Table of Contents

Java OpenNebula Cloud API 3.2

This page contains the OpenNebula Cloud API Specification for Java. It has been designed as a wrapper for the XML-RPC methods, with some basic helpers. This means that you should be familiar with the XML-RPC API and the XML formats returned by the OpenNebula core. As stated in the XML-RPC documentation, you can download the XML Schemas (XSD) here.

Javadoc

You can consult the javadoc online. If you prefer a local copy, read the Compilation and Installation section below.

Code Sample

This is a small code snippet. As you can see, the code flow would be as follows:

- Create a `org.opennebula.client.Client` object, setting up the authorization string

and the endpoint. You only need to create it once.
- Create a pool (e.g. `HostPool`) or element (e.g. `VirtualNetwork`) object.
- You can perform "actions" over these objects right away, like `myVNet.delete();`
- If you want to query any information (like what objects the pool contains, or one of the element attributes), you have to issue an `info()` call before, so the object retrieves the data from OpenNebula.

For more complete examples, please check the `src/oca/java/share/examples` directory included. You may be also interested in the java files included in `src/oca/java/test`.

```
// First of all, a Client object has to be created.
// Here the client will try to connect to OpenNebula using the default
// options: the auth. file will be assumed to be at $ONE_AUTH, and the
// endpoint will be set to the environment variable $ONE_XMLRPC.
Client oneClient;

try
{
    oneClient = new Client();

    // This VM template is a valid one, but it will probably fail to run
    // if we try to deploy it; the path for the image is unlikely to
    // exist.
    String vmTemplate =
          "NAME     = vm_from_java    CPU = 0.1    MEMORY = 64\n"
        + "DISK     = [\n"
        + "\tsource   = \"/home/user/vmachines/ttylinux/ttylinux.img\",\n"
        + "\ttarget   = \"hda\",\n"
        + "\treadonly = \"no\" ]\n"
        + "FEATURES = [ acpi=\"no\" ]";

    System.out.print("Trying to allocate the virtual machine... ");
    OneResponse rc = VirtualMachine.allocate(oneClient, vmTemplate);

    if( rc.isError() )
    {
        System.out.println( "failed!");
        throw new Exception( rc.getErrorMessage() );
    }

    // The response message is the new VM's ID
    int newVMID = Integer.parseInt(rc.getMessage());
    System.out.println("ok, ID " + newVMID + ".");

    // We can create a representation for the new VM, using the returned
    // VM-ID
    VirtualMachine vm = new VirtualMachine(newVMID, oneClient);

    // Let's hold the VM, so the scheduler won't try to deploy it
    System.out.print("Trying to hold the new VM... ");
    rc = vm.hold();

    if(rc.isError())
    {
```

```
            System.out.println("failed!");
            throw new Exception( rc.getErrorMessage() );
        }

        // And now we can request its information.
        rc = vm.info();

        if(rc.isError())
            throw new Exception( rc.getErrorMessage() );

        System.out.println();
        System.out.println(
                "This is the information OpenNebula stores for the new VM:");
        System.out.println(rc.getMessage() + "\n");

        // This VirtualMachine object has some helpers, so we can access its
        // attributes easily (remember to load the data first using the info
        // method).
        System.out.println("The new VM " +
                vm.getName() + " has status: " + vm.status());

        // And we can also use xpath expressions
        System.out.println("The path of the disk is");
        System.out.println( "\t" + vm.xpath("template/disk/source") );

        // We have also some useful helpers for the actions you can perform
        // on a virtual machine, like cancel or finalize:

        rc = vm.finalizeVM();
        System.out.println("\nTrying to finalize (delete) the VM " +
                           vm.getId() + "...");

    }
    catch (Exception e)
    {
        System.out.println(e.getMessage());
    }
```

Usage

To use the OpenNebula Cloud API for Java in your Java project, you have to add to the classpath the org.opennebula.client.jar file and the xml-rpc libraries located in the lib directory.

Compilation and Installation

The Java OCA is part of the OpenNebula core distribution, but it is not installed by default. You will find the packaged code in a tar.gz file following this link: http://dev.opennebula.org/packages.

To compile the Java OCA, untar the source, `cd` to the java directory and use the build

script:

```
$ cd src/oca/java
$ ./build.sh -d
Compiling java files into class files...
Packaging class files in a jar...
Generating javadocs...
```

This command will compile and package the code in `jar/org.opennebula.client.jar`, and the javadoc will be created in `share/doc/`.

You might want to copy the .jar files to a more convenient directory. You could use /usr/lib/one/java/

```
$ sudo mkdir /usr/lib/one/java/
$ sudo cp jar/* lib/* /usr/lib/one/java/
```