is a general and comprehensive interface to managing the life-cycles of Virtual Machines, and offers a lot of flexibility in the way that XenAPI providers may implement specific functionality (e.g. storage provisioning, or console handling). XenServer has several extensions which provide useful functionality used in our own XenCenter interface. The workings of these mechanisms are described in this chapter.

Extensions to the XenAPI are often provided by specifying *other-config* map keys to various objects. The use of this parameter indicates that the functionality is supported for that particular release of XenServer, but *not* as a long-term feature. We are constantly evaluating promoting functionality into the API, but this requires the nature of the interface to be well-understood. Developer feedback as to how you are using some of these extensions is always welcome to help us make these decisions.

6.1. VM console forwarding

Most XenAPI graphical interfaces will want to gain access to the VM consoles, in order to render them to the user as if they were physical machines. There are several types of consoles available, depending on the type of guest or if the physical host console is being accessed:

Console access

Operating System	Text	Graphical	Optimized graphical
Windows	No	VNC, using an API call	RDP, directly from guest
Linux	Yes, through VNC and an API call	No	VNC, directly from guest
Physical Host	Yes, through VNC and an API call	No	No

Hardware-assisted VMs, such as Windows, directly provide a graphical console over VNC. There is no text-based console, and guest networking is not necessary to use the graphical console. Once guest networking has been established, it is more efficient to setup Remote Desktop Access and use an RDP client to connect directly (this must be done outside of the XenAPI).

Paravirtual VMs, such as Linux guests, provide a native text console directly. XenServer provides a utility (called **vncterm**) to convert this text-based console into a graphical VNC representation. Guest networking is not necessary for this console to function. As with Windows above, Linux distributions often configure VNC within the guest, and directly connect to it over a guest network interface.

The physical host console is only available as a `vt100` console, which is exposed through the XenAPI as a VNC console by using **vncterm** in the control domain.

RFB (Remote Framebuffer) is the protocol which underlies VNC, specified in [The RFB Protocol](#). Third-party developers are expected to provide their own VNC viewers, and many freely available implementations can be adapted for this purpose. RFB 3.3 is the minimum version which viewers must support.

6.1.1. Retrieving VNC consoles using the API

VNC consoles are retrieved using a special URL passed through to the host agent. The sequence of API calls is as follows:

1. Client to Master/443: XML-RPC: **Session.login_with_password()**.
2. Master/443 to Client: Returns a session reference to be used with subsequent calls.



3. Client to Master/443: XML-RPC: **VM.get_by_name_label()**.
4. Master/443 to Client: Returns a reference to a particular VM (or the "control domain" if you want to retrieve the physical host console).
5. Client to Master/443: XML-RPC: **VM.get_consoles()**.
6. Master/443 to Client: Returns a list of console objects associated with the VM.
7. Client to Master/443: XML-RPC: **VM.get_location()**.
8. Returns a URI describing where the requested console is located. The URIs are of the form: <https://192.168.0.1/console?ref=OpaqueRef:c038533a-af99-a0ff-9095-c1159f2dc6a0>.
9. Client to 192.168.0.1: HTTP CONNECT `"/console?ref=(...)"`

The final HTTP CONNECT is slightly non-standard since the HTTP/1.1 RFC specifies that it should only be a host and a port, rather than a URL. Once the HTTP connect is complete, the connection can subsequently directly be used as a VNC server without any further HTTP protocol action.

This scheme requires direct access from the client to the control domain's IP, and will not work correctly if there are Network Address Translation (NAT) devices blocking such connectivity. You can use the CLI to retrieve the console URI from the client and perform a connectivity check.

To retrieve a console URI using the CLI

1. Retrieve the VM UUID by running:

```
xe vm-list params=uuid --minimal name-label=name
```

2. Retrieve the console information:

```
xe console-list vm-uuid=uuid
uuid ( RO): 714f388b-31ed-67cb-617b-0276e35155ef
vm-uuid ( RO): 8acb7723-a5f0-5fc5-cd53-9f1e3a7d3069
vm-name-label ( RO): etch
protocol ( RO): RFB
location ( RO): https://192.168.0.1/console?ref=(...)
```

Use command-line utilities like **ping** to test connectivity to the IP address provided in the *location* field.

6.1.2. Disabling VNC forwarding for Linux VM

When creating and destroying Linux VMs, the host agent automatically manages the **vncterm** processes which convert the text console into VNC. Advanced users who wish to directly access the text console can disable VNC forwarding for that VM. The text console can then only be accessed directly from the control domain directly, and graphical interfaces such as XenCenter will not be able to render a console for that VM.

Disabling a Linux VNC console using the CLI

1. Before starting the guest, set the following parameter on the VM record:

```
xe vm-param-set uuid=uuid other-config:disable_pv_vnc=1
```

2. Start the VM.
3. Use the CLI to retrieve the underlying domain ID of the VM with:

```
xe vm-list params=dom-id uuid=<uuid> --minimal
```

4. On the host console, connect to the text console directly by:

```
/usr/lib/xen/bin/xenconsole <domain_id>
```

This configuration is an advanced procedure, and we do not recommend that the text console is directly used for heavy I/O operations. Instead, connect to the guest over SSH or some other network-based connection mechanism.

6.2. Paravirtual Linux installation

The installation of paravirtual Linux guests is complicated by the fact that a Xen-aware kernel must be booted, rather than simply installing the guest using hardware-assistance. This does have the benefit of providing near-native installation speed due to the lack of emulation overhead. XenServer supports the installation of several different Linux distributions, and abstracts this process as much as possible.

To this end, a special bootloader known as **eliloader** is present in the control domain which reads various *other-config* keys in the VM record at start time and performs distribution-specific installation behavior.

- *install-repository* - Required. Path to a repository; 'http', 'https', 'ftp', or 'nfs'. Should be specified as would be used by the target installer, but not including prefixes, e.g. method=.
- *install-vnc* - Default: false. Use VNC where available during the installation.
- *install-vncpasswd* - Default: empty. The VNC password to use, when providing one is possible using the command-line of the target distribution.
- *install-round* - Default: 1. The current bootloader round. Not to be edited by the user (see below)

6.2.1. Red Hat Enterprise Linux 4.1/4.4

eliloader is used for two rounds of booting. In the first round, it returns the installer `initrd` and kernel from `/opt/xen/source/packages/files/guest-installer`. Then, on the second boot, it removes the additional updates disk from the VM, switches the bootloader to **pygrub**, and then begins a normal boot.

This sequence is required since Red Hat does not provide a Xen kernel for these distributions, and so the XenServer custom kernels for those distributions are used instead.

6.2.2. Red Hat Enterprise Linux 4.5/5.0

Similar to the RHEL4.4 installation, except that the kernel and ramdisk are downloaded directly from the network repository that was specified by the user, and switch the bootloader to **pygrub** immediately. Note that `pygrub` is not executed immediately, and so will only be parsed on the next boot.

The network retrieval enables users to install the upstream Red Hat vendor kernel directly from their network repository. An updated XenServer kernel is also provided on the `xs-tools.iso` built-in ISO image which fixes various Xen-related bugs.

6.2.3. SUSE Enterprise Linux 10 SP1

This requires a two-round boot process. The first round downloads the kernel and ramdisk from the network repository and boots them. The second round then inspects the disks to find the installed kernel and ramdisk, and sets the `PV-bootloader-args` to reflect these paths within the guest filesystem. This process emulates the **domUloader** which SUSE use as an alternative to **pygrub**. Finally, the bootloader is set to **pygrub** and is executed to begin a normal boot.

The SLES 10 installation method means that the path for the kernel and ramdisk is stored in the VM record rather than in the guest `menu.lst`, but this is the only way it would ever work since the YAST package manager doesn't write a valid `menu.lst`.

6.2.4. CentOS 4.5 / 5.0

The CentOS installation mechanism is similar to that of the Red Hat installation notes above, save that some MD5 checksums are different which **eliloader** recognizes.

6.3. Adding Xenstore entries to VMs

Developers may wish to install guest agents into VMs which take special action based on the type of the VM. In order to communicate this information into the guest, a special Xenstore name-space known as `vm-data` is available which is populated at VM creation time. It is populated from the `xenstore-data` map in the VM record.

To populate a Xenstore node *foo* in a VM

1. Set the *xenstore-data* parameter in the VM record:

```
xe vm-param-set uuid=<vm_uuid> xenstore-data:vm-data/foo=bar
```
2. Start the VM.
3. If it is a Linux-based VM, install the XenServer Tools and use the **xenstore-read** to verify that the node exists in Xenstore.

Note:

Only prefixes beginning with *vm-data* are permitted, and anything not in this name-space will be silently ignored when starting the VM.

6.4. Security enhancements

The control domain in XenServer 6.1.0 and above has various security enhancements in order to harden it against attack from malicious guests. Developers should never notice any loss of correct functionality as a result of these changes, but they are documented here as variations of behavior from other distributions.

- The socket interface, **xenstored**, access using `libxenstore`. Interfaces are restricted by `xs_restrict()`.
- The device `/dev/xen/evtchn`, which is accessed by calling `xs_evtchn_open()` in `libxenctrl`. A handle can be restricted using `xs_evtchn_restrict()`.
- The device `/proc/xen/privcmd`, accessed through `xs_interface_open()` in `libxenctrl`. A handle is restricted using `xc_interface_restrict()`. Some privileged commands are naturally hard to restrict (e.g. the ability to make arbitrary hypercalls), and these are simply prohibited on restricted handles.
- A restricted handle cannot later be granted more privilege, and so the interface must be closed and re-opened. Security is only gained if the process cannot subsequently open more handles.

The control domain privileged user-space interfaces can now be restricted to only work for certain domains. There are three interfaces affected by this change:

- The **qemu** device emulation processes and **vncterm** terminal emulation processes run as a non-root user ID and are restricted into an empty directory. They use the restriction API above to drop privileges where possible.
- Access to xenstore is rate-limited to prevent malicious guests from causing a denial of service on the control domain. This is implemented as a token bucket with a restricted fill-rate, where most operations take one token and opening a transaction takes 20. The limits are set high enough that they should never be hit when running even a large number of concurrent guests under loaded operation.
- The VNC guest consoles are bound only to the `localhost` interface, so that they are not exposed externally even if the control domain packet filter is disabled by user intervention.

6.5. Advanced settings for network interfaces

Virtual and physical network interfaces have some advanced settings that can be configured using the *other-config* map parameter. There is a set of custom *ethtool* settings and some miscellaneous settings.

6.5.1. ethtool settings

Developers might wish to configure custom *ethtool* settings for physical and virtual network interfaces. This is accomplished with *ethtool-<option>* keys in the *other-config* map parameter.

Key	Description	Valid settings
<code>ethtool-rx</code>	Specify if RX checksumming is enabled	<code>on</code> or <code>true</code> to enable the setting, <code>off</code> or <code>false</code> to disable it

Key	Description	Valid settings
ethtool-tx	Specify if TX checksumming is enabled	on or true to enable the setting, off or false to disable it
ethtool-sg	Specify if scatter-gather is enabled	on or true to enable the setting, off or false to disable it
ethtool-tso	Specify if tcp segmentation offload is enabled	on or true to enable the setting, off or false to disable it
ethtool-ufo	Specify if UDP fragmentation offload is enabled	on or true to enable the setting, off or false to disable it
ethtool-gso	Specify if generic segmentation offload is enabled	on or true to enable the setting, off or false to disable it
ethtool-autoneg	Specify if autonegotiation is enabled	on or true to enable the setting, off or false to disable it
ethtool-speed	Set the device speed in Mb/s	10, 100, or 1000
ethtool-duplex	Set full or half duplex mode	half or full

For example, to enable TX checksumming on a virtual NIC using the xe CLI:

```
xe vif-param-set uuid=<VIF UUID> other-config:ethtool-tx="on"
```

or:

```
xe vif-param-set uuid=<VIF UUID> other-config:ethtool-tx="true"
```

To set the duplex setting on a physical NIC to half duplex using the xe CLI:

```
xe vif-param-set uuid=<VIF UUID> other-config:ethtool-duplex="half"
```

6.5.2. Miscellaneous settings

You can also set a promiscuous mode on a VIF or PIF by setting the *promiscuous* key to on. For example, to enable promiscuous mode on a physical NIC using the xe CLI:

```
xe pif-param-set uuid=<PIF UUID> other-config:promiscuous="on"
```

or:

```
xe pif-param-set uuid=<PIF UUID> other-config:promiscuous="true"
```

The VIF and PIF objects have a *MTU* parameter that is read-only and provide the current setting of the maximum transmission unit for the interface. You can override the default maximum transmission unit of a physical or virtual NIC with the *mtu* key in the *other-config* map parameter. For example, to reset the MTU on a virtual NIC to use jumbo frames using the xe CLI:

```
xe vif-param-set uuid=<VIF UUID> other-config:mtu=9000
```

Note that changing the MTU of underlying interfaces is an advanced and experimental feature, and may lead to unexpected side-effects if you have varying MTUs across NICs in a single resource pool.



6.6. Internationalization for SR names

The SRs created at install time now have an *other_config* key indicating how their names may be internationalized.

other_config["i18n-key"] may be one of

- *local-hotplug-cd*
- *local-hotplug-disk*
- *local-storage*
- *xenserver-tools*

Additionally, *other_config["i18n-original-value-<field name>"]* gives the value of that field when the SR was created. If XenCenter sees a record where *SR.name_label* equals *other_config["i18n-original-value-name_label"]* (that is, the record has not changed since it was created during XenServer installation), then internationalization will be applied. In other words, XenCenter will disregard the current contents of that field, and instead use a value appropriate to the user's own language.

If you change *SR.name_label* for your own purpose, then it no longer is the same as *other_config["i18n-original-value-name_label"]*. Therefore, XenCenter does not apply internationalization, and instead preserves your given name.

6.7. Hiding objects from XenCenter

Networks, PIFs, and VMs can be hidden from XenCenter by adding the key `HideFromXenCenter=true` to the *other_config* parameter for the object. This capability is intended for ISVs who know what they are doing, not general use by everyday users. For example, you might want to hide certain VMs because they are cloned VMs that shouldn't be used directly by general users in your environment.

In XenCenter, hidden Networks, PIFs, and VMs can be made visible, using the **View** menu.

Chapter 7. XenCenter API Extensions

The following section details the assumptions and API extensions that we have made, over and above the documented API. Extensions are encoded as particular key-value pairs in dictionaries such as `VM.other_config`.

7.1. Pool

Key	Semantics
<code>pool.name_label</code>	An empty <code>name_label</code> indicates that the pool should be hidden on the tree view.
<code>pool.rolling_upgrade_in_progress</code>	Present if the pool is in the middle of a rolling upgrade.

7.2. Host

Key	Semantics
<code>host.other_config["iscsi_iqn"]</code>	The host's iSCSI IQN.
<code>host.license_params["expiry"]</code>	The expiry date of the host's license, in ISO 8601, UTC.
<code>host.license_params["sku_type"]</code>	The host licence type i.e. Server or Enterprise.
<code>host.license_params["restrict_pooling"]</code>	Returns <code>true</code> if pooling is restricted by the host.
<code>host.license_params["restrict_connection"]</code>	The number of connections that can be made from XenCenter is restricted.
<code>host.license_params["restrict_qos"]</code>	Returns <code>true</code> if Quality of Service settings are enabled on the host.
<code>host.license_params["restrict_vlan"]</code>	Returns <code>true</code> if creation of virtual networks is restricted on the host.
<code>host.license_params["restrict_pool_attached_storage"]</code>	Returns <code>true</code> if the creation of shared storage is restricted on this host.
<code>host.software_version["product_version"]</code>	Returns the host's product version.
<code>host.software_version["build_number"]</code>	Returns the host's build number.
<code>host.software_version["xapi"]</code>	Returns the host's api revision number.
<code>host.software_version["package-linux"]</code>	Returns "installed" if the Linux pack has been installed.
<code>host.software_version["oem_build_number"]</code>	If the host is the OEM version, return its revision number.
<code>host.logging["syslog_destination"]</code>	Gets or sets the destination for the XenServer system logger (null for local logging).

Key	Semantics
host.logging["multipathing"]	"true" if storage multipathing is enabled on this host.
host.logging["boot_time"]	A floating point Unix time giving the time that the host booted.
host.logging["agent_start_time"]	A floating point Unix time giving the time that the control domain management daemon started.

7.3. VM

Key	Semantics
VM.other_config["default_template"]	This template is one that was installed by Citrix. This is used to selectively hide these in the tree view, to use a different icon for them, and to disallow deletion.
VM.other_config["xensource_internal"]	This template is special, such as the P2V server template. These are completely hidden by the UI.
VM.other_config["install_distro"] == "rhlike"	This template is for RHEL 4.5, RHEL 5, or CentOS equivalents. This is used to prompt for the Install Repository during install, including support for install from ISO / CD on Miami, and to modify NFS URLs to suit these installers.
VM.other_config["install_distro"] in { "rhel41" "rhel44" }	This template is for RHEL 4.1, RHEL 4.4, or CentOS equivalents. This is used to prompt for the Install Repository during install, and to modify NFS URLs to suit these installers. No ISO support is available for these templates.
VM.other_config["install_distro"] == "sleslike"	This template is for SLES 10 and SLES 9. This is used to prompt for the Install Repository during install, like the EL5 ones, but in this case the NFS URLs are not modified. ISO support is available for SLES 10 on XenServer 6.1.0. Use <i>install-methods</i> to distinguish between SLES 9 and SLES 10 on that platform.
VM.other_config["install-repository"] == "cdrom"	Requests an install from a repository in the VM's attached CD drive, rather than a URL.
VM.other_config["auto_poweron"]	Gets or sets whether the VM starts when the server boots, "true" or "false".
VM.other_config["ignore_excessive_vcpus"]	Gets or sets to ignore XenCenter's warning if a VM has more VCPUs than its host has physical CPUs, <code>true</code> to ignore.

Key	Semantics
VM.other_config["HideFromXenCenter"]	Gets or sets whether XenCenter will show the VM in the treeview, "true" to hide.
VM.other_config["import_task"]	Gets the import task that created this VM.
VM.HVM_boot_params["order"]	Gets or sets the VM's boot order on HVM VM's only, e.g. "CDN" will boot in the following order - First boot disk, CD drive, Network.
VM.VCPU_params["weight"]	Gets or sets the IONice value for the VM's VCPUs, ranges from 1 to 65536, 65536 being the highest.
VM.pool_migrate(..., options['live'])	<code>true</code> indicates live migration. XenCenter always uses this.
VM.other_config["install-methods"]	A comma-separated list of install methods available for this template. May include "cdrom", "nfs", "http" or "ftp".
VM.other_config["last_shutdown_time"]	The time that this VM was last shut down or rebooted, formatted as a UTC ISO8601 datetime.
VM.other_config["p2v_source_machine"]	The source machine, if this VM was imported by a P2V process.
VM.other_config["p2v_import_date"]	The date the VM was imported, if it was imported by a P2V process. Formatted as a UTC ISO8601 datetime.

7.4. SR

Key	Semantics
SR.other_config["auto-scan"]	The SR will be automatically scanned for changes. Set on all SRs created by XenCenter.
SR.sm_config["type"]	Set as type <code>cd</code> for SRs which are physical CD drives.

7.5. VDI

Key	Semantics
VDI.type	<code>user</code> instead of <code>system</code> is used to mean "do or do not allow deletion of the VDI through the GUI, if this disk is attached to a VM". The intention here is to prevent you from corrupting a VM (you should uninstall it instead). <code>suspend</code> and <code>crashdump</code> record suspend and core dumps respectively. <code>ephemeral</code> is currently unused.

Key	Semantics
VDI.managed	All unmanaged VDIs are completely hidden in the UI. These are branch points in VHD chains, or unused LUN-per-VDI disks.
VDI.sm_config["vmhint"]	The UUID of the VM that this VDI supports. This is set when VDIs are created through the user interface, to improve performance for certain storage backends.

7.6. VBD

Key	Semantics
VBD.other_config["is_owner"]	If set, then this disk may be deleted when the VM is uninstalled.
VBD.other_config["class"]	Set to an integer, corresponding to the Best Effort setting of ionice .

7.7. Network

Key	Semantics
network.other_config["automatic"]	The New VM wizard will create a VIF connected to this network by default, if this key has any value other than <code>false</code> .
network.other_config["import_task"]	Gets the import task that created this network.

7.8. VM_guest_metrics

Key	Semantics
PV_drivers_version["major"]	Gets the major version of the VM's PV drivers' version.
PV_drivers_version["minor"]	Gets the minor version of the VM's PV drivers' version.
PV_drivers_version["micro"]	Gets the micro (build number) of the VM's PV drivers' version.

7.9. Task

Key	Semantics
task.other_config["object_creation"] == "complete"	For the task associated with a VM import, this flag will be set when all the objects (VMs, networks) have been created. This is useful in the import VM wizard for us to then go and re-map all the networks that need it.