

# Xen Cloud Platform Storage Manager API

Xen Cloud Platform  
Storage Manager API

# **Xen Cloud Platform Storage Manager API**

# Contents

---

<b>1. Overview</b> .....	<b>5</b>
XAPI Agent Handling of SR Operations .....	6
Ordering, Concurrency, and Locking .....	7
SR-Level API Operations .....	7
sr_create() .....	8
sr_delete() .....	8
sr_attach() .....	9
sr_detach() .....	9
sr_content_type() .....	10
sr_scan() .....	10
sr_probe() .....	12
sr_get_driver_config() .....	12
sr_update() .....	12
VDI-Level API Operations .....	12
vdi_create() .....	12
vdi_delete() .....	13
vdi_attach() .....	13
vdi_detach() .....	13
vdi_resize() .....	13
vdi_activate() .....	13
vdi_deactivate() .....	14
vdi_snapshot() .....	14
vdi_clone() .....	14
vdi_update() .....	14
vdi_introduce() .....	14
Non-Primitive Types .....	15
vdi_info .....	15
device_config .....	15
driver_params .....	15
vdi_sm_config .....	15
driver_info .....	16
nil .....	16
Message Syntax .....	16
Call Syntax .....	17
Error Codes .....	18
Current Driver Parameters .....	24
A. Mapping of xe calls to SMAPI calls .....	27
Details .....	27
Introduction .....	27

Storage Repository Calls .....	27
PBD Calls .....	27
Virtual Machine Calls .....	28
VDI Calls .....	29
VBD calls .....	29
The test script .....	30

# Chapter 1. Overview

The Storage Manager API (SMAPI) provides a framework for IHV/ISVs to develop storage manager plugins to integrate new vendor-specific types of storage into Xen Cloud Platform. Rather than prescribing a specific approach to managing storage or placing data on disk, the Xen Cloud Platform architecture exposes storage-related operations through this API and enables storage vendors to directly expose the specific features and best practices of their products to Xen Cloud Platform VMs. The SMAPI makes it possible to do things like directly provisioning new LUNs to be associated with a newly created virtual machine, expose on-array snapshot capabilities, and allowing Xen Cloud Platform systems to directly present advanced storage features such as thin provisioning and deduplication.

The SMAPI is implemented as a series of python classes. There are two main objects that the storage manager uses to describe storage: A Storage Repository (SR) is a single instance of storage (such as a local disk or an NFS mount point) that will contain a set of Virtual Disk Images (VDIs). A VDI is a virtual disk, and may be attached to a virtual machine in Xen Cloud Platform. An SR represents a homogeneous interface to a set of storage and provides a common set of mechanisms to interact with the VDIs that it contains. One example of an SR is that of shared NFS, where individual VDIs are each stored in their own VHD-format image file on an NFS filer. Another example is that of the local LVM SR, in which a directly attached disk is managed using the Linux logical volume manager, and the resulting volumes each represent a VDI. The SMAPI abstractions provide simple mechanisms to accommodate the wide range of storage possibilities directly within Xen Cloud Platform's control and management interfaces. Note that in all cases, VMs are unaware of the specific SR implementation being used; VDIs from any SR instance are presented simply as block devices to the guest VM.

There are several SMAPI plugins currently deployed with the Xen Cloud Platform SDK. Included among these is an example implementation ("`DummySR.py`") which can be used as a starting point for new storage manager plugin development. The DummySR implementation provides a useful debugging interface into the SMAPI calls and how they are accessed by the higher level XAPI operations. Developers hoping to get some insight into how to implement new storage plugins should experiment with the DummySR code and begin by understanding the storage primitives expected of the plugin API.

The following sections describe the Xen Cloud Platform Storage Manager APIs in detail. The SMAPI implementation in Xen Cloud Platform currently has drivers for VHD image files on local disk or a shared NFS filer, and both local and shared block devices (DAS, iSCSI, and HBA-based) in which a single LUN is subdivided using the Linux logical volume manager. There is additionally an integrated driver for Network Appliance filers that maps VDIs to individual LUNs that it provisions on demand. Current SMAPI drivers are written in Python. The source code of these implementations can be found at `/opt/xensource/sm` on a typical Xen Host installation, providing model code for new driver development.

---

## Note

Place your finished storage manager plugin in the `/opt/xensource/sm` directory and ensure that it is suffixed with `SR`, for example, a python plugin entitled

`MyStorageManagerSR.py` should be symlinked to a plugin interface named `MyStorageManagerSR`, so that it can be discovered at run time by the Xen Host.

---

As mentioned above, the SMAPI aims to provide an abstract interface to a heterogeneous pool of storage. When the XenAPI agent (XAPI) needs to find / create / destroy a disk for a virtual machine it will make one or more SMAPI calls to the appropriate plugin. The calls are written in XMLRPC syntax, where the `methodCall` is transmitted on the command line to the plugin, which returns a `methodResponse` (possibly including a fault) on `stdout` with a return code of zero for success, or a plugin defined error greater than zero.

Each SMAPI call (except `sr_get_driver_info()`) provides the driver a pre-authenticated XenAPI session which can be used to perform XenAPI calls on the server. The session has access to normally read-only fields in the XenAPI SR, PBD and VDI objects which are managed by the plugin. Most SMAPI calls use the XenAPI to modify object fields directly rather than return complex results to the server as XMLRPC. Using the python framework, however can greatly simplify the management and manipulation of the driver owned database fields since the inheritance of the basic driver classes will ensure that data is synchronized and fields are updated.

## XAPI Agent Handling of SR Operations

Within the XAPI agent, SR requests translate down to a set of operations on the storage medium itself. All metadata related to the internal consistency of SR data objects is handled by the storage medium driver which is always considered to be authoritative. The availability and implementation of SR operations is based on the ability of the storage medium to report and provide access to storage operations. Specific implementation details are provided throughout this API document to demonstrate how operations are implemented on different storage medium types. It is the responsibility of the backend driver to ensure that database fields, e.g. the read-only VDI status field, are updated correctly.

Furthermore:

- The SR driver does not provide any secure access control to VDIs.
- The SMAPI provides suitable hooks in the VDI attach sequence which enable the backend driver to implement exclusive host access to VDIs, e.g. SCSI persistent reservations or file-based locks, such that concurrent write access can be avoided. Exclusive access to disks is not a requirement, however and it is up to the driver to specify its capabilities as outlined in the DummySR driver.
- The host management tool will always ensure that any cached SR data such as VDI database entries remain consistent with the actual state on disk. The `sr_scan()` driver command is provided for this purpose and is called periodically by the management tool to interrogate the SR's contents. As indicated earlier, inheriting the base python framework will simplify this database record management and is strongly recommended as a starting point.
- The host manager will always explicitly lock and unlock a VDI before and after use by calling `vd.activate()` and `vd.deactivate()`. If locking is not provided at the SR level the driver must not declare the capability in the `get_driver_info()` data.

## Ordering, Concurrency, and Locking

SR implementations depend on the XAPI agent to provide concurrency control. The XAPI agent, which may manage multiple hosts connected to the same SR, will apply the following concurrency properties equally across all hosts in the managed pool:

1. Any operation that involves creating or deleting a specific VDI is serialized with respect to the enclosing SR.
2. SR attach and detach are serialized on a per-host/SR basis. Different hosts may be calling SR attach or detach on the same SR at the same time, but on one single host, these calls to the same SR are serialized.
3. VDI operations, other than creation and deletion are serialized on a per-VDI basis; multiple operations of these types may be performed concurrently within the same SR.
4. The SR scan operation is serialized with respect to all other operations in the enclosing SR so that it can return consistent results.

In light of these guidelines, SR backend implementations may provide the following:

1. Lock and unlock calls for exclusive data access to VDIs for shared SRs. The management tools will call **vdi\_activate()** before any data access to a VDI and will release it using **vdi\_deactivate()** when access is complete.
2. Crash recovery must be provided in the case that a physical host crashes part way through an operation. This recovery may involve documented command line interactions, but the SR should not be corrupted as a result of such crashes.

## SR-Level API Operations

SR-Level operations allow for the management of entire SRs. There are a limited number of these calls, and they serve three broad purposes:

- **sr\_create()** and **sr\_delete()** are used to create and destroy storage repositories. **sr\_create()** takes a storage medium, for example a physical disk or NFS mount point, and would prepare it to hold VDIs, e.g. insert a volume manager, or format the disk in preparation of serving files as VDIs. **sr\_delete()** takes an empty SR and performs any additional clean-up that is required to return the specified device or remote storage medium to a clean state.
- **sr\_attach()** and **sr\_detach()** are used to connect and disconnect an SR from a given XE host. VDI operations are only available on SRs that have first been attached.
- **sr\_scan()** is used to query the contents of an SR and is called periodically by the XE management tools. This allows the tools to discover updates to the SR such as the addition or removal of VDIs that were not explicitly carried out through the XE interfaces.

---

**Note**

Note that if you are using the python framework, the arguments fields in this section describe the arguments that are passed to the associated SR and VDI functions. A more detailed view of parameter passing mechanics is provided later in Sections 7 and 8.

---

**sr\_create()**

Arguments	SR-UUID, Size
Description	Create an SR of size [Size] bytes using the given <code>device-config</code> structure. This operation may fail if it believes that an SR is already installed on the disk.
Result	Create an SR of size [Size] bytes using the given <code>device-config</code> structure. This operation may fail if it believes that an SR is already installed on the disk.
Examples	<p>[LVM] - The operation extracts the physical device list from the <code>device-config</code> structure and attempts to create a Volume Group named <code>VG_XenStorage-<i>&lt;&lt;SR-UUID&gt;&gt;</i></code> across all the physical devices. If a Volume Group already exists for which any physical device is a member, the operation will fail.</p> <p>[NFS] - The driver mounts the target directory and creates a directory named by the SR-UUID. If the directory already exists or any of the target attachment parameters are incorrect, the operation will fail.</p>
Relevant Error codes	ENODEV - No such device EEXIST - SR already exists ENOSYS - Function not implemented

**sr\_delete()**

Arguments	SR-UUID
Description	Deletes the specified SR, leaving the storage medium in a clean state.
Precondition	All VDIs must be deleted in order for the <b>sr_delete()</b> call to succeed. It is the responsibility of the driver to verify that this is the case.
Result	Returns SUCCESS or FAILURE + error string. The operation <i>is</i> idempotent and should succeed if the SR exists and can be deleted or if the SR does not exist. The call may FAIL if any VDIs in the SR are in use or have not been deleted.

Arguments	SR-UUID
Examples	<p>[LVM] - If it exists the Volume Group identified as VG_XenStorage-<i>&lt;&lt;SR-UUID&gt;&gt;</i> is removed and all LVM meta-data removed from each physical device.</p> <p>[NFS] - Not implemented as this is a shared storage medium</p> <p>[EXT] - The device must be in an unmounted state initially, and the driver satisfied that there are no VDIs present. The device is wiped by removing at least 100MB of data from the start of the disk.</p>
Relevant Error codes	<p>ENODEV - No such device</p> <p>ENOSYS - Function not implemented</p>

## sr\_attach()

Arguments	SR-UUID
Description	Initiate local access to the SR. Initialized any device state required to access the storage medium.
Result	Returns SUCCESS or FAILURE and an error string. The operation <i>is</i> idempotent and will return success if the SR can be attached or if the SR is already attached.
Examples	<p>[LVM] - Verify the Volume Group VG_XenStorage-<i>&lt;&lt;SR-UUID&gt;&gt;</i> exists and explicitly set it to active.</p> <p>[NFS] - If not already attached, mount the target NFS directory on the local SR mountpoint path. The call may fail if the target SR directory does not exist.</p>
Relevant Error codes	<p>ENODEV - No such device</p> <p>EIO - General IO error occurred attaching device</p>

## sr\_detach()

Arguments	SR-UUID
Description	Remove local access to the SR. Destroys any device state initiated by the <b>sr_attach()</b> operation.
Result	Returns SUCCESS or FAILURE + error string. The operation <i>is</i> idempotent and will return success if the SR can be detached or if the SR is already detached. All VDIs must be detached in order for the operation to succeed.

Arguments	SR-UUID
Examples	<p>[LVM] - Verify the Volume Group <code>VG_XenStorage-<i>&lt;&lt;SR-UUID&gt;&gt;</i></code> exists and explicitly set it to inactive.</p> <p>[NFS] - If not already detached, unmount the target NFS directory from the local SR mountpoint path and, if successful, remove the mountpoint from the local host filesystem.</p>
Relevant Error codes	<p>ENODEV - No such device</p> <p>EBUSY - Device in use, unable to detach</p>

### sr\_content\_type()

Arguments	SR-UUID
Description	Reports the SR content type used by the agent to determine the Block device driver to initialize
Result	Returns an XML string with a 'type' object
Examples	<p>[LVM] - Returns a value of <code>phy</code> to indicate that the blkback in-kernel IO driver should handle the disk IO:</p> <pre>&lt;?xml version="1.0" ?&gt;&lt;sr&gt;&lt;type&gt;"aio"&lt;/type&gt;&lt;/sr&gt;</pre> <p>[NFS] - Returns a value of <code>vhd</code> to indicate that the blktap VHD driver should handle the disk IO.</p> <pre>&lt;?xml version="1.0" ?&gt;&lt;sr&gt;&lt;type&gt;"vhd"&lt;/type&gt;&lt;/sr&gt;</pre>

### sr\_scan()

Arguments	SR-UUID
Description	Verify the SR contents on the physical storage medium and make any changes in the local host database to ensure that the contents are consistent. The SR backend driver is always considered authoritative in this respect.
Result	<p>Called asynchronously by the agent (the frequency can be adjusted by the XAPI interface) and should always return success given that the <code>sr_attach</code> has previously succeeded:</p> <p><b>Table 1.1. HIGH-LEVEL SR OBJECTS which should be synchronized with the agent database:</b></p> <pre>&lt;physical_utilization&gt;</pre>

Arguments	SR-UUID
	The amount of physical space in bytes consumed by VDIs
<virtual_allocation>	The virtual disk space in bytes allocated to this SR
<physical_size>	The physical disk space in bytes allocated to this SR
<label>	A user generated tag for identifying the SR
<description>	A longer user generated SR description string

**Table 1.2. PER-VDI SUB OBJECTS:**

<uuid>	A globally unique VDI identifier conforming to OSF DCE 1.1.
<utilisation>	The amount of physical space in Bytes consumed by this VDI
<size>	The virtual size of the disk in Bytes.
<label>	A user-generated tag for identifying the VDI
<description>	A longer user generated VDI description string
<shareable>	Whether the VDI can safely be shared across hosts
<read_only>	Whether the disk is read-only accessible
<parent>	If this VDI has a content dependency, specify the UUID of the source
<hidden>	The accessibility status of the VDI, e.g. if this VDI was a Copy-on-Write instance that should not be accessed as an RW VDI, but should reside in the database, e.g. dependency logging, specify the hidden flag.

Arguments	SR-UUID
Relevant Error codes	ENODEV - No such SR

### sr\_probe()

Arguments	none
Description	Discovers and reports information about all SRs.
Result	Returns an XML string containing information about all discovered SRs.

### sr\_get\_driver\_config()

Arguments	none
Description	Reads information about the driver.
Result	Returns an XML string containing driver configuration information.

### sr\_update()

Arguments	SR-UUID
Description	Updates the specified SR and synchronize the database record with the SR status.
Result	If the SR exists, the command should return zero, else an appropriate error code.

## VDI-Level API Operations

VDI-level operations allow the manipulation of individual virtual disks within an SR. To be called, the appropriate SR must have already been attached to the calling host.

### vdi\_create()

Arguments	SRUUID, VDIUUID, size
Description	Creates a new VDI of the specified size in the specified SR.

Result	Returns an XML string containing information about the newly created VDI.
--------	---

### **vdi\_delete()**

Arguments	SRUUUID, VDIUUID
Description	Deletes the specified VDI.
Result	none

### **vdi\_attach()**

Arguments	SRUUUID, VDIUUID
Description	Prepares and verifies the disk for connection to the host. N.B. If exclusive access is not implemented in the backend driver, then vdi_attach should activate and verify the device path accessible.

### **vdi\_detach()**

Arguments	SRUUUID, VDIUUID
Description	Disconnects the specified virtual disk from the calling host.

### **vdi\_resize()**

Arguments	SRUUUID, VDIUUID, size
Description	Increases the capacity of the specified VDI to the given <i>size</i> .
Result	An XML string containing the new VDI parameters.

### **vdi\_activate()**

Arguments	SRUUUID, VDIUUID
Description	Provides exclusive write access on the data path for the specified VDI.
Result	Zero returned on success, or an appropriate error code + string to stderr.

## **vdi\_deactivate()**

Arguments	SRUUID, VDIUUID
Description	Relinquishes exclusive write access on the data path for the specified VDI.
Result	Zero on success or an appropriate error code and string to stderr.

## **vdi\_snapshot()**

Arguments	SRUUID, VDIUUID
Description	Creates a read-only snapshot of the specified VDI.
Result	An XML string containing the new VDI parameters.

## **vdi\_clone()**

Arguments	SRUUID, VDIUUID
Description	Creates a writable snapshot of the specified VDI.

## **vdi\_update()**

Arguments	SRUUID, VDIUUID
Description	Updates the specified VDI and synchronize the database record with the VDI status.
Result	If the VDI exists, the command should return zero, else an appropriate error code.

## **vdi\_introduce()**

Arguments	SRUUID, VDIUUID
Description	Introduces a VDI to an SR. During this operation the backend driver should insert an appropriate database record. For managing lists of VDIs in an SR such as a raw LUN-per-VDI SR type, implementers may choose to manage records using the 'managed' flag. This allows the backend driver to toggle the availability status of a VDI e.g. during vdi_create.

Result	An XML string containing the VDI parameters.
--------	--

## Non-Primitive Types

This section describes the non-primitive types used in the XMLRPC messages.

### vd\_info

`vd_info` structures are returned from calls which usually create a new disk (e.g. snapshot, clone). The structure has two possible key/value pairs:

- location: string (mandatory)
- uuid: string (optional)

The location contains some opaque identifier for the new disk object which is interpreted only by the SM backend. Locations must be unique across VDIs in the same SR. If a UUID string is also returned then this will be used to create the XenAPI VDI record, otherwise a fresh UUID will be generated by XAPI.

### device\_config

`device_config` is an XMLRPC structure whose keys and values are taken directly from the XenAPI `PBD.device_config` (for most calls) or directly from the XenAPI call arguments in the case of `SR.probe`. A `device_config` structure is present on every `SMAPI` call, except the call to query the capabilities of the plugin itself. It contains customized data that is useful to the SR driver type and enables the driver to access and configure VDI objects on the storage medium.

When the same SR is mapped to multiple hosts, the `device_config` may be different in each case. This reflects the fact that the same underlying storage may be exposed differently (e.g. the device nodes may differ) on separate hosts. The backend driver is responsible for verifying the `device_config` and establishing whether the required parameters are provided. The required parameters are:

- SR-UUID: UUID that refers to an SR
- VDI-UUID: UUID that refers to a VDI

### driver\_params

`driver_params` is an XMLRPC structure whose keys and values are taken directly from the XenAPI `VDI.snapshot` and `VDI.clone` calls. It is intended to allow parameters such as epochIDs to be passed directly to backends to permit optimization.

### vd\_sm\_config

`vd_sm_config` is an XMLRPC structure used in the `vd_introduce()` call whose keys and values are taken directly from the XenAPI `VDI.introduce` call.

## driver\_info

See the section called “Current Driver Parameters”.

## nil

`nil` is returned from calls where only return codes are of significance.

## Message Syntax

All requests have the same form: an XMLRPC methodCall with a method name (a string such as `sr_create` or `vdi_attach`) and an XMLRPC structure containing key/value pairs. SMAPI calls differ in the keys present in the XMLRPC structure. The full set of possible keys are:

Key name	Type	Description
<code>host_ref</code>	string	XenAPI reference to the Host representing localhost
<code>args</code>	array	additional arguments (detailed below)
<code>device_config</code>	device_config	PBD.device_config
<code>session_ref</code>	string	pre-authenticated XenAPI session reference
<code>sr_ref</code>	string	XenAPI SR reference
<code>sr_uuid</code>	string	UUID of the SR stored in the XAPI database
<code>vdi_ref</code>	string	XenAPI VDI reference
<code>vdi_location</code>	string	opaque identifier for a disk
<code>vdi_uuid</code>	string	UUID of the VDI stored in the XAPI database
<code>new_uuid</code>	string	suggested UUID of the new object
<code>driver_params</code>	driver_params	allow VDI.clone, VDI.snapshot to be optimized
<code>vdi_sm_config</code>	vdi_sm_config	initial values for VDI.sm_config for the <b>vdi_introduce()</b> call

## Call Syntax

Call	Parameters	Return type
sr_create	sr_ref device_config args=[size]	nil
sr_delete	sr_ref device_config args=[]	nil
sr_attach	sr_ref device_config args=[]	nil
sr_detach	sr_ref device_config args=[]	nil
sr_probe	device_config args=[]	string
sr_scan	sr_ref dconf []	nil
sr_content_type	sr_ref dconf []	string
vdi_create	sr_ref vdi_sm_config device_config args=[size]	vdi_info
vdi_update	sr_ref vdi_ref device_config args=[]	nil
vdi_introduce	sr_ref vdi_location vdi_sm_confignew_uuid device_config args=[]	vdi_info
vdi_delete	sr_ref vdi_ref device_config args=[]	nil
vdi_attach	sr_ref vdi_ref device_config args=[]	string
vdi_detach	sr_ref vdi_ref device_config args=[]	nil
vdi_activate	sr_ref vdi_ref device_config args=[]	nil
vdi_deactivate	sr_ref vdi_ref device_config args=[]	nil
vdi_snapshot	sr_ref vdi_ref driver_params device_config args=[]	vdi_info
vdi_clone		vdi_info

Call	Parameters	Return type
	sr_ref vdi_ref driver_params device_config args=[]	
vdi_resize	sr_ref vdi_ref device_config args=[newszie]	vdi_info

## Error Codes

SUCCESS is defined as a zero return code. FAILURE can comprise any one of the following subset of generic system error codes:

### pre-existing errors and values that must not change

Code Name	Description	Error Code Value
SRInUse	The SR device is currently in use	16
VDIInUse	The VDI is currently in use	24
LockErr	The lock/unlock request failed	37
Unimplemented	The requested method is not supported / implemented	38
SRNotEmpty	The SR is not empty	39
ConfigLUNMissing	The request is missing the LUNid parameter	87

### Codes used by agent-internal backends (such as ISO driver); allocated from 1000

Code Name	Description	Error Code Value
ISODconfMissingLocation	'Location' parameter must be specified in Device Configuration	220
ISOMustHaveISOExtension	ISO name must have .iso extension	221
ISOMountFailure	Could not mount the directory specified in Device Configuration	222

Code Name	Description	Error Code Value
ISOUnmountFailure	Could not unmount the directory specified in Device Configuration	223

#### generic invalid arguments

Code Name	Description	Error Code Value
InvalidArg	Invalid argument	1
BadCharacter	A bad character was detected in the device-config structure	2
InvalidDev	No such device	19

#### errors, starting at 40 (arbitrary)

Code Name	Description	Error Code Value
SRScan	The SR scan failed	40
SRLog	The SR log operation failed	41
SRExists	The SR already exists	42
VDIExists	The VDI already exists	43
SRNoSpace	There is insufficient space	44
VDIUnavailable	The VDI is not available	46
SRUnavailable	The SR is not available	47
SRUnknownType	Unknown repository type	48
SRBadXML	Malformed XML string	49
LVMCreate	Logical Volume creation error	50
LVMDelete	Logical Volume deletion error	51
LVMMount	Logical Volume mount/activate error	52
LVMUnMount	Logical Volume unmount/deactivate error	53

<b>Code Name</b>	<b>Description</b>	<b>Error Code Value</b>
LVMWrite	Logical Volume write error	54
LVMPartCreate	Logical Volume partition creation error	55
LVMPartInUse	Logical Volume partition in use	56
LVMFilesystem	Logical Volume filesystem creation error	57
LVMMaster	Logical Volume request must come from master	58
LVMResize	Logical Volume resize failed	59
LVMSize	Logical Volume invalid size	60
FileSRCreate	File SR creation error	61
FileSRRmdir	File SR failed to remove directory	62
FileSRDelete	File SR deletion error	63
VDIRemove	Failed to remove VDI	64
VDILoad	Failed to load VDI	65
VDIType	Invalid VDI type	66
ISCSIDevice	ISCSI device failed to appear	67
ISCSILogin	ISCSI login failed	68
ISCSILogout	ISCSI logout failed	69
ISCSIInitiator	Failed to set ISCSI initiator	70
ISCSIDaemon	Failed to start ISCSI daemon	71
NFSVersion	Required NFS server version unsupported	72
NFSMount	NFS mount error	73
NFSUnMount	NFS unmount error	74

<b>Code Name</b>	<b>Description</b>	<b>Error Code Value</b>
NFSAttached	NFS mount point already attached	75
NFSDelete	Failed to remove NFS mount point	76
LVMGroupCreate	Logical Volume group creation failed	77
VDICreate	VDI Creation failed	78
VDISize	VDI Invalid size	79
VDIDelete	Failed to mark VDI hidden	80
VDIClone	Failed to clone VDI	81
VDISnapshot	Failed to snapshot VDI	82
ISCSIDiscovery	ISCSI discovery failed	83
ISCSIIQN	ISCSI target and received IQNs differ	84
ISCSIDetach	ISCSI detach failed	85
ISCSIQueryDaemon	Failed to query the iscsi daemon	86
NFSCreate	NFS SR creation error	88
ConfigLUNIDMissing	The request is missing the LUNid parameter	89
ConfigDeviceMissing	The request is missing the device parameter	90
ConfigDeviceInvalid	The device is not a valid path	91
VolNotFound	The volume cannot be found	92
PVSfailed	pvs failed	93
ConfigLocationMissing	The request is missing the location parameter	94
ConfigTargetMissing	The request is missing the target parameter	95

Code Name	Description	Error Code Value
ConfigTargetIQNMissing	The request is missing or has an incorrect target IQN parameter	96
ConfigISCSIIQNMissing	Unable to retrieve the host configuration ISCSI IQN parameter	97
ConfigLUNSerialMissing	The request is missing the LUN serial number	98
LVMOneLUN	Only 1 LUN may be used with shared LVM	99
LVMNoVolume	Cannot find volume	100
ConfigServerPathMissing	The request is missing the serverpath parameter	101
ConfigServerMissing	The request is missing the server parameter	102
ConfigServerPathBad	The serverpath argument is not valid	103
LVMRefCount	Unable to open the ref-count file	104
Rootdev	Root system device, cannot be used for VM storage	105
InvalidIQN	The IQN provided is an invalid format	106

#### Agent database query errors 150+

Code Name	Description	Error Code Value
APISession	Failed to initialize XMLRPC connection	150
APILocalhost	Failed to query Local Control Domain	151
APIPBDQuery	A Failure occurred querying the PBD entries	152

## NetApp-specific error codes

Code Name	Description	Error Code Value
NAPPTarget	Netapp Target parameter missing in device-config structure	120
NAPPUsername	Netapp Username parameter missing in device-config structure	121
NAPPPassword	Netapp Password parameter missing in device-config structure	122
NAPPAggregate	Netapp Aggregate parameter missing in device-config structure	123
NAPPTargetFailed	Failed to connect to Netapp target	124
NAPPAuthFailed	Authentication credentials incorrect	125
NAPPInsufficientPriv	Auth credentials have insufficient access privileges	126
NAPPFVolNum	Max number of flexvols reached on target. Unable to allocate requested resource.	127
NAPPSnapLimit	Max number of Snapshots reached on target Volume. Unable to create snapshot.	128
NAPPSnapNoMem	Insufficient space, unable to create snapshot.	129
NAPPUnsupportedVersion	Netapp Target version unsupported	130
DNSError	Incorrect DNS name, unable to resolve.	140
EIO	General IO error	200

## Current Driver Parameters

Each driver type takes a custom set of parameters that must be provided in the `device_config` structure. The default set of drivers included in the current XE product are:

### Ext

The `EXTSR.py` driver uses a whole disk or partition to contain a file-based SR.

### File

The `FileSR.py` driver uses a local, user-specified directory to contain a file-based SR.

### Lvm

The `LVMSR.py` driver creates a Volume Group with the SR UUID and generates a Logical Volume for each VDI.

### Iscsi

The `ISCSISR.py` driver identifies an SR as all the LUNs that are accessible on an iSCSI target. Each LUN is uniquely identified with a UUID and, by default, is not made active in the host database. The host admin must explicitly enable LUNs that are to be used.

### Lvmoiscsi

The `LVMoISCSISR.py` (LVM over iSCSI) driver takes one or more LUNs as a parameter and creates a Volume Group across all the devices. VDIs are then managed the same way as the default LVM driver.

### Exttoiscsi

The `EXToISCSISR.py` (File-based EXT over iSCSI) driver takes a single LUN as a parameter, formats the disk and attaches it as a single file-based SR. VDI operations are then handled the same way as the EXT driver.

The driver-specific parameters that should be included in the `device_config` structure for each type are outlined below.

### Ext

Parameter	Description
device	The device node on which the filesystem SR should be inserted

### File

Parameter	Description
location	The filesystem path on which the SR should be inserted

### Lvm

Parameter	Description
device	

Parameter	Description
	The device node(s) on which the Volume Group SR should be inserted. Multiple devices are comma separated.

### Iscsi

Parameter	Description
target	The iSCSI target node
targetIQN	The IQN record identifier of the target
localIQN	The IQN that should be advertised by this host to the target
chapuser	[Optional] CHAP authentication username
chappassword	[Optional] CHAP authentication password
port	[Optional] Target connection port
usediscoverynumber	[Optional] For multi-homed targets that advertise connections with the same IQN more than once for different interfaces, you can specify a particular record to use.

### Lvmoiscsi

Parameter	Description
target	The iSCSI target node
targetIQN	The IQN record identifier of the target
localIQN	The IQN that should be advertised by this host to the target
LUNid	A comma separated list of LUNs on which the Volume Group should be created
chapuser	[Optional] CHAP authentication username
chappassword	[Optional] CHAP authentication password
port	[Optional] Target connection port
usediscoverynumber	[Optional] For multi-homed targets that advertise connections with the same IQN

Parameter	Description
	more than once for different interfaces, you can specify a particular record to use.

### Extoiscsi

Parameter	Description
target	The iSCSI target node
targetIQN	The IQN record identifier of the target
localIQN	The IQN that should be advertised by this host to the target
LUNid	A comma separated list of LUNs on which the Volume Group should be created
chapuser	[Optional] CHAP authentication username
chappassword	[Optional] CHAP authentication password
port	[Optional] Target connection port
usediscoverynumber	[Optional] For multi-homed targets that advertise connections with the same IQN more than once for different interfaces, you can specify a particular record to use.

# Appendix A. Mapping of `xe` calls to SMAPI calls

## Details

### Introduction

The Xen Cloud Platform comes bundled with a CLI, called `xe` to perform Virtual Machine life cycle management operations from the command line. Some of these `xe` commands will result in one or more calls to the Storage Manager (SM) framework. This document describes how each `xe` call will interact with the SM-API. If an `xe` call is not documented, one can assume no SM-API level call will be made.

### Storage Repository Calls

#### `xe sr-create`

A `xe sr-create` call results in a `SR.create` call, followed by a `SR.attach` call.

#### `xe sr-forget`

A `xe sr-forget` call does not result in any SR level call.

#### `xe sr-scan`

A `xe sr-scan` call results in a `SR.scan` call.

#### `xe sr-probe`

A `xe sr-probe` call results in a `SR.probe` call.

#### `xe sr-introduce`

A `xe sr-introduce` call does not result in any SR level call.

#### `xe sr-destroy`

A `xe sr-destroy` call results in a `SR.attach` call, followed by a `SR.delete` call.

### PBD Calls

#### `xe pbd-plug`

A `xe pbd-plug` call results in a `SR.attach` call. Please note that a subsequent second `pbd-plug` call will not result in any SMAPI calls.

## **xe pbd-unplug**

A **xe pbd-unplug** call results in a SR.detach call.

## **xe pbd-destroy**

A **xe pbd-destroy** call does not result in any SMAPI calls.

# **Virtual Machine Calls**

## **xe vm-install**

A **xe vm-install** call results in one VDI.create call per VDI in the template used to install the VM with, followed by a sequence of VDI.attach, and VDI.activate for each of the VDIs that are part of the VM. A sr-scan call might be scheduled after the vm-install call.

## **xe vm-start**

A **xe vm-start** call results in one VDI.attach and VDI.activate call per VDI attached to the VM. In case of a PV guest, xapi will execute a VDI.attach, VDI.activate, VDI.deactivate and VDI.detach series to get the kernel from the boot disk, before it attaches and activates each VDI.

## **xe vm-shutdown**

A **xe vm-shutdown** call results in a VDI.deactivate and VDI.detach per VDI attached to the Virtual Machine.

## **xe vm-suspend**

A **xe vm-suspend** call results in a VDI.create call, followed by a VDI.attach and VDI.activate call, and then a VDI.deactivate and VDI.detach call. All these calls are for the VDI that is used to store the memory image into. This is followed by a sequence of VDI.deactivate and VDI.detach for each of the data-VDIs.

## **xe vm-resume**

A **xe vm-resume** call results in a VDI.attach and VDI.activate call for the suspend VDI, followed by a VDI.attach and VDI.activate for each of the data-VDIs of the Virtual Machine. After that, there is a VDI.deactivate, VDI.detach and a VDI.delete call for the suspend VDI.

## **xe vm-migrate**

A **xe vm-migrate** call results in a VDI.attach call per data VDI of the Virtual Machine on the target machine of the migrate operation. At the moment the live migration is really happening, the VM executing will be paused, and a VDI.deactivate per data VDI will be called on the source machine, followed by a VDI.activate on the target machine. At this point the VM will resume on the target. Then in a lazy fashion, one VDI.detach call will be made on the source machine.

## **xe vm-clone**

If the SM plugin supports the VDI\_CLONE capability, this **xe vm-clone** call results in one VDI.clone call for each of the data VDIs of the Virtual Machine. The implementation of the plugin call is expected to ensure that on completion of the clone operation, the data on the target VDI is the same as on the source VDI. If the plugin does not support the VDI\_CLONE capability, the **xe vm-clone** call results in a VDI.create call to provision a VDI for the copy of a data VDI, followed by VDI.attach and VDI.activate calls for both source and target VDI. After this, the xapi daemon will copy the blocks from the source VDI to the target VDI on a block level. After the copy is done, a VDI.deactivate and VDI.detach will be called for each of the data-VDIs.

## **xe vm-uninstall**

A **xe vm-uninstall** call results in a VDI.detach, VDI.deactivate and a VDI.delete call for each of the data VDI's of the Virtual Machine.

## **VDI Calls**

### **xe vdi-create**

A **xe vdi-create** call results in a VDI.create call.

### **xe vdi-destroy**

A **xe vdi-destroy** call results in a VDI.detach call, followed by a VDI.delete call.

### **xe vdi-copy**

A **xe vdi-copy** call results in a VDI.create call, followed by VDI.attach and VDI.activate calls for both the source as well as the target VDIs. After the copy completes, a VDI.deactivate and VDI.detach call will be made for both VDIs.

### **xe vdi-forget**

A **xe vdi-forget** call does not result in any SMAPI calls.

### **xe vdi-introduce**

A **xe vdi-introduce** results in a VDI.introduce call.

## **VBD calls**

### **xe vbd-create**

A **xe vbd-create** call does not result in any SMAPI calls.

### **xe vbd-plug**

A **xe vbd-plug** call results in a VDI.attach and VDI.activate call.

## xe vbd-unplug

A **xe vbd-unplug** call results in a VDI.deactivate and a VDI.detach call.

## xe vbd-destroy

A **xe vbd-destroy** call does not result in any SMAPI calls.

## The test script

```
#!/bin/bash

DEV='/dev/sdb'

LOGFILE='/var/log/CMDlog'

log() {
    echo "$1" >> $LOGFILE
    echo "$1"
}

set_globals()
{
    HOST_NAME=`hostname`
    POOL_UUID=`xe pool-list --minimal`
    HOST_UUID=`xe host-list hostname=$HOST_NAME --minimal`
}

run_ops()
{
    set_globals

    log "---START on $SR_TYPE---"
    # SR create
    log "sr-create"

    SR_UUID=`xe sr-create type=$SR_TYPE physical-size=0 content-type=user \
device-config-device=$DEV name-label=TestSR`
    PBD_UUID=`xe pbd-list sr-uuid=$SR_UUID --minimal`

    log "pool-param-set"
    xe pool-param-set uuid=$POOL_UUID default-SR=$SR_UUID

    # VM install

    log "vm-install"
    VM_UUID=`xe vm-install template="Debian Sarge 3.1" new-name-label=TestVM`

    # VM start and shutdown

    log "vm-start"
    xe vm-start uuid=$VM_UUID
```

```
log "vm-shutdown"
xe vm-shutdown uuid=$VM_UUID

# VM suspend / resume

log "vm-start"
xe vm-start uuid=$VM_UUID

log "sleep 30"
sleep 30

log "vm-suspend"
xe vm-suspend uuid=$VM_UUID

log "vm-resume"
xe vm-resume uuid=$VM_UUID

log "vm-shutdown"
xe vm-shutdown uuid=$VM_UUID

# vm migrate

log "vm-start"
xe vm-start uuid=$VM_UUID

sleep 30

log "vm-migrate"
xe vm-migrate live=true uuid=$VM_UUID host=$HOST_UUID

log "vm-shutdown"
xe vm-shutdown uuid=$VM_UUID

#VM start and force shutdown

log "vm-start"
xe vm-start uuid=$VM_UUID

log "vm-shutdown force"
xe vm-shutdown uuid=$VM_UUID --force
```

```

# VM clone
log "vm-clone"
CLONE_VM_UUID=`xe vm-clone uuid=$VM_UUID new-name-label=CloneTestVM`

# and uninstall clone
log "vm-uninstall clone"
xe vm-uninstall uuid=$CLONE_VM_UUID --force &> /dev/null

# VDI create and remove
log "vdi-create"
VDI_UUID=`xe vdi-create sr-uuid=$SR_UUID name-label=TestVDI type=user \
virtual-size=100MiB`

log "vdi-remove"
xe vdi-destroy uuid=$VDI_UUID

# VDI create and plug into VM
log "vm-start"
xe vm-start uuid=$VM_UUID

log "vdi-create"
VDI_UUID=`xe vdi-create sr-uuid=$SR_UUID name-label=TestVDI type=user \
virtual-size=100MiB`

log "vbd-create"
VBD_UUID=`xe vbd-create vdi-uuid=$VDI_UUID vm-uuid=$VM_UUID device=xvdc`

log "vbd-plug"
xe vbd-plug uuid=$VBD_UUID

sleep 10

log "vbd-unplug"
xe vbd-unplug uuid=$VBD_UUID

log "vdi-copy"
CLONE_VDI_UUID=`xe vdi-copy uuid=$VDI_UUID sr-uuid=$SR_UUID`

log "vm-shutdown force"
xe vm-shutdown uuid=$VM_UUID --force

log "vdi-destroy"
xe vdi-destroy uuid=$VDI_UUID

log "vdi-destroy clone"
xe vdi-destroy uuid=$CLONE_VDI_UUID

```

```

log "vm-uninstall"
xe vm-uninstall uuid=$VM_UUID --force &> /dev/null

log "vdi-create"
VDI_UUID=`xe vdi-create sr-uuid=$SR_UUID name-label=TestVDI type=user \
virtual-size=100MiB`

log "vdi-forget"
xe vdi-forget uuid=$VDI_UUID

log "hack - lvremove the volume, since we don't support vdi-introduce yet"
lvremove -f /dev/VG_XenStorage-$SR_UUID/LV-$VDI_UUID &> /dev/null
rm -f /var/run/sr-mount/$SR_UUID/$VDI_UUID.vhd

# SR destroy
log "pbd-unplug"
xe pbd-unplug uuid=$PBD_UUID

log "pbd-plug"
xe pbd-plug uuid=$PBD_UUID

log "pbd-plug 2x"
xe pbd-plug uuid=$PBD_UUID

log "pbd-unplug"
xe pbd-unplug uuid=$PBD_UUID

log "pbd-unplug 2x"
xe pbd-unplug uuid=$PBD_UUID

log "sr-destroy"
xe sr-destroy uuid=$SR_UUID

log "sr-create for forget"
SR_UUID=`xe sr-create type=$SR_TYPE physical-size=0 content-type=user \
device-config-device=$DEV name-label=TestSR`
PBD_UUID=`xe pbd-list sr-uuid=$SR_UUID --minimal`

log "pool-param-set"
xe pool-param-set uuid=$POOL_UUID default-SR=$SR_UUID

log "sr-unplug"
xe pbd-unplug uuid=$PBD_UUID

log "sr-forget"
xe sr-forget uuid=$SR_UUID

```

```

    log "sr-introduce"
    DUMMY_SR=`xe sr-introduce uuid=$SR_UUID name-label=TestSR type=$SR_TYPE \
content-type=user`

    log "pbd-create"
    PBD_UUID=`xe pbd-create host-uuid=$HOST_UUID sr-uuid=$SR_UUID \
device-config-device=$DEV`

    log "pbd-plug"
    xe pbd-plug uuid=$PBD_UUID

    log "sr-scan"
    xe sr-scan uuid=$SR_UUID

    log "pbd-unplug"
    xe pbd-unplug uuid=$PBD_UUID

    log "sr-destroy"
    xe sr-destroy uuid=$SR_UUID

    log "---FINISH on $SR_TYPE---"
    log ""
    log ""
}

SR_TYPE='lvm'
run_ops

SR_TYPE='ext'
run_ops

```