Oracle® VM

Web Services API Developer's Guide for Release 3.3

E50253-05
July 2015

Oracle Legal Notices

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Abstract

Document generated on: 2015-07-20 (revision: 4803)

Table of Contents

Preface

Table of Contents

The Oracle VM Web Services API Developer's Guide is your reference for the Oracle VM Application Programmer's Interface (API). This guide provides information about the Oracle VM Web Services API (WSAPI) for developers and system administrators who are interested in implementing scripts and programs to manage and automate Oracle VM.

This document is intended to be used alongside the API documentation included in the SDK that is packaged with the Oracle VM Manager installation ISO. This document does not provide comprehensive coverage of the API, but is intended to assist developers to create proof of concept client applications.

1 Audience

This document is intended for experienced developers who are building applications for managing and automating Oracle VM. This guide assumes that you have an in depth knowledge of Oracle VM (see the *Oracle VM Manager User's Guide* and the *Oracle VM Manager Command Line Interface User's Guide*), and that you have knowledge of programming against web services APIs exported via REST and SOAP interfaces.

2 Related Documents

For more information, see the following documents in the Oracle VM documentation set:

- *Oracle VM Release Notes*
- *Oracle VM Installation and Upgrade Guide*
- *Oracle VM Concepts Guide*
- *Oracle VM Manager Getting Started Guide*
- *Oracle VM Manager User's Guide*
- *Oracle VM Manager Command Line Interface User's Guide*
- *Oracle VM Administrator's Guide*
- *Oracle VM Windows Paravirtual Drivers Installation Guide*
- *Oracle VM Web Services API Developer's Guide*
- *Oracle VM Security Guide*
- *Oracle VM Licensing Information User Manual*

You can also get the latest information on Oracle VM by going to the Oracle VM Web site:

http://www.oracle.com/us/technologies/virtualization/oraclevm

3 Command Syntax

Oracle Linux command syntax appears in `monospace` font. The dollar character ($), number sign (#), or percent character (%) are Oracle Linux command prompts. Do not enter them as part of the command. The following command syntax conventions are used in this guide:

Convention	Description

backslash \	A backslash is the Oracle Linux command continuation character. It is used in command examples that are too long to fit on a single line. Enter the command as displayed (with a backslash) or enter it on a single line without a backslash:
	`dd if=/dev/rdsk/c0t1d0s6 of=/dev/rst0 bs=10b \`
	`count=10000`

braces { }	Braces indicate required items: `.DEFINE {macro1}`
brackets [ ]	Brackets indicate optional items: `cvtcrt termname [outfile]`
ellipses ...	Ellipses indicate an arbitrary number of similar items: `CHKVAL fieldname value1 value2 ... valueN`
italics	Italic type indicates a variable. Substitute a value for the variable: *`library_name`*
vertical line |	A vertical line indicates a choice within braces or brackets: `FILE filesize [K|M]`
forward slash /	A forward slash is used as an escape character in the Oracle VM Manager Command Line Interface to escape the special characters ", ', ?, \, /, <, >. Special characters need only be escaped when within single or double quotes: `create Tag name=MyTag description="HR/'s VMs"`

4 Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Chapter 1 Oracle VM Web Services Overview

Table of Contents

The Oracle VM Web Services Application Programming Interface (API) provides a Web Services interface for building applications with Oracle VM. Both SOAP and REST interfaces are supported. Both offer the same underlying functionality; the choice depends on the specific client implementation and which type of mapping is most suitable. A client program should use only one type of API interaction. Mixing SOAP and REST based interactions in the same application is **not** supported and may cause unexpected results.

The Web Services API allows you to write applications to customize, control, and automate Oracle VM. For example, you can

- Create, start, restart and stop *virtual machine*s, and *migrate* them between different *Oracle VM Server*s.
- Expand the capacity of the Oracle VM environment by adding more Oracle VM Servers and storage providers.
- Integrate Oracle VM with other third-party tools such as monitoring applications
- Facilitate repetitive tasks like creating new instances of frequently deployed virtual machines from templates, switching between policies or turning them on or off depending on the time of the day or week, performing maintenance on the hardware or virtualized environment, and so on.

This guide introduces you to the Oracle VM Web Services API, providing the basic concepts and API examples. For the complete API documentation, see the Oracle VM Web Services API documentation included in the SDK on the *Oracle VM Manager* installation ISO.

The Web Services API to Oracle VM can be accessed using any programming language. As the same functionality is available via SOAP and REST interfaces, the developer chooses either type of interaction, based on the specific development objectives and architecture. The general interaction pattern with the API is to first find an object, retrieve a client-side instance of the object, and then execute certain authorized manipulations.

An object must be retrieved by its ID. These object IDs can be obtained via other related objects, or in some cases by requesting a list of all object IDs of a given type. Once the ID is available, various operations can be performed on the object. The exact set of operations depends on the type of object and the interaction mode: SOAP or REST.

In both REST and SOAP, operations that are not simple retrievals are performed asynchronously. These methods return a Job object that can be used to subsequently query the status of the operation. In case the operation is an object creation, the Job object, once completed, will also include the ID of the created object so that other operations can be performed on this object.

1.1 Using the Oracle VM Web Services SDK

The Oracle VM Web Services SDK is included on the *Oracle VM Manager* installation ISO, compressed in zip format. To access the SDK, copy the zip file from the mounted ISO to a working directory and unzip it.

The SDK contains all of the resources that you need to get started developing to the Oracle VM Web Services API:

- Full documentation of the Web Services API as exposed by the provided library and sample source code.
- An Oracle VM web services client library.
- Sample Java source code that can be used to get started building your own client application using either the SOAP or REST API.

1.2 Using the Oracle VM Web Services Client Library and Sample Code

Since *Oracle VM Manager* provides generic SOAP and REST APIs, you are able to use this documentation to develop your own code to directly interact with your chosen API based on your understanding of the API mechanism that you choose to use and the methods that are supported by the exposed API. However, the Oracle VM Web Services SDK includes a Java client library and example Java source code that can dramatically reduce the amount of work that you need to do in order to code an application that uses either of the APIs. The example code also includes a number of utilities that are useful when working with an API, but which are not part of the API itself. For this reason, it is recommended that you take advantage of the provided code and library when programming a Java application that interfaces with Oracle VM Manager.

In this guide, we take a closer look at the Oracle VM Web Services Client Library and how it is commonly used. Full coverage of the methods supported by the library can be found in the documentation included with the SDK.

To get started using the SDK, you can import the source code provided into a new Java project within your preferred IDE. The SDK requires that you use the Oracle JDK 7 for your project. JDK 6 is not supported.

You must add the provided client library, `OvmWsClient.jar`, to your project in order to compile the source code. Additionally, the example source code makes use of some external libraries, such as the Jersey Bundle to handle calls to the REST API. While you are free to choose any other tools to create your own application, the example source code requires this library to compile as the example code is capable of being configured to use either the REST API or the SOAP API when it runs. Information on downloading and installing the Jersey Bundle is provided in Notes on the Java samples provided in these examples.

The sample code includes a properties file called `WsDevClient.properties`. This file contains the default values used for different variables defined in the code. In most cases, many of these variable values must be overridden for your own environment. For values that you wish to override, create a file in the same directory, named `WsDevClient_`*`username`*`.properties`, where *username* matches your the username that you use to log into the environment where you intend to compile the code. Note that this properties file is only required for the sample client and is not required when building your own application using the provided libraries.

Once the code has been compiled, you can run it easily from the command line using:

```
java -cp wsclient.jar com.oracle.ovm.mgr.ws.sample.WsDevClient
```

Tip

The sample client code has already been compiled into the wsclient.jar library file. This means that it can be used directly without requiring you to recompile it. You still need to edit the `WsDevClient.properties` file to override variables. To do this, edit `samples/com/oracle/ovm/mgr/ws/sample/WsDevClient.properties` so that the variables match your own environment. Unzip the `lib/OvmWsClient.jar` Java archive file and copy your new version

> of `WsDevClient.properties` into `com/oracle/ovm/mgr/ws/sample/` from the extracted archive. Finally recreate the archive file and run it using the instruction provided above.

If the value for `wsimpl` variable is set to `SOAP` and `debugHttpTraffic` is set to `true`, the output from the sample code includes all of the SOAP interactions that the sample client application performs, including the HTTP headers and the SOAP messages exchanged within the body of each request.

If the value for `wsimpl` variable is set to `REST` and `debugHttpTraffic` is set to `true`, the output from the sample code includes all of the REST interactions that the sample client application performs, including the XML or JSON body content and the HTTP headers.

Chapter 2 Using the Oracle VM Manager REST API

Table of Contents

2.1 Connecting to the REST Base URI

To access the REST web service, you must use the following base URI:

```
https://hostname:port/ovm/core/wsapi/rest
```

In this URI, replace *`hostname`* with the host name or IP address of *Oracle VM Manager*, and replace *`port`* with the number of the port where the REST web service is available - the default is 7002 (SSL).

Note

Querying this URI directly does not provide any information and results in an error. This URI is only to be used as the base URI that REST interactions are built on, according to the available URI mappings. See Section 2.3, “What URI Mappings are Available and How do They Work?” for more information.

2.2 How do I Authenticate?

To gain access to *Oracle VM Manager*, a client must first authenticate successfully. Oracle VM Manager supports the HTTP Basic Authorization mode for REST. The username and password must be sent as part of the header of the initial request in order for Oracle VM Manager to authenticate the request. In subsequent requests, the session cookies, attached to the initial response, should be sent instead of continuing to send the username and password with each new request.

Note

The login method in the `OvmWsRestClient` class provides an example of how to set the username and password in the request header.

> Examples of the login process as required by REST are also provided in the example code included in this document. See Section 2.6.1, “Authenticating” for further examples.

The REST API also has support for certificate-based authentication. As long as you have a valid certificate that is signed and registered either with the internal Oracle VM Manager CA certificate, or with a third-party CA for which your have imported the CA certificate into the Oracle VM Manager truststore, you can authenticate easily using your certificate instead of including a username and password facility within your code. Signing and registering certificates against the internal Oracle VM Manager CA certificate can either be achieved using the provided certificate management tool, discussed in Setting up SSL on Oracle VM Manager in the *Oracle VM Administrator's Guide*; or can be achieved programmatically using the Oracle VM Manager Utilities REST API discussed in Chapter 4, *Additional Utilities Exposed in the WS-API*.

2.3 What URI Mappings are Available and How do They Work?

Any REST API makes use of existing HTTP methods, such as GET and POST requests, to trigger various operations. The type of operation that is triggered depends on the HTTP method and the URI pattern that is used.

In Oracle VM, URI patterns are built around ObjectType names and the identifier for a particular instance of the ObjectType. In three cases, an identifier for an instance can be omitted:

1. Creation of a new instance of an ObjectType, using the HTTP POST method.
2. Full listing of all instances of an ObjectType, using the HTTP GET method.
3. Listing of the identifiers for all instances of an ObjectType, using the HTTP GET method.

The different ObjectTypes that can be accessed via the API are described in the full API documentation provided in the SDK. Note that ObjectType names may not map perfectly onto objects that you are used to seeing in the Oracle VM Manager Web Interface or Oracle VM Manager Command Line Interface. For instance, the 'resourceGroup' object is the same thing as a 'tag' in UI or CLI nomenclature. These differences are usually mentioned within the API documentation accompanying the SDK.

Since many objects within *Oracle VM Manager* have parent-child relationships, URI mappings are available to create, query and delete these relationships.

This table contains the basic URI patterns to interact with the available objects controlled by Oracle VM Manager:

Operation	HTTP Method	URI Path	Entity Passed In	Return Type
Create	POST	/ObjectType	Object being created	Job
Get all	GET	/ObjectType		List<ObjectType>
Get all IDs	GET	/ObjectType/id		List<SimpleId>
Get by ID	GET	/ObjectType/{id}		ObjectType
Modify	PUT	/ObjectType/{id}	Object being modified	Job
Delete	DELETE	/ObjectType/{id}		Job
Action [a]	PUT	/ObjectType/{id}/action	Object being created	Job

Operation	HTTP Method	URI Path	Entity Passed In	Return Type
Add child association	PUT	/ObjectType/{id}/addChildType	Child ID	Job
Remove child association	PUT	/ObjectType/{id}/removeChildType	Child ID	Job
Get child associations	GET	/ObjectType/{id}/ChildType		List<ChildType>
Get child association IDs	GET	/ObjectType/{id}/ChildType/id		List<SimpleId>
Create child object	POST	/ParentType/{id}/ChildType	New ChildType object	Job
Delete child object	DELETE	/ParentType/{parentId}/ChildType/{childId}		Job

[a] See Section 2.6.8, "Managing Servers in a Server Pool" for two good examples of action-style activities.

Note

- Capitalization as shown in the sample URIs is significant for the actual URI request: *`ObjectTypeNames`* are capitalized, while *`actions`* are not.
- *Actions* available for an ObjectType vary depending on the ObjectType. Each action available for each ObjectType is described in the documentation provided with the SDK.
- Only top level objects are removed using the *Delete* operation. For child objects, use the *Delete child object* operation.
- If an object can exist outside the association of its parent it is created as a normal object and then added to the parent using the *Add child association* operation. The relationship between ServerPool and Server objects is an example of this. If the parent object is deleted, the child still exists.
- If the child object cannot exist without its parent then it is created via a *Create child object* operation. The relationship between Vm and VmDiskMapping objects is an example of this. In this case, if the parent object is deleted, the child is also be deleted.

Tip

For GET type requests you can test out the REST API directly by using a standard web browser, as long as you have a means to modify the HTTP Request Headers to include the HTTP Basic authentication credentials required to access the API. If you are using Mozilla Firefox, you could try the Modify Headers Add-On; while Google Chrome users could avail of the ModHeaders Extension.

Modify your browser headers to include the Authorization header. Set the value to BASIC and append the Base64 encoded string for your username and password in the format: *`username:password`*. Once this is done, you can simply point your browser at any URL that accepts a GET request. For example, to obtain a listing of server IDs, try pointing your browser at `https://`*`hostname`*`:7002/ovm/core/wsapi/rest/Server/id` where *`hostname`* is the IP address or FQDN of the Oracle VM Manager host.

2.4 Internet Media Types Used by the REST API (JSON and XML)

The REST API can use either XML or JSON to encode data sent in requests or received in responses. The Internet media type of the data returned is controlled by setting the appropriate HTTP Headers for each request. Since some languages, such as Python, have better libraries for parsing data in JSON format, as opposed to XML, you may decide to set the media type to JSON for all requests.

To notify Oracle VM Manager to return data in JSON format, you must set the **Accept** Header to **application/json**.

To send data to Oracle VM Manager in JSON format you must set the **Content-Type** Header to **application/json**.

To notify Oracle VM Manager to return data in XML format, set the **Accept** Header to **application/xml**, or rely on the default media type and do not set this parameter at all.

To send data to Oracle VM Manager in XML format, set the **Content-Type** Header to **application/xml**, or rely on the default media type and do not set this parameter at all.

Warning

For all POST and PUT operations where data is included in the body of the HTTP request, element names are case-sensitive.

2.5 Good Practice: Check the Oracle VM Manager Start Up State

The Oracle VM Manager start up process can take a number of minutes to complete. During start up, Oracle VM Manager is capable of accepting web services API calls for various debugging purposes, however jobs requested through the web services API are not initiated during start up and the model contents can change significantly over the course of the start-up process as various objects are refreshed and rediscovered. For this reason, it is good practice for an application, using the web services API, to check the running status of Oracle VM Manager before submitting API requests.

The running status of Oracle VM Manager is contained in the managerRunState property or attribute of the Manager object returned by the Oracle VM Web Services API. During start up, the value of this property is set to 'STARTING'. Once all start up operations and server rediscovery is complete, the value of this property changes to 'RUNNING'.

In your code, you should authenticate against Oracle VM Manager and then check that the managerRunState property of the Manager object is set to 'RUNNING' before performing any further operations. To do this using the REST interface, you can perform a GET query against the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/Manager
```

An example of this and a description of how to wait until the Oracle VM Manager is actually running is provided in

2.6 Example Code Using REST

The example code, provided in this section, steps through many common operations within *Oracle VM Manager*. For each example, we have provided code samples for both Java and Python. By nature, the REST API is language agnostic, so you may decide to use an alternative programming language to interface with the API. The code samples are provided to show how different operations might be performed using one of these two popular languages.

In a guide like this, the programming style and the choice of libraries used very much depend on the author and the version of the language used. More than likely, there are many more ways to achieve the

same result, even within the same language. These samples are not intended to be authoritative in their approach, but can be used as guidelines to developing your own applications.

Notes on the Java samples provided in these examples

Our Java samples are built around the Java code and library provided in the SDK that is bundled with Oracle VM Manager. The sample code uses the Jersey implementation of the JAX-RS specification for Java's REST support. Jersey provides a core client library that facilitates REST communication with a REST server, and also supports the automatic creation and parsing of XML via JAXB. Jersey is not included in the SDK and should be downloaded separately from:

http://jersey.java.net/

To avoid the dependency management of multiple jersey-based jars, it is recommended that you download the Jersey Bundle Jar and import this into your project.

In addition to the Jersey libraries, the Java Sample Client takes advantage of the Jackson JSON Processor library to facilitate JSON data binding, allowing for the quick translation between JSON and POJO. Jackson is not included in the SDK and should be downloaded separately from:

http://wiki.fasterxml.com/JacksonHome

The Java Sample Client provided in the SDK includes all of the code required to perform the majority of supported interactions with the WS-API. The code is separated into two packages:

- `com.oracle.ovm.mgr.ws.client`: Contains the classes for both the REST and SOAP interfaces to the API. Notably, for these examples, the RestClient.java and OvmWsRestClient.java files contain much of the code referenced through this guide. In this guide, we attempt to describe how the actual client API code has been constructed to allow you to abstract many of the REST calls that you would need to make otherwise. In practice, you can use these classes without needing to know all of the underlying mechanics to the code, this is illustrated in the WSDevClient class discussed below.
- `com.oracle.ovm.mgr.ws.sample`: Contains the WsDevClient class, which performs particular actions using the API and which is set as your main class when you run the sample client. The WsDevClient class is an example of how you can use the client API classes to create your own applications drawing on all the abstraction provided by these classes. If you're just following this guide to work out how to use the API to write your own applications, you can concentrate on the code in the WsDevClient class.

The SDK also includes the Oracle VM Manager Web Services Client library in the form of a precompiled jar file called `OvmWsClient.jar`. This library contains models for all of the different ObjectTypes exposed through the API, as well as a variety of utilities that are useful to perform various actions on objects in the API. This library must be included in your project to allow you to work with typical Oracle VM ObjectTypes.

Your code should import the models, that you intend to use, as they are described in the Web Services Client library. Typically, a full-scale Java IDE should handle this on your behalf when you import the library and as you make use of different models. The listing presented is provided for completeness. The following listing provides a full outline of model imports:

```
import com.oracle.ovm.mgr.ws.model.AccessGroup;
import com.oracle.ovm.mgr.ws.model.AffinityGroup;
import com.oracle.ovm.mgr.ws.model.Assembly;
import com.oracle.ovm.mgr.ws.model.AssemblyVirtualDisk;
import com.oracle.ovm.mgr.ws.model.AssemblyVm;
import com.oracle.ovm.mgr.ws.model.BaseObject;
import com.oracle.ovm.mgr.ws.model.CloneType;
import com.oracle.ovm.mgr.ws.model.Cluster;
import com.oracle.ovm.mgr.ws.model.ClusterHeartbeatDevice;
```

```
import com.oracle.ovm.mgr.ws.model.ClusterStorageFs;
import com.oracle.ovm.mgr.ws.model.ControlDomain;
import com.oracle.ovm.mgr.ws.model.Cpu;
import com.oracle.ovm.mgr.ws.model.CpuCompatibilityGroup;
import com.oracle.ovm.mgr.ws.model.EthernetPort;
import com.oracle.ovm.mgr.ws.model.Event;
import com.oracle.ovm.mgr.ws.model.FileServer;
import com.oracle.ovm.mgr.ws.model.FileServerPlugin;
import com.oracle.ovm.mgr.ws.model.FileSystem;
import com.oracle.ovm.mgr.ws.model.FileSystemMount;
import com.oracle.ovm.mgr.ws.model.FsAccessGroup;
import com.oracle.ovm.mgr.ws.model.Id;
import com.oracle.ovm.mgr.ws.model.Job;
import com.oracle.ovm.mgr.ws.model.LoggerManagementAttributes;
import com.oracle.ovm.mgr.ws.model.LoginCertificate;
import com.oracle.ovm.mgr.ws.model.Manager;
import com.oracle.ovm.mgr.ws.model.Network;
import com.oracle.ovm.mgr.ws.model.PeriodicTask;
import com.oracle.ovm.mgr.ws.model.Repository;
import com.oracle.ovm.mgr.ws.model.RepositoryExport;
import com.oracle.ovm.mgr.ws.model.ResourceGroup;
import com.oracle.ovm.mgr.ws.model.Server;
import com.oracle.ovm.mgr.ws.model.ServerController;
import com.oracle.ovm.mgr.ws.model.ServerPool;
import com.oracle.ovm.mgr.ws.model.ServerPoolNetworkPolicy;
import com.oracle.ovm.mgr.ws.model.ServerPoolPolicy;
import com.oracle.ovm.mgr.ws.model.ServerUpdateConfiguration;
import com.oracle.ovm.mgr.ws.model.ServerUpdateRepositoryConfiguration;
import com.oracle.ovm.mgr.ws.model.SimpleId;
import com.oracle.ovm.mgr.ws.model.Statistic;
import com.oracle.ovm.mgr.ws.model.StorageArray;
import com.oracle.ovm.mgr.ws.model.StorageArrayPlugin;
import com.oracle.ovm.mgr.ws.model.StorageElement;
import com.oracle.ovm.mgr.ws.model.StorageInitiator;
import com.oracle.ovm.mgr.ws.model.StoragePath;
import com.oracle.ovm.mgr.ws.model.StorageTarget;
import com.oracle.ovm.mgr.ws.model.VirtualDisk;
import com.oracle.ovm.mgr.ws.model.VirtualNic;
import com.oracle.ovm.mgr.ws.model.VirtualSwitch;
import com.oracle.ovm.mgr.ws.model.VlanInterface;
import com.oracle.ovm.mgr.ws.model.Vm;
import com.oracle.ovm.mgr.ws.model.VmCloneDefinition;
import com.oracle.ovm.mgr.ws.model.VmCloneNetworkMapping;
import com.oracle.ovm.mgr.ws.model.VmCloneStorageMapping;
import com.oracle.ovm.mgr.ws.model.VolumeGroup;
import com.oracle.ovm.mgr.ws.model.WsErrorDetails;
import com.oracle.ovm.mgr.ws.model.WsException;
import com.oracle.ovm.mgr.ws.model.Zone;
```

You can simplify these imports by simply doing:

```
import com.oracle.ovm.mgr.ws.model.*;
```

You can use the `com.oracle.ovm.mgr.ws.client` package within your own projects to reduce your development overhead.

In these examples, we show how the REST client has been implemented within the `com.oracle.ovm.mgr.ws.client` package, and also how it is used for particular actions in the WsDevClient class.

Note

The code included in the library expects that you are using JDK 7. Ensure that your project Source/Binary format is set to JDK7 and that you have the JDK7 libraries imported into your project. JDK 6 is not supported.

Notes on the Python samples provided in these examples

Our Python samples are intended to give the reader a feel for direct access to the API for the purpose of scripting quick interactions with Oracle VM Manager. No abstraction is provided through the use of an additional library.

To keep the code simple, we have opted to make use of a couple of Python libraries that can handle HTTP session management, authentication and JSON. We selected these libraries based on the ease with which they can be used and for the brevity of the code that we are able to use. Depending on your Python version and the support provided for these libraries, you may choose to use alternatives to handle interactions with the REST API. Libraries used in these samples include:

- Requests: An HTTP library that offers good support for HTTP Basic Authentication and session handling. This library is available at http://docs.python-requests.org/en/latest/.
- time: The Python core library for time-related functions, which can be used to pause execution between queries to prevent the rapid succession of job status checking. This core library is described at http://docs.python.org/2/library/time.html.
- json: The Python core library for translating Python data objects to JSON and vice versa. This core library is described at http://docs.python.org/2/library/json.html.

In an effort to keep our examples as simple as possible, we have opted to use JSON as our default media type throughout this guide. It is equally possible to make use of the default XML media type using Python. If you choose to do this, you might consider using lxml, a feature-rich XML toolkit that can be used to parse XML into native Python objects, and which can be used to construct well-formatted XML. This toolkit is available at http://lxml.de/index.html.

Your Python code should start with the following imports for the examples in this guide to work:

```
import requests
import json
from time import sleep
```

Due to our selection of libraries, and the syntax of some of our example code, we assume that you are using Python 2.6 or above.

The Python examples in this guide build on top of each other. Frequently an example may refer to a function defined in a previous example. The reader should be aware that code provided for a particular example may not work without having defined some of the functions specified in the other referenced examples.

2.6.1 Authenticating

Authenticating against the REST API requires that you initially use HTTP Basic Authentication as described in RFC 2617. In subsequent requests the session cookies, returned in the initial response, should be sent instead of continuing to send the username and password with each new request.

Alternatively, you can use SSL certificates to perform authentication. You must use a certificate that has been signed and registered by a CA that is recognized by Oracle VM Manager, as described in Section 4.1.3, “Certificate Management for Certificate-based Authentication Using SOAP”.

2.6.1.1 Java

Authentication against the REST API, in the Java client, makes use of a few Jersey utilities. In the RestClient provided in `com.oracle.ovm.mgr.ws.client`, a login method has been defined:

```
...
public class RestClient
```

```
{
   public static final String                  URI_BASE_PATH     = "/ovm/core/wsapi/rest";
   String                                      SECURE_PROTOCOL   = "https";
   String                                      INSECURE_PROTOCOL = "http";

   private static final Map<Class, GenericType> genericTypeMap = new HashMap<Class, GenericType>();

   private boolean                             debug             = false;
   private URI                                 baseURI;
   private Client                              webClient         = null;
   private List<NewCookie>                     cookies;
   private String                              mediaType         = MediaType.APPLICATION_XML;
   private Locale                              locale            = null;
   private SSLSocketFactory                    sslSocketFactory  = null;
...

   public boolean login(final String username, final String password, final Locale locale,
               final String path) throws WsException
   {
       try
       {
            // Make a dummy request to pass the authorization info to the server get
            // the session id (which is returned in a cookie)
            if (getCookies() == null)
            {
                final WebResource resource =
                getWebClient().resource(getBuilder().path(path).build());
                // Specify media type to accept
                final Builder resourceBuilder = resource.accept(getMediaType());
                // Append auth info
                resourceBuilder.header("Authorization",
                "Basic " + new String(Base64.encode(username + ":" + password)));

                if (locale != null)
                {
                    resourceBuilder.acceptLanguage(locale);
                }

                final ClientResponse response = resourceBuilder.head();
                setCookies(response.getCookies());
            }

            return true;
        }
        catch (final UniformInterfaceException ex)
        {
            throw convertException(ex);
        }
    }
...
}
```

Note that an initial request is sent and an Authorization header is attached with the Base64 encoded username and password required for HTTP Basic authentication. The session information is returned as a cookie in the server response, and this cookie can be used for all subsequent requests.

The OvmWsRestClient class in `com.oracle.ovm.mgr.ws.client` extends the RestClient class. It provides the API function that gets called in the sample application and defines the dummy query that is performed against the API to achieve authentication:

```
public class OvmWsRestClient extends RestClient implements OvmWsClient
{
    public static final String URI_BASE_PATH = "/ovm/core/wsapi/rest";

    public OvmWsRestClient()
    {
```

```
        super();
    }
    @Override
    public boolean login(final String username, final String password,
         final Locale locale) throws WsException
    {
        return super.login(username, password, locale, "Server");
    }
...
}
```

Now in the WsDevClient class (the sample application), all we need to do is call this method:

```
...

public class WsDevClient
{
...
public void run()
    {
        try
        {
         ...
         api = OvmWsClientFactory.getOvmWsClient(wsimpl);
         ...
            api.initialize(hostname, port, true);
         // Authenticate with the OvmApi Service
            api.login(username, password, Locale.getDefault());
         ...
         }
...
}
```

There are a few things to note about how this has been implemented in the sample client. The first point is that we refer to the OvmWsClientFactory class to determine whether we are using the REST client or the SOAP client, using the string variable 'wsimpl'. This class allows us to use the same demo code to show both APIs. It contains a switch statement that sets up the appropriate client object:

```
switch (implementation)
        {
            case SOAP:
                return new OvmWsSoapClient();

            case REST:
                return new OvmWsRestClient();
        }
```

Before the login method is called, an initialize method is used to set the hostname and port number on which the Oracle VM Manager is running. This method is ultimately defined in the RestClient class and simply takes all of the components that make up the base URI and constructs the base URI that is used thereafter.

A final thing to note, is that the call to the initialize and login methods provide some preset variables. In fact, the `com.oracle.ovm.mgr.ws.sample` package also includes a properties file: `WsDevClient.properties`. This file contains the default values for many of the variables referenced throughout this guide. For the sample code to work according to the specifics of your own environment, many of these variables must be overridden. Instructions for handling variable overriding are provided in the comments of this properties file itself.

What about Certificate-based Authentication?

Since the code in the SDK does not assume that you have set up user certificates, there are no examples showing how to authenticate using a signed SSL certificate. However, all that is required for this to happen

is for you to use the certificate for all REST requests in your session. In Java, the easiest way to do this is to ensure that you have the certificate and its associated key stored in a keystore file. You can then use the keystore to load your key manager and trust manager that can be used to initialize an SSL context that is used for all connections.

The following example code, should help to get you started. It has been stripped of any error handling that may obfuscate what is required. This code expects the following variables to be initialized:

```
File keystoreFile;        // A file representing the location of the KeyStore
char[] keystorePassword;  // The KeyStore password
char[] keyPassword;       // The key password - if you didn't specify one when creating your
                          // key, then use the keystorePassword for this as well
```

The code required to use this keystore to initialize an SSL context follows:

```
// Load your keystore
FileInputStream stream = new FileInputStream(keystoreFile);
KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
keystore.load(stream, keystorePassword);
stream.close();

// Initialize the key manager from the keystore
KeyManagerFactory kmFactory =
  KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
kmFactory.init(keystore, keyPassword);
KeyManager km[] = kmFactory.getKeyManagers();

// Initialize the trust manager from the keystore
TrustManagerFactory tmFactory =
  TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
tmFactory.init(keystore);
TrustManager tm[] = tmFactory.getTrustManagers();

// Initialize an SSL context and make it the default for all connections
SSLContext ctx = SSLContext.getInstance("TLS");
ctx.init(km, tm, null);
SSLContext.setDefault(ctx);
```

This code could be substituted in the WsDevClient class where the initial SSL context is set:

```
public void run() {
        try {
            // Configure the SSLContext with an insecure TrustManager
            // This is done to avoid the need to use valid certificates in the
            // development environment.
            // This should not be done in a real / secure environment.
            final SSLContext ctx = SSLContext.getInstance("TLS");
            ctx.init(new KeyManager[0], new TrustManager[]{
                        new InsecureTrustManager()
                    }, new SecureRandom());
            SSLContext.setDefault(ctx);
...
}
```

2.6.1.2 Python

In Python, there are a variety of libraries that can take care of your HTTP request process using REST including Requests, Urllib2 and HTTPLib. In this example, we use the Requests library as it provides good support for authentication and session handling.

```
s=requests.Session()
s.auth=('user','password')
s.verify=False #disables SSL certificate verification
```

In the example code, we use an option to disable SSL certificate verification. This is helpful when testing code against a demonstration environment using the default self-signed certificates, as these may not validate properly causing the HTTP request to fail. In a production environment, this line is not recommended and your SSL certificates should be updated to be fully valid and signed by a recognized certificate authority, or you should at least ensure that the Oracle VM Manager internal CA certificate is installed locally for validation purposes.

Since we would like to use JSON for all of our interactions with the API, it may prove worthwhile to take advantage of the Requests library's ability to append headers to all requests within a session. To do this, we can set the appropriate headers now and save ourselves the effort of doing this for each HTTP request that we make:

```
s.headers.update({'Accept': 'application/json', 'Content-Type': 'application/json'})
```

Finally, throughout this guide we are going to refer to the BaseURI that the REST API can be located on. For the sake of keeping our code relatively brief, we can set a variable for this now:

```
baseUri='https://127.0.0.1:7002/ovm/core/wsapi/rest'
```

The `baseUri` specified above follows the format described in Section 2.1, “Connecting to the REST Base URI”. It may vary depending on your own environment and where you intend your script to run from. Note that you can substitute the hostname and port values according to your own application requirements. In this example, we are assuming that you are running your Python scripts on the same system where Oracle VM Manager is hosted.

What about Certificate-based Authentication?

As already mentioned, it is possible to use a signed SSL certificate to authenticate against Oracle VM Manager via the REST API. This allows you to disable any requirement to enter a username or password to perform authentication. To do this, you must have the certificate and key stored in a single PEM file available to your application. The Requests library allows you to send a certificate with each request, or to set it to be used for every request within a session:

```
s.cert='/path/to/mycertificate.pem
```

As long as the certificate is valid and can be authenticated by Oracle VM Manager, the session is automatically logged in using your certificate.

2.6.2 Checking Oracle VM Manager Run State

As described in Section 2.5, “Good Practice: Check the Oracle VM Manager Start Up State”, it is good practice to check the running status of Oracle VM Manager before making any subsequent API calls. This is achieved by submitting a GET query to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/Manager
```

The running status of Oracle VM Manager is indicated by the managerRunState property, returned in the response to this query.

Your code should loop to repeat this query until the value of the managerRunState property is equivalent to 'RUNNING', before allowing any further operations.

2.6.2.1 Java

By using the Oracle VM Manager Web Services Client library in your code, the REST API is almost completely abstracted. The method to query the Manager object as presented in the OvmWsRestClient class is a simple call that constructs a GET request using the Jersey Builder and returns the Manager object:

```
    public Manager managerGet() throws WsException
    {
        try
        {
            final Builder b = getResourceFromPathElements("Manager", "Manager");

            final Manager manager = b.get(Manager.class);

            return manager;
        }
        catch (final UniformInterfaceException ex)
        {
            throw convertException(ex);
        }
    }
```

To track the manager status in Java, the API can be queried repeatedly using this managerGet() method to access the Manager object and query the ManagerRunState property value to determine whether or not the Oracle VM Manager instance has RUNNING runtime status. The code should sleep between requests to reduce the number of requests submitted to the Oracle VM Manager instance over a period and to allow time for the instance to properly start up. The following code provides an example of how you might perform the loop required to wait until the ManagerRunState is set to 'RUNNING' in your Java client application:

```
// Wait for the manager to be in a running state before doing any work
// Refresh the manager object once every second to get the latest run state
final Manager manager = api.managerGet();
while (manager.getManagerRunState() != ManagerRunState.RUNNING){
  try{
    Thread.sleep(1000);
  }
  catch (final InterruptedException e){}
  OvmWsClientTools.refresh(api, manager);
}
```

Of interest in this loop is the call to `OvmWsClientTools.refresh`, which is used to refresh the manager object model on each iteration of the loop. This code should be executed immediately after authentication before any other operations are attempted using the SOAP API.

2.6.2.2 Python

To track manager status in Python, the API can be queried repeatedly using the URI to access the Manager object and the managerRunState property value can be tested to determine whether or not the Oracle VM Manager instance has RUNNING runtime status. To keep the script polite, use the *time* library to sleep between requests. Since this is functionality that may be used repeatedly at different stages in a program, it is better to define it as a function:

```
...
def check_manager_state(baseUri,s):
        while True:
            r=s.get(baseUri+'/Manager')
            manager=r.json()
            if manager[0]['managerRunState'].upper() == 'RUNNING':
                break
            time.sleep(1)
         return;
```

In this function, the `while True:` statement creates an infinite loop. The requests library session object is used to submit a GET request to the appropriate URI and the response content is converted from JSON into a python data object. The object is a list that contains a dictionary with all of the Manager object properties as keys. We use an `if` statement to test the managerRunState property to see whether it

matches the string value 'RUNNING'. If a match is found, the infinite loop is broken and the function can return. If a match is not found, the `time.sleep` statement causes the function to wait for one second before submitting the next request, so that requests take place over a sensible period.

Usually, you would insert a call to this function directly after you authenticate, as described in Section 2.6.1, “Authenticating”. You may even incorporate this code into your authentication process, so that it is called whenever an authentication request is made within your application.

2.6.3 Listing Servers

The Web Services APIs provide various options to list servers that have already been discovered within an Oracle VM environment.

Get a list of all servers and their unique IDs

It is possible to obtain a concise list of servers that have been discovered within the Oracle VM environment along with their unique identifiers. This call is less bandwidth intensive than obtaining the full details for all of your servers. It can provide a quick method to obtain the IDs for servers within the environment.

The following URI can be used to perform a GET request:

```
https://hostname:port/ovm/core/wsapi/rest/Server/id
```

It is useful to note that the IDs returned by this request contain the URL that can be used to query all the details for each server object, as described below. This makes it simple to search for the URL that you should query to get the details for a particular server.

Get all details for a server based on its unique ID

It is possible to obtain all of the details for a particular server based on its unique ID.

The following URI can be used to perform a GET request:

```
https://hostname:port/ovm/core/wsapi/rest/Server/id
```

The id presented in the URI above should be substituted for the actual unique ID value of a server. Note that this URI is returned as part of the ID object returned in an listing of all server IDs.

Get a list of all details for all servers

Full details for all servers within an environment can be obtained easily. If your environment has a large number of servers, this call may be bandwidth intensive.

The following URI can be used to perform a GET request:

```
https://hostname:port/ovm/core/wsapi/rest/Server
```

2.6.3.1 Java

Since the HTTP query to obtain a listings of objects of a particular type is essentially the same for all ObjectTypes, the RestClient provides generic methods for these purposes, such as the getAll method:

```
    /**
     * Gets all objects of a given type.
     */
    @SuppressWarnings("unchecked")
    public <T> List<T> getAll(final Class<T> type) throws WsException
    {
        try
        {
```

```
            final Builder b = getResourceForType(type);
            return (List) b.get(getGenericListType(type));
        }
        catch (final UniformInterfaceException ex)
        {
            throw convertException(ex);
        }
    }
```

This means that to get a listing of all servers, a serverGetAll method is defined in the OvmWsRestClient class that presents the Server class model to the getAll method as follows:

```
...
    @Override
    public List<Server> serverGetAll() throws WsException
    {
        return getAll(Server.class);
    }
```

From the WsDevClient code (the sample application), a call to list all Servers uses the serverGetAll method defined in the OvmWsRestClient:

```
final List<Server> servers = api.serverGetAll();
```

This populates a list, allowing you to loop through it to work with Server details as required:

```
for (final Server server : servers)
            {
                if (testServerName.equals(server.getName()))
                {
                    testServer = server;
                }
                printServer(server);
            }
```

Using the methods exposed by the Server model class, it is possible to print out various attributes specific to the Server. In the sample code, we reference the printServer method which is contained within the WsDevClient class. This method performs a variety of tasks. For the purpose of illustrating how you are able to work with a Server object, we have truncated the code in the following listing:

```
private void printServer(final Server server)
    {
        System.out.println("Server id: " + server.getId());
        System.out.println("\tname: " + server.getName());
        System.out.println("\tdescription: " + server.getDescription());
        System.out.println("\tGeneration: " + server.getGeneration());
        System.out.println("\tServerPool id: " + server.getServerPoolId());
        System.out.println("\tCluster id: " + server.getClusterId());
...
    }
```

2.6.3.2 Python

Building on the authentication code provided in Section 2.2, “How do I Authenticate?” we continue to use the *requests.Session* object to send a GET request to list all server details to the REST API. The response content is in JSON format. Fortunately, the Requests library is capable of decoding JSON automatically. This means that you can treat the content returned by the HTTP request as a native Python data object:

```
 ...
 r=s.get(baseUri+'/Server')
 for i in r.json():
       # do something with the content
       print '{name} is {state}'.format(name=i['name'],state=i['serverRunState'])
 ...
```

The `baseUri` specified above is based on the variable that we set during authentication as described in Section 2.2, “How do I Authenticate?” and we have appended the Server objecttype to it to notify the API of the type of data that we are requesting.

2.6.4 Discovering Servers

To create a *server pool*, you first need server objects to add to the pool. You add unassigned Oracle VM Servers to your environment by discovering them. For Server discovery, the REST API expects parameters to be passed in the URI, with the exception of sensitive information such as user credentials.

To perform server discovery modify the following URL to suit your requirements, and submit a POST request:

```
https://hostname:port/ovm/core/wsapi/rest/Server/discover?serverName=1.example.org \
  &takeOwnershipIfUnowned=True
```

2.6.4.1 Java

The OvmWsRestClient class provides a method to handle server discovery. Since many of the parameters used to perform server discovery are used to construct the URI that must be used for the POST request, these parameters are passed to a the Jersey Builder after they have been processed for URI construction by a method defined in the RestClient class. The login credentials that Oracle VM Manager requires to connect to the Oracle VM Agent are sent within the body of the POST request as a simple string.

Many of the examples in this guide use basic methods that are defined within the RestClient class in `RestClient.java`. These methods use Jersey to set up a web client and also provide constructors to handle tasks like URI construction and XML construction using JAXB. These base methods are not discussed in detail within this guide, but should be studied by the reader if further understanding is required.

```
@Override
    public Job serverDiscover(final String serverName, final String agentPassword,
            final boolean takeOwnership)
    {
        final Map<String, Object> queryParameters = new HashMap<String, Object>();
        queryParameters.put("serverName", serverName);
        queryParameters.put("takeOwnershipIfUnowned", takeOwnership);

        final Builder b =
                getResourceFromUriAndPathElementsWithQueryParameters(null,
                     queryParameters, Server.class.getSimpleName(), "discover");

        return b.type(getMediaType()).post(Job.class, agentPassword);
    }
```

The WSDevClient class, in the sample code, does not include an example of server discovery, however to do server discovery, the code in your main class can be as simple as the following:

```
final Job serverCreateJob = api.serverDiscover("1.example.com", "p4ssword", true);
System.out.println("create server job id: " + serverCreateJob.getId());
sid=waitForJobComplete(api, serverCreateJob, Server.class);
```

Note that the XML response returned by the API contains information about the *job* that has been created to carry out the process requested. This is why we use the output from the serverDiscover method to populate a Job object. Using this object, you can easily track job status with the waitForJobComplete method, which also returns the server ID object if the job is successful.

Note

ID objects include the URI by which the complete object can be referenced. In Java, the URI can be quickly obtained using the `getUri` method against the ID object.

2.6.4.2 Python

Continuing on from the previous example, in Section 2.6.3, “Listing Servers”, the session object can be used to submit a POST request containing the agent password specified as a simple string:

```
...
uri_params={'serverName':'1.example.com', 'takeOwnershipIfUnowned':'True'}
data='p4ssword'
r=s.post(baseUri+'/Server/discover', data, params=uri_params)
```

If the request has been formatted correctly and the API accepts it, the JSON response returned contains information about the job that has been created to carry out the process requested. You can easily obtain the job URI from the response data and use it to track job status with the wait_for_job function:

```
job=r.json()
print 'Job: {name} for {server}'.format(name=job['id']['name'],server='1.example.org')
joburi=job['id']['uri']
wait_for_job(joburi,s)
```

Note

ID objects include the URI by which the complete object can be referenced. We can easily obtain this URI to perform another query against the API for the Job object so that we can track its status. In this example we reference the wait_for_job function, which has not been defined. Before you attempt to use this function, refer to Section 2.6.5, “Working with Jobs” to find out how you should go about defining it.

2.6.5 Working with Jobs

For most write-requests, using the POST, PUT and DELETE HTTP methods, the response from the API contains an XML or JSON representation of the job data specific to the process that has been queued on Oracle VM Manager for the task that you are performing. Since *jobs* within Oracle VM Manager are sequential and can take time to complete, it is common to check whether a job is complete before continuing. It is also useful to obtain job information to determine whether a task has succeeded or failed, along with error messages for job failure.

You can obtain details for a particular job by sending a GET request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/Job/id
```

Substitute the *id* in the URI with the ID that is generated for the job. Remember that the job URI is sent in the XML or JSON response returned for any POST, PUT or DELETE request.

2.6.5.1 Java

Tracking Job Status

The WSDevClient class, in the sample code, includes an example method to handle tracking Job status, so that your application can wait until job completion before continuing to submit requests that may be dependent on a job completing.

The method expects to be passed a Job object, which it uses to continually query the API to detect whether a Job is done or not. We check the **summaryDone** property of the job to determine whether the job is complete, as this provides a more complete view of the status of any spawned child jobs as opposed to the **done** property. See Section 5.1.4, “Working with Jobs” for more information on this. Usually, when performing a write type request against the API, a Job object is returned in the response from the API. This allows you to obtain the Job ID to perform these repeated queries.

```
...
@SuppressWarnings("unchecked")
    public <T> Id<T> waitForJobComplete(final OvmWsClient api, Job job,
              final Class<T> type) throws WsException
    {
        while (job.isSummaryDone() == false)
        {
            try
            {
                Thread.sleep(1000);
            }
            catch (final Exception e)
            {
            }

            job = api.jobGetById(job.getId());

            if (job.getJobRunState().equals(JobRunState.FAILURE))
            {
                final JobError error = job.getError();
                if (error != null)
                {
                    System.out.println(" error type: " + error.getType());
                    System.out.println(" error message: " + error.getMessage());
                }
                System.out.println(" transcript: " + api.jobGetDebugTranscript(job.getId()));
            }
        }

        @SuppressWarnings("rawtypes")
        final Id resultId = job.getResultId();
        if (type == null)
        {
            return resultId;
        }
        else
        {
            final Id<T> typedResultId = resultId;
            return typedResultId;
        }
    }
```

Note that to track job status, the method makes a call to the jobGetById method exposed in the OvmWsRestClient class, which contains the following code:

```
@Override
    public Job jobGetById(final Id<Job> jobId) throws WsException
    {
        return getById(jobId, Job.class);
    }
```

Earlier in this guide, we mentioned that there were some generic methods available in the OvmWsRestClient class, that can be reused to query different ObjectTypes within Oracle VM Manager. The getById method is one of these methods. In actual fact, this method calls the getResourceById method defined in the RestClient class, which in turn calls the getResourceFromURI method. The method is finally capable of using Jersey's Builder API to construct a REST GET request and to return an object. This chain of methods can get confusing to follow. For the sake of keeping things simple, we present the code for the getById method below:

```
public <T> T getById(final Id<T> id, final Class<T> type) throws WsException
    {
        try
        {
            final Builder b = getResourceById(id);
```

```
            return b.get(type);
        }
        catch (final UniformInterfaceException ex)
        {
            throw convertException(ex);
        }
    }
```

It is recommended that readers continue to follow through each of the methods called in the above code to see how the URI and request is constructed. For most developers, however, reusing the code already provided in the sample client can help to abstract the HTTP level of the REST API.

2.6.5.2 Python

Track Job Status

To track job status in Python, we can repeatedly query the API using the job URI returned in each response. To keep our script polite, we use the *time* library to sleep between requests. Since this is functionality that we require for a variety of tasks it is better to define it as a function:

```
...
def wait_for_job(joburi,s):
        while True:
            time.sleep(1)
            r=s.get(joburi)
            job=r.json()
            if job['summaryDone']:
                print '{name}: {runState}'.format(name=job['name'], runState=job['jobRunState'])
                if job['jobRunState'].upper() == 'FAILURE':
                    raise Exception('Job failed: {error}'.format(error=job['error']))
                elif job['jobRunState'].upper() == 'SUCCESS':
                    if 'resultId' in job:
                        return job['resultId']
                    break
                else:
                    break
```

A completed job that consists of an action where an object is created in Oracle VM Manager also includes the ID of the object that has been created. We check the **summaryDone** property of the job to determine whether the job is complete, as this provides a more complete view of the status of any spawned child jobs as opposed to the **done** property. See Section 5.1.4, “Working with Jobs” for more information on this. For these types of activities, it is useful to return this value after the job is complete, so that it is easier to perform subsequent tasks related to the same object. Only return this value if the job was successful and no error was returned.

2.6.6 Creating a Server Pool

After you have discovered the Oracle VM Servers within your environment, you can create a server pool. A *server pool* is a *domain* of physical and virtual resources that performs *virtual machine* migration, HA, and so on.

To create a server pool you need a Virtual IP address, and for usability reasons, a meaningful name. The Virtual IP address is used by Oracle VM Manager to reconnect to a server pool in case of a network loss, or other loss, to the master Oracle VM Server. For more information about HA, see the *Oracle VM Concepts Guide* .

A server pool is created by sending a POST request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/ServerPool
```

2.6.6.1 Java

Creating a Server Pool in Java is a fairly straightforward operation. The WsDevClient class (the sample application) contains an example of this:

```
...
// Create a new server pool
ServerPool testPool = new ServerPool();
testPool.setName(testServerPoolName);
testPool.setVirtualIp(testServerPoolIp);

final Job serverPoolCreateJob = api.serverPoolCreate(testPool);
System.out.println("create server pool job id: " + serverPoolCreateJob.getId());
testPoolId = waitForJobComplete(api, serverPoolCreateJob, ServerPool.class);
```

In this example, we create a new ServerPool object using the ServerPool model provided by the Oracle VM Manager client model library included in the SDK. Using this model, we set various parameters specific to the object. In this case, we only set the required parameters which include the server pool name and the server pool virtual IP address. In the example code, these are set to the values defined for these variables in the `WsDevClient.properties` file.

As described previously, a Job object is populated immediately in the return value provided by the serverPoolCreate method. We use this object to call the waitForJobComplete method, which ensures that the server pool object is successfully created before continuing. We use the return value from the waitForJobComplete method to populate testPoolId value, which we use later to handle other server pool related activity for the newly created server pool.

The serverPoolCreate method is called from the OvmWsRestClient class, where the following code is defined:

```
    @Override
    public Job serverPoolCreate(final ServerPool serverPool) throws WsException
    {
        return create(serverPool);
    }
```

As you can see, the action uses a more generic method defined within OvmWsRestClient which can be applied to create any ObjectType. This code uses the Jersey Builder to create the required REST request, and creates the required XML using the Java object that is passed to the method:

```
public <T extends BaseObject<T>> Job create(final T newObject) throws WsException
{
    try
     {
        final Builder b = getResourceForType(newObject.getClass());
        return b.type(getMediaType()).post(Job.class, createJAXBElement(newObject));
     }
    catch (final UniformInterfaceException ex)
     {
        throw convertException(ex);
     }
}
```

2.6.6.2 Python

As in our previous example, Section 2.6.4, “Discovering Servers”, the session object can be used to submit a POST request, this time containing the serverPool object which is described as a Python dictionary and converted to a JSON string as it is submitted:

```
...
```

```
data = {
            'name': 'MyServerPool',
            'virtualIp': '10.172.77.196',
        }
uri='{base}/ServerPool'.format(base=baseUri)
r=s.post(uri, data=json.dumps(data))
job=r.json()
# wait for the job to complete
sp_id=wait_for_job(job['id']['uri'],s)
```

As before, the response content returned by Oracle VM Manager, contains a JSON object describing the job that is created to add the server pool. See Track Job Status for more information on how to handle this content. Assuming this server pool is created without any hitch, the wait_for_job function should return the ID for the server pool that has been created, so we populate the **sp_id** variable with the contents returned by the wait_for_job function. This makes it easier for us to perform other immediate actions, such as setting up clustering.

2.6.7 Clustering

To configure HA functionality, you must set up a cluster and create a cluster heartbeat device.

A cluster is a component of the *server pool* and is created by submitting a POST request to the server pool's URI:

```
https://hostname:port/ovm/core/wsapi/rest/ServerPool/id/Cluster
```

The cluster heartbeat device is a component of the cluster itself, and is created by submitting a POST request to the cluster's URI:

```
https://hostname:port/ovm/core/wsapi/rest/Cluster/id/ClusterHeartbeatDevice
```

2.6.7.1 Java

Create the Cluster Object

A cluster, in terms of Oracle VM, is a child object of a server pool. The WsDevClient class in the sample code includes an example where a cluster object is created for the server pool created in the previous example. The code is straightforward:

```
...
// Create a new cluster
Cluster testCluster = new Cluster();
testCluster.setName(testClusterName);

final Job clusterCreateJob = api.serverPoolCreateCluster(testPoolId, testCluster);
System.out.println("create cluster job id: " + clusterCreateJob.getId());
testClusterId = waitForJobComplete(api, clusterCreateJob, Cluster.class);
```

A cluster object only requires that a name is set for the object instance. Once this has been done, the serverPoolCreateCluster method is called from the OvmWsRestClient class. Note that two parameters are passed to this method: the server pool ID value, obtained during the creation of the server pool, and the cluster object itself. The code for the serverPoolCreateCluster method is presented below:

```
@Override
    public Job serverPoolCreateCluster(final Id<ServerPool> serverPoolId,
        final Cluster cluster)
      throws WsException
    {
        return createChildObject(serverPoolId, cluster);
    }
```

As expected, this method uses a more generic method that allows you to create a child object for any ObjectType. This method is reused for other actions later in this guide. Once again, the createChildObject method in OvmWsRestClient uses the Jersey Builder to construct the SOAP message:

```
public <O extends BaseObject<O>, P extends BaseObject<P>> Job
     createChildObject(final Id<P> parentId, final O newObject, final
     Map<String, Object> queryParameters)
   throws WsException
{
   try
   {
     final Builder b = getResourceForCreateChild(parentId, newObject.getClass(),
              queryParameters);
     return b.type(getMediaType()).post(Job.class, createJAXBElement(newObject));
   }
   catch (final UniformInterfaceException ex)
   {
     throw convertException(ex);
   }
}
```

Create a Cluster Heartbeat Device

A cluster requires a heartbeat device that can be located on shared storage accessible to all servers that get added to the server pool. For this purpose, we need to create the heartbeat device as a child object of the cluster object. The WsDevClient class contains an example of this:

```
ClusterHeartbeatDevice hbDevice = new ClusterHeartbeatDevice();
hbDevice.setName(clusterHeartbeatDeviceName);
hbDevice.setStorageType(clusterHeartbeatStorageDeviceType);
switch (clusterHeartbeatStorageDeviceType)
{
   case NFS:
     hbDevice.setNetworkFileSystemId(clusterHeartbeatNetworkFileSystem.getId());
     break;
   case STORAGE_ELEMENT:
     hbDevice.setStorageElementId(clusterHeartbeatStorageElement.getId());
     break;
   default:
     throw new Exception(
     "Invalid cluster heartbeat storage device type: " +
        clusterHeartbeatStorageDeviceType);
}

final Job hbDeviceCreateJob = api.clusterCreateHeartbeatDevice(testClusterId, hbDevice);
System.out.println("create cluster heartbeat device job id: " +
       hbDeviceCreateJob.getId());
testHeartbeatDeviceId = waitForJobComplete(api, hbDeviceCreateJob,
       ClusterHeartbeatDevice.class);
```

The cluster heartbeat device requires a number of parameters that need to be set before the object is created. In the example, the code sets a variety of parameters based on variables defined in the `WsDevClient.properties` file. This provides the user of the example client with the option to define whether to use an NFS storage repository or an alternative such as an ISCSI LUN. Depending on the storage device type selected, the appropriate heartbeat device parameter, indicating the storage ID within Oracle VM Manager must be set.

Once the heartbeat device object parameters are set, the clusterCreateHeartbeatDevice method is called from the OvmWsRestClient class. Since the heartbeat device is a child object of the cluster the method expects the parent cluster ID value, as well as the heartbeat device object:

```
@Override
public Job clusterCreateHeartbeatDevice(final Id<Cluster> clusterId, final
```

```
    ClusterHeartbeatDevice heartbeatDevice)
    throws WsException
{
        return createChildObject(clusterId, heartbeatDevice);
}
```

As expected, this method calls the more generic createChildObject discussed earlier.

2.6.7.2 Python

Obtaining the Server Pool ID Value

In the previous example, Section 2.6.6, “Creating a Server Pool”, we created a server pool and finally obtained the server pool ID as an object that we could reuse immediately when setting up clustering. However, let's presume that we didn't get that ID for some reason, and now need to find it so that we can configure clustering. Using the REST API, we can obtain the ID Value for the server pool that we have created based on the name that we assigned to it:

```
r=s.get(baseUri+'/ServerPool/id')
for id in r.json():
   if id['name']=='MyServerPool':
      sp_id=id
```

Since this is the type of action that may need to be repeated for other objects in the environment, it may be useful to define a function for this type of action:

```
def get_id_from_name(s,baseUri,resource,obj_name):
        uri=baseUri+'/'+resource+'/id'
        r=s.get(uri)
        for obj in r.json():
           if 'name' in obj.keys():
            if obj['name']==obj_name:
                return obj
        raise Exception('Failed to find id for {name}'.format(name=obj_name))
```

Note

In terms of the WS-API, an ID for an object contains a collection of information, including the object name, the model used by the API to define the object, the URI that can be used to query the object, and the unique ID value. ID Values are typically used in URI's to identify a particular object, while IDs are usually passed as XML objects within the body of a POST or PUT request. For this reason, we distinguish between an ID and ID Value throughout this document. In this case, we return the entire ID object as this contains the full collection of information that may be required.

Create the Cluster Object

Now to create the cluster, we submit a POST request to the URI that we construct using the server pool id:

```
...
data={'name':'MyServerPool_cluster'}
uri='{base}/ServerPool/{spid}/Cluster'.format(base=baseUri, spid=sp_id['value'])
r=s.post(uri,data=json.dumps(data))
job=r.json()
# wait for the job to complete
cluster_id=wait_for_job(job['id']['uri'],s)
```

Note that in this piece of code, we assume that you have populated the **sp_id** with the dictionary returned for the server pool ID. In the URL that we are posting to, we specify the server pool ID Value, to attach the

cluster to the correct server pool. Once again, we wait for the job to complete using the function that we defined previously, and expect the successful completion of the job to return the cluster ID. We need this, so that we can create a Cluster Heartbeat Device attached to this new cluster object.

Create the Cluster Heartbeat Device

Finally, to create a cluster heartbeat device you need to choose a shared storage repository that can be used for the heartbeat device. If you have not already set up a storage repository, you must do so before continuing. See Section 2.6.9, “Discovering a Network File Server” and Section 2.6.10, “Creating a Storage Repository” for more information. In our example, we use an NFS repository that is already available within Oracle VM Manager, and obtain the ID for a File System that we have already reserved for the purpose of acting as a server pool file system. For this, we use the **get_id_from_name** function that we defined previously:

```
...
nfs_id=get_id_from_name(s,baseUri,'FileSystem','nfs:/mnt/vol1/poolfs01')
data={
        'name':'MyServerPool_cluster_heartbeat',
        'networkFileSystemId': nfs_id,
        'storageType':'NFS',
        }
uri='{base}/Cluster/{cluster_id}/ClusterHeartbeatDevice'.format(
    base=baseUri,
    cluster_id=cluster_id['value'])
r=s.post(uri,data=json.dumps(data))
job=r.json()
# wait for the job to complete
cl_hb_id=wait_for_job(job['id']['uri'],s)
```

2.6.8 Managing Servers in a Server Pool

Once your server pool is fully set up, you can manage the servers that belong to the server pool.

To add a server to the server pool, you send a PUT request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/ServerPool/id/addServer
```

Note

The first server that you add to a server pool is automatically promoted to the *Master server* for the pool. It is possible to modify the server pool object to set the ID for the Master server at a later stage.

Similarly, to remove a server, send a PUT request to:

```
https://hostname:port/ovm/core/wsapi/rest/ServerPool/id/removeServer
```

Note

Remove the Master server from the pool after all other servers have been removed

In both cases, the body of your request should contain an XML or JSON object that describes the server that you are managing. These two request types are good examples of action-style activities as described in Section 2.3, “What URI Mappings are Available and How do They Work?”.

It is interesting to note that these operations create associations between server objects and the server pool object. In many ways it may appear as though the server is treated as a child object of the server pool, this is not strictly true although the methods may behave similarly. To distinguish between operations on child objects and operations on associations, the API uses the add/remove nomenclature for methods that

add or remove an association, and uses the create/delete nomenclature for methods that add or remove a child object. This may become clearer when studying the examples below.

2.6.8.1 Java

The WsDevClient class includes an example where a server is added to the test server pool. The code checks whether the server belongs to another server pool already, and removes it from the server pool if it does. It then goes on to perform the add operation. This provides an example of both activities. To keep things simple, the code presented in this guide focuses on the actual operations required to perform each action.

Adding a server to a server pool.

Although server objects are not technically child objects, in the case where a server is added to a server pool it is treated as if it was a child object for this action. Therefore, it is necessary to pass the serverPoolAddServer function both the server pool ID value and the server ID object. The code to do this, as extracted from the WsDevClient class in the sample code is as follows:

```
...
final Job job = api.serverPoolAddServer(testPoolId, testServer.getId());
System.out.println("add server to pool job id: " + job.getId());
waitForJobComplete(api, job);
```

Since the action is effectively the same as adding a child object to its parent, the serverPoolAddServer method in the OvmWsRestClient class uses the more generic method addChildObject:

```
@Override
public Job serverPoolAddServer(final Id<ServerPool> serverPoolId,
  final Id<Server> serverId)
  throws WsException
{
    return addChildObject(serverPoolId, serverId);
}
```

The addChildObject method code follows:

```
public Job addChildObject(final Id<?> parent, final Id<?> child) throws WsException
{
    try
    {
        return action(parent, "add" + getSimpleName(child.getType()), child);
    }
    catch (final UniformInterfaceException ex)
    {
        throw convertException(ex);
    }
}
```

Removing a server from a server pool.

Removing a server from a server pool in Java is a similar process to adding one. Example code extracted from the WsDevClient class in the sample code follows:

```
final Id<ServerPool> testServerPoolId = testServer.getServerPoolId();
Job job = api.serverPoolRemoveServer(testServerPoolId, testServer.getId());
System.out.println("remove server from pool job id: " + job.getId());
waitForJobComplete(api, job);
```

In a similar manner to other more generic operations, the serverPoolRemoveServer method in the OvmWsRestClient class actually refers to the generic removeChildObject method:

```
public Job removeChildObject(final Id<?> parent, final Id<?> child)
```

```
throws WsException
{
    try
    {
        return action(parent, "remove" + getSimpleName(child.getType()), child);
    }
    catch (final UniformInterfaceException ex)
    {
        throw convertException(ex);
    }
}
```

It is important to understand that although the behavior of removing a server from a server pool is essentially the same as removing a child object from its parent, a server is not really a child object. The distinction is important, since the removal of a server pool object cannot result in the removal of the server. Technically, Oracle VM Manager does not allow you to remove a server pool until all of the servers have been removed from it, but the distinction remains.

2.6.8.2 Python

Adding a server to a server pool.

By searching for the IDs for each server name we wish to add to the server pool, it is straightforward to add a server to the server pool that we created:

```
...
svrid=get_id_from_name(s,baseUri,'Server','1.example.org')
uri='{base}/ServerPool/{spid}/addServer'.format(base=baseUri,spid=sp_id['value'])
r=s.put(uri,data=json.dumps(svrid))
job=r.json()
# wait for the job to complete
wait_for_job(job['id']['uri'],s)
```

Note that we assume that you have the **sp_id** variable set, from one of the previous examples. If you do not have this set, you must populate it with the id for the server pool that you are configuring.

Removing a server from a server pool.

Using the same logic as we used to add a server, it is also straightforward to remove a server from the server pool that we created:

```
...
svrid=get_id_from_name(s,baseUri,'Server','1.example.org')
uri='{base}/ServerPool/{spid}/removeServer'.format(base=baseUri,spid=sp_id['value'])
r=s.put(uri,data=json.dumps(svrid))
job=r.json()
# wait for the job to complete
wait_for_job(job['id']['uri'],s)
```

Note that we assume that you have the **sp_id** variable set, from one of the previous examples. If you do not have this set, you must populate it with the id for the server pool that you are configuring.

2.6.9 Discovering a Network File Server

In this example, we show how to *discover* a Network File Server using the REST API. Shared storage is required for a number of purposes within Oracle VM, such as storing the pool file system used for the cluster heartbeat device.

A Network File Server is frequently used to provide shared storage, but is not the only form of shared storage that can be used within your environment. For this reason, you need to specify which *Storage*

Connect plug-in you are using when you add a new storage resource. In this case, we use the "Oracle Generic Network File System" plug-in that is configured when Oracle VM is installed. For this reason, we must obtain the plug-in's object ID before we set up the filer, as this is required as one of the parameters that is sent in the API request.

Additionally, a Network File Server requires that at least one *Admin Server* is specified to handle administrative tasks on the filer. This must be the object ID for one of the servers that you have already discovered in your environment.

Once you have this information available, you can discover a Network File Server on your network by submitting a POST request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/FileServer
```

After discovery of a file server is complete, to make file server exports available to Oracle VM, it is usual to perform a file server refresh. This is achieved by obtaining the ID value for the newly added file server, and then submitting a PUT request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/FileServer/id/refresh
```

For the file server exports to become usable within Oracle VM it is also important that you perform a file system refresh for each file system available on the file server. This is achieved by submitting a PUT request to the following URI for each file-system:

```
https://hostname:port/ovm/core/wsapi/rest/FileServer/id/refresh
```

You can get a listing for all the file systems exported on a particular file server, after it has been refreshed, by submitting a GET request on the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/FileServer/id/FileSystem/id
```

2.6.9.1 Java

The WsDevClient class included in the sample code does not include an example to discover a Network File Server. Certainly, it expects that a Network File Server has already been discovered within your Oracle VM environment and that its file systems have already been refreshed. However, the OvmWsRestClient class included in the sample code does include all of the methods required to fully set up and configure a Network File Server. The following examples indicate the steps that you would need to take to perform these tasks.

As this process is fairly intensive, we have concentrated on using the methods available in the OvmWsRestClient class. We encourage the reader to refer to the code for each of these methods, to see how these methods actually use the Jersey REST Client to perform each query against the Oracle VM Manager REST API.

Discovering the Network File Server

The FileServer model requires a number of parameters to be set before it can be created. Significantly, you must obtain the File Server Oracle VM Storage Connect plug-in ID that should be used by the FileServer object. The generic Oracle VM Storage Connect plug-ins are created at installation time. This means that to select the appropriate plug-in you need to obtain its ID by looping through the existing plug-in IDs. In this example, we search for the "Oracle Generic Network File System" which can be used to connect to a Network File Server. The following code populates the FileServerPlugin Id object instance called testfileServerPluginId:

```
...
// Get a list of all File Server Plug-ins
final List<Id<FileServerPlugin>> fileServerPluginIds = api.fileServerPluginGetIds();
```

```
for (final Id<FileServerPlugin> fileServerPluginId : fileServerPluginIds)
{
    if (fileServerPluginId.getName().equals("Oracle Generic Network File System"))
    {
       testfileServerPluginId = fileServerPluginId;
    }
}
```

Also required for the creation of a FileServer object is a list of Server ID object instances for each server that you want to configure as Admin Servers for your Network File Server. You could narrow this down to one or more specific servers. For the sake of simplicity, we create a list that contains all server IDs, which are then used as Admin Servers:

```
// Get a list of all Servers (to keep things simple, all servers are added)
final List<Id<Server>> servIds = api.serverGetIds();
```

Now we can set up the FileServer object, and call the fileServerDiscover method to create the FileServer object. We ensure that we also obtain the FileServer ID value from the waitForJobComplete method, so that we are able to use it for refresh tasks that need to be performed once the Network File Server has been discovered.

```
// Discover FileServer
FileServer fileserver = new FileServer();
// Set the Name for your FileServer object
fileserver.setName("MyNFSServer");
// Set the AccessHost to the IP address or Hostname of the FileServer
fileserver.setAccessHost("10.172.76.125");
// Set the Admin Server IDs, this is a list of ids
// In this example we are adding all servers in the environment
fileserver.setAdminServerIds(servIds);
// Set the Fileserver Type, can be LOCAL, NETWORK or UNKNOWN
fileserver.setFileServerType(FileServer.FileServerType.NETWORK);
// Set the Plugin ID
fileserver.setFileServerPluginId(testfileServerPluginId);
// Set the FileServer as having Uniform Exports
fileserver.setUniformExports(true);

final Job fileserverCreateJob = api.fileServerDiscover(fileserver);
System.out.println("create fileserver job id: " + fileserverCreateJob.getId());
fsid=waitForJobComplete(api, fileserverCreateJob, FileServer.class);
```

Refresh The Network File Server

Once the Network File Server has been successfully discovered, it must be refreshed before it can be used within Oracle VM Manager. This can be achieved by obtaining its ID and then calling the fileServerRefresh method:

```
// Create a Job for FileServer Refreshing
final Job fileserverRefreshJob = api.fileServerRefresh(fsid);
System.out.println("refresh fileserver job id: " + fileserverRefreshJob.getId());
waitForJobComplete(api,fileserverRefreshJob);
```

Refresh File Systems

Finally, you must refresh the file systems so that they can be used within your Oracle VM environment. To do this, it is possible to use the getFileSystemIds method exposed by the FileServer model to populate a list of file system IDs. Loop through this list calling the fileSystemRefresh method for each file system ID:

```
// For all of the fileSystems on the FileServer, do a refresh
fileserver = api.fileServerGetById(fsid);
final List<Id<FileSystem>> fileSystemIds = fileserver.getFileSystemIds();
   for (final Id<FileSystem> fileSystemId : fileSystemIds)
   {
```

```
    final Job filesystemRefreshJob = api.fileSystemRefresh(fileSystemId);
    waitForJobComplete(api,file systemRefreshJob);
   }
```

2.6.9.2 Python

Discovering the Network File Server

In this example, we need to perform a number of tasks. The JSON object that we must send in the POST request contains the server ID for an admin server that is used to perform administrative tasks on the filer. It must also contain the ID for the Oracle VM Storage Connect plug-in that Oracle VM should use to connect to the filer. To handle this, we can rely on the **get_id_from_name** function that we defined in Obtaining the Server Pool ID Value:

```
pluginname="Oracle Generic Network File System"
adminserver="1.example.org"
plugin_id=get_id_from_name(s,baseUri,'FileServerPlugin',pluginname)
admin_id=get_id_from_name(s,baseUri,'Server',adminserver)
```

Now we can construct the JSON object for the network file server discovery and send the initial POST request:

```
...
data={
        'name': 'MyNFSFiler',
        'accessHost': '10.172.76.125',
        'fileServerType': 'NETWORK',
        'fileServerPluginId': plugin_id,
        'adminServerIds': [admin_id],
        'uniformExports': True,
    }
uri='{base}/FileServer'.format(base=baseUri)
r=s.post(uri,data=json.dumps(data))
job=r.json()
# wait for the job to complete
filer_id=wait_for_job(job['id']['uri'],s)
```

Refresh The Network File Server

To make use of the file systems that are exported by the network file server, you need to refresh the network file server:

```
# get the idval for the nfs
uri='{base}/FileServer/{nfsid}/refresh'.format(base=baseUri,nfsid=filer_id['value'])
r=s.put(uri)
job=r.json()
# wait for the job to complete
wait_for_job(job['id']['uri'],s)
```

Refresh File Systems

Now you finally need to obtain the file system ID values for each file system exported by the network file server, and refresh the file system for each of these:

```
uri='{base}/FileServer/{nfsid}/FileSystem/id'.format(base=baseUri,nfsid=filer_id['value'])
r=s.get(uri)
fsids=r.json()
for id in fsids:
    uri='{base}/FileSystem/{id}/refresh'.format(base=baseUri,id=id['value'])
    r=s.put(uri)
    job=r.json()
    # wait for the job to complete
    wait_for_job(job['id']['uri'],s)
```

2.6.10 Creating a Storage Repository

When you have discovered the exposed file systems on a network file server, and have refreshed the file system of your choice, you can create a storage repository on it. A storage repository is a child object of the file system that it is created on. In this example, we use a file system located on a Network File Server, however a repository can equally be created on alternate shared storage, such as an ISCSI LUN.

Add a storage repository by submitting a submitting a POST request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/FileSystem/id/Repository
```

2.6.10.1 Java

The WsDevClient class in the sample code does not contain an example showing how to create a repository. The sample code expects that a repository has already been created within your environment and that its details have been provided in the `WsDevClient.properties` file.

Creating a storage repository is not complicated and the process can be extrapolated from the other examples or from the API documentation included with the SDK. In this example, we use the information that we have already learned to create a storage repository.

A storage repository is a child object of a FileSystem object. Therefore, it is necessary to obtain the ID of the FileSystem object where the storage repository must be created. The following loop can be used to populate the testfileSystemId with the ID of a file system named "nfs:/mnt/vol1/repo01":

```
...
fileserver = api.fileServerGetById(fsid);
final List<Id<FileSystem>> fileSystemIds = fileserver.getFileSystemIds();
for (final Id<FileSystem> fileSystemId : fileSystemIds)
{
   if (fileSystemId.getName().equals("nfs:/mnt/vol1/repo01")){
        testfileSystemId = fileSystemId;
   }
}
```

Creating the repository, once you have the file system ID is straightforward, using the fileSystemCreateRepository method provided by the OvmWsRestClient class:

```
// Create a repository
Repository myrepo = new Repository();
myrepo.setName("MyRepository");
final Job repositoryCreateJob = api.fileSystemCreateRepository(testfileSystemId,
myrepo);
System.out.println("create repository job id: " + repositoryCreateJob.getId());
myrepoId = waitForJobComplete(api, repositoryCreateJob, Repository.class);
```

A quick glance at the code for the fileSystemCreateRepository method confirms that this method uses the generic createChildObject method that we first encountered when we created the cluster object.

```
    @Override
    public Job fileSystemCreateRepository(final Id<FileSystem> fileSystemId,
       final Repository repository) throws WsException
    {
        return createChildObject(fileSystemId, repository);
    }
```

2.6.10.2 Python

In this example, we use the **get_id_from_name** function that we defined in Obtaining the Server Pool ID Value to obtain the file system ID value for a file system with the name "nfs:/mnt/vol1/repo01". We use this value to construct the URI where we need to submit the POST request required to create the repository.

```
...
fsid=get_id_from_name(s,baseUri,'FileSystem',"nfs:/mnt/vol1/repo01")
data={
        'name': 'MyRepository',
}
uri='{base}/FileSystem/{fsid}/Repository'.format(base=baseUri, fsid=fsid['value'])
r=s.post(uri,data=json.dumps(data))
job=r.json()
# wait for the job to complete
repo_id=wait_for_job(job['id']['uri'],s)
```

2.6.11 Presenting a Storage Repository

When you have created the storage repository, you must decide to which Oracle VM Servers you are going to present it.

Present a repository to a server by submitting a PUT request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/Repository/id/present
```

The body of the message must contain an XML representation of the Server ID object to which the repository is to be presented.

2.6.11.1 Java

In the WsDevClient class, there is an example showing how to present a repository to a server, if it has not already been presented. Using the Repository model, it is possible to use the getPresentedServerIds method to check where the repository has already been presented.

Presenting a repository to a server is an action performed against the repository. The following code is extracted from the WsDevClient class:

```
...
final Repository testRepository = api.repositoryGetById(testRepoId);
if (testRepository.getPresentedServerIds() == null ||
             !testRepository.getPresentedServerIds().contains(testServerId))
{
         repoPresentJob = api.repositoryPresent(testRepoId, testServerId);
         System.out.println("present repository job id: " + repoPresentJob.getId());
         waitForJobComplete(api, repoPresentJob);
}
```

The repositoryPresent method is called from the OvmWsRestClient class:

```
@Override
    public Job repositoryPresent(final Id<Repository> repositoryId,
    final Id<Server> serverId)
    throws WsException
    {
        return action(repositoryId, "present", serverId);
    }
```

This method calls the more generic action method in the same class. The action method, in turn, ultimately resolves to calling the actionWithQueryParameters method in the RestClient class, which allows you to send additional parameters to an ObjectType in the API for an action event.

```
public Job actionWithQueryParameters(final Id<?> id, final String actionName, final
   Map<String, Object> queryParameters, final Object argument)
   throws WsException
   {
     if (argument instanceof List || argument instanceof Map)
     {
```

```
            throw new WsException(new WsErrorDetails(null,
                 "Invalid use of a list or map in argument"));
        }
        try
        {
            final Builder b = getResourceForAction(id, actionName, queryParameters);
            if (argument != null)
            {
                b.type(getMediaType()).entity(createJAXBElement(argument));
            }
             return b.put(Job.class);
        }
        catch (final UniformInterfaceException ex)
        {
            throw convertException(ex);
        }
       }
```

The actionWithQueryParameters method relies on the Jersey Builder to construct the correct URI to query, to build the XML object that should be sent, and to submit an HTTP PUT request to the set URI. It finally returns the response as a Job object.

2.6.11.2 Python

For this request, we must obtain the server ID object for the server to which we wish to present the repository. To achieve this, we use the **get_id_from_name** function that we defined in Obtaining the Server Pool ID Value.

```
...
servid=get_id_from_name(s,baseUri,'Server',"1.example.com")
uri='{base}/Repository/{rid}/present'.format(base=baseUri, rid=repo_id['value'])
r=s.put(uri,data=json.dumps(servid))
job=r.json()
# wait for the job to complete
wait_for_job(job['id']['uri'],s)
```

2.6.12 Creating Networks

It is possible to create different network types within Oracle VM Manager. In this example we show how to create a standard network object, as well as a local network limited to a single server.

To create a standard network for your environment to use, submit a POST request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/Network
```

To create a local network for your environment, it must be attached to the Oracle VM Server where it will run. To do this, submit a POST request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/Server/id/Network
```

2.6.12.1 Java

Creating a Network

Network objects are straightforward to create in themselves and don't require that anything other than the network Name attribute is set. The WsDevClient includes the following example code:

```
...
Network network = new Network();
network.setName(testNetworkName);
network.setDescription("Creating a test network named " + testNetworkName);
```

```
final Job networkCreateJob = api.networkCreate(network);
System.out.println("create network job id: " + networkCreateJob.getId());
testNetworkId = waitForJobComplete(api, networkCreateJob, Network.class);
```

As expected, in terms of the REST API, the networkCreate method in the OvmWsRestClient class, calls the more generic create method discussed in other examples within this guide:

```
@Override
public Job networkCreate(final Network network) throws WsException
{
    return create(network);
}
```

Creating a Local Network for a Server

Local networks differ from other network types in that they are flagged differently and the API call must be made against the server object that they are attached to. The following example code exists within the WsDevClient class included in the SDK:

```
// Create a new server local network
Network serverLocalNetwork = new Network();
serverLocalNetwork.setName("MyTestServerLocalNetwork");
testServerId = testServer.getId();
serverLocalNetwork.setServerId(testServerId);

final Job localNetworkCreateJob = api.serverCreateNetwork(testServerId, network);
System.out.println("create server local network job id: " +
     networkCreateJob.getId());
testServerLocalNetworkId = waitForJobComplete(api, localNetworkCreateJob,
     Network.class);
```

The serverCreateNetwork expects the ID for the server that the local network is being created for. Checking the method in the OvmWsRestClient class confirms that the method uses the more generic createChildObject method to construct the XML and URI required to perform the POST request.

2.6.12.2 Python

Creating a Network

Creating a network in Python is straightforward. The JSON object expected by the API consists of the network name, and optionally a description. The following code builds on our previous examples:

```
...
data = {
        'name':'MyNetwork',
        'description': 'A test network using the REST API',
    }
uri='{base}/Network'.format(base=baseUri)
r=s.post(uri,data=json.dumps(data))
job=r.json()
# wait for the job to complete
net_id=wait_for_job(job['id']['uri'],s)
```

Creating a Local Network for a Server

To create a local network for a specific server, we can use the **get_id_from_name** function that we defined in Obtaining the Server Pool ID Value to obtain an ID object for a specific server. We can then construct the JSON body to send in the POST requests and construct the URI to create a local network for the server:

```
...
```

```
svr_id=get_id_from_name(s,baseUri,'Server','1.example.com')
data = {
        'name': 'MyLocalNetwork',
        'description': 'Test network for 1.example.com',
        }
uri='{base}/Server/{sid}/Network'.format(base=baseUri,sid=svr_id['value'])
r=s.post(uri,data=json.dumps(data))
job=r.json()
# wait for the job to complete
localnet_id=wait_for_job(job['id']['uri'],s)
```

2.6.13 Creating Virtual Machines

In this section, we describe how to create a *virtual machine*. Your virtual machine configuration may vary and you may want to explore other aspects of the API to fit your own requirements. The examples given provide a basic guideline to getting started.

To create a virtual machine using the REST API, submit a POST request to:

```
https://hostname:port/ovm/core/wsapi/rest/Vm
```

The body content of the request should contain an XML representation of the virtual machine object, which should also include the server pool id where the virtual machine should be hosted.

2.6.13.1 Java

The WsDevClient class contains an example of the creation of a virtual machine:

```
...
Vm testVm = new Vm();
testVm.setVmDomainType(VmDomainType.XEN_HVM);
testVm.setName(testVmName);
testVm.setRepositoryId(testRepoId);

final Job vmCreateJob = api.vmCreate(testVm, testPoolId);
System.out.println("create vm job id: " + vmCreateJob.getId());
testVmId = waitForJobComplete(api, vmCreateJob, Vm.class);
```

In this code, some basic attributes of the virtual machine are set before it is created. In fact, there are a large number of attributes specific to a virtual machine that can be set to control how the virtual machine is configured. Here the most basic attributes have been selected to create a virtual machine using Xen hardware virtualization with its configuration located on the repository defined in the `WsDevClient.properties` file.

The vmCreate method also requires the server pool ID to be provided so that the virtual machine is created within the correct server pool. In fact, the method sets the value for the ServerPoolId attribute of the VM object before calling the more generic create method:

```
@Override
    public Job vmCreate(final Vm vm, final Id<ServerPool> serverPoolId) throws WsException
    {
        vm.setServerPoolId(serverPoolId);
        return create(vm);
    }
```

The WsDevClient class goes on to demonstrate many other actions that can be performed on a virtual machine via the REST API including removing and restoring server pool association, basic modifications of attributes, assigning VNICs and killing a virtual machine. It is recommended that the reader explore these examples and how they relate to methods within the OvmWsRestClient class to understand how to expand any interactions with virtual machines through the REST API.

2.6.13.2 Python

In the following code, some basic attributes of the virtual machine are set before it is created. In fact, there are a large number of attributes specific to a virtual machine that can be set to control how the virtual machine is configured. Here the most basic attributes have been selected to create a virtual machine using Xen para-virtualization with its configuration located on the repository called 'MyRepository'. The virtual machine is also attached to the server pool called 'MyServerPool'.

```
...
repo_id=get_id_from_name(s,baseUri,'Repository','MyRepository')
sp_id=get_id_from_name(s,baseUri,'ServerPool','MyServerPool')
data={
        'name': 'MyVirtualMachine',
        'description': 'A virtual machine created using the REST API',
        'vmDomainType': 'XEN_PVM',
        'repositoryId': repo_id,
        'serverPoolId': sp_id,
    }
uri='{base}/Vm'.format(base=baseUri)
r=s.post(uri,data=json.dumps(data))
job=r.json()
# wait for the job to complete
vm_id=wait_for_job(job['id']['uri'],s)
```

The VM created in this example is very simple and few attributes have been configured. If you intend to create virtual machines like this, it is worthwhile referring to the API documentation to discover what attributes can be set to fully configure your virtual machine.

2.6.14 Importing Assemblies

The final example in this guide is designed to show how to import an *assembly* containing a configuration of one or more virtual machines along with their *virtual disks* and any inter-connectivity between them, to ease set up and creation of your virtual machines within Oracle VM Manager.

An assembly is imported into a repository and its import is handled by an action request on the repository. You can import an assembly into a repository by submitting a PUT request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/Repository/id/importAssembly?url=url
```

Once the assembly has been imported into the repository, you can import a virtual machine from within the assembly into your environment by submitting a PUT request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/Vm/createFromAssemblyVm/Assemblyid
```

2.6.14.1 Java

Importing the Assembly

The WsDevClient class contains the following code to handle the assembly import:

```
...
final Job importAssemblyJob = api.repositoryImportAssembly(testRepoId, assemblyUrl);
System.out.println("import assembly job id: " + importAssemblyJob.getId());
testAssemblyId = waitForJobComplete(api, importAssemblyJob, Assembly.class);
```

The assemblyUrl is defined in the `WsDevClient.properties` file. The URL must point to a valid *assembly* package that Oracle VM Manager can access and download. The repositoryImportAssembly method uses two more generic methods defined in the OvmWsRestClient class to construct the URI and then to submit the PUT request using the Jersey Builder:

```
@Override
    public Job repositoryImportAssembly(final Id<Repository> repositoryId, final String url)
        throws WsException
    {
        final Map<String, Object> params = createQueryParameterMap("url", url);
        return actionWithQueryParameters(repositoryId, "importAssembly", params, null);
    }
```

Action requests that contain query parameters within the URI are constructed and submitted using the actionWithQueryParameters method in the RestClient class, described previously.

Importing a Virtual Machine From An Assembly

After the assembly has been imported into Oracle VM Manager you may want to import a virtual machine from within the assembly into your environment. First, you need to obtain the assemblyVm ID for the virtual machine that you want to import. For this, the WsDevClient class includes the following code to loop over the Vm ID's within the assembly. The code is simple and selects the first ID that it detects in the loop.

```
assemblyVms = api.assemblyVmGetListById(assembly.getAssemblyVmIds());
Id<AssemblyVm> testAssemblyVmId = null;
for (final AssemblyVm assemblyVm : assemblyVms)
{
    if (testAssemblyVmId == null)
    {
        testAssemblyVmId = assemblyVm.getId();
    }
```

Once an assemblyVmId has been set, the vmCreateFromAssemblyVm method can be called from OvmWsRestClient:

```
if (testAssemblyVmId != null)
{
    final Job importVmFromAssemblyJob = api.vmCreateFromAssemblyVm(testAssemblyVmId);
    System.out.println("import vm from assembly job id: " + importVmFromAssemblyJob.getId());
    importedAssemblyVmId = waitForJobComplete(api, importVmFromAssemblyJob, Vm.class);
}
```

Although this is an action request that functions through a PUT request, the URI construction is different to the format that we used when importing the assembly into Oracle VM Manager. For this reason, the vmCreateFromAssemblyVm method does not rely on a more generic method to construct the request:

```
public Job vmCreateFromAssemblyVm(final Id<AssemblyVm> assemblyVmId)
            throws WsException
{
    try
    {
        final Builder b = getResourceFromPathElements(Vm.class.getSimpleName(),
                    "createFromAssemblyVm", assemblyVmId.getValue());
        final Job job = b.put(Job.class);
        return job;
    }
    catch (final UniformInterfaceException ex)
    {
       throw convertException(ex);
    }
}
```

2.6.14.2 Python

Importing the Assembly

Importing an assembly into Oracle VM Manager using the REST API only requires you to submit a PUT request to a properly constructed URI. As a result, there is no requirement to build a JSON body for the

request, however since there are GET variables passed in the URL, we set these parameters separately to performing URL construction. The following Python code illustrates this:

```
assembly_url='http://example.com/assemblys/my_assembly.ovf
repo_id=get_id_from_name(s,baseUri,'Repository',"MyRepository")
uri='{base}/Repository/{repoid}/importAssembly'.format(
    base=baseUri,
    repoid=repo_id['value'],
    )
params={'url': assembly_url }
r=s.put(uri,params=params)
job=r.json()
# wait for the job to complete
assembly_id=wait_for_job(job['id']['uri'],s)
```

Importing a Virtual Machine From an Assembly

Once the assembly has completed its import, it is possible to query the API to obtain the ID values for virtual machines contained within the assembly. An assembly Vm Id is required to import it as a functional virtual machine within your environment. In this example we import all of the virtual machines in the assembly into our environment:

```
...
r=s.get('{base}/Assembly/{id}'.format(base=baseUri,id=assembly_id['value']))
assembly=r.json()
for i in assembly['assemblyVmIds']:
    uri='{base}/Vm/createFromAssemblyVm/{id}'.format(base=baseUri,id=i['value'])
    r=s.put(uri)
    job=r.json()
    # wait for the job to complete
    wait_for_job(job['id']['uri'],s)
```

By sending a GET request to the API first to obtain the full list of details for the assembly that we are working with, we can loop through each of the assemblyVmIds. In this loop, we are able to construct the URI required for the PUT request that must be submitted to trigger the import.

Chapter 3 Using the Oracle VM Manager SOAP API

Table of Contents

3.1 Connecting To The SOAP API

To access the SOAP web service, you can use the following base URI:

```
https://hostname:port/ovm/core/wsapi/soap
```

In this URI, replace *`hostname`* with the host name or IP address of *Oracle VM Manager*, and replace *`port`* with the number of the port where the SOAP web service is available - the default is 7002 (SSL).

3.2 How Do I Authenticate?

To gain access to *Oracle VM Manager*, a client must first authenticate successfully. The SOAP API has explicit login and logout calls for establishing an authenticated client session. The login method of the `OvmApi` class accepts the username, password and session parameters such as the user's locale.

It is important that the client maintains the session property across all subsequent calls by passing the appropriate session cookie information on subsequent requests. Many SOAP client implementations either do this automatically or can be configured to do so. The login method in the `OvmWsSoapClient` class provides an example of how to call the login method. The initialize method in that class provides an example of setting the *SESSION_MAINTAIN_PROPERTY* property to maintain the session across multiple API calls.

The SOAP API also has support for certificate-based authentication. As long as you have a valid certificate that is signed and registered either with the internal Oracle VM Manager CA certificate, or with a third-party CA for which your have imported the CA certificate into the Oracle VM Manager truststore, you can authenticate easily using your certificate instead of explicitly calling the login method. Signing and registering certificates against the internal Oracle VM Manager CA certificate can either be achieved using the provided certificate management tool, discussed in Setting up SSL on Oracle VM Manager in the

Oracle VM Administrator's Guide; or can be achieved programmatically using the Oracle VM Manager Utilities SOAP endpoint discussed in Chapter 4, *Additional Utilities Exposed in the WS-API*.

If you choose to make use of certificate-based authentication, you do not need to call the login method. However, it is good practice to call the logout method when the application terminates, so that Oracle VM Manager can perform session termination and cleanup.

3.3 Common SOAP API Calls To Retrieve Information

All SOAP API calls available for each object type are described in detail in the API documentation packaged within the SDK. The different ObjectTypes that can be accessed via the API are also described in the documentation included within the SDK. For quick reference, most object types support direct retrieval of the a list of all objects or IDs of a given type. These methods are:

- ***objectType*GetAll**: Retrieves all objects of type objectType.
- ***objectType*GetIds**: Retrieves the IDs of all objects of the given objectType.
- ***objectType*GetById**: Retrieves a full object representation based on the ID provided.
- ***objectType*GetListById**: Retrieves a set of objects based on the list of IDs provided.

3.4 Good Practice: Check the Oracle VM Manager Start Up State

The Oracle VM Manager start up process can take a number of minutes to complete. During start up, Oracle VM Manager is capable of accepting web services API calls for various debugging purposes, however jobs requested through the web services API are not initiated during start up and the model contents can change significantly over the course of the start-up process as various objects are refreshed and rediscovered. For this reason, it is good practice for an application, using the web services API, to check the running status of Oracle VM Manager before submitting API requests.

The running status of Oracle VM Manager is contained in the managerRunState property or attribute of the Manager object returned by the Oracle VM Web Services API. During start up, the value of this property is set to 'STARTING'. Once all start up operations and server rediscovery is complete, the value of this property changes to 'RUNNING'.

In your code, you should authenticate against Oracle VM Manager and then check that the managerRunState property of the Manager object is set to 'RUNNING' before performing any further operations. To do this using the SOAP interface, you can use the **managerGet()** method to obtain the values of all of the Manager object properties.

An example of this and a description of how to wait until the Oracle VM Manager is actually running is provided in

3.5 Example Code Using SOAP

The example code, provided in this section, steps through many common operations within *Oracle VM Manager*. For each example, we have provided code samples for both Java and Python. By nature, the SOAP API is language agnostic, so you may decide to use an alternative programming language to interface with the API. The code samples are provided to show how different operations might be performed using one of these two popular languages.

In a guide like this, the programming style and the choice of libraries used very much depend on the author and the version of the language used. More than likely, there are many more ways to achieve the

same result, even within the same language. These samples are not intended to be authoritative in their approach, but can be used as guidelines to developing your own applications.

Notes on the Java samples provided in these examples

Our Java samples are built around the Java code and library provided in the SDK that is bundled with Oracle VM Manager.

The Java Sample Client provided in the SDK includes all of the code required to perform the majority of supported interactions with the WS-API. The code is separated into two packages:

- `com.oracle.ovm.mgr.ws.client`: Contains the classes for both the REST and SOAP interfaces to the API. Notably, for these examples, the OvmWsSoapClient.java file contains much of the code referenced through this guide. In this guide, we attempt to describe how the actual client API code has been constructed to allow you to abstract many of the SOAP calls that you would need to make otherwise. In practice, you can use these classes without needing to know all of the underlying mechanics to the code, this is illustrated in the WSDevClient class discussed below.

- `com.oracle.ovm.mgr.ws.sample`: Contains the WsDevClient class, which performs particular actions using the API and which is set as your main class when you run the sample client. The WsDevClient class is an example of how you can use the client API classes to create your own applications drawing on all the abstraction provided by these classes. If you're just following this guide to work out how to use the API to write your own applications, you can concentrate on the code in the WsDevClient class.

The SDK also includes the Oracle VM Manager Web Services Client library in the form of a precompiled jar file called `OvmWsClient.jar`. This library contains models for all of the different ObjectTypes exposed through the API, as well as a variety of utilities that are useful to perform various actions on objects in the API. The SOAP API can be easily abstracted by using the methods provided within this library. This library must be included in your project to allow you to work with typical Oracle VM ObjectTypes.

Your code should import the models, that you intend to use, as they are described in the Web Services Client library. Typically, a full-scale Java IDE should handle this on your behalf when you import the library and as you make use of different models. The listing presented is provided for completeness. The following listing provides a full outline of model imports:

```
import com.oracle.ovm.mgr.ws.model.AccessGroup;
import com.oracle.ovm.mgr.ws.model.AffinityGroup;
import com.oracle.ovm.mgr.ws.model.Assembly;
import com.oracle.ovm.mgr.ws.model.AssemblyVirtualDisk;
import com.oracle.ovm.mgr.ws.model.AssemblyVm;
import com.oracle.ovm.mgr.ws.model.BaseObject;
import com.oracle.ovm.mgr.ws.model.CloneType;
import com.oracle.ovm.mgr.ws.model.Cluster;
import com.oracle.ovm.mgr.ws.model.ClusterHeartbeatDevice;
import com.oracle.ovm.mgr.ws.model.ClusterStorageFs;
import com.oracle.ovm.mgr.ws.model.ControlDomain;
import com.oracle.ovm.mgr.ws.model.Cpu;
import com.oracle.ovm.mgr.ws.model.CpuCompatibilityGroup;
import com.oracle.ovm.mgr.ws.model.EthernetPort;
import com.oracle.ovm.mgr.ws.model.Event;
import com.oracle.ovm.mgr.ws.model.FileServer;
import com.oracle.ovm.mgr.ws.model.FileServerPlugin;
import com.oracle.ovm.mgr.ws.model.FileSystem;
import com.oracle.ovm.mgr.ws.model.FileSystemMount;
import com.oracle.ovm.mgr.ws.model.FsAccessGroup;
import com.oracle.ovm.mgr.ws.model.Id;
import com.oracle.ovm.mgr.ws.model.Job;
import com.oracle.ovm.mgr.ws.model.LoggerManagementAttributes;
```

```
import com.oracle.ovm.mgr.ws.model.LoginCertificate;
import com.oracle.ovm.mgr.ws.model.Manager;
import com.oracle.ovm.mgr.ws.model.Network;
import com.oracle.ovm.mgr.ws.model.PeriodicTask;
import com.oracle.ovm.mgr.ws.model.Repository;
import com.oracle.ovm.mgr.ws.model.RepositoryExport;
import com.oracle.ovm.mgr.ws.model.ResourceGroup;
import com.oracle.ovm.mgr.ws.model.Server;
import com.oracle.ovm.mgr.ws.model.ServerController;
import com.oracle.ovm.mgr.ws.model.ServerPool;
import com.oracle.ovm.mgr.ws.model.ServerPoolNetworkPolicy;
import com.oracle.ovm.mgr.ws.model.ServerPoolPolicy;
import com.oracle.ovm.mgr.ws.model.ServerUpdateConfiguration;
import com.oracle.ovm.mgr.ws.model.ServerUpdateRepositoryConfiguration;
import com.oracle.ovm.mgr.ws.model.SimpleId;
import com.oracle.ovm.mgr.ws.model.Statistic;
import com.oracle.ovm.mgr.ws.model.StorageArray;
import com.oracle.ovm.mgr.ws.model.StorageArrayPlugin;
import com.oracle.ovm.mgr.ws.model.StorageElement;
import com.oracle.ovm.mgr.ws.model.StorageInitiator;
import com.oracle.ovm.mgr.ws.model.StoragePath;
import com.oracle.ovm.mgr.ws.model.StorageTarget;
import com.oracle.ovm.mgr.ws.model.VirtualDisk;
import com.oracle.ovm.mgr.ws.model.VirtualNic;
import com.oracle.ovm.mgr.ws.model.VirtualSwitch;
import com.oracle.ovm.mgr.ws.model.VlanInterface;
import com.oracle.ovm.mgr.ws.model.Vm;
import com.oracle.ovm.mgr.ws.model.VmCloneDefinition;
import com.oracle.ovm.mgr.ws.model.VmCloneNetworkMapping;
import com.oracle.ovm.mgr.ws.model.VmCloneStorageMapping;
import com.oracle.ovm.mgr.ws.model.VolumeGroup;
import com.oracle.ovm.mgr.ws.model.WsErrorDetails;
import com.oracle.ovm.mgr.ws.model.WsException;
import com.oracle.ovm.mgr.ws.model.Zone;
```

You can simplify these imports by simply doing:

```
import com.oracle.ovm.mgr.ws.model.*;
```

Additionally, to define how your application connects to the SOAP API and to use the methods provided, you must define the following imports:

```
import com.oracle.ovm.mgr.ws.client.OvmWsClient;
import com.oracle.ovm.mgr.ws.client.OvmWsClientFactory;
```

You can use the `com.oracle.ovm.mgr.ws.client` package within your own projects to reduce your development overhead.

In these examples, we show how the SOAP client has been implemented within the `com.oracle.ovm.mgr.ws.client` package, and also how it is used for particular actions in the WsDevClient class.

Note

The code included in the library expects that you are using JDK 7. Ensure that your project Source/Binary format is set to JDK7 and that you have the JDK7 libraries imported into your project. JDK 6 is not supported.

Notes on the Python samples provided in these examples

Our Python samples are intended to give the reader a feel for direct interaction with the API for the purpose of scripting quick interactions with Oracle VM Manager.

To keep the code simple, we have opted to make use of the latest stable version of the popular Python Suds (0.4.1) library to provide a complete SOAP client that can interact directly with the SOAP interface exposed by Oracle VM Manager's Web Services API. Your Python scripts should have at minimum, the following import defined:

```
from suds.client import Client
```

To properly use the Suds client library, you must provide the URL to the WSDL to the client when you instantiate an instance:

```
url = "https://localhost:7002/ovm/core/wsapi/soap?wsdl"
client=Client(url)
```

The `url` specified above follows the format described in Section 3.1, “Connecting To The SOAP API”. It may vary depending on your own environment and where you intend your script to run from.

Once the WSDL has been loaded, you are able to inspect the service object to obtain a list of methods provided by the Oracle VM Manager Web Services SOAP API:

```
print client
```

Truncated output follows:

```
Suds ( https://fedorahosted.org/suds/ )  version: 0.4 GA  build: R699-20100913

Service ( OvmApi ) tns="http://ws.mgr.ovm.oracle.com/"
   Prefixes (2)
      ns0 = "http://ws.mgr.ovm.oracle.com/"
      ns1 = "http://www.w3.org/2005/08/addressing"
   Ports (1):
      (OvmApiPort)
         Methods (420):
            accessGroupAddStorageInitiator(id accessGroupId, id storageInitiatorId, )
            accessGroupGetAll()
            accessGroupGetById(id accessGroupId, )
            accessGroupGetIds()
            accessGroupGetListById(id[] accessGroupIds, )
            accessGroupModify(accessGroup accessGroup, )
            accessGroupRemoveStorageInitiator(id accessGroupId, id storageInitiatorId, )
            affinityGroupAddServer(id affinityGroupId, id serverId, )
            affinityGroupAddVm(id affinityGroupId, id vmId, )
            affinityGroupGetAll()
            affinityGroupGetById(id affinityGroupId, )
            affinityGroupGetIds()
            affinityGroupGetListById(id[] affinityGroupIds, )
            affinityGroupModify(affinityGroup affinityGroup, )
            affinityGroupRemoveServer(id affinityGroupId, id serverId, )
            affinityGroupRemoveVm(id affinityGroupId, id vmId, )
            assemblyGetAll()
            assemblyGetById(id assemblyId, )
            assemblyGetIds()
            assemblyGetListById(id[] assemblyIds, )
            assemblyModify(assembly assembly, )
            assemblyRefresh(id assemblyId, )
            assemblyVirtualDiskGetAll()
            assemblyVirtualDiskGetById(id assemblyVirtualDiskId, )
            assemblyVirtualDiskGetIds()
            assemblyVirtualDiskGetListById(id[] assemblyVirtualDiskIds, )
            assemblyVirtualDiskModify(assemblyVirtualDisk assemblyVirtualDisk, )
            assemblyVmGetAll()
            assemblyVmGetById(id assemblyVmId, )
            assemblyVmGetIds()
            assemblyVmGetListById(id[] assemblyVmIds, )
            assemblyVmModify(assemblyVm assemblyVm, )
```

```
...
            vmGetAll()
            vmGetById(id vmId, )
            vmGetConsoleUrl(id vmId, )
            vmGetIds()
            vmGetListById(id[] vmIds, )
            vmGetSerialConsoleUrl(id vmId, )
            vmGetSupportedOsTypes()
            vmKill(id vmId, )
            vmMigrate(id vmId, id destinationServerId, )
            vmModify(vm vm, )
            vmMove(id vmId, id repositoryId, id vmCloneDefinitionId, )
            vmRestart(id vmId, )
            vmResume(id vmId, )
            vmSendApiMessage(id vmId, WsKeyValuePair[] message, xs:boolean logFlag, )
            vmStart(id vmId, )
            vmStop(id vmId, )
            vmSuspend(id vmId, )
...
```

If you experience any trouble using the Suds library, you may decide to import the standard Python logging library and set log levels for components in the Suds library that you are using, to help with any troubleshooting that you need to do:

```
import logging
logging.basicConfig(level=logging.INFO)
logging.getLogger('suds.client').setLevel(logging.DEBUG)
logging.getLogger('suds.transport').setLevel(logging.DEBUG)
logging.getLogger('suds.xsd.schema').setLevel(logging.DEBUG)
logging.getLogger('suds.wsdl').setLevel(logging.DEBUG)
```

It is recommended that you refer to the Suds library documentation at:

https://fedorahosted.org/suds/wiki/Documentation

Please also refer to Section 5.2, “Notable Issues for Suds Users” for some common problems that developers using the Suds Library may encounter.

Due to the syntax of some of our example code, we assume that you are using Python 2.6 or above.

3.5.1 Authenticating

Authentication using the SOAP API is handled by calling the login method. Most SOAP client APIs provide some facility for session management. It is also possible to authenticate using a signed certificate, if the certificate has been signed and registered by a CA that is recognized by Oracle VM Manager, as described in Section 4.1.3, “Certificate Management for Certificate-based Authentication Using SOAP”.

3.5.1.1 Java

Authentication against the SOAP API, in the Java client relies on the code defined in the OvmWsSoapClient class, which uses the packaged Oracle VM Manager Web Service libraries contained in the wsclient.jar file:

```
...
public class OvmWsSoapClient implements OvmWsClient
{
...
    @Override
    public boolean initialize(final String hostname, final String port,
        final boolean secure) throws MalformedURLException
    {
```

```
        this.hostname = hostname;
        this.port = port;

        if (secure)
        {
            protocol = SECURE_PROTOCOL;
        }
        else
        {
            protocol = INSECURE_PROTOCOL;
        }

        // Get a handle to the OvmApi Service
        final OvmApi_Service service =
                new OvmApi_Service(new URL(
                   protocol + "://" + hostname + ":" + port + "/ovm/core/wsapi/soap"));
        api = service.getOvmApiPort();
        // Configure the client to maintain the session across multiple API
        // calls
        ((BindingProvider) api).getRequestContext().put(
              BindingProvider.SESSION_MAINTAIN_PROPERTY, true);

        return true;
    }
...
    @Override
    public boolean login(final String username, final String password,
       final Locale locale) throws WsException
    {
        try
        {
            final SessionProperties props = new SessionProperties();
            props.setLocale(locale.toString());

            return api.login(username, password, props);
        }
        catch (final WsException_Exception ex)
        {
            throw convertException(ex);
        }
    }
...
}
```

Two methods are described here. The first sets up the handle that is used throughout your code to access the SOAP API and maintains session information across the application's runtime. The second method actually starts the session and uses the methods provided in the Oracle VM Manager Web Services Client library to handle authentication.

Now in the WsDevClient class all we need to do is call this method:

```
...

public class WsDevClient
{
...
public void run()
    {
        try
        {
         ...
         api = OvmWsClientFactory.getOvmWsClient(wsimpl);
         ...
         // Authenticate with the OvmApi Service
            api.login(username, password, Locale.getDefault());
         ...
```

```
        }
...
}
```

There are a few things to note about how this has been implemented in the sample client. The first point is that we refer to the OvmWsClientFactory class to determine whether we are using the REST client or the SOAP client, using the string variable 'wsimpl'. This class allows us to use the same demo code to show both APIs. It contains a switch statement that sets up the appropriate client object:

```
switch (implementation)
        {
            case SOAP:
                return new OvmWsSoapClient();

            case REST:
                return new OvmWsRestClient();
        }
```

A final thing to note, is that the call to the login method provides some preset variables.
In fact, the `com.oracle.ovm.mgr.ws.sample` package also includes a properties file: `WsDevClient.properties`. This file contains the default values for many of the variables referenced throughout this guide. For the sample code to work according to the specifics of your own environment, many of these variables must be overridden. Instructions for handling variable overriding are provided in the comments of this properties file itself.

What about Certificate-based Authentication?

Since the code in the SDK does not assume that you have set up user certificates, there are no examples showing how to authenticate using a signed SSL certificate. However, all that is required for this to happen is for you to use the certificate for all SOAP requests in your session. In Java, the easiest way to do this is to ensure that you have the certificate and its associated key stored in a keystore file. You can then use the keystore to load your key manager and trust manager that can be used to initialize an SSL context that is used for all connections.

The following example code, should help to get you started. It has been stripped of any error handling that may obfuscate what is required. This code expects the following variables to be initialized:

```
File keystoreFile;        // A file representing the location of the KeyStore
char[] keystorePassword;  // The KeyStore password
char[] keyPassword;       // The key password - if you didn't specify one when creating your
                          // key, then use the keystorePassword for this as well
```

The code required to use this keystore to initialize an SSL context follows:

```
// Load your keystore
FileInputStream stream = new FileInputStream(keystoreFile);
KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
keystore.load(stream, keystorePassword);
stream.close();

// Initialize the key manager from the keystore
KeyManagerFactory kmFactory =
  KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
kmFactory.init(keystore, keyPassword);
KeyManager km[] = kmFactory.getKeyManagers();

// Initialize the trust manager from the keystore
TrustManagerFactory tmFactory =
  TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
tmFactory.init(keystore);
TrustManager tm[] = tmFactory.getTrustManagers();
```

```
// Initialize an SSL context and make it the default for all connections
SSLContext ctx = SSLContext.getInstance("TLS");
ctx.init(km, tm, null);
SSLContext.setDefault(ctx);
```

This code could be substituted in the WsDevClient class where the initial SSL context is set:

```
public void run() {
        try {
            // Configure the SSLContext with an insecure TrustManager
            // This is done to avoid the need to use valid certificates in the
            // development environment.
            // This should not be done in a real / secure environment.
            final SSLContext ctx = SSLContext.getInstance("TLS");
            ctx.init(new KeyManager[0], new TrustManager[]{
                        new InsecureTrustManager()
                    }, new SecureRandom());
            SSLContext.setDefault(ctx);
...
}
```

3.5.1.2 Python

Authentication against the SOAP API using Python and the Suds library is simple. Assuming that you have already instantiated a client object using the instructions provided in Notes on the Python samples provided in these examples, you need only call the API login method:

```
client.service.login('user','password')
```

Note that once the client object is instantiated, all methods supported by the API become available under the service namespace within the client object.

Once authenticated, Suds maintains your session for you and you can continue to make other API calls. Your session is either terminated through a timeout, or by calling the logout method supported by the API.

```
client.service.logout()
```

What about Certificate-based Authentication?

The Suds library does not have built-in support for SSL authentication. This makes it difficult to use Python and SOAP in conjunction with certificate-based authentication. Typically, to achieve this, you need to configure your own HTTP transports to use with Suds and ensure that you include a facility to handle SSL authentication. Often it is easier to change the underlying transport used by Suds to make use of the Requests library, which provides good support for handling SSL certificates. There are many approaches to this, but the following example shows how we override the default transport to use the Requests library to handle all of our HTTP requests:

```
import requests
from suds.transport.http import HttpAuthenticated
from suds.transport import Reply, TransportError

class RequestsTransport(HttpAuthenticated):
    def __init__(self, **kwargs):
        self.cert = kwargs.pop('cert', None)
        HttpAuthenticated.__init__(self, **kwargs)

    def send(self, request):
        self.addcredentials(request)
        resp = requests.post(request.url, data=request.message,
                             headers=request.headers, cert=self.cert, verify=False)
        result = Reply(resp.status_code, resp.headers, resp.content)
        return result
```

Note that in this case, we have chosen not to perform SSL verification on the CA certificate, but this option can be left out of the method if you have added the CA certificate to your trusted certificates on the system where the application runs. Once you have defined this class, you can instantiate your client object in the following way:

```
t = RequestsTransport(cert='/path/to/mycertificate.pem')
headers = {"Content-Type" : "text/xml;charset=UTF-8", "SOAPAction" : ""}
client = Client(url,headers=headers,transport=t)
```

Once your client is instantiated, you are able to perform any of the SOAP API calls available, without the requirement to login using standard plain text authentication credentials.

3.5.2 Checking Oracle VM Manager Run State

As described in Section 3.4, “Good Practice: Check the Oracle VM Manager Start Up State”, it is good practice to check the running status of Oracle VM Manager before making any subsequent API calls. This is achieved by calling the managerGet() method.

The running status of Oracle VM Manager is indicated by the ManagerRunState property, returned in the response to this API call.

Your code should loop to repeat this query until the value of the ManagerRunState property is equivalent to 'RUNNING', before allowing any further operations.

3.5.2.1 Java

By using the Oracle VM Manager Web Services Client library in your code, the SOAP API is almost completely abstracted. As a result, the method to query the Manager object as presented in the OvmWsSoapClient class is a simple call directly to the equivalent method exposed directly through the library:

```
    public Manager managerGet() throws WsException
    {
        try
        {
            return api.managerGet();
        }
        catch (final WsException_Exception ex)
        {
            throw convertException(ex);
        }
    }
```

To track the manager status in Java, the API can be queried repeatedly using the managerGet() method to access the Manager object and query the ManagerRunState property value to determine whether or not the Oracle VM Manager instance has RUNNING runtime status. The code should sleep between requests to reduce the number of requests submitted to the Oracle VM Manager instance over a period and to allow time for the instance to properly start up. The following code provides an example of how you might perform the loop required to wait until the ManagerRunState is set to 'RUNNING' in your Java client application:

```
// Wait for the manager to be in a running state before doing any work
// Refresh the manager object once every second to get the latest run state
final Manager manager = api.managerGet();
while (manager.getManagerRunState() != ManagerRunState.RUNNING){
  try{
    Thread.sleep(1000);
  }
  catch (final InterruptedException e){}
  OvmWsClientTools.refresh(api, manager);
```

```
}
```

Of interest in this loop is the call to `OvmWsClientTools.refresh`, which is used to refresh the manager object model on each iteration of the loop. This code should be executed immediately after authentication before any other operations are attempted using the SOAP API.

3.5.2.2 Python

To track manager status in Python, the API can be queried repeatedly using the managerGet() method to access the Manager object and the managerRunState property value can be tested to determine whether or not the Oracle VM Manager instance has RUNNING runtime status. To keep the script polite, use the *time* library to sleep between requests. Since this is functionality that may be used repeatedly at different stages in a program, it is better to define it as a function:

```
import time
...
def check_manager_state(client):
        while True:
            manager=client.service.managerGet()
            if manager.managerRunState.upper() == 'RUNNING':
                break
            time.sleep(1)
        return;
```

In this function, the `while True:` statement creates an infinite loop. The suds library client object is used to submit a call to the managerGet() method exposed by the SOAP API. We use an `if` statement to test the managerRunState property to see whether it matches the string value 'RUNNING'. If a match is found, the infinite loop is broken and the function can return. If a match is not found, the `time.sleep` statement causes the function to wait for one second before submitting the next request, so that requests take place over a sensible period.

Usually, you would insert a call to this function directly after you authenticate, as described in Section 3.5.1, “Authenticating”. You may even incorporate this code into your authentication process, so that it is called whenever an authentication request is made within your application.

3.5.3 Listing Servers

The Web Services APIs provide various options to list servers that have already been discovered within an Oracle VM environment.

Get a list of all servers and their unique IDs

It is possible to obtain a concise list of servers that have been discovered within the Oracle VM environment along with their unique identifiers. This call is less bandwidth intensive than obtaining the full details for all of your servers. It can provide a quick method to obtain the IDs for servers within the environment.

Get a list of all details for all servers

Full details for all servers within an environment can be obtained easily. If your environment has a large number of servers, this call may be bandwidth intensive.

3.5.3.1 Java

By using the Oracle VM Manager Web Services Client library in your code, the SOAP API is almost completely abstracted. As a result, the method to list all server details as presented in the

OvmWsSoapClient class is a simple call directly to the equivalent method exposed directly through the library:

```
@Override
public List<Server> serverGetAll() throws WsException
{
    try
    {
        return api.serverGetAll();
    }
    catch (final WsException_Exception ex)
    {
        throw convertException(ex);
    }
}
```

From the WsDevClient code, a call to list all Servers uses the serverGetAll method defined in the OvmWsSoapClient:

```
final List<Server> servers = api.serverGetAll();
```

This populates a list, allowing you to loop through it to work with Server details as required:

```
for (final Server server : servers)
            {
                if (testServerName.equals(server.getName()))
                {
                    testServer = server;
                }
                printServer(server);
            }
```

Using the methods exposed by the Server model class, it is possible to print out various attributes specific to the Server. In the sample code, we reference the printServer method which is contained within the WsDevClient class. This method performs a variety of tasks. For the purpose of illustrating how you are able to work with a Server object, we have truncated the code in the following listing:

```
private void printServer(final Server server)
    {
        System.out.println("Server id: " + server.getId());
        System.out.println("\tname: " + server.getName());
        System.out.println("\tdescription: " + server.getDescription());
        System.out.println("\tGeneration: " + server.getGeneration());
        System.out.println("\tServerPool id: " + server.getServerPoolId());
        System.out.println("\tCluster id: " + server.getClusterId());
        System.out.println("\tCpuCompatibilityGroup id: " +
             server.getCpuCompatibilityGroupId());
        System.out.println("\tServerController id: " +
             server.getServerControllerId());
...
    }
```

3.5.3.2 Python

Once authenticated, the Suds client can be used to make any API call directly using the services exposed within the client.services namespace. Obtaining a listing of servers is as simple as:

```
client.service.serverGetAll()
```

Suds returns the response from the API as a workable Python object, so that no parsing is required in your own code. The following example code illustrates the ease with which you can extract information for your application:

```
servers=client.service.serverGetAll()
```

```
for i in servers:
    print '{name} is {state}'.format(name=i.name,state=i.serverRunState)
```

3.5.4 Discovering Servers

To create a *server pool*, you first need server objects to add to the pool. You add unassigned Oracle VM Servers to your environment by discovering them.

3.5.4.1 Java

The OvmWsSoapClient class provides a method to handle server discovery. This method simply requires the serverName or IP address, a text string containing the agentPassword and a boolean value to determine whether or not Oracle VM Manager should take ownership of the server:

```
...
@Override
    public Job serverDiscover(final String serverName,
        final String agentPassword, final boolean takeOwnership)
        throws WsException
    {
        try
        {
            return api.serverDiscover(serverName, agentPassword, takeOwnership);
        }
        catch (final WsException_Exception ex)
        {
            throw convertException(ex);
        }
    }
```

The WSDevClient class, in the sample code, does not include an example of server discovery, however to do server discovery, the code in your main class might look as follows:

```
final Job serverCreateJob = api.serverDiscover(
    "1.example.com", "p4ssword", true);
System.out.println("create server job id: " + serverCreateJob.getId());
sid=waitForJobComplete(api, serverCreateJob, Server.class);
```

Note that the SOAP response returned by the API contains information about the job that has been created to carry out the process requested. This is why we use the output from the serverDiscover method to populate a Job object. Using this object, you can easily track job status with the waitForJobComplete method, which also returns the server ID object if the job is successful.

3.5.4.2 Python

According to the WSDL, the method used to *discover* a server is:

```
serverDiscover(xs:string serverName,
   xs:string agentPassword, xs:boolean takeOwnership, )
```

This process is straightforward, using suds. Simply call the serverDiscover method from directly within your Python script and populate the values that the method expects:

```
job=client.service.serverDiscover('1.example.com', 'p4ssword','true')
```

If the API accepts the request, the SOAP response returned contains information about the job that has been created to carry out the process requested. To see the job information, returned do:

```
print job
```

The following typical output is returned:

```
(job){
   generation = 0
   id =
      (id){
         name = "Server Discover"
         type = "com.oracle.ovm.mgr.ws.model.Job"
         uri = "https://localhost:7002/ovm/core/wsapi/rest/Job/1362504787662"
         value = "1362504787662"
      }
   locked = False
   done = False
   jobGroup = False
   jobRunState = "NONE"
   jobSummaryState = "NONE"
   summaryDone = False
 }
```

You can easily use the job ID from the response data to track job status with the wait_for_job function:

```
sid=wait_for_job(client,job.id)
```

3.5.5 Working with Jobs

For most write-requests, the response from the API contain a SOAP object representing the job data specific to the process that has been queued on Oracle VM Manager for the task that you are performing. Since *jobs* within Oracle VM Manager are sequential and can take time to complete, it is common to check whether a job is complete before continuing. It is also useful to obtain job information to determine whether a task has succeeded or failed, along with error messages for job failure.

3.5.5.1 Java

Tracking Job Status

The WSDevClient class, in the sample code, includes an example method to handle tracking Job status, so that your application can wait until job completion before continuing to submit requests that may be dependent on a job completing.

The method expects to be passed a Job object, which it uses to continually query the API to detect whether a Job is done or not. We check the summaryDone property of the job to determine whether the job is complete, as this provides a more complete view of the status of any spawned child jobs as opposed to the **done** property. See Section 5.1.4, “Working with Jobs” for more information on this. Usually, when performing a write type request against the API, a Job object is returned in the response from the API. This allows you to obtain the Job ID to perform these repeated queries.

```
...
@SuppressWarnings("unchecked")
    public <T> Id<T> waitForJobComplete(final OvmWsClient api, Job job,
             final Class<T> type) throws WsException
    {
        while (job.isSummaryDone() == false)
        {
            try
            {
                Thread.sleep(1000);
            }
            catch (final Exception e)
            {
            }

            job = api.jobGetById(job.getId());

            if (job.getJobRunState().equals(JobRunState.FAILURE))
```

```
            {
                final JobError error = job.getError();
                if (error != null)
                {
                    System.out.println(" error type: " + error.getType());
                    System.out.println(" error message: " + error.getMessage());
                }
                System.out.println(" transcript: " + api.jobGetDebugTranscript(job.getId()));
            }
        }

        @SuppressWarnings("rawtypes")
        final Id resultId = job.getResultId();
        if (type == null)
        {
            return resultId;
        }
        else
        {
            final Id<T> typedResultId = resultId;
            return typedResultId;
        }
    }
```

Note that to track job status, the method makes a call to the jobGetById method exposed in the OvmWsSoapClient class, which contains the following code:

```
@Override
public Job jobGetById(final Id<Job> jobId) throws WsException
{
    try
    {
        return api.jobGetById(jobId);
    }
    catch (final WsException_Exception ex)
    {
        throw convertException(ex);
    }
}
```

Once again, this is essentially a wrapper around the equivalent method exposed by the API, as defined in the Web Services Client library.

3.5.5.2 Python

Track Job Status

To track job status in Python, we can repeatedly query the API using the job ID returned in each response. To keep our script polite, we use the *time* library to sleep between requests. Since this is functionality that is required for a variety of tasks it is better to define it as a function:

```
import time
...
def wait_for_job(client, job_id):
        while True:
            time.sleep(1)
            job = client.service.jobGetById(job.id)
            if job.summaryDone:
                if job.jobRunState == 'FAILURE':
                    raise Exception('Job failed: %s' % job.error)
                elif job.jobRunState == 'SUCCESS':
                    if 'resultId' in job:
                        return job.resultId
                    break
                elif job.jobRunState == 'RUNNING':
```

```
                continue
            else:
                break
```

A completed job that consists of a create style action also includes the ID of the object that has been created. We check the summaryDone property of the job to determine whether the job is complete, as this provides a more complete view of the status of any spawned child jobs as opposed to the **done** property. See Section 5.1.4, “Working with Jobs” for more information on this. For these types of activities, it is useful to return this value after the job is complete, so that it is easier to perform subsequent tasks related to the same object. Only return this value if the job was successful and no error was returned.

3.5.6 Creating a Server Pool

After you have discovered the Oracle VM Servers within your environment, you can create a server pool. A *server pool* is a *domain* of physical and virtual resources that performs *virtual machine* migration, HA, and so on.

To create a server pool you need a Virtual IP address, and for usability reasons, a meaningful name. The Virtual IP address is used by Oracle VM Manager to reconnect to a server pool in case of a network loss, or other loss, to the master Oracle VM Server. For more information about HA, see the *Oracle VM Concepts Guide* .

3.5.6.1 Java

Creating a Server Pool in Java is a straightforward operation. The WsDevClient class contains an example of this:

```
...
// Create a new server pool
ServerPool testPool = new ServerPool();
testPool.setName(testServerPoolName);
testPool.setVirtualIp(testServerPoolIp);

final Job serverPoolCreateJob = api.serverPoolCreate(testPool);
System.out.println("create server pool job id: " + serverPoolCreateJob.getId());
testPoolId = waitForJobComplete(api, serverPoolCreateJob, ServerPool.class);
```

In this example, we create a new ServerPool object using the ServerPool model provided by the Oracle VM Manager Web Services Client library included in the SDK. Using this model, we set various parameters specific to the object. In this case, we only set the required parameters which include the server pool name and the server pool virtual IP address. In the example code, these are set to the values defined for these variables in the `WsDevClient.properties` file.

As described previously, a Job object is populated immediately in the return value provided by the serverPoolCreate method. This object is used by the waitForJobComplete method, which ensures that the server pool object is successfully created before continuing. We use the return value from the waitForJobComplete method to populate testPoolId value, which we use later to handle other server pool related activity for the newly created server pool.

The serverPoolCreate method is called from the OvmWsSoapClient class, where the following code is defined:

```
@Override
public Job serverPoolCreate(final ServerPool serverPool) throws WsException
{
    try
    {
        return api.serverPoolCreate(serverPool);
    }
```

```
    catch (final WsException_Exception ex)
    {
        throw convertException(ex);
    }
}
```

As with most of the SOAP API calls, the methods defined in the OvmWsSoapClient class call equivalent methods directly from the API, as defined in the Oracle VM Manager Web Services Client library.

3.5.6.2 Python

The serverPoolCreate method exposed by the SOAP API expects a complex argument of type serverPool. Suds provides a facility to construct these complex argument types using the factory namespace within the client object. We can use the Suds client factory namespace to create the serverPool object that the method expects as an argument:

```
serverPool=client.factory.create('serverPool')
```

Suds uses the WSDL to define how these objects are created. This makes it easy for you to see what parameters can be populated:

```
print serverPool
```

```
(serverPool){
   description = None
   generation = None
   id =
      (id){
         name = None
         type = None
         uri = None
         value = None
      }
   locked = None
   name = None
   resourceGroupIds[] = <empty>
   userData[] = <empty>
   affinityGroupIds[] = <empty>
...
   unassignedVmIds[] = <empty>
   virtualIp = None
   vmIds[] = <empty>
   vmStartPolicy =
      (vmStartPolicy){
         value = None
      }
   zoneId =
      (id){
         name = None
         type = None
         uri = None
         value = None
      }
 }
```

In this case, we only need to populate some very basic attributes to create a simple server pool:

```
serverPool.name="MyServerPool"
serverPool.virtualIp="10.172.77.196"
```

If you need to configure other parameters for your server pool, you can set the attributes now or edit the server pool later. For now, setting the name and virtualIp parameters are sufficient for this example. Once this is complete, you can call the serverPoolCreate method:

```
job=client.service.serverPoolCreate(serverPool)
svrpool_id=wait_for_job(client,job.id)
```

As before, the response content returned by Oracle VM Manager, contains a SOAP object describing the job that is created to add the server pool. See Track Job Status for more information on how to handle this content.

3.5.7 Clustering

To configure HA functionality, you must set up a cluster and create a cluster heartbeat device.

A cluster is a component of the *server pool* and is created by calling the serverPoolCreateCluster method.

The cluster heartbeat device is a component of the cluster itself, and is created by calling the clusterCreateHeartbeatDevice method.

3.5.7.1 Java

Create the Cluster Object

A cluster, in terms of Oracle VM, is a child object of a server pool. The WsDevClient class in the sample code includes an example where a cluster object is created for the server pool created in the previous example. The sample code follows:

```
// Create a new cluster
Cluster testCluster = new Cluster();
testCluster.setName(testClusterName);

final Job clusterCreateJob = api.serverPoolCreateCluster(testPoolId, testCluster);
System.out.println("create cluster job id: " + clusterCreateJob.getId());
testClusterId = waitForJobComplete(api, clusterCreateJob, Cluster.class);
```

A cluster object only requires that a name is set for the object instance. Once this has been done, the serverPoolCreateCluster method is called from the OvmWsSoapClient class. Note that two parameters are passed to this method: the server pool ID value, obtained during the creation of the server pool, and the cluster object itself. The code for the serverPoolCreateCluster method is presented below:

```
@Override
    public Job serverPoolCreateCluster(final Id<ServerPool> serverPoolId, ]
       final Cluster cluster) throws WsException
    {
        try
        {
            return api.serverPoolCreateCluster(serverPoolId, cluster);
        }
        catch (final WsException_Exception ex)
        {
            throw convertException(ex);
        }
    }
```

As expected, this method is a wrapper around the actual API call.

Create a Cluster Heartbeat Device

A cluster requires a heartbeat device that can be located on shared storage accessible to all servers that get added to the server pool. For this purpose, we need to create the heartbeat device as a child object of the cluster object. The WsDevClient class contains an example of this:

```
ClusterHeartbeatDevice hbDevice = new ClusterHeartbeatDevice();
```

```
hbDevice.setName(clusterHeartbeatDeviceName);
hbDevice.setStorageType(clusterHeartbeatStorageDeviceType);
switch (clusterHeartbeatStorageDeviceType)
{
   case NFS:
     hbDevice.setNetworkFileSystemId(clusterHeartbeatNetworkFileSystem.getId());
     break;
   case STORAGE_ELEMENT:
     hbDevice.setStorageElementId(clusterHeartbeatStorageElement.getId());
     break;
   default:
     throw new Exception(
     "Invalid cluster heartbeat storage device type: " +
        clusterHeartbeatStorageDeviceType);
}

final Job hbDeviceCreateJob = api.clusterCreateHeartbeatDevice(testClusterId, hbDevice);
System.out.println("create cluster heartbeat device job id: " +
       hbDeviceCreateJob.getId());
testHeartbeatDeviceId = waitForJobComplete(api, hbDeviceCreateJob,
       ClusterHeartbeatDevice.class);
```

The cluster heartbeat device requires a number of parameters that need to be set before the object is created. In the example, the code sets a variety of parameters based on variables defined in the `WsDevClient.properties` file. This provides the user of the example client with the option to define whether to use an NFS storage repository or an alternative such as an ISCSI LUN. Depending on the storage device type selected, the appropriate heartbeat device parameter, indicating the storage ID within Oracle VM Manager must be set.

Once the heartbeat device object parameters are set, the clusterCreateHeartbeatDevice method is called from the OvmWsSoapClient class. As for most of the methods defined in the OvmWsSoapClient class, this method is a wrapper for the actual API call.

3.5.7.2 Python

Obtaining the Server Pool ID Value

If you have only just created the server pool, and you made use of the wait_for_job function that we defined in Track Job Status, you should already have a dictionary populated with the new server pool ID. If you are creating a cluster for an existing server pool and need to obtain an ID value for it, you can use the SOAP API to obtain the ID for any server pool based on the name assigned to it:

```
spids=client.service.serverPoolGetIds()
for sp in spids:
    if sp.name=='MyServerPool':
       svrpool_id=sp
```

Create the Cluster Object

Using the Suds Client Factory, it is simple to define the cluster object that is required for the serverPoolCreateCluster method. In this case, we only set the name of the cluster object:

```
cluster=client.factory.create('cluster')
cluster.name='MyServerPool_cluster'
```

Using the server pool ID and the cluster object that we have just created, we can call the serverPoolCreateCluster method to create the cluster for the server pool:

```
job=client.service.serverPoolCreateCluster(svrpool_id,cluster)
clusterid=wait_for_job(client,job.id)
```

Create the Cluster Heartbeat Device

Finally, to create a cluster heartbeat device you need to choose a shared storage repository that can be used for the heartbeat device. If you have not already set up a storage repository, you must do so before continuing. See Section 3.5.9, “Discovering a Network File Server” and Section 3.5.10, “Creating a Storage Repository” for more information. In our example, we use an NFS repository that is already available within Oracle VM Manager:

```
for id in client.service.fileSystemGetIds():
    if hasattr(id,'name'):
        if id.name=="nfs:/mnt/vol1/poolfs01":
            fsid=id
```

Now use the Suds Client Factory to create a clusterHeartbeatDevice object:

```
clhbdev=client.factory.create('clusterHeartbeatDevice')
clhbdev.name="MyServerPool_cluster_heartbeat"
clhbdev.storageType="NFS"
clhbdev.networkFileSystemId=fsid
job=client.service.clusterCreateHeartbeatDevice(clusterid,clhbdev)
clhbdevid=wait_for_job(client,job.id)
```

3.5.8 Managing Servers in a Server Pool

Once your server pool is fully set up, you can manage the servers that belong to the server pool.

To add a server to the server pool use the serverPoolAddServer method, which requires the ID of the server pool and the ID of the server.

Note

The first server that you add to a server pool is automatically promoted to the *Master server* for the pool. It is possible to modify the server pool object to set the ID for the Master server at a later stage.

Similarly, to remove a server, use the serverPoolRemoveServer method.

Note

Remove the Master server from the pool after all other servers have been removed

3.5.8.1 Java

The WsDevClient class includes an example where a server is added to the test server pool. The code checks whether the server belongs to another server pool already, and removes it from the server pool if it does. It then goes on to perform the add operation. This provides an example of both activities. To keep things simple, the code presented in this guide focuses on the actual operations required to perform each action.

Adding a server to a server pool.

Although server objects are not technically child objects, in the case where a server is added to a server pool it is treated as if it was a child object for this action. Therefore, it is necessary to pass the serverPoolAddServer function both the server pool ID value and the server ID object. The code to do this, as extracted from the WsDevClient class in the sample code is as follows:

```
...
final Job job = api.serverPoolAddServer(testPoolId, testServer.getId());
```

```
System.out.println("add server to pool job id: " + job.getId());
waitForJobComplete(api, job);
```

Removing a server from a server pool.

Removing a server from a server pool in Java is a similar process to adding one. Example code extracted from the WsDevClient class in the sample code follows:

```
final Id<ServerPool> testServerPoolId = testServer.getServerPoolId();
Job job = api.serverPoolRemoveServer(testServerPoolId, testServer.getId());
System.out.println("remove server from pool job id: " + job.getId());
waitForJobComplete(api, job);
```

3.5.8.2 Python

Obtaining the server ID.

Since the API expects an object containing the server ID that you are either adding or removing from the server pool, you should obtain these details beforehand by querying for a list of Server IDs. See Section 3.5.3, “Listing Servers” for more information on this. Example code to extract a particular server ID based on its name within follows:

```
for i in client.service.serverGetIds():
    if i.name=='1.example.com':
       sid=i
```

Adding a server to a server pool.

Using the server ID that we obtained in Obtaining the server ID., we can add the server to the server pool that we created:

```
job=client.service.serverPoolAddServer(svrpool_id,sid)
wait_for_job(client,job.id)
```

Removing a server from a server pool.

Using the server ID that we obtained in Obtaining the server ID., we can remove the same server from the server pool that we created:

```
job=client.service.serverPoolRemoveServer(svrpool_id,sid)
wait_for_job(client,job.id)
```

3.5.9 Discovering a Network File Server

In this example, we show how to *discover* a Network File Server using the SOAP API. Shared storage is required for a number of purposes within Oracle VM, such as storing the pool file system used for the cluster heartbeat device.

A Network File Server is frequently used to provide shared storage, but is not the only form of shared storage that can be used within your environment. For this reason, you need to specify which *Storage Connect* plug-in you are using when you add a new storage resource. In this case, we use the "Oracle Generic Network File System" plug-in that is configured when Oracle VM is installed. For this reason, we must obtain the plug-in's object ID before we set up the filer, as this is required as one of the parameters that is sent in the API request.

Additionally, a Network File Server requires that at least one *Admin Server* is specified to handle administrative tasks on the filer. This must be the object ID for one of the servers that you have already discovered in your environment.

Once you have this information available, you can discover a Network File Server on your network using the fileServerDiscover method.

After discovery of a file server is complete, to make file server exports available to Oracle VM, it is usual to perform a file server refresh. Obtain the ID value for the newly added file server, and then perform a fileServerRefresh.

For the file server exports to become usable within Oracle VM it is also important that you perform a file system refresh for each file system available on the file server. This is achieved by calling the fileSystemRefresh method.

You can get a listing for all the file systems exported on a particular file server, after it has been refreshed, by calling the fileSystemGetIds method.

3.5.9.1 Java

The WsDevClient class included in the sample code does not include an example to discover a Network File Server. Certainly, it expects that a Network File Server has already been discovered within your Oracle VM environment and that its file systems have already been refreshed. However, the OvmWsSoapClient class included in the sample code does include all of the methods required to fully set up and configure a Network File Server. The following examples indicate the steps that you would need to take to perform these tasks.

Discovering the Network File Server

The FileServer model requires a number of parameters to be set before it can be created. Significantly, you must obtain the File Server *Storage Connect* plug-in ID that should be used by the FileServer object. The generic Oracle VM Storage Connect plug-ins are created at installation time. This means that to select the appropriate plug-in you need to obtain its ID by looping through the existing plug-in IDs. In this example, we search for the "Oracle Generic Network File System" which can be used to connect to a Network File Server. The following code populates the FileServerPlugin Id object instance called testfileServerPluginId:

```
...
// Get a list of all File Server Plug-ins
final List<Id<FileServerPlugin>> fileServerPluginIds = api.fileServerPluginGetIds();
for (final Id<FileServerPlugin> fileServerPluginId : fileServerPluginIds)
{
    if (fileServerPluginId.getName().equals("Oracle Generic Network File System"))
    {
       testfileServerPluginId = fileServerPluginId;
    }
}
```

Also required for the creation of a FileServer object is a list of Server ID object instances for each server that you want to configure as Admin Servers for your Network File Server. You could narrow this down to one or more specific servers. For the sake of simplicity, we create a list that contains all server IDs, which are then used as Admin Servers:

```
// Get a list of all Servers (to keep things simple, all servers are added)
final List<Id<Server>> servIds = api.serverGetIds();
```

Now we can set up the FileServer object, and call the fileServerDiscover method to create the FileServer object. We ensure that we also obtain the FileServer ID value from the waitForJobComplete method, so that we are able to use it for refresh tasks that need to be performed once the Network File Server has been discovered.

```
// Discover FileServer
FileServer fileserver = new FileServer();
// Set the Name for your FileServer object
```

```
fileserver.setName("MyNFSServer");
// Set the AccessHost to the IP address or Hostname of the FileServer
fileserver.setAccessHost("10.172.76.125");
// Set the Admin Server IDs, this is a list of ids
// In this example we are adding all servers in the environment
fileserver.setAdminServerIds(servIds);
// Set the Fileserver Type, can be LOCAL, NETWORK or UNKNOWN
fileserver.setFileServerType(FileServer.FileServerType.NETWORK);
// Set the Plugin ID
fileserver.setFileServerPluginId(testfileServerPluginId);
// Set the FileServer as having Uniform Exports
fileserver.setUniformExports(true);

final Job fileserverCreateJob = api.fileServerDiscover(fileserver);
System.out.println("create fileserver job id: " + fileserverCreateJob.getId());
fsid=waitForJobComplete(api, fileserverCreateJob, FileServer.class);
```

Refresh The Network File Server

Once the Network File Server has been successfully discovered, it must be refreshed before it can be used within Oracle VM Manager. Obtain its ID and then call the fileServerRefresh method:

```
// Create a Job for FileServer Refreshing
final Job fileserverRefreshJob = api.fileServerRefresh(fsid);
System.out.println("refresh fileserver job id: " + fileserverRefreshJob.getId());
waitForJobComplete(api,fileserverRefreshJob);
```

Refresh File Systems

Finally, you must refresh the file systems so that they can be used within your Oracle VM environment. To do this, it is possible to use the getFileSystemIds method exposed by the FileServer model to populate a list of file system IDs. Loop through this list calling the fileSystemRefresh method for each file system ID:

```
// For all of the fileSystems on the FileServer, do a refresh
fileserver = api.fileServerGetById(fsid);
final List<Id<FileSystem>> fileSystemIds = fileserver.getFileSystemIds();
   for (final Id<FileSystem> fileSystemId : fileSystemIds)
   {
    final Job filesystemRefreshJob = api.fileSystemRefresh(fileSystemId);
    waitForJobComplete(api,file systemRefreshJob);
   }
```

3.5.9.2 Python

Discovering the Network File Server

In this example, we need to perform a number of tasks. First, it is necessary to obtain the server ID for an Admin Server that is used to perform administrative tasks on the filer. Second, it is also necessary to obtain the ID for the Oracle VM Storage Connect plug-in used to connect to the filer. In this example, we use the server ID object that we created earlier, and we loop through the Oracle VM Storage Connect plug-in options to select the ID for the "Oracle Generic Network File System".

```
adminserver=sid
for i in client.service.fileServerPluginGetIds():
    if i.name=="Oracle Generic Network File System":
        plugin_id=i
```

Now we can use the Suds Client Factory to construct the SOAP object for the network file server discovery and send the initial request:

```
fileServer=client.factory.create('fileServer')
fileServer.adminServerIds=adminserver
```

```
fileServer.name="MyNFSServer"
fileServer.accessHost='10.172.76.125'
// Set the Fileserver Type, can be LOCAL, NETWORK or UNKNOWN
fileServer.fileServerType="NETWORK"
fileServer.fileServerPluginId=plugin_id
// Set the FileServer as having Uniform Exports
fileserver.setUniformExports="true"
job=client.service.fileServerDiscover(fileServer)
nfsid=wait_for_job(client,job.id)
```

Refresh The Network File Server

To make use of the file systems that are exported by the network file server, you need to refresh the network file server. :

```
job=client.service.fileServerRefresh(nfsid)
wait_for_job(client,job.id)
```

Refresh File Systems

Now you finally need to obtain the file system ID values for each file system exported by the network file server, and refresh the file system for each of these:

```
nfs=client.service.fileServerGetById(nfsid)
for fsid in nfs.fileSystemIds:
    job=client.service.fileSystemRefresh(fsid)
    wait_for_job(client,job.id)
```

3.5.10 Creating a Storage Repository

When you have discovered the exposed file systems on a network file server, and have refreshed the file system of your choice, you can create a storage repository on it. A storage repository is a child object of the file system that it is created on. In this example, we use a file system located on a Network File Server, however a repository can equally be created on alternate shared storage, such as an ISCSI LUN.

You can add a storage repository using the fileSystemCreateRepository method.

3.5.10.1 Java

The WsDevClient class in the sample code does not contain an example showing how to create a repository. The sample code expects that a repository has already been created within your environment and that its details have been provided in the `WsDevClient.properties` file.

Creating a storage repository is not complicated and the process can be extrapolated from the other examples or from the API documentation included with the SDK. In this example, we use the information that we have already learned to create a storage repository.

A storage repository is a child object of a FileSystem object. Therefore, it is necessary to obtain the ID of the FileSystem object where the storage repository must be created. The following loop can be used to populate the testfileSystemId with the ID of a file system named "nfs:/mnt/vol1/repo01":

```
...
fileserver = api.fileServerGetById(fsid);
final List<Id<FileSystem>> fileSystemIds = fileserver.getFileSystemIds();
for (final Id<FileSystem> fileSystemId : fileSystemIds)
{
   if (fileSystemId.getName().equals("nfs:/mnt/vol1/repo01")){
        testfileSystemId = fileSystemId;
   }
}
```

Use the fileSystemCreateRepository method provided by the OvmWsSoapClient class to create the repository for the file system identified by its file system ID:

```
// Create a repository
Repository myrepo = new Repository();
myrepo.setName("MyRepository");
final Job repositoryCreateJob = api.fileSystemCreateRepository(testfileSystemId,
    myrepo);
System.out.println("create repository job id: " + repositoryCreateJob.getId());
myrepoId = waitForJobComplete(api, repositoryCreateJob, Repository.class);
```

3.5.10.2 Python

In this example, we build on the previous example, reusing objects that we have already created. The repository must be attached to a particular file system. We loop through the file systems available on the Network File Server and extract the ID object for this file system:

```
for fsid in nfs.fileSystemIds:
    if fsid.name=="nfs:/mnt/vol1/repo01":
        repofs_id=fsid
```

Now create a repository object using the Suds Client Factory, and then call the fileSystemCreateRepository method:

```
repository=client.factory.create('repository')
repository.name="MyRepository"
job=client.service.fileSystemCreateRepository(repofs_id,repository)
repo_id=wait_for_job(client,job.id)
```

3.5.11 Presenting a Storage Repository

When you have created the storage repository, you must decide to which Oracle VM Servers you are going to present it.

Presentation of a repository to a server is done using the repositoryPresent method.

3.5.11.1 Java

In the WsDevClient class, there is an example showing how to present a repository to a server, if it has not already been presented. Using the Repository model, it is possible to use the getPresentedServerIds method to check where the repository has already been presented.

Presenting a repository to a server is an action performed against the repository. The following code is extracted from the WsDevClient class:

```
...
final Repository testRepository = api.repositoryGetById(testRepoId);
if (testRepository.getPresentedServerIds() == null ||
      !testRepository.getPresentedServerIds().contains(testServerId))
{
         repoPresentJob = api.repositoryPresent(testRepoId, testServerId);
         System.out.println("present repository job id: " + repoPresentJob.getId());
         waitForJobComplete(api, repoPresentJob);
}
```

3.5.11.2 Python

For this request, we must obtain the server ID object for the server to which we wish to present the repository. If you do not have an ID object for your server populated, refer to Obtaining the server ID.. Here, we assume that the server ID object is named sid:

```
job=client.service.repositoryPresent(repo_id,sid)
wait_for_job(client,job.id)
```

3.5.12 Creating Networks

It is possible to create different network types within Oracle VM Manager. In this example we show how to create a standard network object, as well as a local network limited to a single server.

To create a standard network for your environment, use the networkCreate method.

To create a local network for a specific server, use the serverCreateLocalNetwork method.

3.5.12.1 Java

Creating a Network

Network objects only require the network Name attribute to be set before they are created. The WsDevClient includes the following example code:

```
...
Network network = new Network();
network.setName(testNetworkName);
network.setDescription("Creating a test network named " + testNetworkName);

final Job networkCreateJob = api.networkCreate(network);
System.out.println("create network job id: " + networkCreateJob.getId());
testNetworkId = waitForJobComplete(api, networkCreateJob, Network.class);
```

Creating a Local Network for a Server

Local networks differ from other network types in that they are flagged differently and the API call must be made against the server object that they are attached to. The following example code exists within the WsDevClient class included in the SDK:

```
// Create a new server local network
Network serverLocalNetwork = new Network();
serverLocalNetwork.setName("MyTestServerLocalNetwork");
testServerId = testServer.getId();
serverLocalNetwork.setServerId(testServerId);

final Job localNetworkCreateJob = api.serverCreateNetwork(testServerId, network);
System.out.println("create server local network job id: " +
     networkCreateJob.getId());
testServerLocalNetworkId = waitForJobComplete(api, localNetworkCreateJob,
     Network.class);
```

The serverCreateNetwork expects the ID for the server that the local network is being created for.

3.5.12.2 Python

Creating a Network

The SOAP object expected by the API for a new network consists of the network name, and optionally a description. The following Python code illustrates this:

```
net=client.factory.create('network')
net.name="MyNetwork"
net.description="A network set up to test the SOAP API"
job=client.service.networkCreate(net)
net_id=wait_for_job(client,job.id)
```

Creating a Local Network for a Server

To create a local network for a specific server, you need the server ID. If you do not have the server ID available for use in your code already, refer to Obtaining the server ID. to obtain an ID object for a specific server. Here, we assume the server ID object is already populated:

```
lo_net=client.factory.create('network')
lo_net.name="MyLocalNetwork"
lo_net.description="A local network set up to test the SOAP API"
job=client.service.serverCreateNetwork(sid,lo_net)
lo_net_id=wait_for_job(client,job.id)
```

Notice that the serverCreateNetwork method defined in the API helps to distinguish this network type from other network types, and expects the server ID object required to flag it as a local network.

3.5.13 Creating Virtual Machines

In this section, we describe how to create a *virtual machine*. Your virtual machine configuration may vary and you may want to explore other aspects of the API to fit your own requirements. The examples given provide a basic guideline to getting started.

To create a virtual machine using the SOAP API, use the vmCreate method.

3.5.13.1 Java

The WsDevClient class contains an example of the creation of a virtual machine:

```
Vm testVm = new Vm();
testVm.setVmDomainType(VmDomainType.XEN_HVM);
testVm.setName(testVmName);
testVm.setRepositoryId(testRepoId);

final Job vmCreateJob = api.vmCreate(testVm, testPoolId);
System.out.println("create vm job id: " + vmCreateJob.getId());
testVmId = waitForJobComplete(api, vmCreateJob, Vm.class);
```

In this code, some basic attributes of the virtual machine are set before it is created. In fact, there are a large number of attributes specific to a virtual machine that can be set to control how the virtual machine is configured. Here the most basic attributes have been selected to create a virtual machine using Xen hardware virtualization with its configuration located on the repository defined in the `WsDevClient.properties` file.

The vmCreate method also requires the server pool ID to be provided so that the virtual machine is created within the correct server pool.

The WsDevClient class goes on to demonstrate many other actions that can be performed on a virtual machine via the SOAP API including removing and restoring server pool association, basic modifications of attributes, assigning VNICs and killing a virtual machine. It is recommended that the reader explore these examples and how they relate to methods within the OvmWsSoapClient class to understand how to expand any interactions with virtual machines through the SOAP API.

3.5.13.2 Python

In the following code, some basic attributes of the virtual machine are set before it is created. In fact, there are a large number of attributes specific to a virtual machine that can be set to control how the virtual machine is configured. Here the most basic attributes have been selected to create a virtual machine using Xen para-virtualization with its configuration located on the repository called 'MyRepository'. The virtual machine is also attached to the server pool called 'MyServerPool'.

In the following example, we assume that many of the ID objects required to create a virtual machine have already been set based on the code used in the previous examples. If you need to populate these objects beforehand, refer to the example provided in Obtaining the Server Pool ID Value.

```
vm=client.factory.create('vm')
vm.name="MySOAPVm"
vm.repositoryId=repo_id
vm.vmDomainType='XEN_PVM'
job=client.service.vmCreate(vm,svrpool_id)
vm_id=wait_for_job(client,job.id)
```

3.5.14 Importing Assemblies

The final example in this guide is designed to show how to import an *assembly* containing a configuration of one or more virtual machines along with their *virtual disks* and any inter-connectivity between them, to ease set up and creation of your virtual machines within Oracle VM Manager.

An assembly is imported into a repository and is performed by calling the repositoryImportAssembly method.

Once the assembly has been imported into the repository, you can import a virtual machine from within the assembly into your environment using the vmCreateFromAssemblyVm method.

3.5.14.1 Java

Importing the Assembly

The WsDevClient class contains the following code to handle the assembly import:

```
final Job importAssemblyJob = api.repositoryImportAssembly(testRepoId, assemblyUrl);
System.out.println("import assembly job id: " + importAssemblyJob.getId());
testAssemblyId = waitForJobComplete(api, importAssemblyJob, Assembly.class);
```

The assemblyUrl is defined in the `WsDevClient.properties` file. The URL must point to a valid assembly package that Oracle VM Manager can access and download.

Importing a Virtual Machine From An Assembly

After the assembly has been imported into Oracle VM Manager you may want to import a virtual machine from within the assembly into your environment. First, you need to obtain the assemblyVm ID for the virtual machine that you want to import. For this, the WsDevClient class includes the following code to loop over the Vm ID's within the assembly. The code is simple and selects the first ID that it detects in the loop.

```
assemblyVms = api.assemblyVmGetListById(assembly.getAssemblyVmIds());
Id<AssemblyVm> testAssemblyVmId = null;
for (final AssemblyVm assemblyVm : assemblyVms)
{
    if (testAssemblyVmId == null)
    {
        testAssemblyVmId = assemblyVm.getId();
    }
```

Once an assemblyVmId has been set, the vmCreateFromAssemblyVm method can be called to create the new virtual machine:

```
if (testAssemblyVmId != null)
{
    final Job importVmFromAssemblyJob = api.vmCreateFromAssemblyVm(testAssemblyVmId);
    System.out.println("import vm from assembly job id: " + importVmFromAssemblyJob.getId());
    importedAssemblyVmId = waitForJobComplete(api, importVmFromAssemblyJob, Vm.class);
```

```
}
```

3.5.14.2 Python

Importing the Assembly

Importing an assembly into Oracle VM Manager in Python using the SOAP API is straightforward. Obtain the ID for the repository where you wish to import the assembly and then call the repositoryImportAssembly method, providing it with the URL that Oracle VM Manager should use to download the assembly.

```
assembly_url='http://example.com/assemblys/my_assembly.ovf
job=client.service.repositoryImportAssembly(repo_id,assembly_url)
assembly_id=wait_for_job(client,job.id)
```

Importing a Virtual Machine From an Assembly

Once the assembly has completed its import, it is possible to query the API to obtain the ID values for virtual machines contained within the assembly. An assembly Vm ID is required to import it as a functional virtual machine within your environment. In this example we import all of the virtual machines in the assembly into our environment:

```
assembly=client.service.assemblyGetById(assembly_id)
if has_attr(assembly,'assemblyVmIds'):
    for vm in assembly.assemblyVmIds:
        job=client.service.vmCreateFromAssemblyVm(vm)
        wait_for_job(client,job.id)
```

Chapter 4 Additional Utilities Exposed in the WS-API

Table of Contents

Utility methods that are not directly related to core *Oracle VM Manager* behavior are exposed through a second endpoint available via the Web Services API. The Oracle VM Web Service Utilities API provides additional methods both for obtaining information within Oracle VM Manager and for configuring or triggering particular Oracle VM Manager functionality. Typical examples include determining which *Oracle VM Servers* are available to be added to a new *server pool*; obtaining a list of all file systems which are on *local storage*; setting statistics collection attributes used to configure how often statistics are gathered and for how long they are stored; and starting a backup process.

The Utilities API is available via both the SOAP and REST interfaces.

Java examples providing fully implemented SOAP and REST clients are included in the SDK. These are available in `com.oracle.ovm.mgr.ws.client`:

1. `OvmWsRestUtilitiesClient.java` contains the source code for the REST implementation of the Utilities Client
2. `OvmWsSoapUtilitiesClient.java` contains the source code for the SOAP implementation of the Utilities Client

4.1 SOAP

4.1.1 Authentication

To get a reference to this endpoint, use the `getOvmWsUtilitiesEndpoint()` method exposed through the *OvmWsApi* endpoint. If retrieved in this way, login credentials are handed off such that the client using this new endpoint can access the Utilities API without the need to re-authenticate.

The `getOvmWsUtilitiesEndpoint()` method returns an Endpoint Reference. Use the facilities provided by your client library to follow the reference.

Note

Due to difficulties redefining endpoints according to WS-addressing in the Python Suds library, Python users should consider taking advantage of the REST interface if there is a requirement to access the Utilities API.

The following example method, taken from OvmWsSoapClient, shows how to use the `getOvmWsUtilitiesEndpoint()` method to access the Utilities API using Java:

```
@Override
  public OvmWsUtilitiesClient getWsUtilities() throws WsException, MalformedURLException
  {
    final OvmUtilities_Service utilitiesService =
```

```
            new OvmUtilities_Service(new URL
             (protocol + "://" + hostname + ":" + port + "/ovm/core/wsapi/soap/utilities"));
    final OvmUtilities utilityApi =
            utilitiesService.getPort(api.getOvmWsUtilitiesEndpoint(), OvmUtilities.class);
    return new OvmWsUtilitiesSoapClient(utilityApi);
  }
```

4.1.2 Calling Utility Methods Using SOAP and Java

Once the new OvmWsUtilitiesSoapClient has been created, the methods exposed through the Utilities SOAP API can be used in the same fashion as those exposed by the core SOAP API. The Java example source code provided in `OvmWsUtilitiesSoapClient.java`, in the SDK, provides excellent examples of how to access the methods exposed through the Utilities SOAP API. The truncated sample program listing, presented below, shows how the `getAvailableEthernetPorts` method is set up in the OvmWsUtilitiesSoapClient.

```
...
public class OvmWsUtilitiesSoapClient implements OvmWsUtilitiesClient
{
    private final OvmUtilities api;

    public OvmWsUtilitiesSoapClient(final OvmUtilities utilityApi)
    {
        this.api = utilityApi;
    }

    @SuppressWarnings("unchecked")
    @Override
    public BusinessSelection<EthernetPort> getAvailableEthernetPorts(final Id<Server> serverId,
            final Id<Network> networkId)
        throws WsException
    {
        try
        {
            return api.getAvailableEthernetPorts(serverId, networkId);
        }
        catch (final WsException_Exception e)
        {
            throw convertException(e);
        }
    }
...
}
```

4.1.3 Certificate Management for Certificate-based Authentication Using SOAP

The Utilities API significantly includes a set of methods that allow you to manage certificate generation and registration within Oracle VM Manager. This is important as the WS-API also allows for certificate-based authentication, allowing you to further secure how custom-developed applications authenticate and interact with Oracle VM Manager. This section explores some of these methods briefly in the context of the SOAP API.

Certificate management within Oracle VM Manager is discussed in a variety of contexts throughout the documentation. For more information on authenticating using an SSL certificate using SOAP, please see Section 3.5.1, “Authenticating”. Please also refer to Setting up SSL on Oracle VM Manager in the *Oracle VM Administrator's Guide* for more information on SSL certificate management.

How to Obtain the CA Certificate Using SOAP

Once authenticated, either using an existing SSL certificate, or by using the API login method, it is possible to query the Utility API endpoint to obtain the internal Oracle VM Manager CA certificate. This is achieved using the certificateGetCaCertificate() method. This method simply returns the CA certificate as a string.

It is useful to obtain the CA certificate and to add it to your trusted certificates or to your keystore, so that it can be used to validate SSL interactions with Oracle VM Manager.

How to Sign and Register a Certificate Using SOAP

The API provides options to sign and register an SSL certificate using the internal Oracle VM Manager CA certificate. There are equally options to only sign a certificate, or to register an already signed certificate. This can be useful if you have added a trusted third-party CA certificate to Oracle VM Manager's own truststore, and wish to use a certificate issued by that third-party. These additional API methods are discussed in the API documentation. In this case, we assume that you need to sign and register a certificate with the internal CA.

The Utilities API provides a certificateSignAndRegister() method that can be used for this purpose. The method can either take a certificate that you have generated locally and sign and register it for use with Oracle VM Manager; or if called without any argument, can automatically generate the certificate with a passphraseless key before signing and registering it for use with Oracle VM Manager. For security reasons, it is usually more sensible to generate your certificate locally and set a passphrase for the key, before calling this method. If using Java, you can do this by setting up a keystore using the keytool:

```
$ keytool -genkey
```

Once you have set up a keystore and certificate, you can use the keytool to export your certificate in PEM format, so that you have it available for signing:

```
$ keytool -export -rfc
```

In your web services client, you must create a new LoginCertificate object and place your new certificate into the certificate field. For example:

```
LoginCertificate cert = new LoginCertificate();
cert.setCertificate("-----BEGIN CERTIFICATE-----" +
    "MIIDTzCCAw2gAwIBAgIEIIUUWjALBgcqhkjOOAQDBQAweTELMAkGA1UEBhMCVVMxCzAJBgNVBAgT" +
    "AkNBMRUwEwYDVQQHEwxSZWR3b29kIENpdHkxDzANBgNVBAoTBk9yYWNsZTEaMBgGA1UECxMRT3Jh" +
    "Y2xlIFZNIE1hbmFnZXIxGTAXBgNVBAMTEENlcnRpZmljYXRlIERlbW8wHhcNMTMwODIxMTYzOTUz" +
    "WhcNMTMxMTE5MTYzOTUzWjB5MQswCQYDVQQGEwJVUzELMAkGA1UECBMCQ0ExFTATBgNVBAcTDFJl" +
    "ZHdvb2QgQ2l0eTEPMA0GA1UEChMGT3JhY2xlMRowGAYDVQQLExFPcmFjbGUgVk0gTWFuYWdlcjEZ" +
    "MBcGA1UEAxMQQ2VydGlmaWNhdGUgRGVtbzCCAbgwggEsBgcqhkjOOAQBMIIBHwKBgQD9f1OBHXUS" +
    "KVLfSpwu7OTn9hG3UjzvRADDHj+AtlEmaUVdQCJR+1k9jVj6v8X1ujD2y5tVbNeBO4AdNG/yZmC3" +
    "a5lQpaSfn+gEexAiwk+7qdf+t8Yb+DtX58aophUPBPuD9tPFHsMCNVQTWhaRMvZ1864rYdcq7/Ii" +
    "Axmd0UgBxwIVAJdgUI8VIwvMspK5gqLrhAvwWBz1AoGBAPfhoIXWmz3ey7yrXDa4V7l5lK+7+jrq" +
    "gvlXTAs9B4JnUVlXjrrUWU/mcQcQgYC0SRZxI+hMKBYTt88JMozIpuE8FnqLVHyNKOCjrh4rs6Z1" +
    "k$ keytool -import -file newcertW6jfwv6ITVi8ftiegEkO8yk8b6oUZCJqIPf4VrlnwaSi" +
    "6bc9ozDyK1cgNyZWl4kq1efzjsyolIr1i4CiM/MqnEZO43hVVtXex3V+VWd9i/CLn0I/ZC9Lfi5X" +
    "HlQOEzWKK/esvf64Mv96DbZna/XRj6JhTEPGoStizNhXrVJCF4DaiIP+l53qYKJEtrNoR+tToRt8" +
    "OimE3PzLCXILvwwaCaMhMB8wHQYDVR0OBBYEFAtyjCpfkznpsUf2Lj8iBmRS3/0oMAsGByqGSM44" +
    "BAMFAAMvADAsAhRTm5NW8HDcM8jG5a7QIowNLN+fEQIUZXMogTvKbcXu6NN6fh0KY09hokI=" +
    "-----END CERTIFICATE-----");
```

Now you can invoke the method to sign and register your certificate, this returns the signed version of the certificate:

```
LoginCertificate signed = ovmUtil.certificateSignAndRegister(cert);
System.out.println(signed.getCertificate());
```

The output returned contains the newly signed certificate. You can dump this output to file, so that you can use it to import the newly signed certificate into your keystore:

```
$ keytool -import -file newcert
```

Note that you should also import the CA certificate into your keystore, so that your certificates can be validated. Be sure to use a different alias when importing the CA certificate into your keystore, to avoid overwriting your newly signed certificate and key.

4.2 REST

4.2.1 Authentication

Authentication for the REST interface to the Web Services Utilities API follows the same approach as that used to authenticate against the Oracle VM Manager REST API, see Section 2.2, “How do I Authenticate?” for more information. If your client is already authenticated, there should be no reason to perform authentication again in order to access the Utilities API.

The Utilities API can be accessed by appending "/Utilities" to the URI. Therefore, the base URI to access the utilities via the REST interface is:

```
https://hostname:port/ovm/core/wsapi/rest/Utilities
```

In this URI, replace *`hostname`* with the host name or IP address of Oracle VM Manager, and replace *`port`* with the number of the port where the REST web service is available - the default is 7002 (SSL).

4.2.2 Utilities Paths and Examples

In terms of the REST interface to the Web Services Utilities API, different methods exposed by the API are grouped together based on their functional relationships to Oracle VM Manager to create separate utilities. Each utility has its own relative path beyond the base URI that provides access to the methods that are available for that utility. The relative paths and a description of the utility is provided below:

- `/ArchiveManagement`: utility to control the event and job logs housekeeping schedule
- `/BackupManagement`: utility to access the Oracle VM Manager backup configuration and to provide the ability to start a backup operation on demand
- `/BusinessManagement`: utility that provides tools to extend the functionality provided by the core API
- `/Certificate`: utility to manage certificate registration for certificate-based authentication
- `/LogManagement`: utility to control runtime selection of which OVM software components (identified by the java package name) can write to the system log file and the level of detail of the messages written.
- `/MacManagement`: utility to control how MAC addresses are generated for VNICs
- `/StatisticsManagement`: utility to control the statistics configuration, including how often statistics are collected and how long they are stored for

The methods available for each of the utilities is documented fully in the documentation provided with the SDK. The reader is encouraged to refer to this documentation to *discover* the full range of functionality provided by the Web Services Utilities API.

The Java example source code provided in `OvmWsUtilitiesRestClient.java`, in the SDK, provides excellent examples of how to access the methods exposed through the REST API using the Jersey Bundle tools. In the highly truncated sample below, it is easy to see that the different utility paths defined above are set as string values that can be combined to construct the correct URI to access each utility:

```
...
public class OvmWsUtilitiesRestClient extends RestClient implements OvmWsUtilitiesClient
{
    private static final String UTILITIES_PATH             = "Utilities";
    private static final String BUSINESS_MANAGEMENT_PATH   = "BusinessManagement";
    private static final String STATISTICS_MANAGEMENT_PATH = "StatisticsManagement";
    private static final String STATISTICS_ATTRIBUTES_PATH = "StatisticsAttributes";
    private static final String MAC_MANAGEMENT_PATH        = "MacManagement";
```

```
    private static final String MAC_ADDRESS_RANGE_PATH     = "MacAddressRange";
    private static final String ARCHIVE_MANAGEMENT_PATH    = "ArchiveManagement";
    private static final String ARCHIVE_ATTRIBUTES_PATH    = "ArchiveAttributes";
    private static final String BACKUP_MANAGEMENT_PATH     = "BackupManagement";
    private static final String BACKUP_ATTRIBUTES_PATH     = "BackupAttributes";
    private static final String LOG_MANAGEMENT_PATH        = "LogManagement";
    private static final String LOG_LOGGERATTRIBUTES_PATH  = "LoggerAttributes";

    private final URI           businessManagementURI;
    private final URI           backupManagementURI;
    private final URI           logManagementURI;

    public OvmWsUtilitiesRestClient(final RestClient parentClient)
    {
        super(parentClient, UTILITIES_PATH);

        businessManagementURI =
           UriBuilder.fromUri(getBaseURI()).segment(BUSINESS_MANAGEMENT_PATH).build();
        backupManagementURI =
           UriBuilder.fromUri(getBaseURI()).segment(BACKUP_MANAGEMENT_PATH).build();
        logManagementURI = UriBuilder.fromUri(getBaseURI()).segment(LOG_MANAGEMENT_PATH).build();
    }

    @Override
    public BusinessSelection<EthernetPort> getAvailableEthernetPorts(final Id<Server>
          serverId, final Id<Network> networkId)
        throws WsException
    {
        try
        {
            final Map<String, Object> queryParameters =
                    createQueryParameterMap("serverId", serverId.getValue(), "networkId",
                            networkId.getValue());
            final Builder b =
                    getResourceFromUriAndPathElementsWithQueryParameters(businessManagementURI,
                              queryParameters, "availableEthernetPorts");

            @SuppressWarnings("unchecked")
            final BusinessSelection<EthernetPort> ethernetPorts = b.get(BusinessSelection.class);

            return ethernetPorts;
        }
        catch (final UniformInterfaceException ex)
        {
            throw convertException(ex);
        }
    }
...
}
```

In the code above, also included, is an example definition for the `getAvailableEthernetPorts` method. Here you can that since this class extends the RestClient class, it is able to use the `getResourceFromUriAndPathElementsWithQueryParameters` method in conjunction with the Jersey Builder to submit an HTTP GET request that returns the available Ethernet ports.

According to the documentation provided with the SDK, and extrapolating from the Java code, it should be easy to replicate this example in Python. The sample presented below provides a complete example of the Python code required to construct a similar query:

```
# import the requests library to handle HTTP requests and session maintenance
import requests
# import the json library for JSON translation (not required for this example)
import json

# instantiate a session object and populate it with authentication credentials
s=requests.Session()
```

```
s.auth=('user','password')
s.verify=False #disables SSL certificate verification
# configure the session to always use JSON
s.headers.update({'Accept': 'application/json', 'Content-Type': 'application/json'})

# set up a baseUri object to contain the URI to the Utilities API
baseUri='https://127.0.0.1:7002/ovm/core/wsapi/rest/Utilities'

# construct the URI according to the requirements set out in the documentation
uri='{base}/BusinessManagement/availableEthernetPorts'.format(
    base=baseUri)

# configure the query parameters
params={
    "serverId": "00:e0:81:4d:40:f5:00:e0:81:4d:40:be:00:e0:81:4d",
    "networkId": "0aac4c00"
}

# submit a get request to the uri and store the response
r=s.get(uri,params=params)
# use the requests library's native json parser to obtain a usable python object
availPorts=r.json()
```

4.2.3 Certificate Management for Certificate-based Authentication Using REST

The Utilities API significantly includes a set of methods that allow you to manage certificate generation and registration within Oracle VM Manager. This is important as the WS-API also allows for certificate-based authentication, allowing you to further secure how custom-developed applications authenticate and interact with Oracle VM Manager. This section explores some of these methods briefly in the context of the REST API.

Certificate management within Oracle VM Manager is discussed in a variety of contexts throughout the documentation. For more information on authenticating using an SSL certificate using REST, please see Section 2.6.1, “Authenticating”. Please also refer to Setting up SSL on Oracle VM Manager in the *Oracle VM Administrator's Guide* for more information on SSL certificate management.

How to Obtain the CA Certificate Using REST

Once authenticated, either using an existing SSL certificate, or using the HTTP BASIC authentication mechanism, it is possible to query the Utility API to obtain the internal Oracle VM Manager CA certificate. This is achieved by simply sending an HTTP GET request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/Utilities/Certificate/CaCertificate
```

This method simply returns the CA certificate as a string.

It is useful to obtain the CA certificate and to add it to your trusted certificates or to your keystore, so that it can be used to validate SSL interactions with Oracle VM Manager.

How to Sign and Register a Certificate Using REST

The API provides options to sign and register an SSL certificate using the internal Oracle VM Manager CA certificate. There are equally options to only sign a certificate, or to register an already signed certificate. This can be useful if you have added a trusted third-party CA certificate to Oracle VM Manager's own keystore, and wish to use a certificate issued by that third-party. These additional API methods are discussed in the API documentation. In this case, we assume that you need to sign and register a certificate with the internal CA.

The REST API allows you to sign and register a certificate by sending a POST request to the following URI:

```
https://hostname:port/ovm/core/wsapi/rest/Utilities/Certificate
```

The body of the POST request can contain either a JSON or XML representation of a loginCertificate object. Only the certificate element of the object need be populated. If no object is submitted within the POST request, the API automatically generates a certificate and passphraseless key that can be used for authentication. For security reasons, it is usually a good idea to generate a certificate locally and to set a passphrase for the key beforehand, so that the certificate can be passed in the body of the request.

Important

By default, the API only registers a certificate sent in the body of a POST request sent to this URI. This is to enable the possibility of using certificates already signed by a third party. To force the API to also sign a certificate submitted using this method, the boolean `sign` parameter must also be set to true within the URI. Therefore, in this case, you should post to the URI:

```
https://hostname:port/ovm/core/wsapi/rest/Utilities/Certificate?sign=True
```

In the SOAP version of this process, we focussed on how this could be performed using Java. In this REST-based example, we achieve the same thing using Python. The principles are much the same.

First create a key and certificate locally using a tool like OpenSSL. If you are using Java, you would probably do this by generating a keystore using the Java `keytool` command as presented in the SOAP version of this example.

```
$ openssl genrsa -des3 -out mykey.pem 2048
$ openssl req -new -key mykey.pem -out mycertreq.csr
$ openssl x509 -req -days 365 -in mycertreq.csr -signkey mykey.pem -out mycert.pem
```

For security reasons, it is advisable to set a passphrase for your key. Note that this is a typical self-signed certificate, so you first generate a Certificate Signing Request (CSR) and then use your key to sign the certificate that you intend to generate from this CSR.

Once you have created a certificate, you can use the REST API to sign it. In Python, the following code could be used, assuming that you have already set up a Requests session and authenticated as described in Section 2.6.1.2, “Python”:

```
cert=open('/path/to/mycert.pem').read()
body={'certificate': cert}
r=s.post('https://127.0.0.1:7002/ovm/core/wsapi/rest/Utilities/Certificate?sign=True',
          data=json.dumps(body))
signed_cert=r.json()['certificate']
f=open('/path/to/signed.pem','w')
f.write(signed_cert)
f.close()
```

In the above example, we read the contents of the self-signed certificate into a variable named 'cert' and use this to create a python data structure that we can convert to a JSON string similar to the content that the API expects in the body of the request. We submit the POST request and set the `sign` parameter to True in the URI. We extract the newly signed certificate from the body of the response and then write the content of this into a file at */path/to/signed.pem*.

To use this certificate to authenticate to either the REST or SOAP API using Python, it should be combined with its key. You can do this easily on the command line:

```
$ cat /path/to/signed.pem /path/to/mykey.pem >> /path/to/OVMSignedCertificate.pem
```

You can now use this new certificate to authenticate against either the SOAP or REST API from within any of your programs, as described in Section 2.6.1.2, “Python”.

Chapter 5 Programming Issues and Considerations

Table of Contents

5.1 General Programming Considerations

There are a number of considerations that programmers should take into account when developing against the Web Services API. This section provides a breakdown of common obstacles that users of the API encounter, and how they should be handled.

5.1.1 Object Modification

When you read an object into your client application, the object should be considered as a copy of the object actually contained within Oracle VM Manager. Changes to the client side representation of an object don't affect anything until you call an object modify method and pass it the updated copy of the object that you have modified locally within your client application. If you populate a local object with the data returned from a get object method, the local object is only a copy of the actual object. If you make changes to the local object, the server-side object remains unchanged. It is only when you call an object modify method that the object on the server-side is actually affected.

Since Oracle VM Manager can be accessed by multiple users or clients at any moment in time, it is always possible that two users are attempting to modify an object at the same time. The API handles race conditions, but if the object that you are modifying has already been modified by another operation and your object is out of date the API will generate an exception. Therefore, when performing an object modification, you should always ensure that your object is up to date before you call an object modify method.

This behavior prevents random changes being made that you didn't intend. For example, if you have an object that you change the description for, but the content of your object is old, someone else may have made all kinds of changes to that object. If an exception wasn't thrown, the object would revert to the previous state for all of those other attributes even though that wasn't the callers intention.

Object IDs can't be modified, even if there are methods that seem like they might achieve this for you. See Section 5.1.3, “Read-only Methods and Object Properties” for more information on why these methods exist. Object IDs are generated within Oracle VM Manager and are used to maintain consistency across object relationships and must remain unique. Therefore, it is not possible for a client application to directly modify an object ID.

5.1.2 Object Associations

Associations aren't modified by calling setters on the object and then calling modify. Associations are changed with explicit API calls. Examples include, serverPoolAddServer, networkAddEthernetPort, etc.

Associations are changed in one direction only - typically from the "parent". Using the above cases, you add servers to a server pool, you don't set the serverPoolId on a server, even though it appears as a property of the server object. See Section 5.1.3, “Read-only Methods and Object Properties” for more information on these properties and why they exist for any object.

As mentioned previously in Section 5.1.1, “Object Modification”, the IDs of objects that have an association with a "parent" object cannot be modified even if there are methods that seem like they might achieve this for you. These associated object IDs that appear as properties of a "parent" object are not modifiable directly and are automatically associated when an association method is called from the API.

5.1.3 Read-only Methods and Object Properties

There are methods documented as "read-only" on the model objects. For example, if you look at the API documentation for Server.setServerPoolId, it explicitly states that the serverPoolId is read only. If you attempt to call these methods or modify these properties for an object, the value that you attempt to set is simply ignored. No exception is thrown if you call one of these methods, or even if you attempt to modify the value of such a property using an object modify API call.

The reason that these methods exist is to allow particular web services frameworks (such as the Jackson and Jersey libraries) to de-serialize these objects. Therefore, these methods and properties need to be available on the client side, but they do not affect the manager's representation of the object.

5.1.4 Working with Jobs

Jobs can have child jobs. For an operation to have fully completed, the child jobs must be complete as well. However, there are certain operations in which the child jobs can take a very long time to complete or which never complete. This can happen when a child job is spawned to deal with each server. However, if the server is offline, that child job remains active until the server comes back online.

Jobs have properties indicating firstly whether the parent job has completed, and secondly whether all child jobs spawned by the parent job have also completed. The **done** property relates to the parent job, while the **summaryDone** property relates to the parent job as well as all spawned child jobs. In some cases, the parent job may appear to be complete if you check the **done** property, but the **summaryDone** property indicates that some child jobs are still running. Therefore, it is usually better practice to check the value of the **summaryDone** property when waiting for job completion.

To see the updates to an object after a job completes (including a modify), you must refresh the client's view of the object.

5.1.5 Dealing with Exceptions

Oracle VM Manager is a multi-layered application. As such, checks on the legitimacy of operations are performed at various levels within the Oracle VM Manager Core. Some errors may be caught within the Web Services layer itself, while others may only be triggered by the Core rule evaluations. This should not make any difference within your application itself. An exception should be treated as such, regardless of where it originates from.

For instance, the following exception returned as a Job error for a web services request, is as much of an exception as a rule exception returned by Core:

```
ovm.mgr.ws.model.WsException: NETWORK_000008: Cannot perform operation Modify Role
List on a Server local Network: Modify Role List
```

Equally, a rule exception from within the core, is as important as an error generated at the Web Services layer:

```
com.oracle.ovm.mgr.api.exception.RuleException: OVMRU_002043E Cannot release
ownership of Repository: MyRepo. Virtual Disks/CDroms:
[0004fb0000120000eb8cd3defdc71ba5.img on 0004fb0000060000e3ea97cd6b8d45bd,
0004fb0000120000f2ba448c46f4fc12.img on 0004fb0000060000e3ea97cd6b8d45bd],
are still assigned to VMs/Templates that have configuration files in another
repository.
```

It is also worth noting that exceptions that are returned from the API may vary in terms of the error message or exception code that is returned depending on whether you are using the SOAP API or the REST API. If you are coding an application that can use either API, you need to ensure that your code is consistent in handling these exceptions regardless of the API used.

5.2 Notable Issues for Suds Users

5.2.1 Dealing with an Externally Hosted XSD

The SOAP API references an externally hosted XSD:

```
<xs:import namespace="http://www.w3.org/2005/08/addressing"
schemaLocation="http://www.w3.org/2006/03/addressing/ws-addr.xsd"/>
```

In environments where your client application is unable to directly access the Internet, this could pose a problem for the Suds library. In this case, you should host a localized copy of this XSD and bind the new schema location for the namespace, as described at https://fedorahosted.org/suds/wiki/Documentation#BindingSchemaLocationsURLtoNamespaces. An example function follows to show how to bind the namespace to an XSD hosted locally in a temporary directory:

```
from suds.xsd.sxbasic import Import

def bind_schema_locations():
    Import.bind(
        'http://www.w3.org/2005/08/addressing',
        'file:///tmp/xsd/www.w3.org/2006/03/addressing/ws-addr.xsd')
```

5.2.2 Null Properties and Empty Lists

When an instance has a null property or an empty list, the instance in Suds removes the property. For example, when instance has no name:

```
instance.name = None
```

the Suds instance removes the name property. Therefore, in order to check the name property, you must first check that the instance actually contains the property:

```
if 'name' in instance and instance.name == 'foo':
      # do something
```

Equally, empty lists result in a similar behavior, where Suds removes the property. In order to iterate on a property that contains a list, you must first check that the property exists:

```
if 'ethernetPortIds' in server:
      for ethernet_port_id in server.ethernetPortIds:
      # do something
```

Attempting to iterate through a property containing an empty list results in an exception, as the property is removed from the instance. As a result, it is not a good idea to simply attempt to iterate on a property expecting it to be available:

```
for ethernet_port_id in server.ethernetPortIds:
      # do something
```

In the example above, if the `ethernetPortIds` property is an empty list, Suds does not attach it as a property of the `server` instance. The result is an exception similar to the following:

```
AttributeError: server instance has no attribute ‘ethernetPortIds’ exception.
```

5.2.3 Unable to Access the OvmWsUtilities Endpoint

While it is possible to load the WSDL for the *OvmWsUtilities* endpoint in Suds, it is not possible to authenticate for this endpoint. This is because authentication is handled using the login method exposed by the *OvmWsApi* endpoint.

Technically, access to the Utilities endpoint is designed to be achieved by calling the `getOvmWsUtilitiesEndpoint` method from the *OvmWsApi* endpoint, however this method returns an Endpoint Reference which the client is meant to follow. The Suds library does not appear to easily cater for this functionality.

Therefore, in the case that you decide to use the methods exposed by the *OvmWsUtilities* endpoint, it is recommended that you either resort to using the REST interface to the API, or that you consider using the `OvmWsh` to program the functionality that you desire.

5.3 Notable Issues for Jackson and Jersey Library Users

5.3.1 Null Properties and Empty Lists

When empty lists are serialized or deserialized through web services, they can be converted into nulls by the Jackson and Jersey libraries. Therefore, your code must be able to handle getting back a null property instead of an empty list. Also, an empty list passed into the web services api can be converted into a null. Therefore, the api treats them as equivalent.

Examples illustrating checks for null properties are included in the WsDevClient class in the sample client provided with the SDK.