Orchestration Developer Reference

Cloud Manager 2.1.1

June 20, 2012

Legal Notice

Contents

About This Guide

This *Orchestration Developer Guide and Reference* is a component of the documentation library for NetIQ Cloud manager. While Orchestration components provides the broad framework and networking tools to manage complex virtual machines and high performance computing resources in a datacenter, this guide explains how to develop grid application jobs and polices that form the basis of Cloud Manager Orchestration functionality. This guide provides developer information to create and run custom Orchestration jobs. It also helps provides the basis to build, debug, and maintain policies using Cloud Manager Orchestration.

This guide contains the following sections:

Audience

The developer has control of a self-contained development system where he or she creates jobs and policies and tests them in a laboratory environment. When the jobs are tested and proven to function as intended, the developer delivers them to the NetIQ Cloud Manager Orchestration administrator.

Prerequisite Skills

As data center managers or IT or operations administrators, it is assumed that users of the product have the following background:

- General understanding of network operating environments and systems architecture.
- Knowledge of basic Linux* shell commands and text editors.

Feedback

We want to hear your comments and suggestions about this manual and the other documentation included with this product. Please use the User Comments feature at the bottom of each page of the online documentation, or go to www.novell.com/documentation/feedback.html (http://www.novell.com/documentation/feedback.html) and enter your comments there.

Additional Documentation

1 Getting Started With Development

This *Developer Guide* for the NetIQ Cloud Manager Orchestration Server is intended for individuals acting as Orchestration job developers. This document discusses the tools and technology required to create discrete programming scripts—called "jobs"—that control nearly every aspect of the NetIQ Cloud Manager product. The guide also explains how to create, debug, and maintain policies that can be associated with jobs running on the Cloud Manager Orchestration Server.

As a job developer, you need your own self-contained, standalone system with full access to your network environment. As a job developer, you might eventually assume all system roles: job creator, job deployer, system administrator, tester, etc. For more information about jobs, see

This section includes the following information:

1.1 What You Should Know

This section includes the following information:

1.1.1 Prerequisite Knowledge

This guide assumes you have the following background:

- Sound understanding of networks, operating environments, and system architectures.
- Familiarity with the Python development language. For more information, see the following online references:
  - **Python Development Environment (PyDEV):** The PyDEV plug-in (http://www.pydev.org/) enables developers to use Eclipse* for Python and Jython development. The plug-in makes Eclipse a more robust Python IDE and comes with tools for code completion, syntax highlighting, syntax analysis, refactoring, debugging, etc.
  - **Python Reference Manual:** This reference (http://python.org/doc/2.1/ref/ref.html) describes the exact syntax and semantics but does not describe the Python Library Reference, (http://python.org/doc/2.1/lib/lib.html) which is distributed with the language and assists in development.
  - **Python Tutorial:** This online tutorial (http://python.org/doc/2.1/ref/ref.html)helps developers get started with Python.

The Orchestration Job Development Language (JDL) incorporates compact Python scripts to create job definitions to manage nearly every aspect of the Orchestration grid. For more information, see Appendix B, "Cloud Manager Orchestration Job Classes and JDL Syntax," on page 223.

- Knowledge of basic UNIX shell commands or the Windows command prompt, and text editors.
- An understanding of parallel computing and how applications are run on Orchestration infrastructure.
- Familiarity with the Orchestration Console layout and use, as explained in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*.
- Familiarity with basic administrative tasks, as explained in the *NetIQ Cloud Manager 2.1.1 Orchestration Administrator Reference* and in the "The zosadmin Command Line Tool" section of the *NetIQ Cloud Manager 2.1.1 Orchestration Server Command Line Reference*
- Familiarity with on-line Orchestration API Javadoc as you build custom client applications. For more information see Appendix A, "The Cloud Manager Orchestration Client SDK," on page 203.
- Assumption of both Cloud Manager OrchestrationOrchestration Server administrative and end-user roles while testing and debugging jobs.

1.1.2 Setting Up Your Development Environment

To set up a development environment for creating, deploying, and testing jobs, we recommend the following procedure:

1. Initially set up a simple, easy-to-manage server, agent, and client on a single machine. Even on a single machine, you can simulate multiple servers by starting extra agents (see "Configuring the Orchestration Agent" in the *NetIQ Cloud Manager 2.1.1 Orchestration Installation Guide*.
2. As you get closer to a production environment, your setup might evolve to handle more complex system demands, such as any of the following:
   - An Orchestration Server instance deployed on one computer.
   - An Orchestration Agent installed on every managed server.
   - An Orchestration Console installed on your desktop machine.

     From your desktop machine, you can build jobs/policies, and then remotely deploy them using zosadmin command linetool. You can then remotely modify the jobs and other grid object through the Orchestration Console.
3. Use a version control system, such as Subversion*, to organize and track development changes.
4. Put the job version number inside the deployed file. This will help you keep your job versions organized.
5. Create make or Ant scripts for bundling and deploying your jobs.

By leveraging the flexibility of the Orchestration environment, you should not have to write jobs targeted specifically for one hypervisor technology (Xen, VMware, etc.).

1.2 Prerequisites for the Development Environment

- Install the Java* Development Kit (https://sdlc3d.sun.com/ECom/EComActionServlet;jsessionid=DCA955A842E56492B469230CC680B2E1), version 1.6 or later, to create jobs and to compile a Java SDK client in the Orchestration environment. The Cloud Manager installer ships with a Java Runtime Environment (JRE) suitable for running Orchestration jobs.
- **Components to write Python-based Job Description Language (JDL) scripts:**
  - Eclipse version 3.2.1 or later. (http://www.eclipse.org/).
- **Development Environment:** Set up your environment according to the guidelines outlined in "Cloud Manager System Requirements" in the *NetIQ Cloud Manager 2.1.1 Installation Planning Guide*. In general, the installed Orchestration Server requires 2 (minimum for 100 or fewer managed resources) to 4 gigabytes (recommended for more than 100 managed resources) of RAM.
- **Network Capabilities:** For Virtual Machine Management, you need a high-speed Gigabit Ethernet. For more information about network requirements, see "Requirements for Machines Designated as VM Hosts" in the *NetIQ Cloud Manager 2.1.1 Installation Planning Guide*.
- **Initial Configuration:** After you install and configure Cloud Manager Orchestration components, start in the agent and user auto registration mode as described in "Automatically Registering a Resource" in the *NetIQ Cloud Manager 2.1.1 Orchestration Installation Guide*. As a first-time connection, the server creates an account for you as you set up a self-contained system.

IMPORTANT: Because auto registration mode does not provide high security, make sure you prevent unauthorized access to your network from your work station during development. As you migrate to a production environment, make sure that this mode is deactivated.

---

2 Job Development Concepts

This section provides advanced conceptual information to help you create your own NetIQ Cloud Manager Orchestration jobs:

2.1 Contents of a Job Package

A job package might consist of the following elements:

- Job description language (JDL) code (the Python-based script containing the bits to control jobs).
- One or more policy files, which apply constraints and other job facts to control jobs.
- Any other associated executables or data files that the job requires. For example, the `cracker.jdl` sample job includes a set of Java code that discovers the user password in every configured agent before the Java class is run.

  Many discovery jobs that measure performance of Web servers or monitor any other applications can also include resource discovery utilities that enable resource discovery.

Section 3.2, “JDL Package,” on page 34 provides more information about job elements.

2.2 JDL Job Scripts

This section contains the following information:

2.2.1 What is JDL?

The Orchestration job definition language (JDL) is an extended and embedded implementation of Python. The Orchestration system provides additional constructs to control and access the following:

- Interaction with the infrastructure under management (requesting resources, querying load, etc.)
- Distributed variable space with job, user and system-wide scoping
- Extensible event callbacks mechanism
- Job logging
- Datagrid for efficient movement of files across the infrastructure.
- Automatic distribution of parallel operations
- Failover logic

For more information about the Orchestration JDL script editor, see Section 3.2, "JDL Package," on page 34.

The JDL language allows for the scripted construction of logic that can be modified by external parameters and constraints (through one or more associated policies) at the time the job instance is executed. Development of a job with the JDL (Python) language is very straightforward. For a listing of the job, joblet, and utility classes, see Appendix B, "Cloud Manager Orchestration Job Classes and JDL Syntax," on page 223.

A simple "hello world" Python script example that runs a given number of times (*numTests*) in parallel (subject to resource availability and policy) is shown below:

```
class exampleJob(Job):
    def job_started_event(self):
        print 'Hello world started: got job_started_event'
        # Launch the joblets
        numJoblets = self.getFact("jobargs.numTests")
        pspace = ParameterSpace()
        i = 1
        while i <= numJoblets:
            pspace.appendRow({'name':'test'+str(i)})
            i += 1
        self.schedule(exampleJoblet, pspace, {})

class exampleJoblet(Joblet):
    def joblet_started_event(self):
        print "Hello from resource%s" % self.getFact("resource.id")
```

This example script contains two sections:

- The class that extends the job and runs on the server.
- The class that extends the joblet that will run on any resource employed by this job.

Because the resources are not requested explicitly, they are allocated based on the resource constraints associated with this job. If none are specified, all resources match. The `exampleJoblet` class would typically execute some process or test based on unique parameters.

2.2.2 Using Facts in Job Scripts

This section contains the following information:

Fact Values

Facts can be retrieved, compared against, and written to (if not read-only) from within jobs. Every Grid object has a set of accessor and setter JDL functions. For example, to retrieve the `cryptpw` job argument fact in the job example listed in Section 2.3.2, "Job Arguments and Parameter Lists in Policies," on page 19, you would write the following JDL code:

```
1  def job_started_event(self):
2      pw = self.getFact("jobargs.cryptpw")
```

In line 2, the function `getFact()` retrieves the value of the job argument fact. `getFact()` is invoked on the job instance Grid object.

The following set of JDL Grid object functions retrieve facts:

```
getFact()
factExists()
getFactLastModified()
getFactNames()
```

The following set of JDL Grid object functions modify fact values (if they are not read-only) and remove facts (if they are not deleteable):

```
setFact
setDateFact
setTimeFact
setArrayFact
setBooleanArrayFact
setDateArrayFact
setIntegerArrayFact
setTimeArrayFact
setStringArrayFact
deleteFact
```

For more complete information on these fact values, see GridObjectInfo (page 249).

Fact Operations in the Joblet Class

Each joblet is also a Grid object with its own set of well known facts. These facts are listed in Section B.2, "Joblet Class," on page 223. An instance of the Joblet class runs on the resource. The joblet instance on the resource has access to the fact set of the resource where it is running. The resource fact set has no meaning outside of this execution context, because the Joblet can be scheduled to run on any of the resources that match the resource and allocation constraints.

For example, using the `cracker` job example shown in "Job Arguments and Parameter Lists in Policies" on page 19, you would write the following JDL code to retrieve the `cryptpw` job argument fact, the OS family fact for the resource, the Job instance ID fact, and the joblet number:

```
1 class CrackerJoblet(Joblet):
2    def joblet_started_event(self):
3        pw = self.getFact("jobargs.cryptpw")
4        osfamily = self.getFact("resource.os.family")
5        jobid = self.getFact("jobinstance.id")
6        jobletnum = self.getFact("joblet.number")
```

In line 3, the function `getFact()` retrieves the value of the job argument fact. `getFact()` is invoked on the joblet instance grid object. In line 4, the `resource.os.family` fact is retrieved for the resource where the Joblet is being executed. This varies, depending on which resource the Joblet is scheduled to run on. In line 5, the ID fact for the job instance is retrieved. This changes for every job instance. In line 6, the joblet index number for this joblet instance is returned. The index is 0 based.

Using the Policy Debugger to View Facts

The Policy Debugger page of the Orchestration Console provides a table view of all facts in a running or completed job instance. This view includes the Job instance facts (`jobinstance.*` namespace) and the facts from the job context. After you select the Policy Debugger tab in the Job Monitor view, the right side panel displays this fact table. For more details, see "The Policy Debugger" in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*.

2.3 Policies

Policies are XML-based files that aggregate the resource facts and constraints that are used to control resources. This section includes the following information about policies:

- Section 2.3.1, "Policy Types," on page 18
- Section 2.3.2, "Job Arguments and Parameter Lists in Policies," on page 19
- Section 2.3.3, "The Role of Policy Constraints in Job Operation," on page 21

For information about facts, see Chapter 4, "Understanding Grid Object Facts, Computed Facts, and Custom Facts," on page 51.

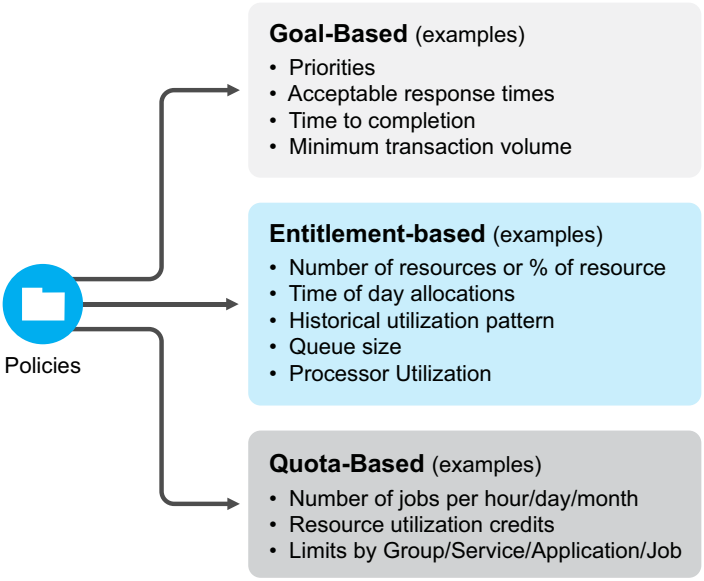
2.3.1 Policy Types

Policies are used to enforce quotas, job queuing, resource restrictions, permissions, etc. They can be associated with various grid objects (jobs, users, resources, etc.). The policy example below shows a constraint that limits the number of running jobs to a defined value, while exempting certain users from this limit. Jobs started that exceed the limit are queued until the running jobs count decreases and the constraint passes:

```
<policy>
    <constraint type="start" reason="too busy">
        <or>
            <lt fact="job.instances.active" value="5" />
            <eq fact="user.name" value="canary" />
        </or>
    </constraint>
</policy>
```

Policies can be based on goals, entitlements, quotas, and other factors, all of which are controlled by jobs.

Figure 2-1 *Policy Types and Examples*



2.3.2 Job Arguments and Parameter Lists in Policies

Part of a job's static definition might include job arguments. A job argument defines what values can be passed in when a job is invoked. This allows the developer to statically define and control how a job behaves, while the administrator can modify policy values.

You define job arguments in an XML policy file, which is typically given the same base name as the job. The example job `cracker.jdl`, for example, has an associated policy file named `cracker.policy`. The `cracker.policy` file contains entries for the <*jobargs*> namespace, as shown in the following partial example from `cracker.policy`.

```
<jobargs>
     <fact name="cryptpw"
      type="String"
      description="Password of abc"
      value="4B3lzcNG/Yx7E"
      />
   <fact name="joblets"
      type="Integer"
      description="joblets to run"
      value="100"
      />
</jobargs>
```

The above policy defines two facts in the `jobargs` namespace for the cracker job. One is a String fact named `cryptpw` with a default value. The second `jobargs` fact is an integer named `joblets`. Both of these facts have default values, so they do not require being set on job invocation. If the default value was omitted, then job would require that the two facts be set on job invocation. The job will not start unless all required job argument facts are supplied at job invocation. The default values of job argument facts can be overridden at job invocation. Job arguments are passed to a job when the job is invoked. This is done in one of the following ways:

- From the zos `run` command, as shown in the following example:

```
>zos run cracker cryptpw="dkslsl"
```

- From within a JDL job script when invoking a child job, as shown in the following job JDL fragment:

  ```
  self.runjob("cracker", { "cryptpw" : "asdfa" } )
  ```

- From the Job Scheduler, either with the Orchestration Console or by a `.sched` file.

When you deploy a job, you can include an XML policy file that defines constraints and facts. Because every job is a Grid object with its own associated set of facts (`job.id`, etc.), it already has a set of predefined facts, so jobs can also be controlled by changing job arguments at run time.

As a job writer, you define the set of job arguments in the `jobargs` fact space. Your goal in writing a job is to define the specific elements a job user is permitted to change. These job argument facts are defined in the job's XML policy for every given job.

The job argument fact values can be passed to a job with any of the following methods used for running a job:.

- as command-line arguments to the `zos run` command
- from the Job Arguments tab of the Job Scheduler in the Development Client
- from the Server Portal
- through the `runJob()` method of the JDL Job class

Consequently, the Orchestration Server `run` command passes in the job arguments. Similarly, for the Job Scheduler, you can define which job arguments you want to schedule or run a job. You can also specify job arguments when you use the Server Portal.

For example, in the following `quickie.job` example the number of joblets allowed to run and the amount of sleep time between running joblets are set by the arguments `numJoblets` and `sleeptime` as defined in the policy file for the job. If no job arguments are defined, the client cannot affect the job:

```
...
          # Launch the joblets
          numJoblets = self.getFact("jobargs.numJoblets")
          print 'Launching ', numJoblets, ' joblets'

          self.schedule(quickieJoblet, numJoblets)

class quickieJoblet(Joblet):

     def joblet_started_event(self):
          sleeptime = self.getFact("jobargs.sleeptime")
          time.sleep(sleeptime)
```

To view the complete example, see quickie.job (page 142).

As noted, when running a job, you can pass in a policy to control job behavior. Policy files define additional constraints to the job, such as how a resource might be selected or how the job runs. The policy file is an XML file defined with the `.policy` extension.

For example, as shown below, you can pass in a policy for the job named `quickie`, with an additional constraint to limit the chosen resources to those with a Linux OS. Suppose a policy file name `linux.policy` in the directory named `/mypolicies` with this content:

```
<constraint type="resource">
    <eq fact="resource.os.family" value="linux" />
</constraint>
```

To start the `quickie` job using the additional policy, you would enter the following command:

```
>zos run quickie --policyfile=/mypolicies/linux.policy
```

2.3.3 The Role of Policy Constraints in Job Operation

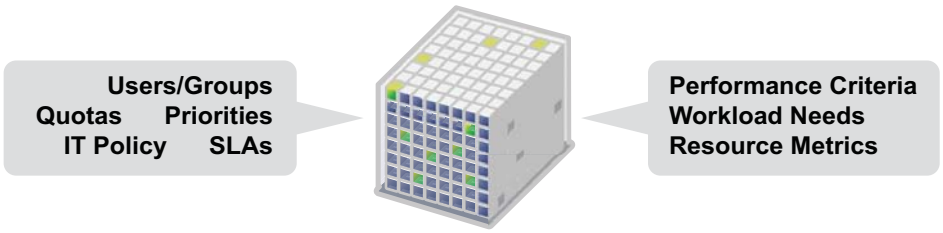
This section includes the following information:

- "How Constraints Are Used" on page 21
- "Constraint Types" on page 22
- "Scheduling with Constraints" on page 26
- "Constraints Constructed in JDL" on page 27

How Constraints Are Used

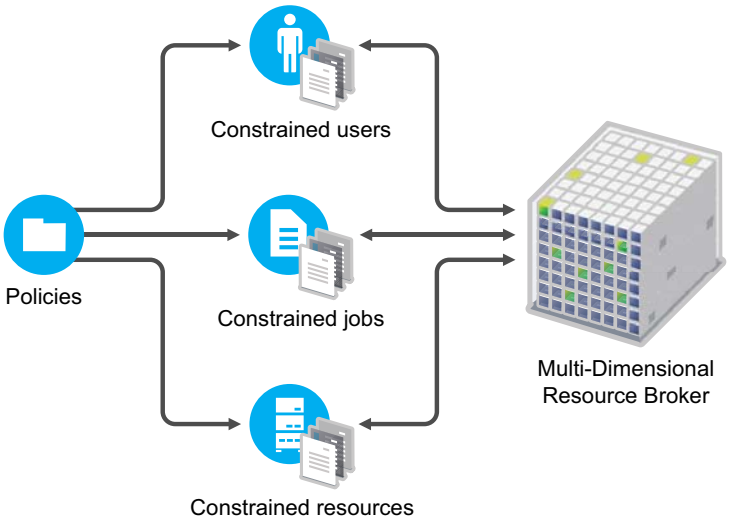
NetIQ Cloud Manager Orchestration lets you create jobs that meet the infrastructure scheduling and resource management requirements of your data center, as illustrated in the following figure.

Figure 2-2 *Multi-Dimensional Resource Scheduling Broker*



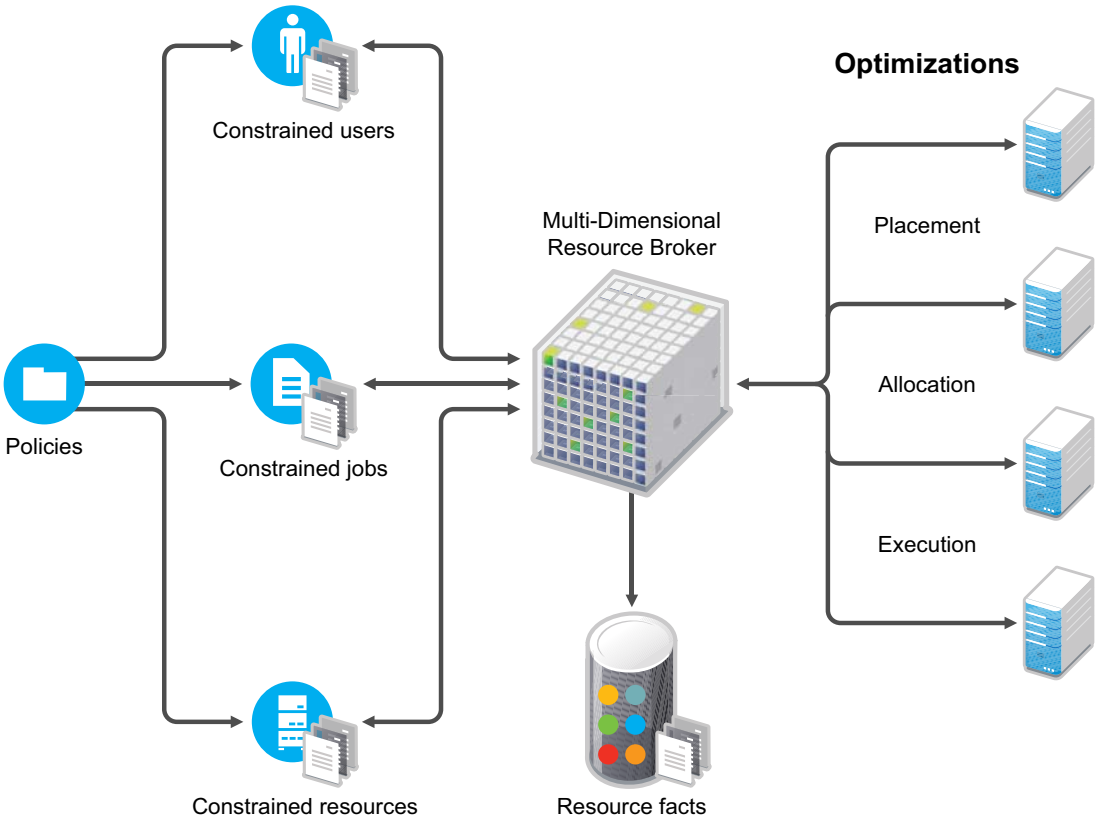
There are many combinations of constraints and scheduling demands on the system that can be managed by the highly flexible Orchestration Resource Broker. As shown in the figure below, many object types are managed by the Resource Broker. Resource objects are discovered (see Section 2.4, "Resource Discovery," on page 28). Other object types such as users and jobs can also be managed. All of these object types have "facts" that define their specific attributes and operational characteristics. Orchestration compares these facts to requirements set by the administrator for a specific data center task. These comparisons are called "constraints."

Figure 2-3 *Policy-Based Resource Management Relying on Various Constraints*



A policy is an XML file that specifies (among other things) constraints and fact values. Policies govern how jobs are dynamically scheduled based on various job constraints. These job constraints are represented in the following figure.

Figure 2-4 *Policy-Based Job Management*



The Resource Broker allocates or "dynamically schedules" resources based on the runtime requirements of a job (for example, the necessary CPU type, OS type, and so on) rather than allocating a specific machine in the data center to run a job. These requirements are defined inside a job policy and are called "resource constraints." In simpler terms, in order to run a given job, the Resource Broker looks at the resource constraints defined in a job and then allocates an available resource that can satisfy those constraints.

Constraint Types

The constraint element of a policy can define the selection and allocation of Grid objects (such as resources) in a job. The required type attribute defines the selection type.

The following list explains how constraint types are applied in a job's life cycle through policies:

- **accept:** A job-related constraint used to prevent work from starting; enforces a hard quota on the jobs. If the constraint is violated, the job fails.
- **start:** A job-related constraint used to queue up work requests; limits the quantity of jobs or the load on a resource. If the constraint is violated, the job stays queued.
- **continue:** A job-related constraint used to cancel jobs; provides special timeout or overrun conditions. If the constraint is violated, the job is canceled.

- **provision:** A joblet-related constraint (for resource selection) used to control automatic provisioning.

  Provision constraints are used by the Orchestration Broker as it evaluates VMs or VM templates tthat could be automatically provisioned to satisfy a scheduled joblet. By default, a job's `job.provision.maxcount` fact is set to 0, which means no automatic provisioning. If this value is greater than 0 and a joblet cannot be allocated to a physical resource, the provision constraints are evaluated to find a suitable VM or a VM template to provision that also satisfies the allocation and resource constraints.
- **allocation:** A joblet-related constraint (for resource selection) used to put jobs in a waiting state when the constraint is violated.
- **resource:** A joblet-related constraint (for resource selection) used to select specific resources. The joblet is put in a waiting state if the constraint is violated.
- **vmhost:** A VM-related constraint used to define a suitable VM host and repository for VM provisioning.
- **repository:** A VM-related constraint used to define a suitable repository for the storage of a VM.

  It is possible to create or edit a policy that constrains a repository during VM provisioning; however, a `vmhost` constraint type must be used, rather than a `repository` constraint type.

  For example:

  ```
  <constraint type="vmhost">
      <eq fact="repository.id" value="XXXX"/>
  </constraint>
  ```
- **authorize:** A VM-related constraint evaluated before a `vmhost` or `repository` constraint.

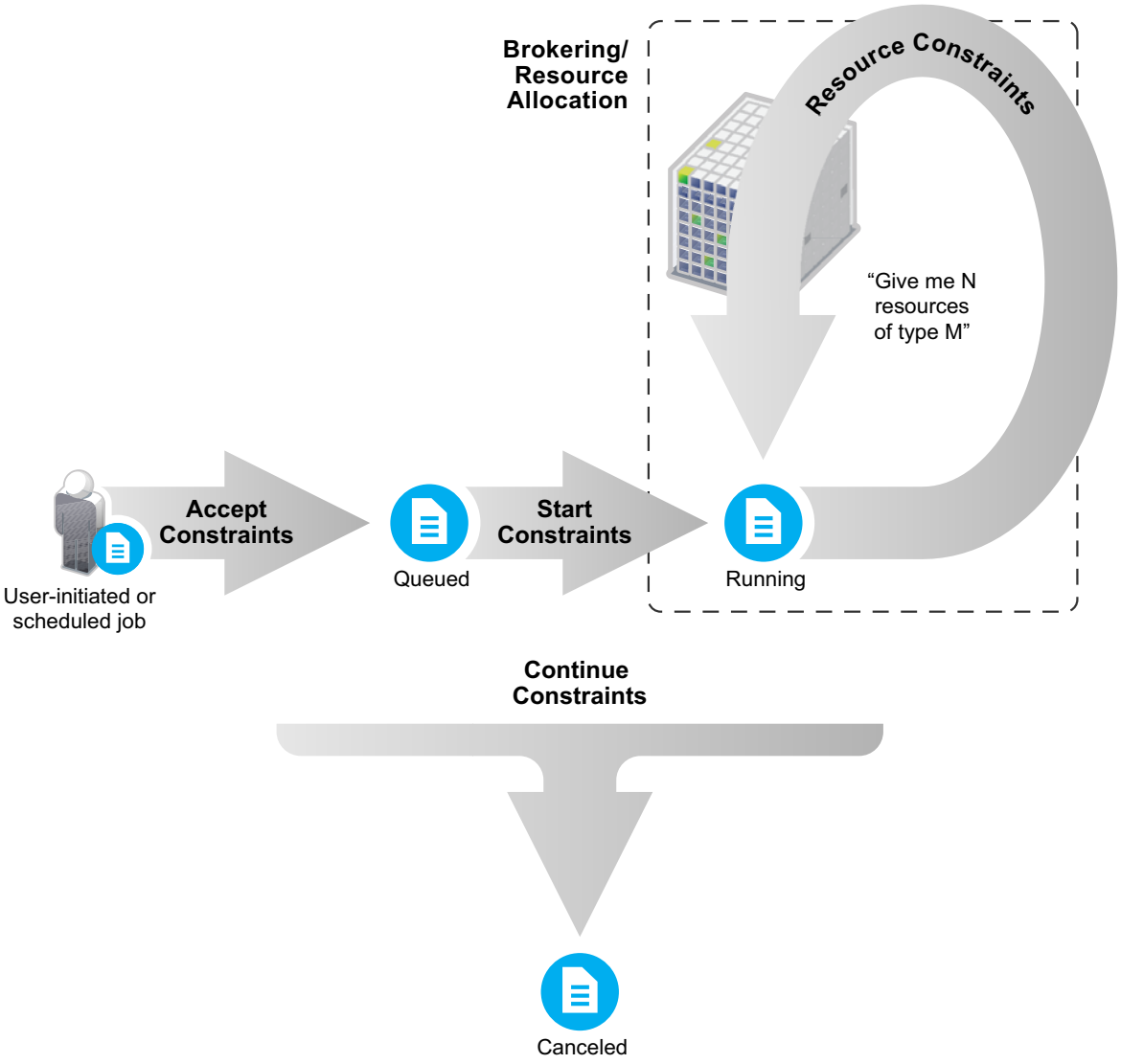
While `allocation` constraints are continuously evaluated, the results of `resource` constraints are cached for a short period of time (30 seconds). This difference allows you as a job writer to separate constraints between those that require an immediate check of a constantly changing fact, and those that require fewer checks because the fact changes infrequently.

For example, a resource's OS type might be unlikely to change, so a constraint that checks this fact fits in the `resource` constraint type (the assumption is that resource facts change infrequently, especially if they are used for determining joblet assignment). In contrast, a job instance fact can be changed frequently by a job instance, so a constraint that checks a job instance fact should fit in the `allocation` constraint type.

All of these constraints are visible and can be tested in the Policy Debugger in the Orchestration Console.

A job's life cycle as determined by constraints is illustrated in the following figure.

Figure 2-5 *Constraint-Based Job State Transition*



The following three figures provide more detail about the sequence of a job initiated by a user and the constraints it must satisfy before it runs. The diagrams are not intended to represent a finely-detailed flow, with every possible constraint, action, or state, but they do illustrate a high-level constraint workflow.

Figure 2-6 *VM-related Constraint Workflow*

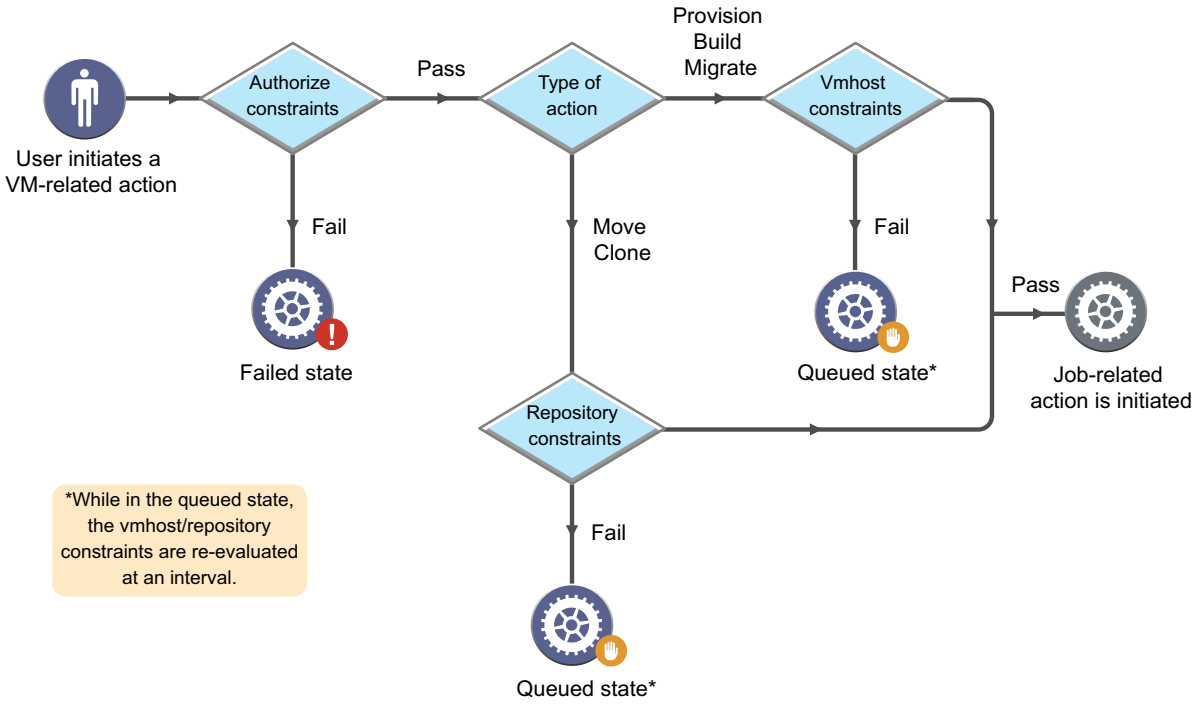


Figure 2-7 *Job-related Constraint Workflow*

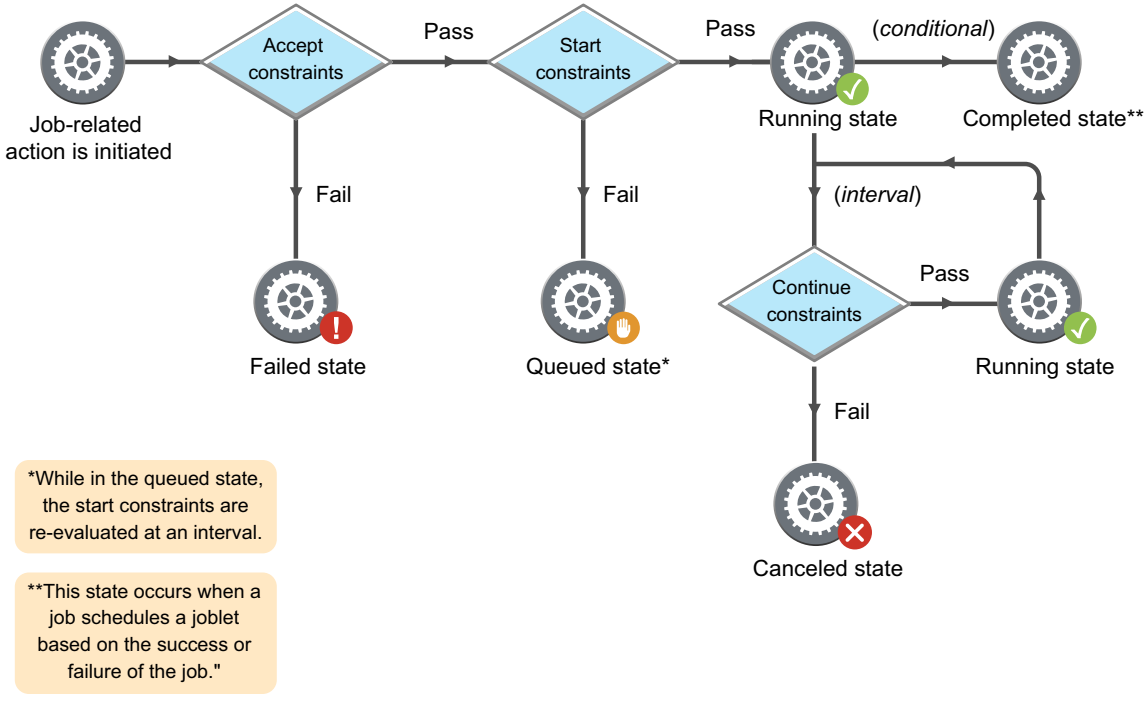
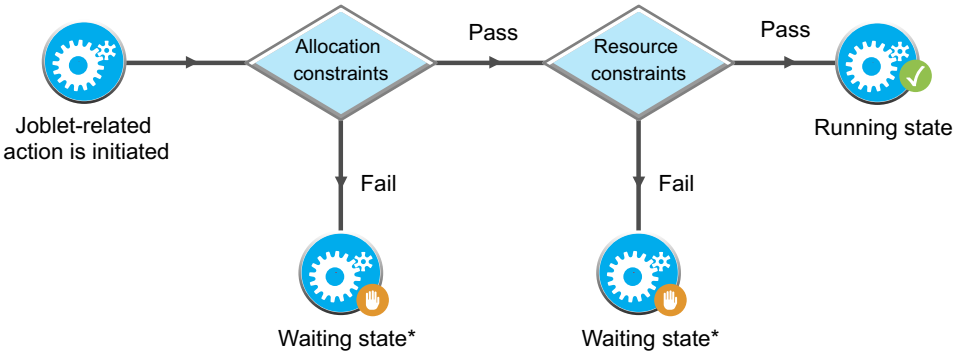


Figure 2-8 *Joblet-related Constraint Workflow*



*While in the waiting state, the resource and allocation constraints are re-evaluated at an interval.

Scheduling with Constraints

The constraint specification of the policies is comprised of a set of logical clauses and operators that compare property names and values. The grid server defines most of these properties, but they can also be arbitrarily extended by the user/developer.

All properties appear in the job context, which is an environment where constraints are evaluated. Compound clauses can be created by logical concatenation of earlier clauses. A rich set of constraints can thus be written in the policies to describe the needs of a particular job.

You can also set constraints through the use of deployed policies, and you can use jobs to specify additional constraints that can further restrict a particular job instance. The figure below shows the complete process employed by the Orchestration Server to constrain and schedule jobs.

When a user issues a work request, the user facts (`user.*` facts) and job facts (`job.*` facts) are added to the job context. The server also makes all available resource facts (`resource.*` facts) visible by reference. This set of properties creates an environment where constraints can be executed.

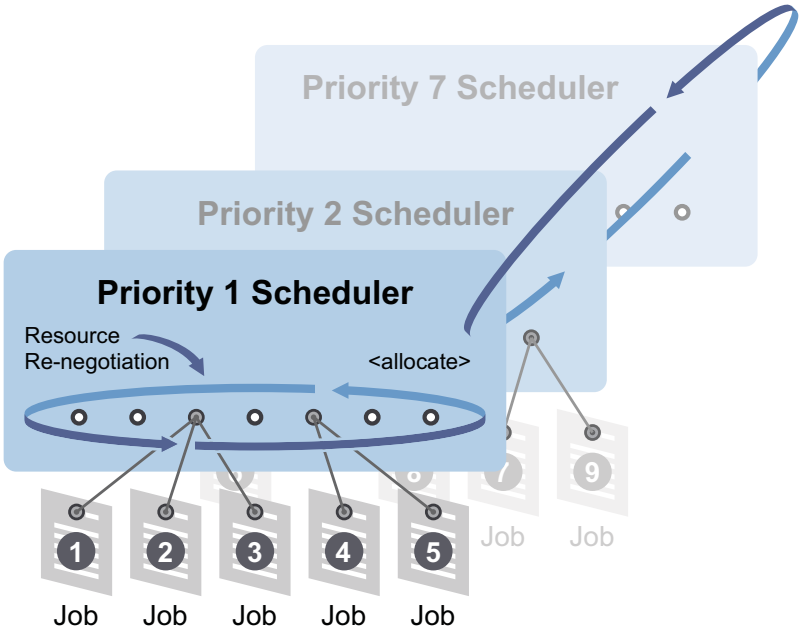
The Job Scheduling Manager applies a logic ANDing of job constraints (specified in the policies), grid policy constraints (set on the server), optionally additional user-defined constraints specified on job submission, and optional constraints specified by the resources.

This procedure results in a list of matching resources. The Orchestration solution returns three lists:

- Available resources
- Pre-emptable resources (nodes running lower priority jobs that could be suspended)
- Resources that could be “stolen” (nodes running lower-priority jobs that could be killed)

These lists are then passed to the resource allocation logic where, given the possible resources, the ordered list of desired resources is returned along with minimum acceptable allocation information. The Job Scheduling Manager uses all of this data to appropriate resources for all jobs within the same priority group.

Figure 2-9 *Job Scheduling Priority*



As the Job Scheduling Manager continually re-evaluates the allocation of resources, it relies on the job policies as part of its real-time algorithm to help provide versatile and powerful job scheduling.

Setting up a constraint for use by the Job Scheduling Manager is accomplished by adding a constraint in the job policy. For example, you might write just a few lines of policy code to describe a job requiring a node with a x86 machine, greater than 512 MB of memory, and a resource allocation strategy of minimizing execution time. Below is an example.

```
<constraint type=”resource”>
  <and>
    <eq fact="cpu.architecture" value="x86" />
    <gt fact="memory.physical.total" value="512" />
  </and>
</constraint>
```

Constraints Constructed in JDL

Constraints can also be constructed in JDL and in the Java Client SDK. A JDL-constructed constraint can be used for grid search and for scheduling. A Java Client SDK-constructed constraint can only be used for Grid object search.

When you create constraints, it is sometimes useful to access facts on a Grid object that is not in the context of the constraint evaluation. An example scenario would be to sequence the running of jobs triggered by the Job Scheduler.

In this example, you need to make `job2` run only when all instances of `job1` are complete. To do this, you could add the following start constraint to the `job2` definition:

```
<constraint type="start">
  <eq fact="job[job1].instances.active" value="0"/>
</constraint>
```

Here, the job in the context is `job2`, however the facts on `job1` (instances.active) can still be accessed. The general form of the fact name is:

`<grid_object_type>[<grid_object_name>].rest.of.fact.space`

Cloud Manager Orchestration supports specific Grid object access for the following Grid objects:

- Jobs
- Resources (physical or virtual machines)
- VM hosts (physical machines hosting guest VMs)
- Virtual Disks
- Virtual NICs
- Repositories
- Virtual Bridges
- Users

Currently, explicit group access is not supported.

For more detailed information, see the following JDL constraint definitions:

- AndConstraint (page 230)
- BinaryConstraint (page 232)
- Constraint (page 237)
- ContainerConstraint (page 238)
- ContainsConstraint (page 239)
- DefinedConstraint (page 243)
- EqConstraint (page 244)
- GeConstraint (page 248)
- GtConstraint (page 251)
- LeConstraint (page 257)
- LtConstraint (page 258)
- NeConstraint (page 263)
- NotConstraint (page 264)
- OrConstraint (page 265)
- UndefinedConstraint (page 278)

2.4 Resource Discovery

Resource discovery jobs inspect a resource's environment to set resource facts that are to be stored with the Resource grid object. These jobs automatically discover the resource attributes (fully extensible facts relating to such things as CPU, memory, storage, bandwidth, load, software inventory) of the resources being managed by the Orchestration Server. These facts are later used during Orchestration runtime from within a policy or constraint to select resources that have certain identifiable attributes.

- Section 2.4.1, "Resource Discovery in Provisioning Jobs," on page 29
- Section 2.4.2, "Some Specific Resource Discovery Jobs," on page 29

2.4.1 Resource Discovery in Provisioning Jobs

Provisioning jobs (also known as "provisioning adapters") are used to discover VM hosts (those resources running a VM technology such as Xen, VMware, or Hyper-V) and VM image repositories (such as Amazon EC2), as well as VM images residing in those repositories.

For more information, see "Orchestration Provisioning Adapter Information" in the *NetIQ Cloud Manager 2.1.1 VM Orchestration Reference*.

2.4.2 Some Specific Resource Discovery Jobs

Some of the commonly used resource discovery jobs include:

- "cpuInfo.job" on page 29
- "demoInfo.job" on page 29
- "findApps.job" on page 29
- "osInfo.job" on page 29

cpuInfo.job

Gets CPU information of a resource.

demoInfo.job

Generates the CPU, operating system, and application information for testing.

findApps.job

Finds and reports what applications are installed on the datagrid.

osInfo.job

Gets the operating system of a grid resource. On Linux, it reads the `/proc/cpuinfo`; on Windows, it reads the registry; on UNIX, it executes `uname`.

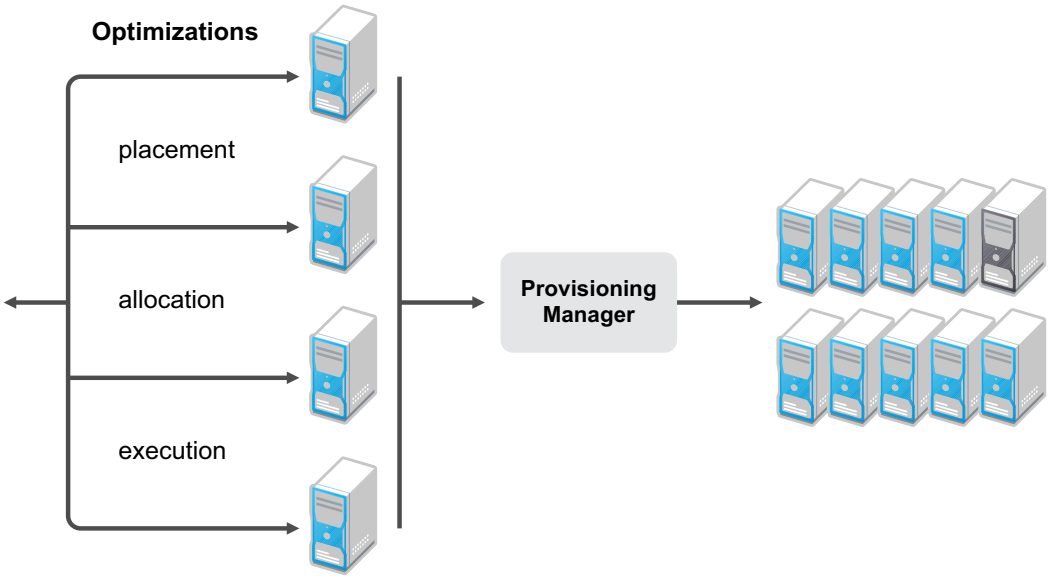
```
resource.cpu.mhz (integer) e.g., "800" (in Mhz)
resource.cpu.vendor (string) e.g. "GenuineIntel"
resource.cpu.model (string) e.g. "Pentium III"
resource.cpu.family (string) e.g. "i686"
```

2.5 Workload Management Performed by the Provisioning Manager

The Orchestration Server uses a Provisioning Manager to allocate (assign) and preempt (reassign) resources.

The Provisioning Manager preempts a resource by monitoring the job queue that is waiting for allocation. The manager then compares the job's relative priority to jobs already consuming resources. Higher priority jobs can preempt lower priority jobs.

Figure 2-10 *Workload Management*



Depending on the tasks that applications might require, the Orchestration Server submits the required jobs to one or more of the connected managed resources to perform specific tasks.

For more information about how job scheduling and provisioning works, see the following sections:

- Chapter 8, "Job Scheduling," on page 193
- Section 6.6, "Automatically Provisioning a VM," on page 124
- Examples: dgtest.job (page 147)

2.6 Deploying Packaged Job Files

After jobs are created, you deploy `.jdl` or multi-element `.job` files to the Orchestration Server by using any of the following methods:

- Copying job files into the "hot" Orchestration Server deployment directory. See "Deploying a Sample System Job" in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*.
- Using the Orchestration Console. This process is discussed in Section 3.5.1, "Using the Orchestration Console to Deploy Jobs," on page 38.
- Using the Orchestration Server command line (CLI) tools. This process is discussed in Section 3.5.2, "Using the zosadmin Command Line Tool to Deploy Jobs," on page 38.

2.7 Running Your Jobs

After your jobs are deployed, you can execute them by using the following methods:

- **Command Line Interface:** Nearly all Orchestration functionality, including deploying and running jobs, can be performed using the command line tool, as shown in the following example:

```
zos run buildTest [testlist=mylist]
JobID: paul.buildTest.14
```

More detailed CLI information is available in the zos command line tool.

- **Custom Client:** The Orchestration toolkit provides an SDK that provides a custom client that can invoke your custom jobs. This process is discussed in Appendix A, "The Cloud Manager Orchestration Client SDK," on page 203.
- **Built-in Job Scheduler:** The Orchestration Server uses a built-in Job Scheduler to run deployed jobs. This tool is discussed in Chapter 8, "Job Scheduling," on page 193 and in "The Orchestration Server Job Scheduler" in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*.
- **From Other Jobs:** As part of a job workflow, jobs can be invoked from within other jobs. You integrate these processes within your job scripts as described in the Chapter 8, "Job Scheduling," on page 193.

2.8 Monitoring Job Results

Cloud Manager Orchestration lets you monitor jobs by using the same methods outlined in Section 2.7, "Running Your Jobs," on page 30.

This section includes the following information:

- Section 2.8.1, "Monitoring Jobs from the Command Line," on page 31

2.8.1 Monitoring Jobs from the Command Line

The following example shows the status of the job `ray.buildtest.18` using different monitoring interfaces:

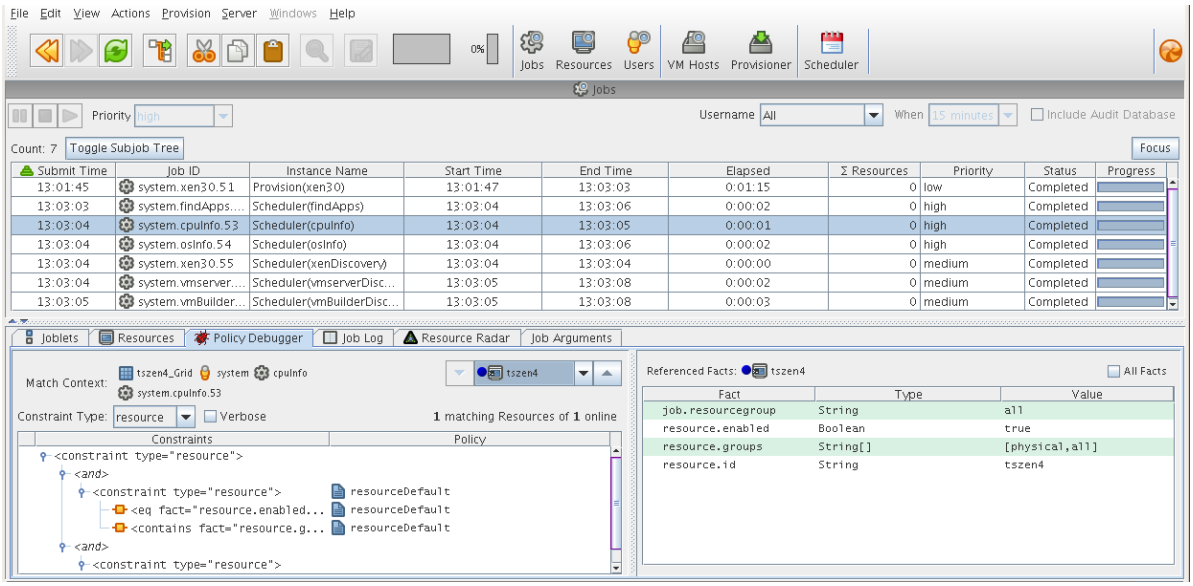
```
zos status -e ray.buildtest.18

Job Status for ray.buildtest.18
-------------------------------
              State: Completed      (0 this job)
     Resource Count: 0
   Percent Complete: 100%
          Queue Pos: 1 of 1 (initial pos=1)
    Child Job Count: 0              (0 this job)


      Instance Name: Buildtest
           Job Type: buildtest
               Memo: Build Test BuildID 02-02-09 1705 , failed: 1, Run: 5,
                     Passed: 4
           Priority: medium
          Arguments: <none>

        Submit Time: 02/02/2009 01:46:12
     Delayed Start: n/a
        Start Time: 02/02/2009 01:46:12
          End Time: 01/01/1009 01:46:14
      Elapsed Time: 0:00:01
        Queue Time: 0:00:00
        Pause Time: 0:00:00

    Total CPU Time: 0:00:00        (0:00:00 this job)
     Total GCycles: 0:00:00        (0:00:00 this job)
        Total Cost: $0.0002        ($0.0002 this job)
         Burn Rate: $0.0003/hr     (0.0003/hr this job)
```

The bottom section of the status report shows that you can also monitor job costing metrics, which are quite minimal in this example. More sophisticated job monitoring is possible.

2.9 Debugging Jobs

The following view of the Development Client shows how you can determine that the `buildTest` job was not able to find or match any resources because resources were not added to the buildtest group as required by the policy.

Figure 2-11 *Debugging Jobs Using the Orchestration Console*



The policy debugger shows the blocking constraint, and the tooltip gives the reason. If you drag and drop to add resources to the required group, the job continues quickly with no restart.

3 Job Architecture

The Cloud Manager Orchestration Job Scheduler is a sophisticated scheduling engine that maintains high performance network efficiency and quality user service when running jobs on the grid. Such efficiencies are managed through a set of grid component facts that operate in conjunction with job constraints. Facts and constraints operate together like a filter system to maintain both the administrator's goal of high quality of service and the user's goal to run fast, inexpensive jobs.

This section explains the following job architectural concepts:

3.1 Understanding JDL

The Orchestration Grid Management system uses an embedded Python-based language for describing jobs (called the Job Definition Language or JDL). This scripting language is used to control the job flow, request resources, handle events and generally interact with the Grid server as jobs proceed.

Jobs run in an environment that expects facts (information) to exist about available resources. These facts are either set up manually through configuration or automatically discovered via discovery jobs. Both the end-user jobs and the discovery jobs have the same structure and language. The only difference is in how they are scheduled.

The job JDL controls the complete life cycle of the job. JDL is a scripting language, so it does not provide compile-time type checking. There are no checks for infinite loops, although various precautions are available to protect against runaway jobs, including job and joblet timeouts, maximum resource consumption, quotas, and limited low-priority JDL thread execution.

As noted, the JDL language is based on the industry standard Python language, which was chosen because of its widespread use for test script writing in QA departments, its performance, its readability of code, and ease to learn.

The Python language has all the familiar looping and conditional operations as well as some powerful operations. There are various books on the language including O'Reilly's *Python in a Nutshell* and *Learning Python*. Online resources are available at http://www.python.org (http://www.python.org)

Within the Orchestration Server and grid jobs, JDL not only adds a suite of new commands but also provides an event-oriented programming environment. A job is notified of every state change or activity by calling an appropriately named event handler method.

A job only defines handlers for events it is interested in. In addition to built-in events (such as, `joblet_started_event`, `job_completed_event`, `job_cancelled_event`, and `job_started_event`) it can define handlers for custom events caused by incoming messages. For example, if a job (Job (page 252) class) defines a method as follows:

```
def my_custom_event(self, params):
  print Got a my_custom event carrying ", params
```

And the joblet (Joblet (page 254) class) sends an event/message as follows:

```
self.sendEvent("my_custom_evemt", {"arg1":"one"})
```

NOTE: The event being sent has to be the same name as the defined method receiving the event.

The following line is added to the job log:

```
Got a my_custom event carrying {'arg1':'one'}
```

JDL can also define timer events (periodic and one-time) with similar event handlers.

Each event handler can run in a separate thread for parallel execution or can be synchronized to a single thread. A separate thread results in better performance, but also incurs the development expense of ensuring that shared data structures are thread safe.

3.2 JDL Package

The job package consists of the following elements:

- Job Description Language (JDL) code, consisting of a Python-based script containing the bits to control jobs.
- An optional policy XML file, which applies constraints and other job facts to control jobs.
- Any other associated executables or data files that the job requires.

The `cracker.jdl` sample job, for example, includes a set of Java code that discovers the user password in every configured agent before the Java class is run. Or, many discovery jobs, which measure performance of Web servers or monitor any other applications, might include resource discovery utilities that enable resource discovery.

Jobs include all of the code, policy, and data elements necessary to execute specific, predetermined tasks administered either through the Orchestration Server Console user interface or from the command line. Because each job has specific, predefined elements, jobs can be scripted and delivered to any agent, which ultimately can lead to automating almost any datacenter task.

3.2.1 .sched Files

Job packages also can contain optional XML `.sched` files that describe the scheduling requirements for any job. This file defines when the job is run.

For example, jobs might be run whenever an agent starts up, which is defined in the `.sched` file. The discovery job "osInfo.job" on page 29 has a schedule XML file that specifies to always run a specified job whenever a specific resource is started and becomes available.

3.3 Job Class

The Job class is a representation of a running job instance. This class defines functions for interacting with the server, including handling notification of job state transitions, child job submission, managing joblets and for receiving and sending events from resources and from clients. A job writer defines a subclass of the job class and uses the methods available on the job class for scheduling joblets and event processing.

For more information about the methods this class uses, see Section 3.3.1, "Job State Transition Events," on page 35.

The following example demonstrates a job that schedules a single joblet to run on one resource:

```
class Simple(Job):
     def job_started_event(self):
         self.schedule(SimpleJoblet)

class SimpleJoblet(Joblet):
     def joblet_started_event(self):
         print "Hello from Joblet"
```

For the above example, the class `Simple` is instantiated on the server when a job is run either by client tools or by the job scheduler. When a job transitions to the started state, the method `job_started_event` is invoked. Here the `job_started_event` invokes the base class method `schedule()` to create a single joblet and schedule the joblet to run on a resource. The `SimpleJoblet` class is instantiated and run on a resource. A Resource is a physical or virtual machine on which the Orchestration Agent is installed and running and where the Joblet code is executed.

3.3.1 Job State Transition Events

Each job has a set of events that are invoked at the state transitions of a job. On the starting state of a job, the `job_started_event` is always invoked.

The following is a list of job events that are invoked upon job state transitions:

```
job_started_event
job_completed_event
job_cancelled_event
job_failed_event
job_paused_event
job_resumed_event
```

The following is a list of job events that are invoked upon child job state transitions:

```
child_job_started_event
child_job_completed_event
child_job_cancelled_event
child_job_failed_event
```

The following is a list of provisioner events that are invoked upon provisioner state transitions:

```
provisioner_completed_event
provisioner_cancelled_event
provisioner_failed_event
```

The following is a list of joblet events that are invoked as the joblet state transitions:

```
joblet_started_event
joblet_completed_event
joblet_failed_event
joblet_cancelled_event
joblet_retry_event
```

NOTE: Only the `job_started_event` is required; other events are optional.

3.3.2 Handling Custom Events

A job writer can also handle and invoke custom events within a job. Events can come from clients, other jobs, and from joblets.

The following example defines an event handler named `mycustom_event` in a job:

```
class Simple(Job):
     def job_started_event(self):
         ...

     def mycustom_event(self,params):
         dir = params["directory_to_list"]
         self.schedule(MyJoblet,{ "dir" : dir } )
```

In this example, the event retrieves a element from the params dictionary that is supplied to every custom event. The dictionary is optionally filled by the caller of the event.

The following example invokes the custom event named `mycustom_event` from the Orchestration command line tool:

```
zos event <jobid_of_running_job> mycustom_event directory_to_list="/tmp"
```

In this example, a message is sent from the client tool to the job running on the server. The following example invokes the same custom event from a joblet:

```
class SimpleJoblet(Joblet):
     def joblet_started_event(self):
         ...
         self.sendEvent("mycustom_event", {"directory_to_list":"/tmp"} )
```

In this example, a message is sent from the joblet running on a resource to the job running on the server. The running job has access to a factset which is the aggregation of the job instance factset (jobinstance.*), the deployed job factset (job.*, jobargs.*), the User factset (user.*), the Matrix factset (matrix.*) and any jobargs or policy facts supplied at the time the job is started.

Fact values are retrieved using the GridObjectInfo (page 249) functions that the job class inherits.

The following example retrieves the value of the job instance fact state.string from the jobinstance namespace:

```
class Simple(Job):
     def job_started_event(self):
         jobstate = self.getFact("jobinstance.state.string")
         print "job state=%s" % (jobstate)
```

For further details about each of the events above, see Section B.1, “Job Class,” on page 223.

The following example uses the `joblet_started_event` to determine the resource a Joblet is running on. If you implement the `joblet_started_event` job method, your job is notified when a Joblet has started execution:

```
1  class test(Job):
2     def job_started_event(self):
3         self.schedule(TestJoblet)
4
5   def joblet_started_event(self,jobletNum,resourceId):
6        print "joblet %d is running on  %s" % (jobletNum, resourceId)
7
8  class TestJoblet(Joblet):
9    def joblet_started_event(self):
10       import time
11       time.sleep(10)
```

In lines 5 and 6, the `joblet_started_event` is notified when the instance of `TestJoblet` is executing on a resource.

3.4 Job Invocation

Jobs can be started using either the zos command line tool, scheduling through a `.sched` file, or manually through the Orchestration Console. Internally, when a job is invoked, an XML file is created. It can be deployed immediately or it can be scheduled for later deployment, depending upon the requirements of the job.

Jobs also can be started within a job. For example, you might have a job that contains JDL code to run a secondary job. Jobs also can be started through the Web portal.

Rather than running jobs immediately, there are many benefits to using the Job Scheduling Manager:

- Higher priority jobs can be run first and jump ahead in the scheduling priority band.
- Jobs can be run on the least costly node resources when accelerated performance is not as critical.
- Jobs can be run on specific types of hardware.
- User classes can be defined to indicate different priority levels for running jobs.

3.5 Deploying Jobs

A job must be deployed to the Orchestration Server before that job can be run. Deployment to the server is done in either of the following ways:

- Section 3.5.1, “Using the Orchestration Console to Deploy Jobs,” on page 38
- Section 3.5.2, “Using the zosadmin Command Line Tool to Deploy Jobs,” on page 38

3.5.1 Using the Orchestration Console to Deploy Jobs

1 In the *Actions* menu, click *Deploy Job*.

3.5.2 Using the zosadmin Command Line Tool to Deploy Jobs

From the CLI, you can deploy a component file (`.job`, `.jdl`, `.sar`) or refer to a directory containing job components.

`.job` files are Java jar archives containing `.jdl`, `.policy`, `.sched` and any other files required by your job. A `.sar` file is a Java jar archive for containing multiple jobs and policies.

1 To deploy a `.job` file from the command line, enter the following command:

```
>zosadmin deploy <myjob>.job
```

2 To deploy a job from a directory where the directory `/jobs/myjob` contains `.jdl`, `.policy`, `.sched`, and any other files required by your job, enter the following command:

```
>zosadmin deploy /jobs/myjob
```

Deploying from a directory is useful if you want to explode an existing job or `.sar` file and redeploy the job components without putting the job back together as a `.job` or `.sar` file.

3 Copy the job file into the "hot" deploy directory by entering the following command:

```
>cp <install dir>/examples/whoami.job <install dir>/deploy
```

As part of an iterative process, you can re-deploy a job from a file or a directory again after specified local changes are made to the job file. You can also undeploy a job out of the system if you are done with it. Use `zosadmin redeploy` and `zosadmin undeploy` to re-deploy and undeploy jobs, respectively.

A typical approach to designing, deploying, and running a job is as follows:

1. Identify and outline the job tasks you want the Orchestration Server to perform.
2. Use the preconfigured JDL files for specific tasks listed in Appendix B, "Cloud Manager Orchestration Job Classes and JDL Syntax," on page 223.
3. To configure jobs, edit the JDL file with an external text editor.
4. Repackage the job as a `.jar` file.

   NOTE: The job could also be packaged and sent as an "exploded" file.

5. Run the zos CLI administration tool to redeploy the packaged job into the Orchestration Server.
6. Run the job using the zos command line tool.
7. Monitor the results of the job in the Orchestration Console.

Another method to deploy jobs is to edit JDL files through the Orchestration Console. The console has a text editor that enables you to make changes directly in the JDL file as it is stored on the server ready to deploy. After changes are made and the file is saved using the Orchestration Console, you simply re-run the job without redeploying it. The procedure is useful when you need to fix typos in the JDL file or have minor changes to make in the job functionality.

NOTE: Redeploying a job overwrites any job that has been previously saved on the Orchestration Server. The Orchestration Console has a *Save File* menu option if you want to preserve JDL modifications you made using the console.

There is no way "undo" a change to a `.jdl` file after the JDL editor in the Orchestration console has saved the file, nor is there a method for rolling back to a previously deployed version. We recommend that you use an external source code control system such as CVS or SVN for version control.

3.6 Starting Orchestration Jobs

Jobs can be started by using any of the following options:

- Running jobs from the `zos` command line (see "The zos Command Line Tool" in the *NetIQ Cloud Manager 2.1.1 Orchestration Server Command Line Reference*).
- Running jobs from the Orchestration Job Scheduler (see "The Orchestration Server Job Scheduler" in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*).
- Running jobs from within jobs (see Section 2.2.2, "Using Facts in Job Scripts," on page 17).

3.7 Using Other Grid Objects

Grid objects can be created and retrieved using jobs. This is done when facts from other objects are needed for job decision processing or when joblets are executed on a resource.

The MatrixInfo (page 261) Grid object represents the system and from the `MatrixInfo` object, you can retrieve other grid objects in the system. For example, to retrieve the resource grid object named `webserver` and a fact named `resource.id` from this object, you would enter the following JDL code:

```
webserver = getMatrix().getGridObject(TYPE_RESOURCE,"webserver")
id = webserver.getFact("resource.id")
```

In Line 1, the `ResourceInfo` Grid object for `webserver` is retrieved. The `getMatrix()` built-in function retrieves the `MatrixInfo` object instance. `getGridObject()` is a method on the `MatrixInfo` class. In Line 2, the fact value for the resource fact `resource.id` is retrieved.

The `MatrixInfo` Grid object also provides functions for creating other Grid objects. For more complete information about these functions, see MatrixInfo (page 261).

The `MatrixInfo` object can be used in both `Job` and `Joblet` classes. In the `Joblet` class, `MatrixInfo` cannot create new Grid objects. If your job is required to create Grid objects, you must use `MatrixInfo` in the Job class.

3.8 Communicating Through Job Events

JDL events are how the server communicates job state transitions to your job. The required `job_started_event` is always invoked when the job transitions to the `starting` state.

Likewise, all the other state transitions have JDL equivalents that can be optionally implemented in your job. For example, the `joblet_completed_event` is invoked when a joblet has transitioned to completed. You could implement `joblet_completed_event` to launch another job or joblet or send a custom event to a Client, another job, or another joblet.

You can also use your own custom events for communicating between Client, job, child jobs and joblets.

Every partition of a job (client, job, joblet, child jobs) can communicate directly or indirectly with any other partition of a job by using Events. Events are messages that are communicated to each of the job partitions. For example, a joblet running on a resource can send an event to the job portion running on the server to communicate the completion of a stage of operation.

A job can send an event to a Java Client signaling a stage completion or just to send a log message to display in a client GUI.

Every event carries a dictionary as a payload. You can put any key/values you want to fulfill the requirements of your communication. The dictionary can be empty.

For more information about events are invoked at the state transitions of a job, see Job (page 252) and Section B.7, "Joblet State Values," on page 226.

3.8.1 Sending and Receiving Events

To send an event from a joblet to a job running on a server, you would input the following:

1. The portion in the joblet JDL to send the event:

```
self.sendEvent("myevent", { "message": "hello from joblet" } )
```

2. The portion in job JDL to receive the event:

```
def myevent(self,params):
    print "hello from myevent. params=",params
```

To send an event from a job running on the server to a client, you would input the following:

```
self.sendClientEvent("notifyClient", { "log" : "Web server installation completed"
} )
```

In your Java client, you must implement AgentListener and check for an Event message.

For testing, you can use the `zos run ... --listen` option to print events from the server. For additional details about the `sendEvent()` and `sendClientEvent()` methods in the Job (page 252) and Joblet (page 254) documentation.

3.8.2 Synchronization

By default, no synchronization occurs on job events. However, synchronization is necessary when you update the same grid objects from multiple events.

In that case, you must put a synchronization wrapper around the critical section you want to protect. The following JDL script is how this is done:

```
1  import synchronize
2  def objects_discovered_event(self, params):
3      print "hello"
4  objects_discovered_event =
synchronize.make_synchronized(objects_discovered_event)
```

Line 1 specifies to use the synchronization wrapper, which requires you to import the synchronize package.

Lines 2 and 3 provide the normal definition to an event in your job, while line 4 wraps the function definition with a synchronized wrapper.

3.9 Executing Local Programs

Running local programs is one of the main reasons for scheduling joblets on resources. Although you are not allowed to run local programs on the server side job portion of JDL, there are two ways to run local programs in a joblet:

1. Use the built-in `system()` function.

   This function is used for simple executions requiring no output or process handling. It simply runs the supplied string as a shell command on the resource and writes `stdout` and `stderr` to the job log.

2. Use the `Exec` JDL class.

   The Exec class provides flexibility in how to invoke executables, to process the output, and to manage the process once running. There is provision for controlling `stdin`, `stdout`, and `stderr` values. `stdout` and `stderr` can be redirected to a file, to the job log, or to a stream object.

   Exec provides control of how the local program is run. You can choose to run as the agent user or the job user. The default is to run as the job user, but fallback to agent user if the job user does not exist on the resource.

   For more information, see Exec (page 245).

3.9.1 Output Handling

The Exec (page 245) function provides controls for specifying how to handle `stdout` out `stderr`. By default, Exec discards the output.

The following example runs a program that directs `stdout` and `stderr` to the job log:

```
e = Exec()
e.setShellCommand(cmd)
e.writeStdoutToLog()
e.writeStderrToLog()
e.execute()
```

The following example runs a program that directs `stdout` and `stderr` to files and opens the `stdout` file if there is no error in execution:

```
e = Exec()
e.setCommand("ps -aef")
e.setStdoutFile("/tmp/ps.out")
e.setStderrFile("/tmp/ps.err")
result = e.execute()
if result == 0:
    output = open("/tmp/ps.out").read()
    print output
```

3.9.2 Local Users

You can choose to run local programs and have file operations done as the agent user or the job user. The default is to run as the job user, but fallback to agent user if the job user does not exist on the resource. These controls are specified on the job. The job.joblet.runtype fact specifies how file and executable operations run in the joblet in behalf of the job user, or not.

The choices for `job.joblet.runtype` are defined in the following table:

Table 3-1 *Job Run Type Values*

Option	Description
RunAsJobUserFallingBackToNodeUser	Default. This means if the job user exists as a user on the resource, then executable and file operations is done on behalf of that user. By falling back, this means that if the job user does not exist, the agent will still execute the joblet executable and file operation as the agent user. If the executable or file operation still has a permission failure, then the agent user is not allowed to run the local program or do the file operation.
RunOnlyAsJobUser	This means resource can only run the executable or file operation as the job user and will fail immediately if the job user does not exist on the resource. You want to use this mode of operation if you wish to strictly enforce execution and file ownership. You must have your resource setup with NIS or other naming scheme so that your users will exist on the resource.
RunOnlyAsNodeUser	This means the resource will only run executables and do file operations as the agent user.

There is also a fact on the resource grid object that can override the `job.joblet.runtype` fact. The fact `resource.agent.config.exec.asagentuseronly` on the resource grid object can overwrite the `job.joblet.runtype` fact.

This ability to run as the job user is supported by the `enhanced exec` feature of the Orchestration Agent. A resource might not support Orchestration enhanced execution of running as job users. If the capability is not supported, the fact `resource.agent.config.exec.enhancedused` is False. This fact is provided so you can create a resource or allocation constraint to exclude such a resource if your grid mixes resource with/without the enhanced exec support and your job requires enhanced exec capabilities.

3.9.3 Safety and Failure Handling

An exception in JDL will fail the job. By default, an exception in the joblet will fail the joblet. The `job.joblet.*` facts provide controls on how many times a failure will fail the joblet. For more information, see Section 3.11, “Improving Job and Joblet Robustness,” on page 45.

```
try :
    < JDL >
except:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print "Exception:", exc_type, exc_value
```

JDL also provides the `fail()` function on the Job and Joblet class for failing a job and joblet. The `fail()` function takes an optional reason message.

You would use `fail()` when you detect an error condition and wish to end the job or joblet immediately. Usage of the joblet `fail()` fails the currently running instance of the joblet. The actual `failed` state of the joblet occurs when the maximum number of retries has been reached.

3.10 Logging and Debugging

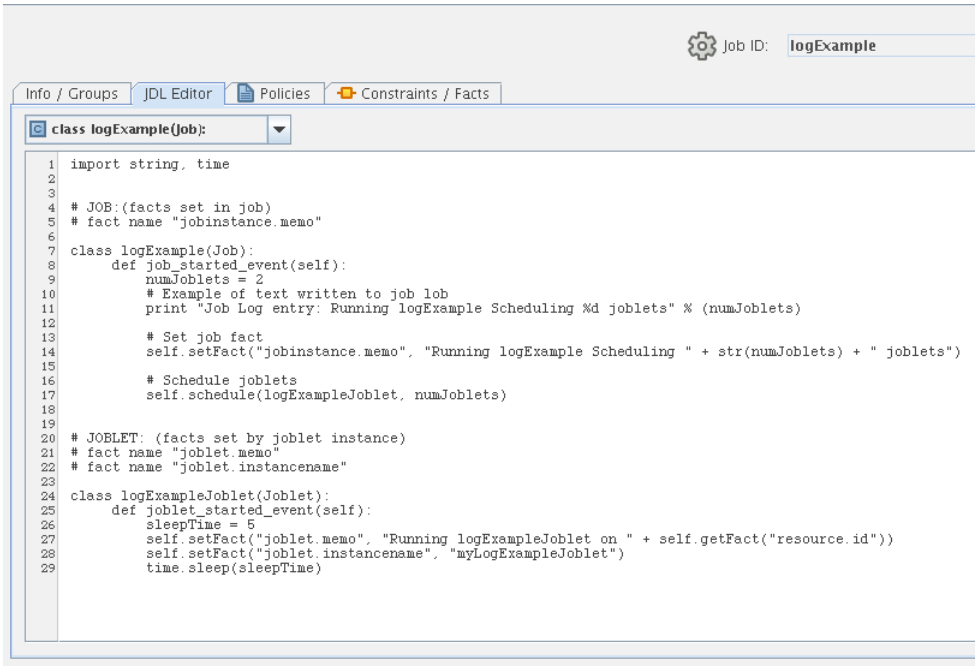
The following sections show some examples how jobs can be logged and debugged:

- Section 3.10.1, “Creating a Job Memo,” on page 43
- Section 3.10.2, “Tracing,” on page 44

3.10.1 Creating a Job Memo

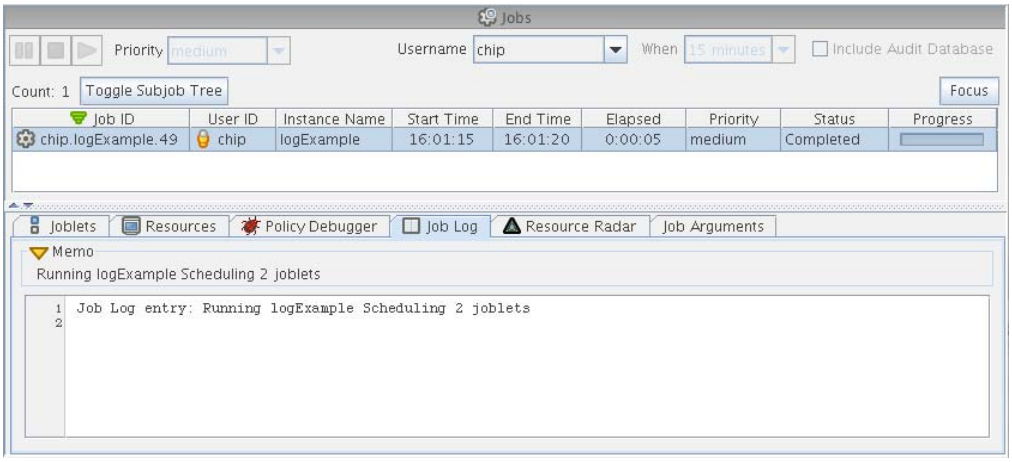
The following job example shows `logExample.jdl` output inthe JDL editor of the Orchestration Console.

Figure 3-1 *Example Job Displayed in the JDL Editor of the Orchestration Console*



In the job section of this example (lines 7-17), the fact `jobinstance.memo` (line 14) is set by the job instance. The job log text is emitted on line 11. Both of those are visible in the following example.

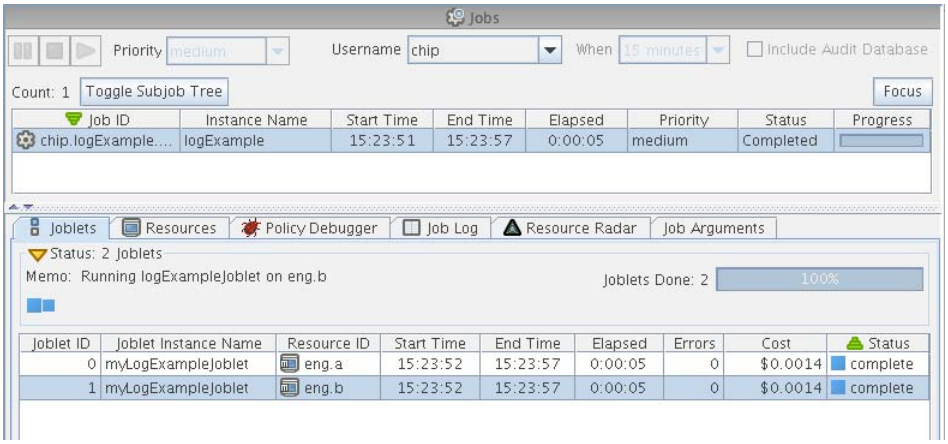
Figure 3-2 *Example Displaying the* `jobinstance.memo` *Fact and Job Log Text in the Jobs Monitor View of the Development Client*



In the `joblet` section of this example (lines 24-29), the fact named `joblet.memo` (line 27) is set by the joblet instance and consists of a brief memo for each joblet. This is typically used for providing detailed explanations, such as the name of the executable being run.

The name of the joblet is specified by the fact named `joblet.instancename` (line 28). This is typically a simple word displayed in the Development Client joblet column view. The following example shows the joblet facts `joblet.memo` and `joblet.instancename` in the Development Client.

Figure 3-3 *Example of Joblet Facts Displayed in the Development Client*



3.10.2 Tracing

There are two facts on the job grid object to turn tracing on or off. The tracing fact writes a message to the job log when a job and/or joblet event is entered and exited. The facts are `job.tracing` and `job.joblet.tracing`. You can turn these on using the Orchestration Console or you can use the `zos run` command tool.

3.11 Improving Job and Joblet Robustness

The job and joblet grid objects provide several facts for controlling the robustness of job and joblet operation.

The default setting of these facts is to fail the job on first error, since failures are typical during the development phase. Depending on your job requirements, you adjust the retry maximum on the fact to enable your joblets either to failover or to retry.

The fact `job.joblet.maxretry` defaults to 0, which means the joblet is not retried. On first failure, the joblet is considered failed. This, in turn, fails the job. However, after you have written and tested your job, you should introduce fault tolerance to the joblet.

For example, suppose you know that your resource application might occasionally timeout due to network or other resource problems. Therefore, you might want to introduce the following behavior by setting facts appropriately:

- On timeout of 60 seconds, retry the joblet.
- Retry a maximum of two times. This may cause a retry on another resource matching your resource and allocation constraints.
- On the third timeout, fail the joblet.

To configure this setup, you use the following facts in either the job policy (using the Orchestration Console to edit the facts directly) or within the job itself:

```
job.joblet.timeout set to 60    job.joblet.maxretry set to 2
```

In addition to timeout, there are different kinds of joblet failures for which you can set the maximum retry. There are forced (job errors) and unforced connection errors. For example, an error condition detected by the JDL code (forced) might require more retries than a network error, which might cause resource disconnections. In the connection failure case, you might want to lower the retry limit because you probably do not want a badly setup resource with connection problems to keep retrying and getting work.

3.12 Using an Event Notification in a Job

Jobs can be notified of an Orchestration event in two ways.

- A running Job can subscribe to receive Orchestration event notifications. (See Section 3.12.1, "Receiving Event Notifications in a Running Job," on page 45)
- A Job can be scheduled to start upon an Event notification.

  For more information about job scheduling, see Chapter 8, "Job Scheduling," on page 193 or "The Orchestration Server Job Scheduler" in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*.

3.12.1 Receiving Event Notifications in a Running Job

Subscribe

For a job to receive notifications, a job subscribes to an event and must remain running for the notification to occur.

How to subscribe to an event is accomplished using the `subscribeToEvent()` Job method, shown below:

```
def subscribeToEvent( <event Name>, <Job callback method> )
```

In this method, `<event name>` is the string name of the event being subscribed to; `<Job callback method>` is the reference to a Job method. Joblets and globals are not supported.

Example: The following is an example of the `subscribeToEvent()` method.

```
self.subscribeToEvent(  "vmhost" ,self.eventHandler)
```

In this example, `vmhost` is the name of the event and `self.eventHandler` is a reference to the callback method.

Unsubscribe

To unsubscribe, use the `unscribeFromEvent()` Job method. For the subscribe example shown above, the following is how to unsubscribe.

```
self.unsubscribeFromEvent("vmhost",self.eventHandler)
```

Callback Method Signature

```
def <Job callback method>( self, context):
```

The callback method must be a Job method. The context argument is a dictionary containing name/value pairs. The dictionary contents passed to the callback vary depending on the event type.

Example: The following is an example of the callback method.

```
def eventHandler(self,context):
```

In this method, `context` is the required dictionary argument passed to every callback. The contents of the dictionary vary depending on event type (for details, see Section 3.12.2, “Event Types,” on page 47).

How an Event Notification Can Start a Job

You can create a schedule using the Job Scheduler or deploy a `.sched` file to start a job on an event notification. For more information, see Chapter 8, “Job Scheduling,” on page 193 or “The Orchestration Server Job Scheduler” in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*.

The job to be started must match a required job argument signature where the job must define at least one job argument.

The required job argument must be called “context” and be of type Dictionary. The contents of the dictionary vary depending on event type (refer to "Event Types " below for details).

The contents of `EventResponse.jdl` is an example of a job and policy that can be scheduled on an event notification:

```
1class EventResponse(Job):
2
3   def job_started_event(self):
4       context = self.getFact("jobargs.context")
5
6       print "Context:"
7       keys = context.keys()
8       keys.sort()
9       for k in keys:
10          v = context[k]
11          print "  key(%s) type(%s) value(%s)" % (k,type(v),str(v))
```

Line 4: This line pulls out the job argument for the event context.

Lines 6-11: These lines print out the contents of the context dictionary.

The contents of `EventResponse.policy` are shown below:

```
1<policy>
2  <jobargs>
3    <fact name="context" type="Dictionary"
4         description="Dictionary containing the context for the event " />
5   </jobargs>
6</policy>
```

Lines 3-4: These lines define the required job argument containing the Event context. The running job receives the job argument named `context` with the dictionary completed by the Orchestration Event Manager with the context that matches the trigger rules.

3.12.2 Event Types

Event Objects

Event objects are defined in an XML document and deployed to a server and managed using the Orchestration console. In the console, these objects are shown in the tree view.

The callback method context argument dictionary contains every grid object type and a value or list of values. The dictionary depends on the event XML definition and the matching grid objects of the `<trigger>` rule.

The following example event file (`vmhost.event`) shows the contents of the dictionary that will be passed as either a jobarg to a job to be scheduled to start, or as a argument to an event callback for a running job.

```
1<event>
2
3   <context>
4      <vmhost />
5      <user>system</user>
6   </context>
7
8   <trigger>
9       <gt fact="vmhost.resource.loadaverage" value="2" />
10   </trigger>
11
12</event>
```

Lines 3-6: Define the context for the Event object.

Line 4: Defines the match for the trigger rule that iterates over all vmhosts.

Line 5: Defines the context, and contains the user grid object named `system`.

Assuming that there are 10 vmhosts named "`vmhost1`, `vmhost2`, ... `vmhost10`, but only the first three vmhosts match the trigger rule, the context includse a list of the matching vmhosts. In this case, the context dictionary contains the following:

```
        {
1          "vmhost" : [  "vmhost1", "vmhost2", "vmhost3" ],
2         "user" : "system",
3          "repository" : "" ,
4          "resource" : "",
5          "job" : ""
         }
```

Line 1: List of the matching VM hosts that passed the `<trigger>` rule.

Line 2: The user object that is defined in the `<context>` XML. In this case, `system`.

Lines 3-5: These grid objects are not defined in the context. Their value is empty.

In this example, the dictionary is passed as a job argument to a scheduled job that triggers on the event or is passed to a callback method in a running job that has subscribed to the event.

Built-in Events

Built-in events occur when a managed object comes online or offline or when that object has a health status change. For built-in events, the dictionary contains the name of the grid object type. The value is the name of the grid object.

The Orchestration built-in events are named as follows:

- `RESOURCE_ONLINE`
- `RESOURCE_NEEDS_UPGRADE`
- `USER_ONLINE`
- `RESOURCE_HEALTH`
- `USER_HEALTH`
- `VMHOST_HEALTH`
- `REPOSITORY_HEALTH`

For example, when the resource `xen1` comes online, the built-in event called `RESOURCE_ONLINE` is invoked. Any scheduled jobs are started and any running jobs that have subscribed are invoked. The context dictionary contains the following:

```
{
    "resource" : "xen1"
}
```

The dictionary shown above is passed as a job argument to a scheduled job that triggers on the event or that is passed to a callback method in a running job that has subscribed to the event.

External Events

External events are events that are invoked by an outside process. In this case, the callback method context dictionary is free form and contains what was supplied to the external event.

For example, if the external event was invoked with a dictionary mapping of two elements like this:

```
{ "name" : "foo", "age" : 40 }
```

The corresponding JDL callback method receives the same dictionary.

Example: The following example of an external event subscribes to a previously deployed event named vmhost. This job continue running until the job is canceled or an error occurs.

```
1 EVENT = "vmhost"
2
3 class EventDaemon(Job):
4
5   def job_started_event(self):
6
7       self.setFact("job.autoterminate",False)
8       self.subscribeToEvent(EVENT,self.eventHandler)
9       print "Waiting for notification for event '%s'" % (EVENT)
10
11
12  def eventHandler(self,context):
13       print "Context:"
14
15       keys = context.keys()
16       keys.sort()
17       for k in keys:
18          v = context[k]
19          print "  key(%s) type(%s) value(%s)" % (k,type(v),str(v))
```

Line 7: Sets the autoterminate fact so that the job remains running upon completion of the `job_started_event()`.

Line 8: Subscribes to the named event vmhost and passes in a reference to the Job method `eventHandler()` to callback on when the event notification occurs.

Line 12: Definition of the callback method to invoke when the event notification occurs.

Lines 15-19: Prints out the context dictionary received upon event notification.

4 Understanding Grid Object Facts, Computed Facts, and Custom Facts

This section includes the following information:

4.1 Grid Object Facts and Fact Junctions

Every component discovered in a NetIQ Cloud Manager Orchestration-enabled network is identified and abstracted as an object. Within the Orchestration management framework, objects are stored within an addressable database called a Grid. Every Grid object has an associated set of facts and constraints that define its properties and characteristics. Essentially, by building, deploying, and running jobs on the Orchestration Server, you can individually change the functionality of any and all system resources by managing an object’s facts and constraints.

The components that have facts include Jobs, Resources (including physical machines, virtual machines and VM hosts), Virtual Disks (vDisks), Virtual NICs (vNICs), Repositories, Virtual Bridges, and Users. The Orchestration Server assigns default values to each of the component facts, although they can be changed at any time by the administrator (unless they are read-only).

The XML fact element defines a fact to be stored in the grid object’s fact namespace. The name, type and value of the fact are specified as attributes. For list or array fact types, the element tag defines list or array members. For dictionary fact types, the dict tag defines dictionary members.

See the examples in the directory, `/allTypes.policy`. This example policy has an XML representation for all the fact types.

Facts can also be created and modified in JDL and in the Java Client SDK

As a Job Developer, you might want certain constraints to be used for a job and you might specify these in the policy. These comprise a set of logical clauses and operators that are compared with the respective component’s fact values when the job is run by the Job Scheduling Manager. See

Remember, all properties appear in the job context, which is an environment where constraints are evaluated. These constraints provide a multilevel filter for a job in order to ensure the best quality of service the grid can provide.

This section includes the following information:

- Section 4.1.4, “Resource Object Facts and Fact Junctions,” on page 66
- Section 4.1.5, “Virtual Disk Object Facts and Fact Junctions,” on page 87
- Section 4.1.6, “Virtual NIC Object Facts and Fact Junctions,” on page 89
- Section 4.1.7, “Repository Object Facts and Fact Junctions,” on page 91
- Section 4.1.8, “Virtual Bridge Object Facts and Fact Junctions,” on page 94
- Section 4.1.9, “User Object Facts and Fact Junctions,” on page 96
- Section 4.1.10, “Matrix Object Facts,” on page 101

For further fact information found in jobs, see Chapter 7, “Job Examples,” on page 127 and Section 2.2.2, “Using Facts in Job Scripts,” on page 17.

4.1.1 Fact Type Definitions

The following table explains the abbreviated codes used to describe facts for Orchestration Grid objects:

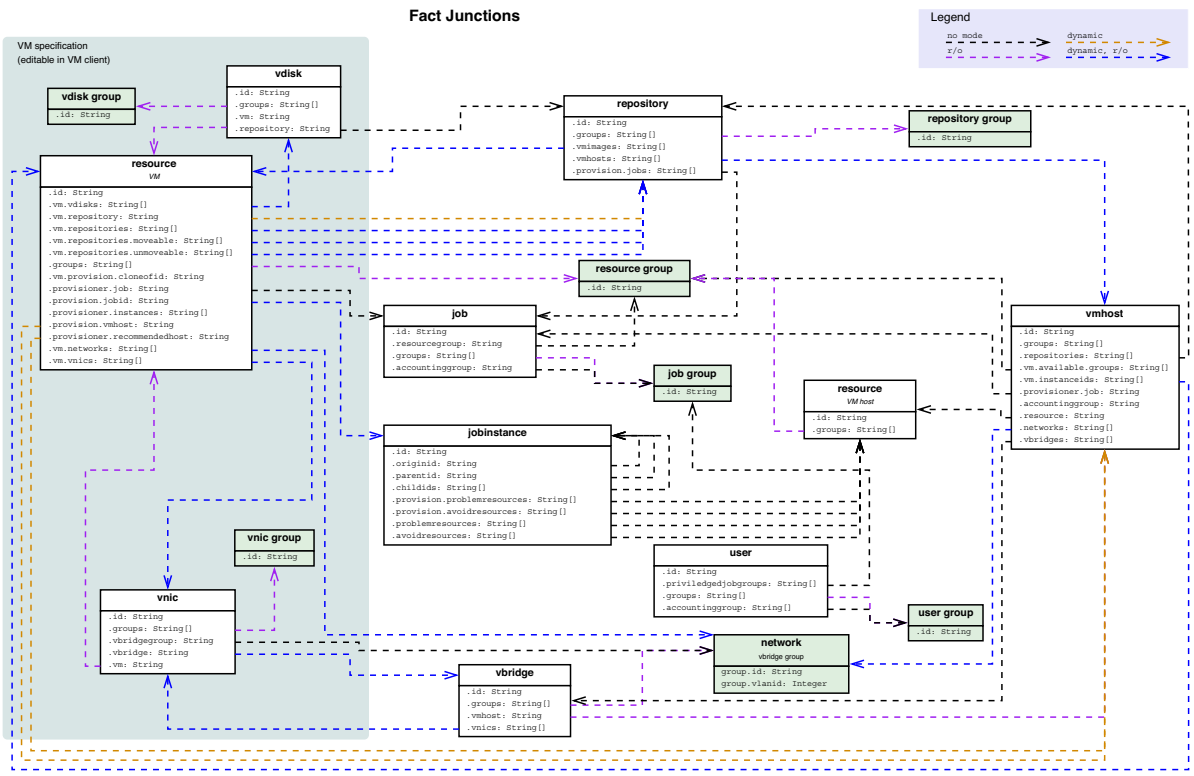
Table 4-1 *Cloud Manager Orchestration Fact Types*

Type	Description
Automatic	The fact should be automatically set after the successful discovery of virtual resources (VM Hosts and VMs).
Boolean	The fact is a Boolean value.
Default	The specified default value of the fact is set.
Dictionary	The fact is selected from a specified dictionary listing.
Dynamic	The fact is dynamically generated.
Enumerate	The fact is a specified enumerated value.
Example	When available, provides an example how a fact might be applied to an object.
Integer	The fact is an integer value.
Real	The fact is a real number.
String	The fact is a string value.
Datagrid	Facts relate to datagrid object types.
Local	Facts relate to local object types.
NAS	Facts relate to Network Attached Storage (NAS) object types.
SAN	Facts relate to Storage Area Network (SAN) object types.
Virtual	Facts relate to virtual object types.

4.1.2 Understanding Fact Junctions

A fact junction is a special type of fact that provides a convenient way to access facts on Grid objects related to the one where the fact lookup is being performed. The following diagram shows the fact junction relationships of all the Grid objects:

Figure 4-1 *Fact Junctions Between Objects in a Sample Orchestration Grid*



As an example of how a fact junction works, a fact lookup on `vmhost.resource.id` is redirected from the VM host object, through the junction onto the underlying physical Resource object. In other words, the value returned is the same as if a fact lookup for `resource.id` was performed on the underlying physical resource. This is accomplished with the following JDL:

```
vmhost1 = getMatrix().getGridObject(TYPE_VMHOST, "vmhost1")
print vmhost1.getFact("vmhost.resource.id")
```

Another example is `vdisk.repository.freespace`, which returns the amount of free space in the repository that is associated with the virtual disk where the fact lookup is being performed:

```
vdisk = getMatrix().getGridObject(TYPE_VDISK, "vm1_vdisk1")
print vdisk.getFact("vdisk.repository.freespace")
```

Note that the fact junction refers to the related Grid object rather than to any of its facts. Therefore, to obtain the ID of the repository associated with a given virtual disk, you must perform a lookup on `vdisk.repository.id` rather than `vdisk.repository`.

It is important to understand how the fact name is constructed from the junction, otherwise certain usages of fact junctions can be confusing, especially when used with facts that contain the dot (“.”) character. For example, starting with a vDisk Grid object as above:

```
vdisk = getMatrix().getGridObject(TYPE_VDISK, "vm1_vdisk1")
```

vDisk objects have a fact junction, `vdisk.vm`, that points to the VM associated with the vDisk. If you want to find all VNICs associated with this VM, remember that VMs have a fact `resource.vm.vnics` that provides the desired array. However, because Cloud Manager Orchestration accesses this fact by using the `vdisk.vm` fact junction, you must replace the resource component of the `resource.vm.vnics` fact with `vdisk.vm`. Therefore, the required code is:

```
print vdisk.getFact(“vdisk.vm.vm.vnics”)
```

Some fact junctions return an array of values rather than a single value. For example, the junction `vmhost.repositories` returns an array of all the repositories visible to the VM host where the lookup is being performed:

```
vmhost1 = getMatrix().getGridObject(TYPE_VMHOST, "vmhost1")
print host1.getFact("vmhost.repositories")
```

In this case, you can also single out one of the Grid objects returned in the array and perform fact lookups on that object. For example, if the repository `san1` is accessible by `vmhost1`, then

```
print vmhost1.getFact("vmhost.repositories[san1].freespace")
```

returns the amount of free space available in the `san1` repository.

```
repo_host1 = getMatrix().getGridObject(TYPE_REPOSITORY, "host1")
print repo_host1.getFact("repository.vmhosts[host1_demoAdapter].networks")
```

Fact junction lookups can be chained multiple times, even mixing use of single-valued and array-valued junctions:

```
vdisk = m.getGridObject(TYPE_VDISK, "vm1_vdisk1")
print vdisk.getFact("vdisk.repository.vmhosts[vmhost1].networks")

host1 = m.getGridObject(TYPE_RESOURCE, "host1")
print
host1.getFact("resource.vmhosts[host1_demoAdapter].repositories[san1].freespace")
```

For a more comprehensive list of available fact junctions, see the example `factJunction.job` stored at `/opt/novell/zenworks/zos/server/examples` on your server installation system.

4.1.3 Job, Jobinstance, and Joblet Object Facts and Fact Junctions

This section includes the following information:

Job Object Facts

The Constraints/Facts tab opens a page that shows all of the effective constraints and facts for a Grid object. Each Grid object has an associated set of facts and constraints that define its properties. In essence, by changing the policy constraints and fact values for a job, you can change the behavior of the job and how the Orchestration Server allocates available system resources to it. The Orchestration Server assigns default values to each of the component facts, although they can be changed at any time by the administrator, unless they are read-only.

The following table lists the default facts created by the Orchestration Server for the Job object.

NOTE: Facts with mode `dynamic` are dyamic read/write facts, which means you can dynamically change the values for that fact.

Facts with mode `r/o` have read-only values, which means they can be viewed but changes cannot be made.

Facts with mode `del` are deleteable, which means they can be deleted at any time. Where facts can be deleted in the Development Client, they can also be deleted in the `GridObjectInfo.deleteFact()` method in JDL.

Table 4-2 *Job Facts*

Fact Name	Fact Type	Mode	Description
`job.accountinggroup`	String		The default job group whose statistics are updated when this job runs.
`job.autoterminate`	Boolean		Whether this job ends when all child jobs and joblets end
`job.cacheresourcematches.ttl`	Integer		Indicates the jobs willingness to allow resource maches to be cached if the scheduler becomes too loaded. The value is the TTL of the cache in seconds ('`<=0`' to disable caching)
`job.description`	String		Textual description of job
`job.enabled`	Boolean		True if the job is enabled (able to be run).
`job.groups`	String[]	r/o	The groups where this job is a member.
`job.history.cost.average`	Real	dynamic, r/o	The average cost of running this job measured since job deployment or last modification. Only updated if the job finishes successfully.
`job.history.cost.total`	Real	dynamic, r/o	The total cost of running this job measured since job deployment or last modification
`job.history.gcycles.average`	Integer	dynamic, r/o	The average amount of normalized grid time (gcycles) taken by running this job. Only updated if the job finishes successfully.
`job.history.gcycles.total`	Integer	dynamic, r/o	The total amount of normalized grid time (gcycles) consumed by this job since deployment
`job.history.jobcount`	Integer	dynamic, r/o	The total number of job instances of this job ever initiated on the system (includes those denied by 'accept' constraints

Fact Name	Fact Type	Mode	Description
job.history.jobcount.cancelled	Integer	dynamic, r/o	The total number of job instances of this job that were cancelled
job.history.jobcount.complete	Integer	dynamic, r/o	The total number of job instances of this job that were completed
job.history.jobcount.failed	Integer	dynamic, r/o	The total number of job instances of this type that failed
job.history.queuetime.average	Integer	dynamic, r/o	The average wall time spent waiting for this job to start in seconds
job.history.queuetime.total	Integer	dynamic, r/o	The total amount of time this job has spent in a queued state since last deployment.
job.history.runtime.average	Integer	dynamic, r/o	The average runtime of this job in seconds. Only updated if the job finishes successfully.
job.history.runtime.total	Integer	dynamic, r/o	The total runtime of the job since deployment in seconds
job.history.samplesize	Integer		The number of points used in the trailing average calculation for all historical averages
job.history.time.average	Integer	dynamic, r/o	The average wall time taken by running this job in seconds. This fact is updated only if the job finishes successfully.
job.history.time.total	Integer	dynamic, r/o	The total combined resource wall time of all work performed on behalf of this job since deployment in seconds
job.id	String	r/o	The name of the job.
job.instances.active	Integer	dynamic, r/o	The number of job instances of this type that are active in the system ('running' or 'paused')c
job.instances.queued	Integer	dynamic, r/o	The number of job instances of this type that are in a queued state
job.instances.total	Integer	dynamic, r/o	The total number of job instances of this type that exist in the sytem
job.joblet.immediateretry	Boolean		Specifies whether the system to attempt to immediately retry a joblet rather than waiting until all others are running/ complete before retrying

Fact Name	Fact Type	Mode	Description
job.joblet.max	Integer		The absolute maximum number of joblets this job may schedule.
job.joblet.maxfailures	Integer		Specifies the number of non-fatal joblet errors to tolerate before failing completely or '-1' to attempt to continue after errorsc
job.joblet.maxperresource	Integer		The absolute maximum number of joblets this job may occupy on any one resource ('-1' indicates no limit)
job.joblet.maxretry	Integer		The maximum number of joblet retries of any type that will be attempted before considering the joblet as failed (0 means don't retry, <0 means keep retrying)
job.joblet.maxrunning	Integer		The absolute maximum number of joblets a job is allowed to have running at one time. This value will never be exceeded ('-1' indicates no limit)
job.joblet.maxwaittime	Integer		The maximum (resource) wait time permitted by a joblet in seconds ('-1' indicates no timeout)
job.joblet.retrylimit.disconnect	Integer		The number of joblet retries caused by unexpected resource disconnect that will be allowed before considering the joblet as failed (0 means don't retry, <0 means keep retrying). Can never exceed job.joblet.maxretry
job.joblet.retrylimit.forced	Integer		The number of forced joblet retries (requested by the joblet to run an another resource) that will be allowed before considering the joblet as failed (0 means don't retry, <0 means keep retrying). Can never exceed job.joblet.maxretry

Fact Name	Fact Type	Mode	Description
job.joblet.retrylimit.timeout	Integer		The number of joblet retries caused by server inititated joblet timeout that will be allowed before considering the joblet as failed (0 means don't retry, <0 means keep retrying). Can never exceed job.joblet.maxretry.
job.joblet.retrylimit.unforced	Integer		The number of unforced joblet retries that will be allowed before considering the joblet as failed (0 means don't retry, <0 means keep retrying). Can never exceed job.joblet.maxretry
job.joblet.runtype	String		Specify file and executable operations run in Joblet are in behalf of the Job user or not.
job.joblet.timeout	Integer		The timeout after which the server will take action to cancel the joblet (seconds, '-1' indicates no timeout)
job.joblet.tracing	Boolean		Indicates whether the joblet should include tracing information in the job log when executing joblet events
job.jobtime			The average wall time this job should take to run. Used to override the computed average when job is of type 'fixedtime' (in seconds)
job.jobtype	String		The type of job -- used in completion time calulation (normal, workflow, pspace, fixedtime, fixedcycles)
job.maxnodefailures	Integer		The maximum number of resource failures that are to be tolerated before excluding the node from future joblet processing. A value of -1 indicates that unlimited failures are acceptable.
job.maxresources	Integer		The absolute maximum number of resources that a job is allow to use at one time. This value will never be exceeded. A value of -1 indicates no limit.

Fact Name	Fact Type	Mode	Description
job.persistfactsonfinish	Boolean		Whether the Grid objects that this job modifies are to be persisted at job end. Used and applicable only when installed in a high availablity cluster.
job.preemptible	Boolean		Indicates whether this job is willing or able to be preempted. Turned on by setting joblet.preemptible. (can be overridden by the job instance).
job.preemption.rankby	String[]		The ranking specification used to select suitable jobs to automatically preempt a resource on. The syntax for each element in the list is <fact>/<*order*> where *order* is either “a” for ascending or “d” for descending.
job.provision.hostselection	String		The strategy used in finding a host for any automatically provisioned resource (queue, immediate).
job.provision.maxcount	Integer		The number of resources that can be automatically provisioned on behalf of this job. A value of 0 turns off automatic provisioning behavior. A value of -1 allows unlimited provisioning.
job.provision.maxnodefailures	Integer		The maximum number of provision failures that will be tolerated before excluding the node from future automatic provisioning. A value of -1 indicates that unlimited failures are acceptable.
job.provision.maxpending	Integer		The number of resources tha can be automatically provisioned at one time (simulataneously) on behalf of this job. A value of <=0 turns off automatic provisioning behavor.

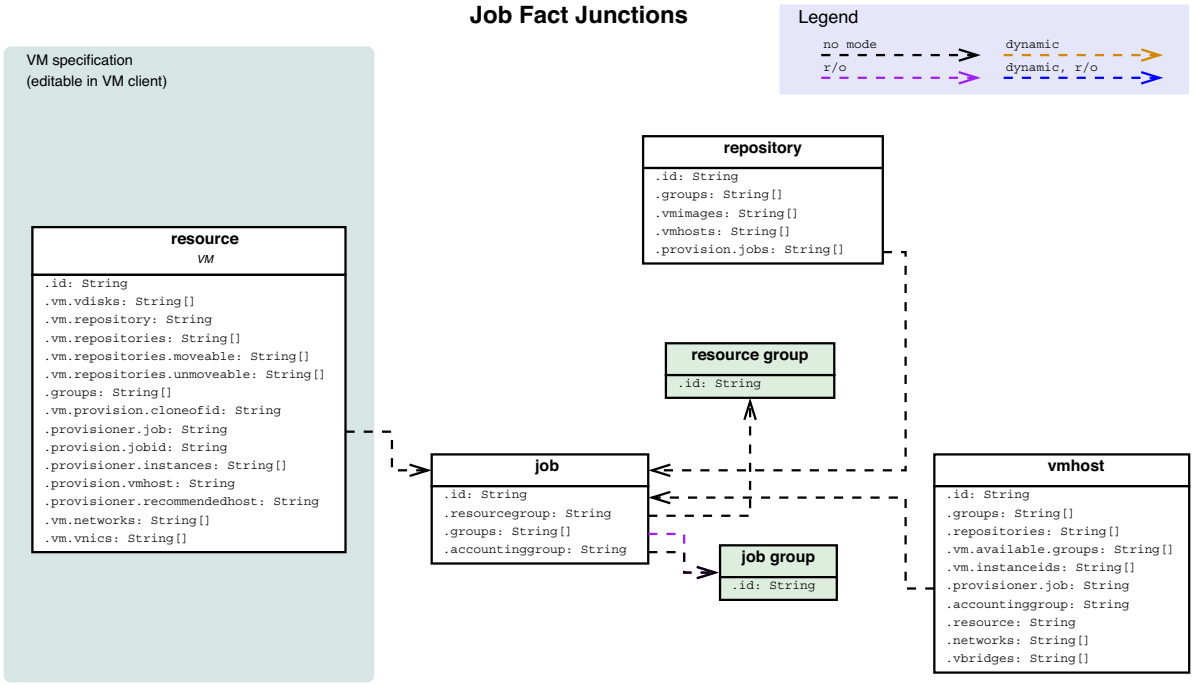
Fact Name	Fact Type	Mode	Description
job.provision.rankby	String[]		The ranking specification used to select suitable resources to automatically provision. Element syntax is <fact>/<order> where order is either ascending or descending.
job.queuedtimeout	Integer		The timeout (measured in seconds) after which the server takes action to cancel a queued job, including all joblets and subjobs. A value of -1 indicates no timeout.
job.queuetime	String		The average wall time (measured in seconds) spent waiting for this job to start . Used to override the computed average when queue is of type fixedtime.
job.queuetype	String		The type of queue from which this job is typically accessed. This is used in start time calculation (none, pfifo, fifo, lifo, fixedtime).
job.resourcegroup	String		The default resource group from which resources will be selected for this job (in addition to any resource policies).
job.resources.rankby	String[]		The ranking specification used to select suitable resources. Element syntax is <fact>/<order> where order is either ascending or descending.
job.restartable	Boolean		Indicates whether this job is willing to be restarted on server restarts (can be overridden by the job instance).
job.timeout	Integer		The timeout (measured in seconds) after which the server will take action to cancel the whole job including all joblets and subjobs. A value of -1 indicates no timeout.
job.tracing	Boolean		Indicates whether this job should include tracing information in the job log when executing job events.

Fact Name	Fact Type	Mode	Description
`job.visible`	Boolean		Whether this job should be visible to the client in queries or UI (it does not stop it being run).

Job Object Fact Junctions

The following diagram illustrates the relationship between the Job Grid object facts and other Grid objects. It also shows the relationship between other discrete Grid object facts and the Job Grid object itself.

Figure 4-2 *Job Fact Junctions*



Job Group Facts

Table 4-3 *Job Group Facts*

Fact Name	Fact Type	Mode	Description
`group.id`	String	r/o	The name of the group.
`group.jobinstances.active`	Integer	dynamic, r/o	The number of job instances that are active on resources in this group ('running' or 'paused').
`group.jobinstances.queued`	Integer	dynamic, r/o	The number of job instances that are in a queued state awaiting start on resources in this group.
`group.jobinstances.total`	Integer	dynamic, r/o	The total number of job instances running on, or awaiting start on resources in this group.

Jobinstance Facts

A job instance is a currently running or recently completed job. All jobinstance facts are viewable only in the Policy Debugger page of the Jobs Monitor. You need to select the *All Facts* check box to view these facts. These facts are not editable.

NOTE: If the job you want to view finished previously, it is possible that it can no longer be viewed in the Policy Debugger if other jobs followed it on the Orchestration Server.

Table 4-4 *Jobinstance Facts*

Fact Name	Fact Type	Description
`jobinstance.childids`	String[]	String array of child job IDs. If no child jobs were launched, the arrayis empty.
`jobinstance.cost`	Real	The cost (measured in dollars) of this job.
`jobinstance.cost.burnrate`	Real	The computed moving average burn rate (measured in dollars per hour) of the job.
`jobinstance.errors`	String	The error messages recorded for a failed job.
`jobinstance.id`	String	The job instance unique identifier
`jobinstance.instancename`	String	The optional, human readable name for this job instance.
`jobinstance.joblet.pspace`	Integer	The number of rows in a fully expanded p-space definition. Will be equal to the number of joblets only if `jobinstance.joblet.size` is `1`.

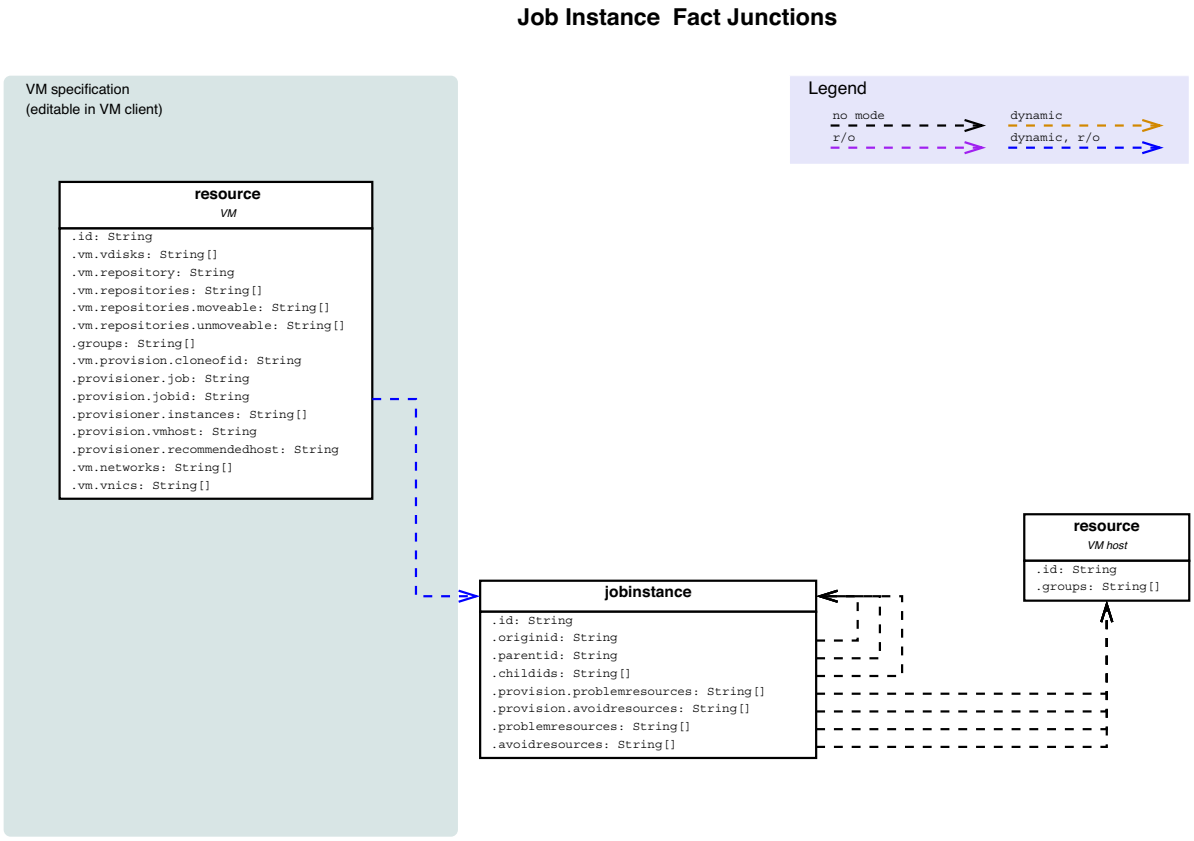
Fact Name	Fact Type	Description
`jobinstance.joblet.size`	Integer	The number of p-space rows encapsulated in each joblet. May be explicitly set of derived as a result of specifing joblet count
`jobinstance.joblets.cancelled`	Integer	The number of joblets that ended in a cancelled state.
`jobinstance.joblets.complete`	Integer	Number of joblets that completed successfully.
`jobinstance.joblets.count`	Integer	The number of joblets under management of this job instance.
`jobinstance.joblets.failed`	Integer	The number of joblets that ended in a failed state.
`jobinstance.joblets.running`	Integer	The number of joblets that are actively running on resources.
`jobinstance.joblets.waiting`	Integer	The number of joblets that are waiting for available resources.
`jobinstance.matchingresources`	Integer	The number of currently active resources that match the `resource` contraints.
`jobinstance.memo`	String	A brief memo set by this job instance that can be seen in user/administrator portals.
`jobinstance.originid`	String	The job identifier of the root job of this hierarchy or this job if a top level job.
`jobinstance.parentid`	String	The job identifier of the parent job of this hierarchy or empty if this is a top level job with no parent.
`jobinstance.preemptible`	Boolean	Specifies that this job instance is willing to give up resources if required. Initially set from the equivalent job fact.
`jobinstance.priority`	Integer	The numeric representation of the current priority of this job instance (1=lowest, 9=highest). Default value is the user's default priority.
`jobinstance.priority.string`	String	The string representation of the current priority of this job instance.
`jobinstance.privileged`	Boolean	Indicates whether the current workflow is running as a privileged job, and if so, that its joblets can be run in a resource's extra system joblet slots. The fact value is true when either of the following conditions are met: ◆ The job instance represents a *Start*, *Stop*, *Pause*, *Suspend*, *Resume*, or *CheckStatus* provisioning action. ◆ The job belongs to a job group that is referenced by the user's privileged job group (`user.privilegedjobgroups`) fact.

Fact Name	Fact Type	Description
jobinstance.problemresources	String[]	The number of resources that are excluded from this job instance due to reaching the job.maxnodefailures limit.
jobinstance.provision.avoidresources	String[]	The names of automatically provisioned resources that are currently being avoided (not yet excluded) because of prior provisioning errors.
jobinstance.provision.count	Integer	The total number of resources that have been automaticaly provisioned (or are in progress) for this job.
jobinstance.provision.pending	Integer	The total number of automatically provisioned resources that are pending online status.
jobinstance.provision.problemresources	String[]	The names of automatically provisioned resources that encountered provisioning errors and have been excluded.
jobinstance.provision.problemresources.count	Integer	The number of automatically provisioned resources that encountered provisioning errors and have been excluded.
jobinstance.resources	String[]	The resources currently in use by this job
jobinstance.resources.count	Integer	Number of resources currently in use by this job
jobinstance.restarted	Boolean	Whether this job instance was been restarted due to server restart.
jobinstance.startat	Date	The date/time that this job was requested to start atc
jobinstance.starttime		The formated start date and time for this job instance
jobinstance.state	Integer	The numeric state of this job instance
jobinstance.state.string	String	String representation of the current state of this job instance
jobinstance.terminationtype	String	The cause of the termination for a cancelled or failed job.
jobinstance.time.completed	Date	The time this job instance completed or an estimation if still active
jobinstance.time.elapsed	Integer	The elapsed wall time this job instance has been running or ran (in seconds)
jobinstance.time.elapsed.string	String	The elapsed wall time this job instance has been in a running
jobinstance.time.queued	Integer	The elapsed wall time this job instance has been a queued (in second)s
jobinstance.time.started	Date	The time this job instance was actually started or an estimating if queued
jobinstance.time.submitted	Date	The time this job instance was submitted

Jobinstance Fact Junctions

The following diagram illustrates the relationship between the Jobinstance facts and other Grid objects.

Figure 4-3 *Jobinstance Fact Junctions*



Joblet Facts

Joblet facts can be accessed only if you write code within the Joblet subclass of a JDL file. If you code a job to expose the joblet fact values, the Orchestration Server runs the scheduled joblets and you can see the joblet fact values in the Job Log tab of the Orchestration Console.

The available joblet facts and their descriptions are listed in the following table.

Table 4-5 *Joblet Facts*

Fact Name	Description
`joblet.autoterminate`	Whether the joblet ends when all events for the joblet ends.
`joblet.errors`	The list of error dictionaries encapsulating the error history for this joblet. Dictionary keys: ◆ **ts:** timestamp in milliseconds ◆ **node:** the node name where execution failed ◆ **error:** the error message

Fact Name	Description
joblet.history	The list of resource IDs where the joblet has run.
joblet.id	The unique identifier for this joblet.
joblet.instancename	A human readable name for this joblet instance.
joblet.memo	An (optional) memo field for this joblet that can be displayed in the management console.
joblet.number	The joblet number.
joblet.preemptible	Indicates whether this joblet is willing or able to be preempted.
joblet.retrynumber	The number of retries for this joblet (0 on first attempt).
joblet.state	The numeric state of this joblet instance.
joblet.state.string	String representation of the current state of this joblet instance.
joblet.timeout	The time after which this joblet will be cancelled/retried. (seconds, defaults to job.joblet.timeout).

4.1.4 Resource Object Facts and Fact Junctions

This section includes the following information:

Resource Object Facts

The Resource object (a physical or virtual machine) has an associated set of facts and constraints that define its properties. The Orchestration Server assigns default values to each of the component facts, although they can be changed at any time by the administrator, unless they are read-only. Facts with mode r/o have read-only values, which can be viewed but changes cannot be made.

The following table lists the default facts created by the Orchestration Server for the Resource object.

NOTE: Facts with mode dynamic are dyamic read/write facts, which means you can dynamically change the values for that fact.

Facts with mode r/o have read-only values, which means they can be viewed but changes cannot be made.

Facts with mode del are deleteable, which means they can be deleted at any time.

Table 4-6 *Resource Facts*

Fact Name	Fact Type	Mode	Description
`resource.agent.clustered`	Boolean		Whether the agent is clustered (converts duplicate logins to failover logins)
`resource.agent.config.datagrid.cache`	Boolean		Whether the agent caches datagrid files.
`resource.agent.config.datagrid.cache.cleanupinterval`	Integer		Wait time (measured in minutes) between cleanup sweeps of the datagrid cache.
`resource.agent.config.datagrid.cache.lifetime`	Integer		How long inactive files should remain in the agent's datagrid cache (in minutes). 0 turns off the cache.
`resource.agent.config.exec.asagentusernly`	Boolean		Whether the agent always runs executables only as the agent user. This overrides any Job fact settings ('job.joblet.runtype').
`resource.agent.config.exec.daemon.timeout`	Integer		How long for the enhanced exec daemon to remain running (in seconds). 0 means remain running.
`resource.agent.config.exec.enhancedused`	Boolean		Whether the agent uses the enhanced exec or not.
`resource.agent.config.gmond.port`	Integer		Setting for gmond port (8649 is the default). 0 or less means values will not be read.
`resource.agent.config.joblet.cleanup`	Boolean		Whether the agent cleans up temporary directories created for each joblet. Can be turned off for debugging purposes and then “catches up” when re-enabled.
`resource.agent.config.logdebug`	Boolean		Turns on agent debugging (additive to log level)
`resource.agent.config.loglevel`	String		Controls the verbosity of agent logging (quiet, normal, verbose)

Fact Name	Fact Type	Mode	Description
resource.agent.exec.installed	Boolean		Whether the agent is installed with the enhanced exec or not.
resource.agent.home	String		The home directory of the agent install.
resource.agent.jvm.home	String		The home directory of the JVM in use by the agent.
resource.agent.jvm.memory	Integer		The memory available to the agent (megabytes).
resource.agent.jvm.runtime	String		The Java JVM runtime in use by the agent.
resource.agent.jvm.vendor	String		The Java JVM vendor in use by the agent.
resource.agent.jvm.version	String		The Java JVM version in use by the agent.
resource.agent.version	String	dynamic, r/o	The agent version and build number <major>.<minor>.<point>_<build>
resource.auth.provider	String	dynamic, r/o	The authentication provider to which the node authenticates
resource.becameidle	Date	dynamic, r/o	The date/time the resource became idle or '-1' if not idle
resource.billfor	String		Whether to bill on wall time or grid time (walltime, gcycles)
resource.billingrate	Real		The billing rate for this resources in $/hr
resource.cpu.architecture	String	del	The CPU architecture e.g. x86, x86_64, sparc (requires cpuInfo system job).
resource.cpu.hvm	Boolean	del	True if the CPU has hardware virtualization support.
resource.cpu.mhz	Integer	del	The speed (measured in MHz) of the processor (requires cpuInfo system job)
resource.cpu.model	String	del	The CPU model number (requires cpuInfo system job)

Fact Name	Fact Type	Mode	Description
resource.cpu.number	Integer		The number of available CPU cores available for processing. This counts each core in a multicore package as its own core, so a Core 2 duo physical CPU displays as two CPUs.
resource.cpu.vendor	String	del	The CPU vendor (requires cpuInfo system job).
resource.cpuload	Integer	dynamic, r/o	The percent CPU load on the resource.
resource.enabled	Boolean		True if the resource is enabled (allowed to log in and accept work).
resource.groups	String[]	r/o	The groups this node is a member of.
resource.hardware.model	String		The vendor-specific hardware model (for example, Dell PowerEdge 1850).
resource.hardware.vendor	String		The name of the hardware vendor (for example, Dell, IBM, or similar).
resource.health	Boolean		The health of this resource. True indicates good health.
resource.history.cost.total	Real	dynamic, r/o	The cost (measured in matrix dollars ($)) of all work performed on this resource.
resource.history.gcycles.total	Integer	dynamic, r/o	The total grid time (gcycles) of all work performed on this resource.
resource.history.jobletcount	Integer	dynamic, r/o	The total number of joblets ever run on this resource.
resource.history.jobletcount.cancelled	Integer	dyanmic, r/o	The total number of joblets that were canceled.
resource.history.jobletcount.completed	Integer	dynamic, r/o	The total number of joblets that completed successfully.
resource.history.jobletcount.failed	Integer	dynamic, r/o	The total number of joblets that failed.

Fact Name	Fact Type	Mode	Description
resource.history.time.total	Integer	dynamic, r/o	The total wall time (measured in seconds) of all work performed on this resource.
resource.hostname	String	del	The resource's hostname
resource.hostname.full	String	del	The fully qualified hostname.
resource.id	String	r/o	The node's login name
resource.installed.apps	String[]	del	The installed applications
resource.ip	String	del	The resources IP address
resource.joblets.active	Integer	dynamic, r/o	The number of joblets currently active on this resource
resource.joblets.maxslots	Integer	dynamic, r/o	The absolute maximum number of work slots availabe to regular and privileged joblets.
resource.joblets.slots	Integer		The number of regular joblets that this resource runs at one time.
resource.joblets.systemslots	Integer		The number of extra slots that will be made available to privileged "system" joblets.
resource.loadaverage	Real	dynamic, r/o	The load average on the resource (from 'uptime' or equivalent).
resource.memory.physical.available	Integer	del	The amount (measured in Mb) of free physical memory available on the resource.
resource.memory.physical.total	Integer	del	The total amount (measured in Mb) of physical memory on the resource.
resource.memory.swap.available	Integer	del	The amount (measured in Mb) of free swap space.

Fact Name	Fact Type	Mode	Description
resource.memory.swap.total	Integer	del	The total amount (measured in Mb) of external VM swap space configured on the host. Wwap space allows the hypervisor or OS to swap out infrequently used memory pages to disk or other storages to make it appear that virtual memory is larger than physical memory.
resource.memory.virtual.available	Integer	del	The amount of available (free) virtual memory (measured in Mb) on the system. This might be more than the amount of physical memor if the host hypervisor or Operating system supports paging of VM to disk or other swap storage.
resource.memory.virtual.total	Integer	del	The total amount (measured in Mb) of virtual memory on the resources.
resource.network.agent.address			The agent side IP address for the current connection
resource.network.agent.port			The agent side TCP port number for the current connection
resource.network.config.server.address			The IP address used by the agent to connect to the server
resource.network.config.server.hostname			The host name used by the agent to connect to the server
resource.network.config.server.port			The TCP port number used by the agent to connect to the server
resource.network.server.address			The server side IP address for the current connection
resource.network.server.port			The server side TCP port number for the current connection

Fact Name	Fact Type	Mode	Description
resource.network.throughput			The measured average network connection throughput in Mbits/sec to neighouring resource (requires netInfo system job)
resource.network.throughput.max			The maximium network connection speed in Mbits/sec (requires netInfo system job)
resource.online	Boolean	dynamic, r/o	True if the agent is online
resource.os.arch	String	del	The operating system architecture e.g. x86, amd64, i386, sparc
resource.os.family	String		The family of operating system (windows, linux, solaris, unix, aix, mac
resource.os.file.separator	String	del	The resource operating system file separator
resource.os.name	String		The name of the resource operating system
resource.os.type	String		Unique string identifier for each OS release (e.g. 'sles11')
resource.os.vendor	String		The operating system vendor (SuSE for SLES/ SLED)
resource.os.vendor.string	String	del	The operating system full identification string (requires osInfo system job)
resource.os.vendor.version	String	del	The vendor defined version number, for example, 10 for SUSE v10.
resource.os.version	String	del	The operating system version numberc
resource.os.version.string	String	del	The operating system vendor full identification string (requires osInfo system job)
resource.password	string	dynamic	The agents's login password

Fact Name	Fact Type	Mode	Description
resource.powerfactor	Real	dynamic, r/o	The normalized power index of this machine relative to a 2.0Ghz, Intel Pentium 4
resource.provision.automatic	Boolean	dynamic, r/o	Signifies that this resource was cloned/ provisioned automatically and thus will be shutdown/destroyed automatically as well
resource.provision.currentaction	String	dynamic, r/o	The current management action in progress on this provisionable resource
resource.provision.jobid	String	dynamic, r/o	The current or last job id that performed a provisioning action on this resource. Useful for viewing job log.
resource.provision.resync	Boolean	dynamic	Specifies that the provisioned resource's state needs to be resynced with the underlying provisioning technology and the next opportunity
resource.provision.state	String	dynamic, r/o	The current state of this provisioned instance ('down', 'suspended', 'up', 'paused') or 'unknown' if an admin action is currently being performed.
resource.provision.status	String	dynamic, r/o	The current descriptive status of the provisioned resource
resource.provision.template	String	dynamic, r/o	The id of the template resource that this instance was created from (if applicable)
resource.provision.time.hostwait	Integer	dynamic, r/o	The time (seconds) this resource has been waiting / waited for a suitable host
resource.provision.time.request	Date	dynamic, r/o	The time when the last provision (or other administrative action) request was made
resource.provision.time.shutdown	Date	dynamic, r/o	The time when the resource was last shutdown

Fact Name	Fact Type	Mode	Description
resource.provision.time.start	Date	dynamic, r/o	The time when the resource was last successfully provisionedc
resource.provision.vmhost	String	dynamic	The id of the host currently housing this provisioned resource
resource.provisionable	Boolean	dynamic, r/o	True if the resources is a provisionable type
resource.provisioner.autoprep.DNSServers			List of DNS servers for name lookup. This is only for cloning/personalize actions.
resource.provisioner.autoprep.DNSSuffixes			List of suffixes to append to a name for lookup. This is only for cloning/ personalize actions.
resource.provisioner.autoprep.Gateways			List of internet gateways available to this VM. This is only for cloning/ personalize actions.
resource.provisioner.autoprep.linuxglobal.ComputerName			Host name of new VM. An asterisk (*) means use the new VM's ID.
resource.provisioner.autoprep.linuxglobal.Domain			The name of the domain where the new new VM belongs.
resource.provisioner.autoprep.options.changeSID			The Windows Security ID. If true, sysprep generates a new Security ID.
resource.provisioner.autoprep.options.deleteAccounts			If true, removes all accounts from the destination VM. If false, existing accounts from the source VM are retained.
resource.provisioner.autoprep.sysprep.GuiRunOnce.Command			List of commands that r un the first time a user logs on after the new VM is created. Commands are sched uled using the HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce registry key.
resource.provisioner.autoprep.sysprep.GuiUnattended.AdminPassword.plainText			True if the AdminPassword is plain text.

Fact Name	Fact Type	Mode	Description
resource.provisioner.autoprep.sysprep.GuiUnattended.AdminPassword.value			The AdminPassword.
resource.provisioner.autoprep.sysprep.GuiUnattended.AutoLogon			If true, the VM auto logs into the Administrator account using AdminPassword. If false, login is prompted..
resource.provisioner.autoprep.sysprep.GuiUnattended.AutoLogonCount			The limit count for the VM to auto log in with the Administrator account. `AutoLogon` must be True.
resource.provisioner.autoprep.sysprep.GuiUnattended.TimeZone			The time zone of the new VM. See provisioning reference for values, for example: `04` indicates PST, `10` indicates MST, `20` indicates CST, and `35` indicates EST.
resource.provisioner.autoprep.sysprep.Identification.DomainAdmin			Windows domain administrator name.
resource.provisioner.autoprep.sysprep.Identification.DomainAdminPassword.plainText			True if `DomainAdminPassword` is in plain text.
resource.provisioner.autoprep.sysprep.Identification.DomainAdminPassword.value			Windows domain administrator account password.
resource.provisioner.autoprep.sysprep.Identification.JoinDomain			Windows domain n ame. If joining a workgroup, then use JoinWorkgroup. For joining a domain, `Domai nAdmin` and `DomainAdminPassword` must be defined.
resource.provisioner.autoprep.sysprep.Identification.JoinWorkgroup			Windows workgroup name. If joining a domain, use `JoinDomain`.
resource.provisioner.autoprep.sysprep.LicenseFilePrintData.AutoMode			Value is either `PerServer` or `PerSeat`. If `PerServer`,`AutoUsers` must be set.
resource.provisioner.autoprep.sysprep.LicenseFilePrintData.AutoUsers			The number of client licenses. Used only if `AutoMode` is `PerServer`.

Fact Name	Fact Type	Mode	Description
resource.provisioner.autoprep.sysprep.UserData.ComputerName			The VM's new host name. An asterisk (*) means to generate the name based on source VM name.
resource.provisioner.autoprep.sysprep.UserData.FullName			The user's full name.
resource.provisioner.autoprep.sysprep.UserData.OrgName			The organization name.
resource.provisioner.autoprep.sysprep.UserData.ProductID			The Windows product key.
resource.provisioner.count	Integer	dynamic, r/o	The total count of operational instances and provisions in progress
resource.provisioner.debug	Boolean		Controls the debug log level in the provisioner
resource.provisioner.host.maxwait	Integer		The maximum time to wait for a suitable host before timing out (in seconds, '<0' to wait indefinitely
resource.provisioner.host.preferredwait	Integer		The time after which some vmhost constraints will be lifted to increase the available pool by, for example, considering moving the disk image (in seconds, '<0' to wait indefinitely
resource.provisioner.instancecount	Integer	dynamic, r/o	The total count of cloned instances of the template
resource.provisioner.instances	String[]	dynamic, r/o	The list of id's of the instances of this template resource (if applicable)
resource.provisioner.job	String		The name of the provisioning job that manages the lifecycle of this resource
resource.provisioner.maxinstances	Integer		The maximum allowed number of instances of this provisionable resource (applicable only to templates)c

Fact Name	Fact Type	Mode	Description
resource.provisioner.recommendedhost	String	dynamic	The host on which the image for this resource is associated. E.g. was suspended or is the preferred host for quick start up. Combined with the 'resource.provisioner.host .preferredwait' can lock a VM to one host.
resource.remotedesktop	Boolean		Whether the resource has a remote desktop (or VNC) access enabled.
resource.repositories			The list of VM repositories visible by this resource (aggregated from VM host containers)
resource.runningjobs	String[]	dynamic, r/o	The list of jobs currently running on this resource
resource.sessions	Integer	dynamic, r/o	The number of active sessions (resource instances with active agent). Will be 0 or 1 unless a resource template
resource.shuttingdown	Boolean	dynamic, r/o	True if the node is attempting to shutdown/ pause/suspend and does not want new workc
resource.type	String	dynamic	The type of resource (whether or not the resource is a VM and if so what type of image (physical, vm, vmTemplate)
resource.vm.basepath	String		The filesystem location of the VM file(s) either absolute or relative to the 'repository.location'.
resource.vm.cpu.architecture	String		The required cpu architecture e.g. x86, x86_64, sparc, ppcc
resource.vm.cpu.hvm	Boolean		True if the VM requires host HVM support (for para virtualization otherwise only full virtualization will be possible).

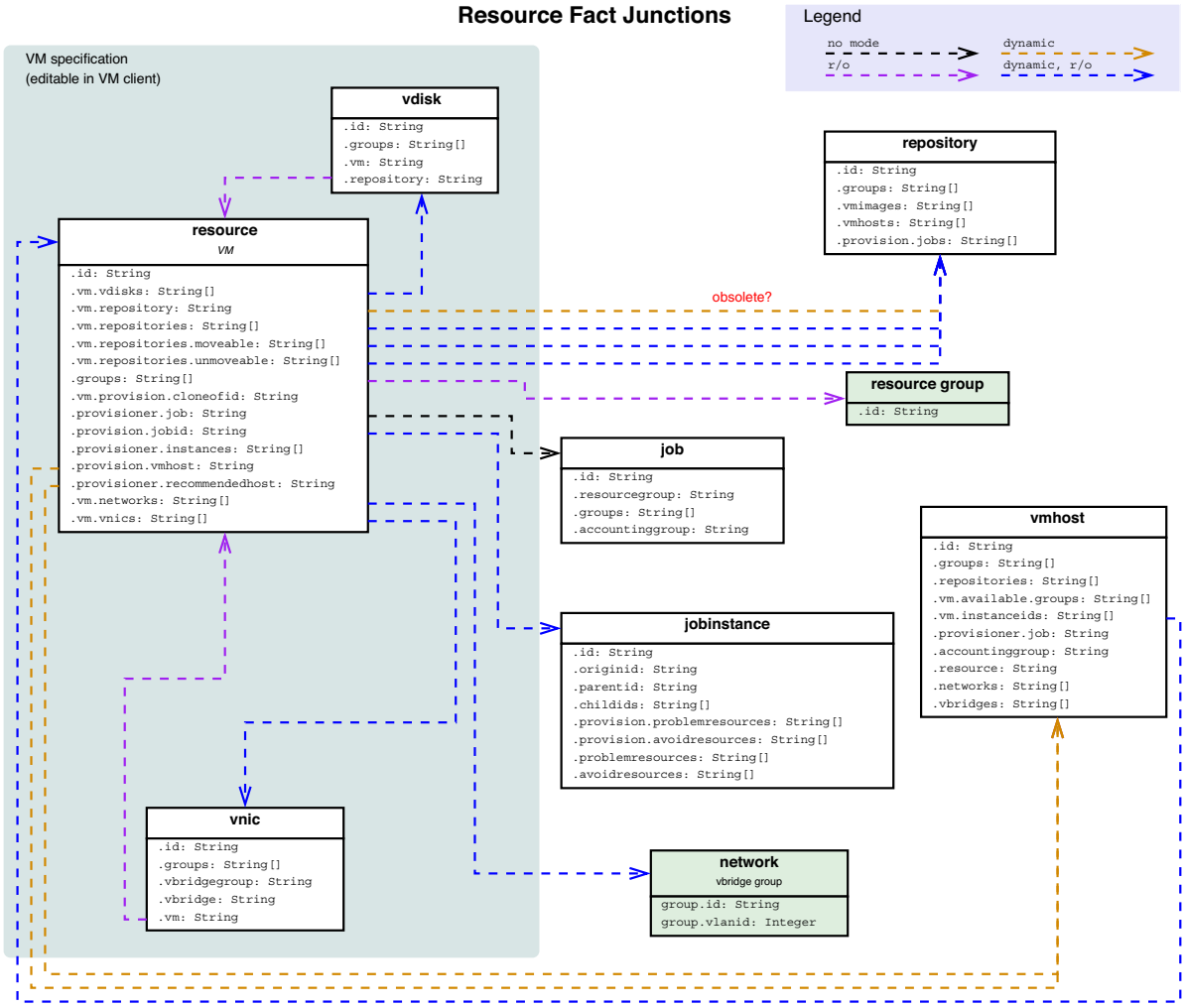
Fact Name	Fact Type	Mode	Description
resource.vm.cpu.weight	Real		The CPU weight for this VM. A value of 1.0 represents normal weighting; setting another VM to a weight of 2.0 would mean it would get twice as much cpu as this VM.
resource.vm.files	Dictionary		Files that make up this VM. The dictionary key (String) represents the file type (adapter specific), the value is the file path either absolute or relative to repository.location of the resource.vm.repository.
resource.vm.maxinstancespervmhost	Integer		The maximum allowed number of instances of this VM image per vmhost
resource.vm.memory	Integer		The configured virtual memory requirement of this VM image (megabytes)
resource.vm.networks	String[]	dynamic, r/o	The networks associated with the vm network interfaces.
resource.vm.preventmove	Boolean		Administrator set attribute that will prevent moving VM disks and thus consideration potential other hosts.
resource.vm.repositories	String[]	dynamic, r/o	The repositories where the vm disk images are stored.
resource.vm.repositories.moveable	String[]	dynamic, r/o	The repositories where the moveable vm disk images are stored.
resource.vm.repositories.unmoveable	String[]	dynamic, r/o	The repositories where the unmoveable vm disk images are stored.c
resource.vm.repository	String	dynamic	The default repository where this vm disk images and other config files are/will be stored.

Fact Name	Fact Type	Mode	Description
resource.vm.spec			Dictionary containing the specification for building this VM. Interpreted by the Provisioning Adapter.
resource.vm.underconstruction	Boolean		True if the VM is currently under construction
resource.vm.uuid	String		The UUID of a virtual machine (vendor/adapter specific).
resource.vm.vcpu.number	Integer		The number of virtual CPUs for this VM.
resource.vm.vdisks	String[]	dynamic, r/o	The list of virtual disks that make up this VM.c
resource.vm.vdisksize	Integer	dynamic, r/o	The total size of all the moveable virtual disks for this VM image (megabytes)
resource.vm.vendor	String		The vendor of a virtual machine
resource.vm.version	Integer		The version number for this VM.
resource.vm.vmhost.rankby	String[]		The ranking specification used to select suitable vm hosts. Element syntax is '<fact>/<order>' where order is either 'ascending' or 'descending'
resource.vm.vnics	String[]	dynamic, r/o	The list of virtual nics that make up this VM. (aggregated from the VNIC containers)
resource.vmhosts	String[]	dynamic, r/o	The list of VM host containers supported by this resource (aggregated from VM host containers)
resource.vnc.ip	String		The port number for a vnc session running on the resource.
resource.vnc.port	Integer		The port number for a vnc session running on the resource.

Resource Object Fact Junctions

The following diagram illustrates the relationship between the Resource Grid object facts and other Grid objects. It also shows the relationship between other discrete Grid object facts and the Resource Grid object itself.

Figure 4-4 *Resource Fact Junctions*



VM Host Object Facts

The VM Host Resource object has an associated set of facts and constraints that define its properties. The Orchestration Server assigns default values to each of the component facts, although they can be changed at any time by the administrator, unless they are read-only. Facts with mode r/o have read-only values, which can be viewed but changes cannot be made.

The following table lists the default facts created by the Orchestration Server for the VM Host Grid object.

NOTE: Facts with mode `dynamic` are dyamic read/write facts, which means you can dynamically change the values for that fact.

Facts with mode `r/o` have read-only values, which means they can be viewed but changes cannot be made.

Facts with mode `del` are deleteable, which means they can be deleted at any time.

Table 4-7 *VM Host Facts*

Fact Name	Fact Type	Mode	Description
`vmhost.accountinggroup`	String		The default VM host group which will be adjusted for VM stats
`vmhost.controllingjob`	String	dynamic	The ID of a running job that manages VM operations on this host. Setting this informs the VM Manager to prevent other jobs from initiating provisioning actions. This fact is cleared when the managing job ends.
`vmhost.enabled`	Boolean		True if the VM host is enabled (new VM instances can be provisioned)
`vmhost.groups`	String[]	r/o	The groups this VM host is a member of
`vmhost.health`	Boolean		The health of this VM host. True indicates good health
`vmhost.hvm`	Boolean		True if the hypervisor supports the hardware virtualization.
`vmhost.id`	String	r/o	The VM host's unique name
`vmhost.loadindex.slots`	Real	dynamic, r/o	The loading index (the ratio of active hosted VMs to the specified max)
`vmhost.loadindex.virtualmemory`	Real	dynamic, r/o	The loading index (ratio of consumed memory to the specifed max)
`vmhost.location`	String		The VM host's physical location
`vmhost.maxvmslots`	Integer		The maximum number of hosted VM instances
`vmhost.memory.available`	Integer	dynamic, r/o	The amount of memory available to new virtual machines
`vmhost.memory.max`	Integer		The maximum amount of memory available to virtual machines (in megabytes)
`vmhost.migration`	Boolean		True if the VM host can support VM migration (also subject to provision adapter cababilities

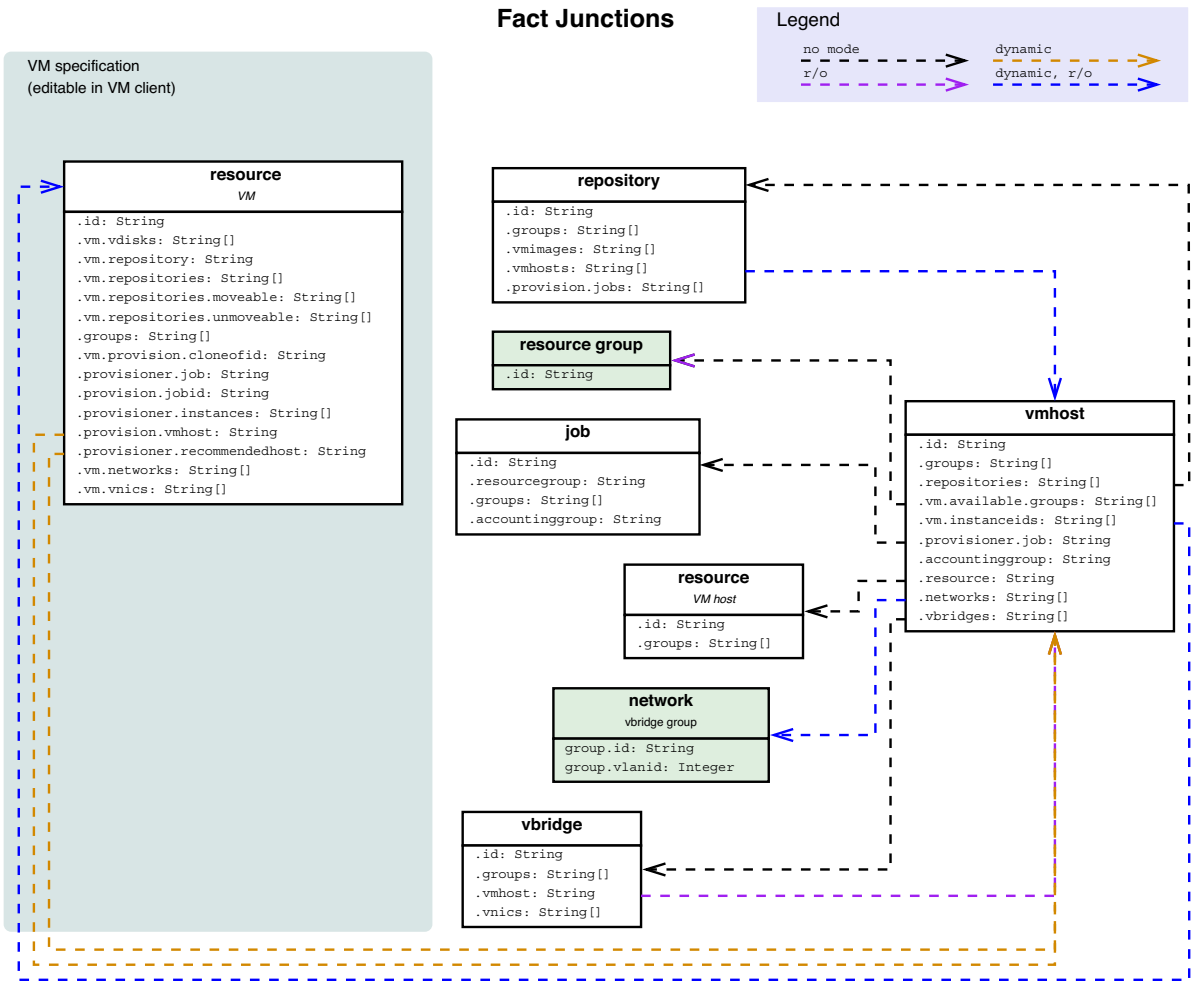
Fact Name	Fact Type	Mode	Description
vmhost.networks	String[]	dynamic, r/o	List of Networks visible to this VM host
vmhost.online	Boolean	dynamic, r/o	True if the agent on the physical host is online
vmhost.provisioner.job	String		The name of the provisioning adapter job that manages VM discovery on this host
vmhost.provisioner.password	String		The password required for provisioning on the VM host (Used by provisioning adapter)
vmhost.provisioner.username	String		The username required for provisioning on the VM host (Used by provisioning adapter)
vmhost.repositories	String[]		This list of repositories (VM disk stores) visible to this VM host
vmhost.resource	String	r/o	The name of the resource that houses this vm host container
vmhost.resync	Boolean	dynamic	Specifies that the host should be probed to resync all VMs managed on this host at the next opportunity
vmhost.shuttingdown	Boolean	dynamic, r/o	True if the VM host is attempting to shutdown and does not want to be provisioned
vmhost.vbridges	String[]	dynamic, r/o	List of Vbridge objects visible to this VM host
vmhost.vm.available.groups	String[]		The list of resource groups containing VMs that are allowed to run on this host
vmhost.vm.count	Integer	dynamic, r/o	The current number of active VM instances
vmhost.vm.instanceids	String[]	dynamic, r/o	The list of active VM instances
vmhost.vm.placement.score	Integer	dynamic, r/o	The 'cost' (score) of moving the disks for a VM to this vmhost. This is *only* visible and valid during a 'vmhost' constraint match (-1 = not possible, 0 = affinity, >0 = expense)

Fact Name	Fact Type	Mode	Description
vmhost.vm.templatecounts	Dictionary	dynamic, r/o	A dictionary of running instance counts for each running VM templatec

VM Host Resource Object Fact Junctions

The following diagram illustrates the relationship between the Vm Host Resource Grid object facts and other Grid objects. It also shows the relationship between other discrete Grid object facts and the VM Host Resource Grid object itself.

Figure 4-5 *VM Host Fact Junctions*



Resource Group Facts

Table 4-8 *Resource Group Facts*

Fact Name	Fact Type	Mode	Description
group.id	String	r/o	The group's name

Fact Name	Fact Type	Mode	Description
group.loadaverage	Real	dynamic, r/o	The aggregated load average of all the resource in this group (the membership may be dynamic).
group.loadpercent	Integer	dynamic, r/o	The percentage of online resources in this group that are currently busy.
group.resources.busy	Integer	dynamic, r/o	The total number of available resources that are currently busy performing work in this group.
group.resources.idle	Integer	dynamic, r/o	The total number of available resources that are ready for work in this group.
group.resources.online	Integer	dynamic, r/o	The total number of online resources (busy and idle) in this group.

Understanding Resource Metrics Facts

When you install the Orchestration Agent on a machine, you can optionally install the Cloud Manager Monitoring Agent along with it. The Monitoring Agent uses the Ganglia Monitoring Daemon (gmond) to automatically collect metrics and send them to the Cloud Manager Monitoring Server. You can use the following command to check the status of an installed Monitoring Agent:

```
# /etc/init.d/novell-gmond status
```

If the daemon is operating normally, it returns a `running` status.

When you install and configure the Cloud Manager Monitoring Agent (gmond), it is set by default to report metrics on port 8649, which is also detected by the Orchestration Agent. When communication is established, the gmond daemon sends out metrics data, which are then gathered by the Orchestration Agent and set as fact values associated with the resource where the daemon is running. You can verify the connection with the following command:

```
telnet localhost 8649
```

If gmond is running and communicating properly, an XML document listing the reported metrics is displayed.

This section includes information about the resource metrics facts that are gathered, the unit conversion performed by Cloud Manager Orchestration on the Ganglia-provided values, and how you can use these facts to help you manage the resources in the grid.

- "Resource Metrics Facts" on page 85
- "Interpreting the Units of Metrics Fact Values" on page 86

Resource Metrics Facts

The Orchestration Agent uses the metrics collected by gmond to create fact values for a given resource. These facts are therefore externally generated and are not among the default facts reported by the Orchestration Agent. The agent updates these externally generated fact values every 30 seconds. All of these fact values have a `resource.metrics.` prefix.

For example, gmond collects a metrics value called `load_one`. The Orchestration Agent sets this value as the `resource.metrics.load_one` fact.

To see a list of these facts in the Orchestration Console,

1. In the Explorer panel, select a resource.
2. In the Workspace panel, select *Constraints/Facts*.

The names of the resource metrics facts are displayed in bold font (in the Development Client interface) because they were added as new facts to the default fact list. The following sample is a list of the default Ganglia-generated metrics facts with data type and an example value:

```
<fact name="resource.metrics.boottime" value="1239122234.0000" type="Real" />
<fact name="resource.metrics.bytes_in" value="208.8800" type="Real" />
<fact name="resource.metrics.bytes_out" value="68.9700" type="Real" />
<fact name="resource.metrics.cpu_aidle" value="76.9000" type="Real" />
<fact name="resource.metrics.cpu_idle" value="95.2000" type="Real" />
<fact name="resource.metrics.cpu_nice" value="0.0000" type="Real" />
<fact name="resource.metrics.cpu_num" value="2" type="Integer" />
<fact name="resource.metrics.cpu_speed" value="1596" type="Integer" />
<fact name="resource.metrics.cpu_system" value="0.3000" type="Real" />
<fact name="resource.metrics.cpu_user" value="4.0000" type="Real" />
<fact name="resource.metrics.cpu_wio" value="0.4000" type="Real" />
<fact name="resource.metrics.disk_free" value="27090" type="Integer" />
<fact name="resource.metrics.disk_total" value="48213" type="Integer" />
<fact name="resource.metrics.gexec" value="OFF" type="String" />
<fact name="resource.metrics.load_fifteen" value="0.2000" type="Real" />
<fact name="resource.metrics.load_five" value="0.4100" type="Real" />
<fact name="resource.metrics.load_one" value="1.1900" type="Real" />
<fact name="resource.metrics.machine_type" value="x86" type="String" />
<fact name="resource.metrics.mem_buffers" value="299" type="Integer" />
<fact name="resource.metrics.mem_cached" value="761" type="Integer" />
<fact name="resource.metrics.mem_free" value="65" type="Integer" />
<fact name="resource.metrics.mem_shared" value="0" type="Integer" />
<fact name="resource.metrics.mem_total" value="1989" type="Integer" />
<fact name="resource.metrics.os_name" value="Linux" type="String" />
<fact name="resource.metrics.os_release" value="2.6.27.19-5-pae" type="String" />
<fact name="resource.metrics.part_max_used" value="70.8000" type="Real" />
<fact name="resource.metrics.part_max_used.units" value="" type="String" />
<fact name="resource.metrics.pkts_in" value="0.4500" type="Real" />
<fact name="resource.metrics.pkts_out" value="0.6300" type="Real" />
<fact name="resource.metrics.proc_run" value="0" type="Integer" />
<fact name="resource.metrics.proc_total" value="411" type="Integer" />
<fact name="resource.metrics.swap_free" value="2039" type="Integer" />
<fact name="resource.metrics.swap_total" value="2047" type="Integer" />
<fact name="resource.metrics.vm_type" value="" type="String" />
<fact name="resource.metrics.vm_type.units" value="" type="String" />
```

These are the metrics reported in Orchestration systems that use the `gmond.conf` created when the Cloud Manager Monitoring Agent was installed and configured. The open source gmond might include other metrics that can be monitored. You can modify the default Orchestration gmond configuration file to report these metrics after it is initially installed and configured. For information about modifying the file, see the gmond.conf man page (http://linux.die.net/man/5/gmond.conf).

By using the XML constraint language, you can utilize these resource metrics facts as you would use any other fact in Cloud Manager Orchestration. For example, you could create an Event that sets thresholds for the amount of incoming network packets. When that threshold is exceeded, a Scheduled Job could be triggered or a notification e-mail sent. For more information, see Section 3.12, "Using an Event Notification in a Job," on page 45.

Interpreting the Units of Metrics Fact Values

The Orchestration Agent converts most of the Ganglia metrics values to Cloud Manager Orchestration standard units. This allows fact values to be compared in constraints without the need to perform conversions explicitly. In cases where units are not known or cannot be converted, a separate fact with a `.units` suffix is included. For example:

```
<fact name="resource.metrics.bytes_in" value="bytes/sec" type="String" />
```

The following table lists the `resource.metrics` facts and the units of measure used for each fact value:

Table 4-9 *Resource Metrics Facts*

Resource Metric Fact With Reported Value	Orchestration Measurement Unit of the Value
boottime	32-bit seconds timestamp
bytes_in	byte rate measured in bytes per second
bytes_out	byte rate measured in bytes per second
cpu_aidle	percentage
cpu_idle	percentage
cpu_nice	percentage
cpu_num	number of CPUs
cpu_speed	megahertz as an integer
cpu_system	percentage
cpu_user	percentage
cpu_wio	percentage
disk_total	total in binary megabytes
disk_free	total in binary megabytes
gexec	simple string
load_fifteen	real number
load_five	real number
load_one	real number
machine_type	simple string
mem_buffers	memory in megabytes (integer)
mem_cached	memory in megabytes (integer)

Resource Metric Fact With Reported Value	Orchestration Measurement Unit of the Value
mem_free	memory in megabytes (integer)
mem_shared	memory in megabytes (integer)
mem_total	memory in megabytes (integer)
os_name	simple string
os_release	simple string
pkts_in	packet rate in packets per second
pkts_out	packet rate in packets per second
proc_run	processes run (integer)
proc_total	total processes (integer)
swap_free	memory in megabytes (integer)
swap_total	memory in megabytes (integer)

4.1.5 Virtual Disk Object Facts and Fact Junctions

This section includes the following information:

- "Virtual Disk Object Facts" on page 87
- "Virtual Disk Object Fact Junctions" on page 88

Virtual Disk Object Facts

The vDisk object has an associated set of facts and constraints that define its properties. The Orchestration Server assigns default values to each of the component facts, although they can be changed at any time by the administrator, unless they are read-only. Facts with mode r/o have read-only values, which can be viewed but changes cannot be made.

The following table lists the default facts created by the Orchestration Server for the vDisk Grid object.

NOTE: Facts with mode `dynamic` are dynamic read/write facts, which means you can dynamically change the values for that fact.

Facts with mode `r/o` have read-only values, which means they can be viewed but changes cannot be made.

Facts with mode `del` are deleteable, which means they can be deleted at any time.

Table 4-10 *vDisk Facts*

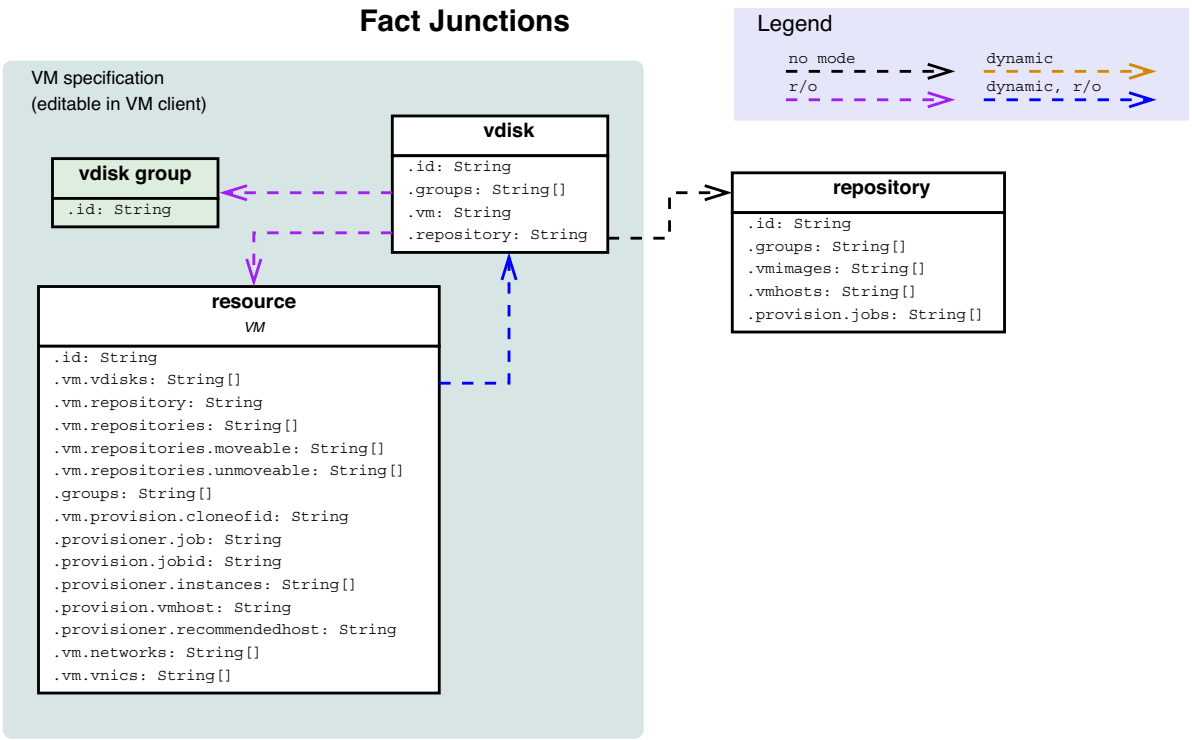
Fact Name	Fact Type	Mode	Description
`vdisk.description`	String		Description of vDisk
`vdisk.groups`	String[]	r/o	The groups this vDisk is a member of

Fact Name	Fact Type	Mode	Description
vdisk.health	Boolean		The health of this vDisk. True indicates good health
vdisk.id	String	r/o	The vDisk's unique ID.
vdisk.location	String		The repository dependent location definition
vdisk.moveable	Boolean		True if the vDisk is moveable The vDisk is not deleted if this fact is set to false.
vdisk.repository	String		The name of the repository containing the vDisk image
vdisk.size	Integer		The size of this virtual disks (megabytes)
vdisk.type	String		The type of vDisk: - file (file backed disk) - block (block device)
vdisk.vm	String	r/o	The name of the VM that uses this vDisk

Virtual Disk Object Fact Junctions

The following diagram illustrates the relationship between the Virtual Disk object facts and other Grid objects. It also shows the relationship between other discrete Grid object facts and the Virtual Disk Grid object itself.

Figure 4-6 *Virtual Disk Fact Junctions*



4.1.6 Virtual NIC Object Facts and Fact Junctions

This section includes the following information:

Virtual NIC Object Facts

The VNIC object has an associated set of facts and constraints that define its properties. The Orchestration Server assigns default values to each of the component facts, although they can be changed at any time by the administrator, unless they are read-only.

The following table lists the default facts created by the Orchestration Server for the VNIC Grid object.

NOTE: Facts with mode `dynamic` are dyamic read/write facts, which means you can dynamically change the values for that fact.

Facts with mode `r/o` have read-only values, which means they can be viewed but changes cannot be made.

Facts with mode `del` are deleteable, which means they can be deleted at any time.

Table 4-11 *VNIC Facts*

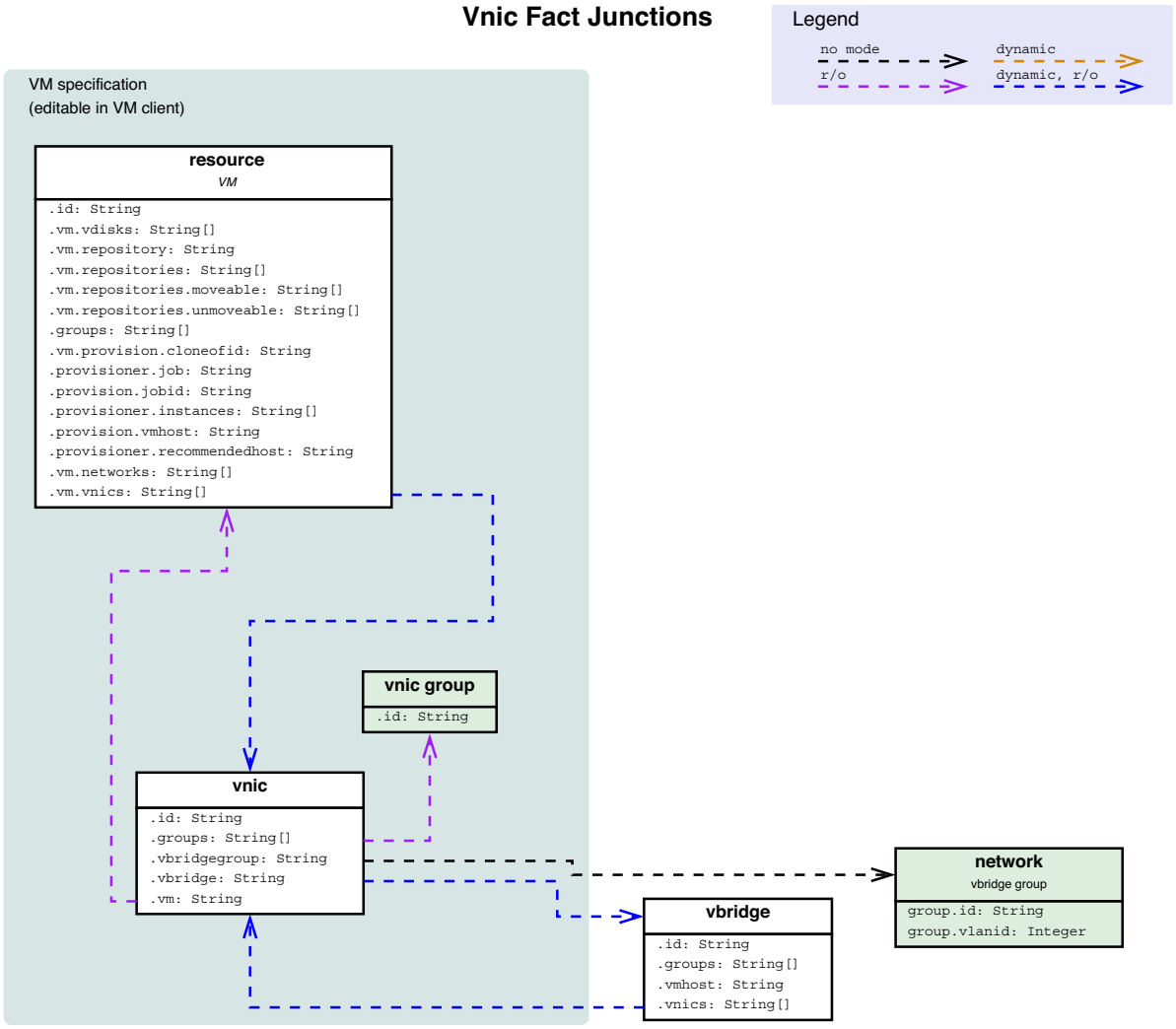
Fact Name	Fact Type	Mode	Description
`vnic.description`	String		Description of vNIC
`vnic.groups`	String[]	r/o	The groups that this vNIC belongs to.
`vnic.health`	Boolean		The health of this vNIC. True indicates good health
`vnic.id`	String	r/o	The unique name of this VNIC.
`vnic.mac`	String		The MAC address assigned to this VNIC. A empty string implies auto-generate MAC address.
`vnic.network`	String		The network (vbridge group) on which this Vnic is provisioned or wishes to be provisioned.
`vnic.provisioner.autoprep.DNSDomain`	String		Windows only. Adapter's Domain.
`vnic.provisioner.autoprep.DNSFromDHCP`	Boolean		(Optional, SUSE VM only.) If true, then the SUSE VM is configured to retrieve its DNS server settings from DHCP.

Fact Name	Fact Type	Mode	Description
vnic.provisioner.autoprep.DNSServers			Adapter's list of DNS servers for name look up.
vnic.provisioner.autoprep.DNSSuffixes			Adapter's suffix appended to name for lookup.
vnic.provisioner.autoprep.Gateways			List of Internet gateways available to the interface.
vnic.provisioner.autoprep.IPAddress	String	del	IP address for this adapter.
vnic.provisioner.autoprep.MACAddress	String	del	MAC address for the interface. Asterisk (*) or not set means to generate a new MAC.
vnic.provisioner.autoprep.UseDHCP	Boolean	del	If true, new VM retrieves its network settings from a DHCP server and any adapter settings are ignored. If false, then any req uired adapter settings must be defined.
vnic.provisioner.autoprep.netBIOS	String	del	NetBios options for VM. The values are: ◆ EnableNetBIOSViaDhcp ◆ EnableNetBIOS ◆ DisableNetBIOS
vnic.provisioner.autoprep.primaryWINS	String	del	Windows only. Adapter's Primary WINS server.
vnic.provisioner.autoprep.secondaryWINS	String	del	Windows only. Adapter's Secondary WINS server.
vnic.provisioner.autoprep.subnetMask	String	del	Subnet mask for this adapter.
vnic.vbridge	String	dynamic, r/o	The name of the Vbridge used by this vNIC
vnic.vm	String	r/o	The name of the VM that uses this vNIC

Virtual NIC Object Fact Junctions

The following diagram illustrates the relationship between the VNIC object facts and other Grid objects. It also shows the relationship between other discrete object facts and the VNIC object itself.

Figure 4-7 *Virtual NIC Fact Junctions*



4.1.7 Repository Object Facts and Fact Junctions

This section includes the following information:

Repository Object Facts

The Repository object has an associated set of facts and constraints that define its properties. The Orchestration Server assigns default values to each of the component facts, although they can be changed at any time by the administrator, unless they are read-only.

The following table lists the default facts created by the Orchestration Server for the Repository Grid object.

NOTE: Facts with mode `dynamic` are dyamic read/write facts, which means you can dynamically change the values for that fact.

Facts with mode `r/o` have read-only values, which means they can be viewed but changes cannot be made.

Facts with mode `del` are deleteable, which means they can be deleted at any time.

Table 4-12 *Repository Facts*

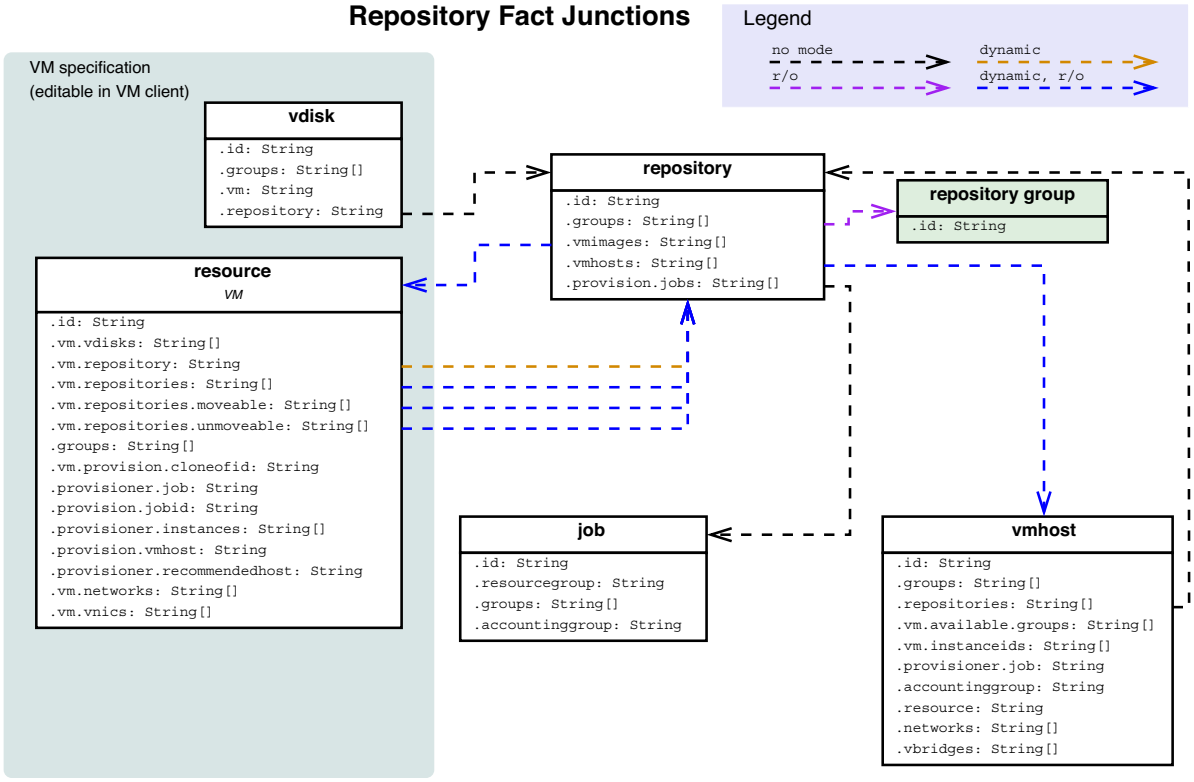
Fact Name	Fact Type	Mode	Description
`repository.capacity`	Integer		The maximum amount of storage space available to virtual machines (in megabytes). `-1` means unlimited.
`repository.description`	String		Description of repository
`repository.efficiency`	Real		The efficiency coefficient used to calculate the cost of moving VM disk images to and from the repository. This value is multiplied by the disk image size in Mb to determine score (thus `0` means no cost -- very efficient).
`repository.enabled`	Boolean		True if the Repository is enabled (new VM instances can be provisioned)
`repository.freespace`	Integer	Dynamic r/o	The amount of storage space available to new virtual machines (in megabytes). `-1` means unlimited.
`repository.groups`	String[]	r/o	The groups to which this repository belongs.
`repository.health`	Boolean		The health of this repository. True indicates good health
`repository.id`	String	r/o	The Repository's unique name
`repository.location`	String		The Repository's logical root location, for example, `/` or `c:/vm` or `nas:/vol1`
`repository.preferredpath`	String		The relative path from `repository.location` to search and place VM files for movement and cloning

Fact Name	Fact Type	Mode	Description
repository.provisioner.jobs	String[]		The names of the provisioning adapter jobs that can manage VM's on this repository
repository.san.type	String		The type of SAN (Adapter specific, iSCSI or Fibre Channel)
repository.san.vendor	String		The vendor of SAN (Adapter specific, iqn, npiv, emc). An empty string indicates bind/unbind is a noop.
repository.searchpath	String[]		The relative path from repository.location to search for VM configuration files (implicitly includes resource.preferredpath)
repository.type	String		The type of repository: ◆ local (for example, local disk) ◆ NAS (for example, NFS mount) ◆ SAN (a Storage area Network such as iSCSI or Fibre Channel) ◆ datagrid (an Orchestration built-in, datagrid-backed store) ◆ virtual (an externally managed store such as VMware Virtual Center)
repository.usedspace	Integer	Dynamic r/o	The amount of storage space used for virtual machines
repository.vmhosts	String[]	Dynamic r/o	The amount of storage space used for virtual machines
repository.vmimages	String[]	Dynamic r/o	The list of VM images stored in this repository (aggregated from individual VM fact)

Repository Object Fact Junctions

The following diagram illustrates the relationship between the Repository object facts and other Grid objects. It also shows the relationship between other discrete Grid object facts and the Repository object itself.

Figure 4-8 *Repository Fact Junctions*



Repository Group Facts

Table 4-13 *Repository Group Facts*

Fact Name	Fact Type	Mode	Description
`group.id`	String	r/o	The name of the group.

4.1.8 Virtual Bridge Object Facts and Fact Junctions

This section includes the following information:

- "Virtual Bridge Object Facts" on page 95
- "Virtual Bridge Object Fact Junctions" on page 95
- "Network Group Facts" on page 96

Virtual Bridge Object Facts

The VNIC object has an associated set of facts and constraints that define its properties. The Orchestration Server assigns default values to each of the component facts, although they can be changed at any time by the administrator, unless they are read-only.

The following table lists the default facts created by the Orchestration Server for the Vbridge Grid object.

NOTE: Facts with mode `dynamic` are dyamic read/write facts, which means you can dynamically change the values for that fact.

Facts with mode `r/o` have read-only values, which means they can be viewed but changes cannot be made.

Facts with mode `del` are deleteable, which means they can be deleted at any time.

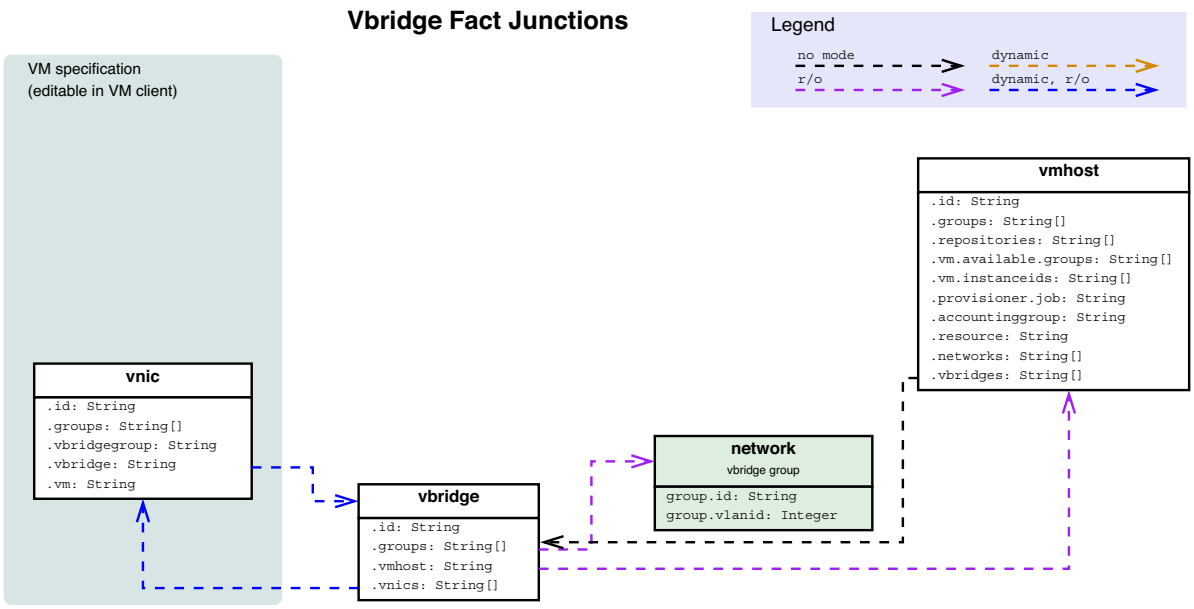
Table 4-14 *Vbridge Facts*

Fact Name	Fact Type	Mode	Description
`vbridge.description`	String		Description of Vbridge
`vbridge.enabled`	Boolean		True if the vbridge is enabled
`vbridge.groups`	String[]	r/o	The groups this Vbridge is a member of
`vbridge.health`	Boolean		True if the vbridge is in a healthy state
`vbridge.id`	String	r/o	The unique identifier for the Vbridge.
`vbridge.vmhost`	String	r/o	The ID of the vmhost containing this vbridge
`vbridge.vnics`	String[]	dynamic, r/o	The virtual NICs attached to this vbridge

Virtual Bridge Object Fact Junctions

The following diagram illustrates the relationship between the Virtual Bridge (Vbridge) object facts and other Grid objects. It also shows the relationship between other discrete Grid object facts and the Vbridge object itself.

Figure 4-9 *Virtual Bridge Fact Junctions*



Network Group Facts

Table 4-15 *Network Group Facts*

Fact Name	Fact Type	Mode	Description
`group.id`	String	r/o	The group name.
`group.provisioner.jobs`	String[]	dynamic, r/o	Provision adapters with vBridge instances on this network.
`group.provisioner.preferred`	String[]		Provision adapters explicitly associated with this network.
`group.vlanid`	String		The name of the VLAN accessed by this network (Vbridge group).

4.1.9 User Object Facts and Fact Junctions

This section includes the following information:

- "User Object Facts" on page 97
- "User Object Fact Junctions" on page 100
- "User Group Facts" on page 100

User Object Facts

The User object has an associated set of facts and constraints that define its properties. The Orchestration Server assigns default values to each of the component facts, although they can be changed at any time by the administrator, unless they are read-only.

The following table lists the default facts created by the Orchestration Server for the User Grid object.

NOTE: Facts with mode `dynamic` are dyamic read/write facts, which means you can dynamically change the values for that fact.

Facts with mode `r/o` have read-only values, which means they can be viewed but changes cannot be made.

Facts with mode `del` are deleteable, which means they can be deleted at any time.

Table 4-16 *User Facts*

Fact Name	Fact Type	Mode	Description
`user.account.balance`	Real	dynamic	The amount of matrix dollars spent by this user since last reset ($). Useful for implementation of quotas
`user.account.gcycles`	Integer	dynamic	The amount of grid time (gcycles) spent by this user since last reset. Useful for implementation of quotas
`user.account.maxspendrate`	Real		This value is used by the resource scheduler to throttle the rate at which matrix computing cycles are comsumed by the user ($/hr, <=0 turns feature off)
`user.account.spendrate`	Real	dynamic, r/o	The computed moving average spending over the last hour of user activity ($/hr)
`user.account.time`	Integer	dynamic	The amount of wall time spent by this user since last reset. Useful for implementation of quotas
`user.accountinggroup`	String		The default user group which will be billed for work conducted by this user
`user.auth.provider`	String		The authentication provider to which the user authenticates
`user.datagrid.maxhistory`	Integer		The maximum number job instance directories that should be kept in the datagrid for this user
`user.enabled`	Boolean		True if the user is enabled (allowed to log in and run jobs)

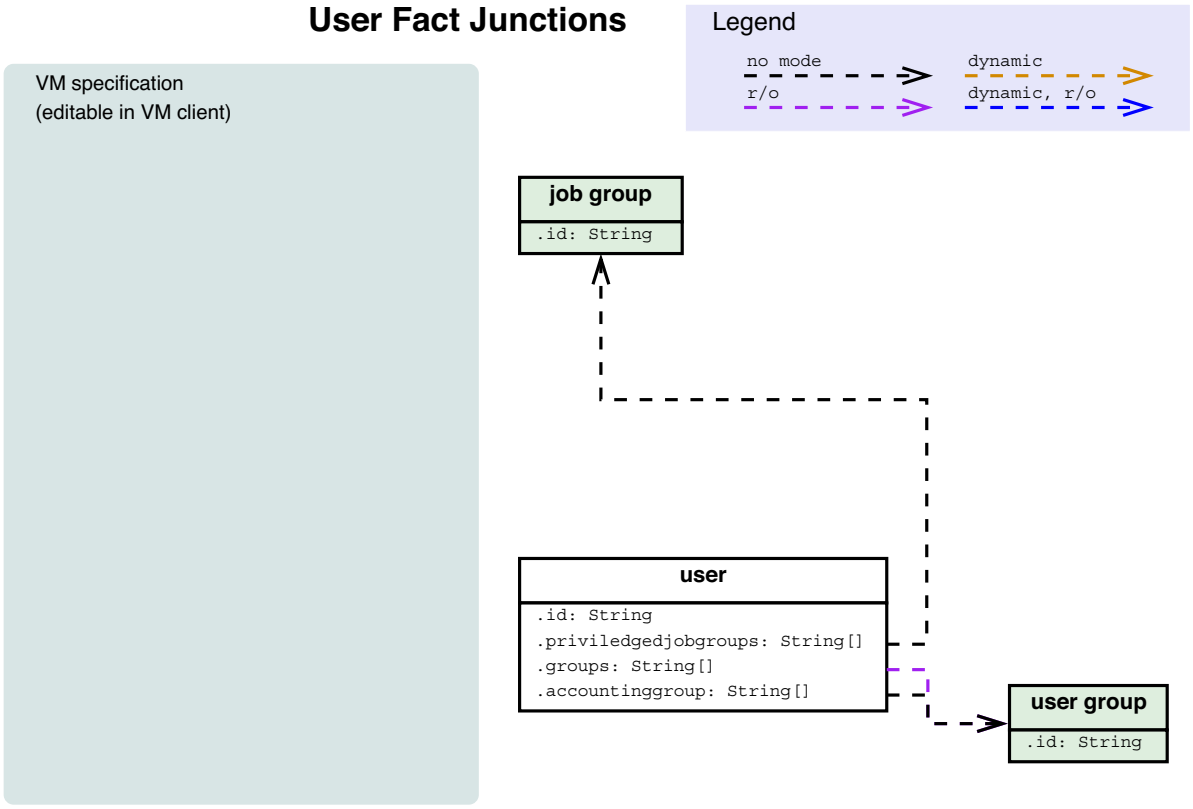
Fact Name	Fact Type	Mode	Description
user.env	Dictionary		The uploaded and default user environment
user.external.groups	String[]	dynamic, r/o	The external groups (e.g. LDAP) this user is a member ofc
user.groups	String	r/o	The groups this user is a member of
user.health	Boolean		The health of this user. True indicates good health
user.history.cost.total	Real	dynamic, r/o	The total amount of matrix dollars spent by this user on the matrix
user.history.gcycles.total	Integer	dynamic, r/o	The total amount of grid time (gcycles) spent by this user on the matrix
user.history.jobcount	Integer	dynamic, r/o	The total number of jobs a user has ever initiated on the matrix
user.history.time.total	Integer	dynamic, r/o	The total amount of wall time spent by this user on the matrix
user.id	String	r/o	The user's login name
user.jobcount	Integer	dynamic	A counter that records the number of jobs a user has initiated since last reset. Useful for implementation of quotas
user.jobs.active	Integer	dynamic, r/o	The number of top-level jobs a user has in an active state
user.jobs.queued	Integer	dynamic, r/o	The number of top-level jobs that are queued for this user
user.jobs.total	Integer	dynamic, r/o	The total number of top-level jobs a user currently has in the matrix
user.location.city	String		The city of location of the user
user.location.country	String		The country of location of the user
user.location.site	String		The site where the users work, for example, a building number.
user.location.state	String		The state of location of the user
user.name.email	String		The user's email address
user.name.first	String		The user's first name
user.name.last	String		The user's last name
user.online	Boolean	dynamic. r/o	Whether the user is currently logged into the matrix
user.password	String	dynamic	The user's login password

Fact Name	Fact Type	Mode	Description
user.preemption.enabled	Boolean		Allows this user to preempt willing jobs of a priority less than the priority of the running job instance
user.preemption.priority.delta	Integer		Indicates the highest job priority band that this user is allowed to preempt resources from (acts as a delta from the current job instance priority). The maximum preemptible priority is always <= user.priority.max
user.priority.default	Integer		The numeric representation of the default priority at which this user's jobs will run (1=lowest, 9=highest)
user.priority.default.string	String	dynamic, r/o	The string representation of the default priority at which this user can run a job (matches user.priority.default)
user.priority.max	Integer		The numeric representation of the maximum priority that this user can run a job at (1=lowest, 9=highest)
user.priority.max.string	String	dynamic, r/o	The string representation of the maximum priority at which this user can run a job (matches user.priority.max)
user.privilegedjobgroups	String[]		Allows this user's to run jobs and thus joblets on resources that have reached their slot maximum or are provisioned resources that have been reserved for another user/job. This is primarily to allow discovery jobs to be 'squeezed' onto the resource
user.sessions	Integer	dynamic, r/o	The number of active sessions (connections) a user currently has with the matrix
user.stealing.enabled	Boolean		Allows this user to steal resources running jobs of a priority less than the priority of running job instance
user.stealing.priority.delta	Integer		Indicates the highest job priority band that this user is allowed to steal resources from (acts as a delta from the current job instance priority, must be < '0')

User Object Fact Junctions

The following diagram illustrates the relationship between the User object facts and other Grid objects. It also shows the relationship between other discrete Grid object facts and the User object itself.

Figure 4-10 *User Fact Junctions*



User Group Facts

Table 4-17 *User Group Facts*

Fact Name	Type	Mode	Description
`group.account.balance`	Real	dynamic	he aggregated cost of work done by members of this group since last reset ($).
`group.account.gcycles`	Integer	dynamic	The aggregated amount of grid time (gcycles) spent by members of this group since last reset.
`group.account.time`	Integer	dynamic	The aggregated amount of wall time spent by members of this group since last reset
`group.id`	String	r/o	The name of the group.

Fact Name	Type	Mode	Description
group.jobcount	Integer	dynamic	The aggregated number of jobs run by members of this group since last reset.

4.1.10 Matrix Object Facts

The Matrix object has an associated set of facts and constraints that define its properties. The Orchestration Server assigns default values to each of the component facts, although they can be changed at any time by the administrator, unless they are read-only.

The following table lists the default facts created by the Orchestration Server for the Matrix Grid object.

NOTE: Facts with mode dynamic are dyamic read/write facts, which means you can dynamically change the values for that fact.

Facts with mode r/o have read-only values, which means they can be viewed but changes cannot be made.

Facts with mode del are deleteable, which means they can be deleted at any time.

Table 4-18 *Matrix Facts*

Fact Name	Fact Type	Mode	Description
matrix.activejobs	Integer	dynamic, r/o	The number of active jobs (including child jobs) for this Orchestration Server.
matrix.clustered	Boolean	dynamic, r/o	Server is configured in a high availabilitly cluster.
matrix.datagrid.root	String	dynamic, r/o	The root directory of the datagrid.
matrix.date	Date	dynamic, r/o	The date of this server.
matrix.date.dayofmonth	Integer	dynamic, r/o	The numerical representation of the current day of the month.
matrix.date.dayofweek	String	dynamic, r/o	The string representation of the current week day.
matrix.date.dayofweek.number	Integer	dynamic, r/o	The numerical representation of the current week day.
matrix.date.month	String	dynamic, r/o	The string representation of the current month.
matrix.date.month.number	Integer	dynamic, r/o	The numerical representation of the current month.
matrix.date.year	Integer	dynamic, r/o	The current year.
matrix.groups.job	String[]	dynamic, r/o	List of Group Names of type Job.

Fact Name	Fact Type	Mode	Description
matrix.groups.repository	String[]	dynamic, r/o	List of Group Names of type Repository.
matrix.groups.resource	String[]	dynamic, r/o	List of Group Names of type Resource.
matrix.groups.user	String[]	dynamic, r/o	List of Group Names of type User.
matrix.groups.vbridge	String[]	dynamic, r/o	List of Group Names of type Vbridge.
matrix.hostname	String	dynamic, r/o	The server hostname.
matrix.hostname.full	String	dynamic, r/o	The fully qualified server hostname.
matrix.id	String	r/o	The name of the matrix
matrix.loadaverage	Real		Indicates the average load of the grid server. A value less than 1.0 is unloaded.
matrix.maxactive	Integer		The hard limit for the maximum number of active jobs (including child jobs) allowed on this server at one time. Jobs exceeding this value will be queued.
matrix.maxqueued	Integer		The maximum number of queued jobs that will be accepted by this matrix server
matrix.maxtopjobs	Integer		The maximum number of active jobs after which new top-level jobs will be queued (child jobs may be be able to start). Should be less than matrix.maxactive.
matrix.physical.hostname	String	dynamic, r/o	The physical server hostname
matrix.physical.hostname.full	String	dynamic, r/o	The fully qualified physical server hostname.
matrix.queuedjobs	Integer	dynamic, r/o	The number of queued jobs for this matrix server.
matrix.time	Time	dynamic, r/o	The time of this server.
matrix.timeout.jobfinishing	Integer		The approximate number of seconds to wait for a job to complete its finishing state. That is, to run any failed/canceled/ completed JDL event handlers.
matrix.timezone	String	dynamic, r/o	The string description of the time zone in which this matrix server resides.
matrix.uptime	String	dynamic, r/o	The time since the last server restart.

Fact Name	Fact Type	Mode	Description
`matrix.version`	String	dynamic, r/o	The version of this server in form *<major>.<minor>.<point>_<buildNumber>*

4.2 Computed Facts

This section includes the following information:

4.2.1 What is a Computed Fact?

There are situations when facts with static or even simple dynamic values don’t provide adequate data to enable implementation of more complex policy constraints. In such cases, Cloud Manager Orchestration can use “computed facts” whose values are dynamically and programmatically calculated using more sophisticated scripted l

A computed fact is meant to be used in a policy’s constraint. When applied through the execution of a Jython thread, the computed fact calculates values on demand and returns them to the Orchestration logic (Job Definition Language or “JDL”) which evaluates the job context and extends the built-in factsets for a Grid Object. This methodology lets you create facts that represent other metrics on the system that are not necessarily available in the default factset.

A computed fact implementation uses the `ComputedFactContext` class to access the evaluation context. This context contains the Grid Objects that the constraint engine uses to evaluate constraints, so this class could access the current job instance, deployed job, User, Resource, vBridge, and Repository Grid Objects if they are available within the context of the policy constraint consuming the computed fact's value.

Some computed facts are bundled with the product and used to handle more complex scenarios relating to provisioning adapter actions. However, computed facts can also be used by jobs which are not related to provisioning adapters or virtual machines in any way. More specifically, the VM host, vBridge and Repository grid objects are in the context only to evaluate provisioning constraints, such as vmHost, whereasthe Job and Job Instance objects are in the context only to evaluate resource or allocation constraints.

The following computed fact example illustrates how you can use a computed fact to calculate how many instances of the job named “foo” that the user in the current job context is running. The value of this computed fact can be used in a constraint to limit the number of instances of a particular job that a user can run:

```
class ActiveFooJobs(ComputedFact):

    def compute(self):
        count = 0
        ctx = self.getContext()
        if ctx == None:
            print "No context"
        else:
            jobInstance = ctx.getJobInstance()
            if jobInstance == None:
                print "No job instance in context"
            else:
                activejobs = getMatrix().getActiveJobs()
                for j in activejobs:
                    state = j.getFact("jobinstance.state.string")
                    if j.getFact("job.id") == "foo" and \
                        j.getFact("user.id") == ctx.getFact("user.id") and \
                        (state == "Running" or state == "Starting"):
                        count+=1
        return count
```

Example of a Computed Fact

The following is an actual computed fact used by the xen and vsphere provisioning adapters:

```
"""
Computed Fact to check whether vmHost in given context has access to all shared
repositories.

This fact is to be used in a vmhost constraint.
The vmhostDefault policy can be augmented to evaluate on this fact.

First you must define the usage of the computed fact on the grid object
by creating a linked fact.  Create a policy with the following text and
apply the new policy to any repository objects you want to constrain
(or for simplification apply this new policy to All repositories)

<policy>
    <resource>
        <fact name="candidateHost" type="Boolean" cfactvalue="cfact.resource"/>
    </resource>
</policy>

This creates a new fact in the repository namespace that links to the computed
fact.

Second, create a constraint that uses the new repository fact.
Example constraint to add to the vmhostDefault policy:

      <eq fact="resource.candidateHost" value="True" reason="vmHost doesn't have
access to all shared repositories" />

To test:
  Select a VM and choose "Check Host Assignment". This brings up a dialog of what
host/repository
  plans match.  In this case, all the plans will not match unless they have access
to shared repositories.

"""

class resource(ComputedFact):

    def compute(self):

        context = self.getContext()
        if context is not None:

            #Get vm resource and vmhost Info object
            resource = context.getResource()
```

```
            vmhost = context.getVmHost()

            if resource == None:
                print "no resource"
                return False

            elif vmhost == None:
                print "no vmhost"
                return False

            else:
                #get the vm disks list
                vmdisks = resource.getFact("resource.vm.vdisks")
                #print 'disk info is ',vmdisks
                for diskId in vmdisks:
                    disk = getMatrix().getGridObject(TYPE_VDISK,diskId)
                    if disk != None and disk.factExists("vdisk.moveable") and not
disk.getFact(vdisk.moveable"):

                        # If the disk is not moveable, get repository where it is
stored
                        if disk.factExists("vdisk.repository"):
                            repo = disk.getFact("vdisk.repository")

                           #Get list of repositories to which vmhost has access to
                            repos = vmhost.getFact("vmhost.repositories")

                          #If vmhost doesn't have access to disk repo, return False
                            if not repo in repos:
                                return False
            #vmhost in context has access to all shared repositories
            return True

        # no context (console fact table)
        return False
```

4.2.2 Creating a New Computed Fact

`ComputedFact` is the base class for creating custom computed Facts. To create a new computed fact, you need to subclass the `ComputedFact` class with the `.cfact` extension.

You can create a standard fact to include the cfact just as you would in a policy structure. For example:

```
<fact name="network.score" type="Integer" cfactvalue="cfact.networkScore"/>
```

4.2.3 Using a Computed Fact

To use a computed fact you must deploy to the server the file with the `.cfact` extension referred to above. The `ComputedFact` subclass name is not required to match the computed fact file name. The file name is the computed fact name that you define. To use the Computed Fact, create a linked fact that references the deployed `ComputedFact` class. Cloud Manager Orchestration uses the linked fact in constraints.

The following is an example you would use to retrieve the current job instance where a computed fact is being executed in a resource constraint:

```
        class myComputedFact(ComputedFact):
            def compute(self):
                ctx = self.getContext()
                if ctx == None:
                    print "No context"
                    ...
                else:
                    jobInstance = ctx.getJobInstance()
                    if jobInstance == None:
                        print "No job instance in context"
                        ...
                    else:
                        print "jobInstance.id=%s" %
(jobInstance.getFact("jobinstance.id"))
                        ...
```

4.2.4 Caching and Performance Considerations

Due to spawning of Jython threads, cfacts place a considerably higher load on the server than normal dynamic fact computations. You should enable caching when an excess load occurs, such as when a fact is associated with a large number of resources. An example of this is using the `vmbuilderPXEBoot` cfact when the grid is large, to avoid performance degradation.

To enable caching of cfacts,

1. In the Explorer treee of the Orchestration Console, expand the *Computed Facts* group object and select *vmbuilderPXEBoot* to open the admin view.
2. In the admin view, select the *Attributes* tab to open the *Attributes* page.
3. On the *Attributes* page, select the *Cache Result For* check box and enter an amount of time greater than 30 seconds (for example, enter 10 minutes).
4. Click the *Save* icon in the main toolbar to save the new settings.

Use this procedure only on Orchestration resources that are also VMs – for example, on Xen VMs where the agent is installed.

NOTE: The caching setting is unselected by default because Cloud Manager Orchestration requires some facts to be evaluated frequently for VM host plans. As an administrator, you should also be aware of and select the amount of time for cache refresh. The `vmbuilderPXEBoot` fact does not change, so setting the cache for this fact has no undesirable effects.

4.3 Custom Facts

This section includes the following information:

4.3.1 What Is a Custom Fact?

A “custom” fact is one that is not internally defined in the Orchestration Server as a common or “well-known” fact. You can determine whether a fact is well known by looking at the list of facts for a given Grid Object using the Orchestration Console. If you select the *Constraints / Facts* tab in the

admin view of any object, the custom fact is displayed in the fact table in bold face type, indicating that it was set by non-core components such as provisioning adapters or by extensions or jobs set up by a Cloud Manager Orchestration administrator or developer.

Many custom facts are set and are used by the Orchestration provisioning adapter code, across most of the grid object types. You should consider these as "internal" to the system and not to be modified under normal circumstances. For example, the vsphere and xen provisioning adapters use certain custom facts to hold internal information used by their implementations to configure and discover virtual network interface card (vNIC) devices. The same is true for VMs, VM hosts, vDisks, and other Grid Objects.

4.3.2 Can I Create a Custom Fact?

You can set up a custom facts directly in JDL code or by editing a policy associated with an object. For example, in vSphere client hosts (where the SDK is available for use by Cloud Manager Orchestration) you would associate the vsphere_client policy to that physical (or virtual) resource. Within that policy, you would use the Policy Editor in the Orchestration Console to set the `vsphere.client` Boolean fact to true.

Under normal circumstances, a custom fact has `deletable` and `read/write` attributes set by default. These are the only attributes that you can set in the Policy Editor; however, it is possible to set the attributes by using the Java API.

5 The Cloud Manager Orchestration Datagrid

This section explains concepts related to the datagrid of the NetIQ Cloud Manager Orchestration Server datagrid and specifies many of the objects and facts that are managed in the grid environment:

5.1 Defining the Datagrid

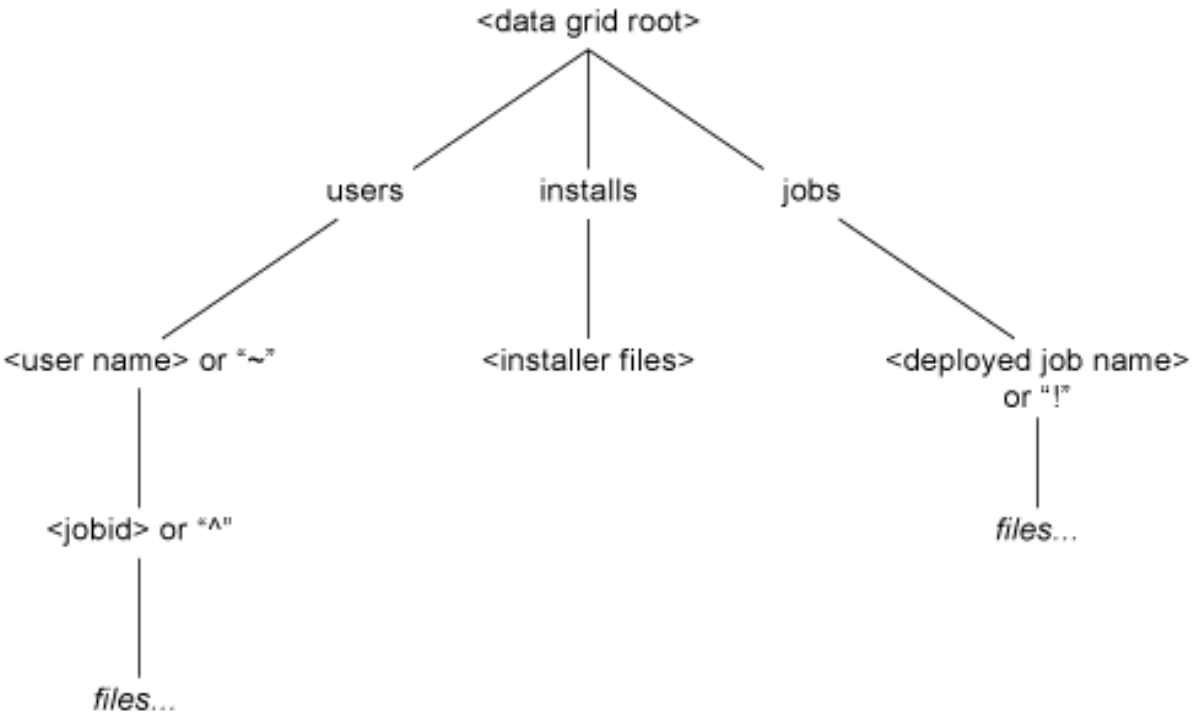
Within the Orchestration environment, the datagrid has three primary functions:

5.1.1 Cloud Manager Orchestration Datagrid Filepaths

The Cloud Manager Orchestration datagrid provides a file naming convention that is used in JDL code and by the Orchestration CLI for accessing files in the datagrid. The naming convention is in the form of a URL. For more information, see “Jobs”in the *NetIQ Cloud Manager 2.1.1 Orchestration Administrator Reference*.

The datagrid defines the root of the namespace as `grid://`, with further divisions under the root as illustrated in the figure below:

Figure 5-1 *File Structure of Data Nodes in a Datagrid*



The grid URL naming convention is the form `grid://<gridID>/<file path>`. Including the grid ID is optional and its absence means the host default grid.When writing jobs and configuring a datagrid, you can use the symbol ^ as a shortcut to the <*jobid*> directory either standalone, indicating the current job, or followed by the jobid number to identify a particular job. Likewise, the symbol ! can be used as a shortcut to the deployed jobs' home directory either standalone, indicating the current jobs' type, or followed by the deployed jobs' name. The symbol ~ is also a shortcut to the user's home directory in the datagrid, either by itself, indicating the current user, or followed by the desired user ID to identify a particular user.

The following examples show address locations in the datagrid using the `zos` command line tool. These examples assume you have logged in using `zos login` to the Orchestration Server you are using:

- "Directory Listing of the Datagrid Root Example" on page 110
- "Directory Listing of the Jobs Subdirectory Example" on page 111

Directory Listing of the Datagrid Root Example

```
$ zos dir grid:///
     <DIR>          Jun-26-2007 9:42 installs
     <DIR>          Jun-26-2007 9:42 jobs
     <DIR>          Jun-26-2007 14:26 users
     <DIR>          Jun-26-2007 9:42 vms
     <DIR>          Jun-26-2007 10:09 warehouse
```

Directory Listing of the Jobs Subdirectory Example

```
$ zos dir grid:///jobs
     <DIR>          Jun-26-2007 9:42 cpuInfo
     <DIR>          Jun-26-2007 9:42 findApps
     <DIR>          Jun-26-2007 9:42 osInfo
     <DIR>          Jun-26-2007 9:42 vcenter
     <DIR>          Jun-26-2007 9:42 vmHostVncConfig
     <DIR>          Jun-26-2007 9:42 vmprep
     <DIR>          Jun-26-2007 9:42 vmserver
     <DIR>          Jun-26-2007 9:42 vmserverDiscovery
     <DIR>          Jun-26-2007 9:42 xen30
     <DIR>          Jun-26-2007 9:42 xenDiscovery
     <DIR>          Jun-26-2007 9:42 xenVerifier
```

5.1.2 Distributing Files

The Orchestration datagrid provides a way to distribute files in the absence of a distributed file system. This is an integrated service of Cloud Manager Orchestration that performs system-wide file delivery and management.

5.1.3 Simultaneous Multicasting to Multiple Receivers

The datagrid provides a multicast distribution mechanism that can efficiently distribute large files simultaneously to multiple receivers. This is useful even when a distributed file system is present. For more information, see Section 5.2, “Datagrid Communications,” on page 112.

5.1.4 Cloud Manager Orchestration Datagrid Commands

The following datagrid commands can be used when creating job files. To see where these commands are applied in the Orchestration Console, see .

Command	Description
`cat`	Displays the contents of a datagrid file.
`copy`	Copies files and directories to and from the datagrid.
`delete`	Deletes files and directories in the datagrid.
`dir`	Lists files and directories in the datagrid.
`head`	Displays the first of a datagrid file.
`log`	Displays the log for the specified job.
`mkdir`	Makes a new directory in the datagrid.
`move`	Moves files and directories in the datagrid.
`tail`	Displays the end of a datagrid file.

5.2 Datagrid Communications

There is no set limit to the number of receivers (nodes) that can participate in the datagrid or in a multicast operation. Indeed, multicast is rarely more efficient when the number of receivers is small. Any type of file or file hierarchy can be distributed via the datagrid.

The datagrid uses both a TCP/IP and IP multicast protocols for file transfer. Unicast transfers (the default) are reliable because of the use of the reliable TCP protocol. Unicast file transfers use the same server/node communication socket that is used for other job coordination datagrid packets are simply wrapped in a generic DataGrid message. Multicast transfers use the persistent socket connection to setup a new multicast port for each transfer.

After the multicast port is opened, data packets are received directly. The socket communication is then used to coordinate packet resends. Typically, a receiver will loose intermittent packets (because of the use of IP multicast, data collisions, etc.). After the file is transferred, all receivers will respond with a bit map of missed packets. The logically ANDing of this mask is used to initiate a resend of commonly missed packets. This process will repeat a few times (with less data to resend on each iteration). Finally, any receiver will still have incomplete data until all the missing pieces are sent in a reliable unicast fashion.

The data transmission for a multicast datagrid transmission is always initiated by the Orchestration Server. Currently this is the same server that is running the grid.

With the exception of multicast file transfers, all datagrid traffic goes over the existing connection between the agent/client and the server. This is done transparently to the end user or developer. As long as the agent is connected and/or the user is logged in to the grid, the datagrid operations function.

5.2.1 Multicast Example

Multicast transfers are currently only supported through JDL code on the agents. In JDL, after you get the Datagrid object, you can enable and configure multicasting like this:

```
dg.setMulticast(true)
```

Additional multicast tuneables can be set on the object as well, such as the following example:

```
dg.setMulticastRate(20000000)
```

This would set the maximum data rate on the transfer to 20 million bytes/sec. There are a number of other options as well. Refer to the JDL reference for complete information.

The actual multicast copy is initiated when a sufficient number of JDL joblets on different nodes issue the JDL command:

```
dg.copy(...)
```

to actually copy the requested file locally. See the `setMulticastMin` and `setMulticastQuorum` options to change the minimum receiver count and other thresholds for multicasting.

For example, to set up a multicast from a joblet, where the data rate is 30 million bytes/sec, and a minimum of five receivers must request multicast within 30 seconds, but if 30 receivers connect, then start right away, use the following JDL:

```
dg = DataGrid()
dg.setMulticast(true)
dg.setMulticastRate(30000000)
dg.setMulticastMin(5)
dg.setMulticastQuorum(30)
dg.setMulticastWait(30000)
dg.copy('grid:///vms/huge-image.dsk', 'image.dsk')
```

In the above example, if at least five agents running the joblet request the file within the same 30 second period, then a multicast is started to all agents that have requested multicast before the transfer is started. Agents requesting after the cutoff have to wait for the next round. Also, if fewer than 5 agents request the file, then each agent will simply fall back to plain old unicast file copy.

Furthermore, if more than 30 agents connect before 30 seconds is up, then the transfer begins immediately after the 30th request. This is useful for situations where you know how many agents will request the file and want to start as soon as all of them are ready.

5.2.2 Grid Performance Factors

The multicast system performance is dependent on the following factors:

- **Network Load:** As the load increases, there is more packet loss, which results in more retries.
- **Number of Nodes:** The more nodes (receivers) there are, the greater the efficiency of the multicast system.
- **File Size:** The larger the file size, the better. Unless there are a large number of nodes, files less than 2 Mb are probably too small.
- **Tuning:** The datagrid facility has the ability to throttle network bandwidth. Best performance has been found at about maximum bandwidth divided by 2. Using more bandwidth leads to more collisions. Also the number of simultaneous multicasts can be limited. Finally the minimum receiver size, receiver wait time and quorum receiver size can all be tuned.

Access to the datagrid is typically performed via the CLI tool or JDL code within a job. There is also a Java API in the Client SDK (on which the CLI is implemented). See "Interface ClientAgent" on page 213.

5.2.3 Plan for Datagrid Expansion

When planning your datagrid, you need to consider where you want the Orchestration Server to store its data. Much of the server data is the contents of the datagrid, including ever-expanding job logs. Every job log can become quite large and quickly exceed its storage constraints.

In addition, every deployed job with its job package—JDL scripts, policy information, and all other associated executables and binary files—is stored in the datagrid. Consequently, if your datagrid is going to grow very large, store it in a directory other than `/opt`.

5.3 datagrid.copy Example

This example fetches the specified source file to the destination. A recursive copy is then attempted if `setRecursive(True)` is set. The default is a single file copy. A multicast also is attempted if `setMulticast(True)` is set. The default is to do a unicast copy. The following example copies a file from the datagrid to a resource, then read the lines of the file:

```
1       datagrid = DataGrid()
2       datagrid.copy("grid:///images/myFile","myLocalFile")
3       text = open("myLocalFile").readlines()
```

This is an example to recursively copy a directory and its sub directories from the datagrid to a resource:

```
4       datagrid = DataGrid()
5       datagrid.setRecursive(True)
6       datagrid.copy("grid:///testStore/testFiles","/home/tester/myLocalFiles")
```

Here's an example to copy down a file from the job deployment area to a resource and then read the lines of the file:

```
7      datagrid = DataGrid()
8      datagrid.copy("grid:///!myJob/myFile","myLocalFile")
9     text = open("myLocalFile").readlines()
```

Here are the same examples without using the shortcut characters. This shows the job "myJob" is under the "jobs" directory under the Datagrid root:

```
10      datagrid = DataGrid()
11      datagrid.copy("grid:///jobs/myJob/myFile","myLocalFile")
12      text = open("myLocalFile").readlines()
```

6 Virtual Machine Job Development

This section explains the following concepts related to developing virtual machine (VM) management jobs with NetIQ Cloud Manager Orchestration:

6.1 VM Job Best Practices

This section discusses some of VM job architecture best practices to help you understand and get started developing VM jobs:

6.1.1 Plan Robust Application Starts and Stops

An application is required for a service, and a VM is provisioned on its behalf. As part of the provisioning process, the VM’s OS typically must be prepared for specific work; for example, NAS mounts, configuration, and other tasks. The application might also need customizing, such as configuring file transfer profiles, client/server relationships, and other tasks.

Then, the application is started and its “identity” (IP address, instance name, and other identifying characteristics) might need to be transferred to other application instances in the service, or a load balancer).

If the Orchestration Server loses the job/joblet communication state machine, such as when a server failover or job timeout occurs, all of the state information must be able to be recovered from “facts” that are associated with the server. This kind of job should also work in a disaster recovery mode, so it should be implemented in jobs regularly when relevant services from Data Center A must be started in Data Center B in a DR case. These jobs require special precautions.

6.1.2 Managing VM Systems

A series of VMs must typically be provisioned in order to run system-wide maintenance tasks. Because there might not be enough resources to bring up every VM simultaneously, you might consider running discovery jobs to limit how many resources (RAM, cores, etc.) that can be used at any given time. Then, you should consider running a task that writes a consolidated audit trail.

6.1.3 Managing VM Images

Similar to how the job `installagent` searches for virtual machine grid objects using specified Constraints and runs a VM operation (`installAgent`) on the VMs that are returned, an Orchestration image must be modified when the VM is not running. Preferably, this should occur without having to provision the VM itself.

6.1.4 Managing VM Hypervisors

The management engine ("hypervisor") underlying the host server must be "managed" while a VM is running. For example, VM memory or CPU parameters must be adjusted on behalf of a monitoring job or a Development Client action.

6.1.5 VM Job Considerations

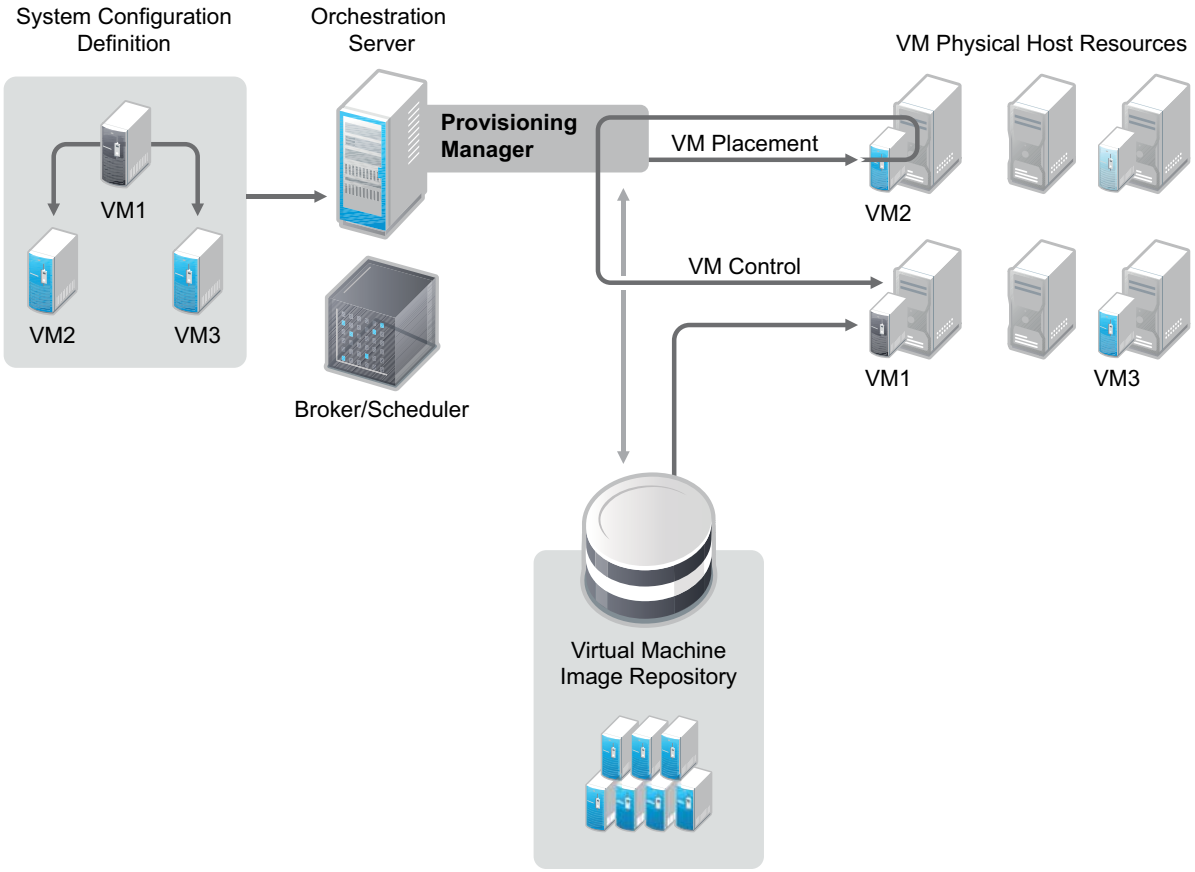
In some instances, some managed resources might host VMs that do not contain an Orchestration Agent. Such VMs can only be controlled by administrators interacting directly with them.

Long-running VMs can be modified or migrated while the job managing the VM is not actively interacting with it. If you have one joblet running on the container and one inside the VM, that relationship might have to be re-established.

6.2 Virtual Machine Management

The Cloud Manager Orchestration provisioning manager provides the ability to manage the use of virtual machines, as shown in the following figure:

Figure 6-1 *VM Management*



For more information about managing virtual machines, see the .

While Cloud Manager Orchestration enables you manage many aspects of your virtual environment, as a developer, you can create custom jobs that do the following tasks:

- **Create and clone VMs:** These jobs create virtual machine images to be stored or deployed. They also create templates for building images to be stored or deployed (see "VM Instance:" on page 121 and "VM Template:" on page 121).
- **Discover resources that can be used as VM hosts.**
- **Provision, migrate, and move VMs:** Virtual machine images can be moved from one physical machine to another.
- **Provide checkpoints, restoration, and re-synchronization of VMs:** Snapshots of the virtual machine image can be taken and used to restore the environment if needed. For more information, refer to the documentation for your hypervisor or contact technical support organization for that hypervisor.
- **Monitor VM operations:** Jobs can start, shut down, suspend and restart VMs.
- **Manage on, off, suspend, and restart operations.**

6.3 VM Life Cycle Management

The life cycle of a VM includes its creation, testing, modifications, use in your environment, and removal when it's no longer needed.

For example, in setting up your VM environment, you might want to first create basic VMs from which you can create templates. Then, to enable the most efficient use of your current hardware capabilities, you can use those templates to create the many different specialized VMs that you need to perform the various jobs. You can create and manage VM-related jobs through the Development Client interface.

Life cycle functions are performed one at a time per given VM in order to prevent conflicts in using the VM. Life cycle events include:

- Creating a VM
- Starting and stopping a VM
- Pausing and resuming a VM
- Suspending and provisioning a VM
- Installing the Orchestration Agent on a VM
- Creating a template from a VM
- Using the VM (starting, stopping, pausing, suspending, restarting, and shutting down)
- Running jobs for the VM
- Editing a VM
- Editing a template
- Moving a stopped VM to another host server
- Migrating a running VM to another host server
- Resynchronizing a VM to ensure that the state of the VM displayed in the Development Client is accurate
- Cloning a VM

6.4 Manual Management of a VM Lifecycle

The example provided in this section is a general purpose job that only provisions a resource.

You might use a job like this, for example, each day at 5:00 p.m. when your accounting department requires extra SAP servers to be available. As a developer, you would create a job that provisions the required VMs, then use the Orchestration Scheduler to schedule the job to run every day at the time specified.

In this example, the `provision` job retrieves the members of a resource group (which are VMs) and invokes the provision action on the VM objects. For an example of a provision job JDL, see Section 6.4.3, “Provision Job JDL,” on page 119.

To setup to create the `provision.job`, use the following procedure:

1. Create your VMs and follow the discovery process in the Development Client so that the VMs are contained in the Orchestration inventory.
2. In the Development Client, create a Resource Group called `sap` and add the required VMs as members of the group.

3 Given the `.jdl` and `.policy` below you would create a `.job` file (jar them):

```
>jar cvf provision.job provision.jdl provision.policy
```

4 Deploy the `provision.job` file to the Orchestration Server using either the Development Client or the `zosadmin` command line.

To run the job, use either of the following procedures:

- Section 6.4.1, "Manually Using the zos Command Line," on page 119
- Section 6.4.2, "Automatically Using the Development Client Job Scheduler," on page 119

6.4.1 Manually Using the zos Command Line

1 At the command line, enter:

```
>zos login <zos server>
>zos run provision VmGroup="sap"
```

For more complete details about entering CLI commands, see "The zos Command Line Tool" in the *NetIQ Cloud Manager 2.1.1 Orchestration Server Command Line Reference*.

6.4.2 Automatically Using the Development Client Job Scheduler

1 In the Development Client, create a *New* schedule.

2 Fill in the job name (`provision`), user, and priority.

3 For the jobarg `VmGroup`, enter `sap`.

4 Create a Trigger for the time you want this job to run.

5 Save the Schedule and enable it by clicking *Resume*.

You can manually force scheduling by clicking *Test Schedule Now*.

For more complete details about using the Job Scheduler, see "The Orchestration Server Job Scheduler" in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*. You can also refer to Section 6.6, "Automatically Provisioning a VM," on page 124 in this guide.

6.4.3 Provision Job JDL

```
"""Job that retrieves the members of a supplied resource group and invokes the
provision action on all members. For more details about this class, see Job
(page 252). See also ProvisionSpec (page 271).

The members must be VMs.

"""
class provision(Job):

    def job_started_event(self):

        # Retrieves the value of a job argument supplied in
        # the 'zos run' or scheduled run.
        VmGroup = self.getFact("jobargs.VmGroup")

        #
        # Retrieves the resource group grid object of the supplied name.
        # The job Fails if the group name does not exist.
        #
```

```
        group = getMatrix().getGroup(TYPE_RESOURCE,VmGroup)
        if group == None:
            self.fail("No such group '%s'." % (VmGroup))

        #
        # Gets a list of group members and invokes a provision action on each one.
        #
        members = group.getMembers()
        for vm in members:
           vm.provision()
           print "Provision action requested for VM '%s'" %
(vm.getFact("resource.id"))


Job Policy:
<!--
    The policy definition for the provision example job.

    This specifies the job argument VmGroup' which is required
-->
<policy>

    <jobargs>

        <fact name="VmGroup"
              type="String"
              description="Name of a VM resource group whose members will be
provisioned"
              />

    </jobargs>

</policy>
```

6.5 Provisioning Virtual Machines

VM provisioning adapters run just like regular jobs on Cloud Manager Orchestration. The system can detect a local store on each VM host and if a local disk might contain VM images. The provisioner puts in a request for a VM host. However, before a VM is brought to life, the system pre-reserves that VM for exclusive use.

That reservation prevents a VM from being stolen by any other job that's waiting for a resource that might match this particular VM. The constraints specified to find a suitable host evaluates machine architectures, CPU, bit width, available virtual memory, or other administrator configured constraints, such as the number of virtual machine slots.

This process provides heterogeneous virtual machine management using the following virtual machine adapters (also called "provisioning adapters"):

- **Xen Adapter:** For more information, see XenSource* (http://www.xensource.com/).
- **VMware vSphere 4.*x*:** For more information, see VMware (http://www.vmware.com).
- **Hyper-V:** For more information, see Microsoft* Windows Server 2008 Virtualization with Hyper-V (http://www.microsoft.com/windowsserver2008/en/us/hyperv.aspx).

For more information, see "Provisioning a Virtual Machine" in the *NetIQ Cloud Manager 2.1.1 VM Orchestration Reference*.

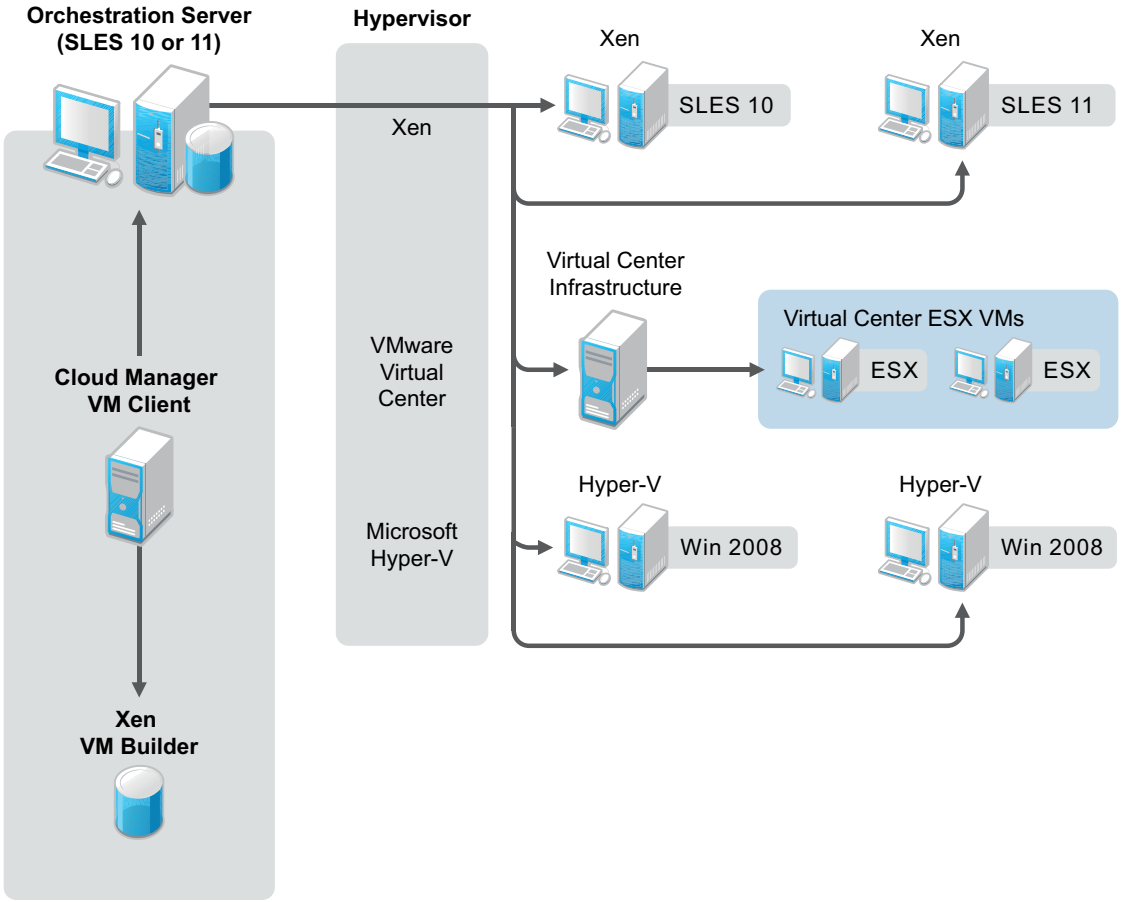
There are two types of VMs that can be provisioned:

- **VM Instance:** A VM instance is a VM that is "state-full." This means there can only ever be one VM that can be provisioned, moved around the infrastructure, and then shut down, yet maintains its state.
- **VM Template:** A VM template represents an image that can be cloned. After it is finished its services, it is shut down and destroyed.

  It can be thought of as a "golden master." The number of times a golden master or template can be provisioned or cloned is controlled though constraints that you specify when you create a provisioning job.

The following graphic is a representation of the provisioning adapters and the way they function to communicate joblets to VMs:

Figure 6-2 *VM Management Provisioning Communications*



NOTE: The Xen VM Monitor can support more than just SUSE Linux Enterprise (SLE) 10 (which uses Xen 3.0.4) and Red Hat Enterprise Linux (RHEL) 5 (which uses Xen 3.0.3) VMs. For a complete list of supported guest operating systems, see the Xen Web site (http://www.xen.org/).

The following sections provide more information on provision of VMs:

- Section 6.5.1, "Provisioning VMs Using Jobs," on page 122
- Section 6.5.2, "VM Placement Policy," on page 123
- Section 6.5.3, "Provisioning Example," on page 124

6.5.1 Provisioning VMs Using Jobs

The following actions can be performed by jobs:

- Provision (schedule or manually provision a set of VMs at a certain time of day).
- Move
- Clone (clone a VM, an online VM, or a template)
- Migrate
- Destroy
- Restart
- Check status
- Create a template to instance
- Create an instance to template
- Affiliate with a host
- Make it a stand-alone VM
- Create checkpoints
- Restore
- Delete
- Cancel Action.

You might want to provision a set of VMs at a certain time of day before the need arises. You also might create a job to shut down all VMs or a constrained group of VMs. You can perform these tasks programmatically (using a job), manually (through the Development Client), or automatically on demand.

When performing tasks automatically, a job might make a request for an unavailable resource, which triggers a job to look for a suitable VM image and host. If located, the image is provisioned and the instance is initially reserved for calling a job to invoke the required logic to select, place, and use the newly provisioned resource.

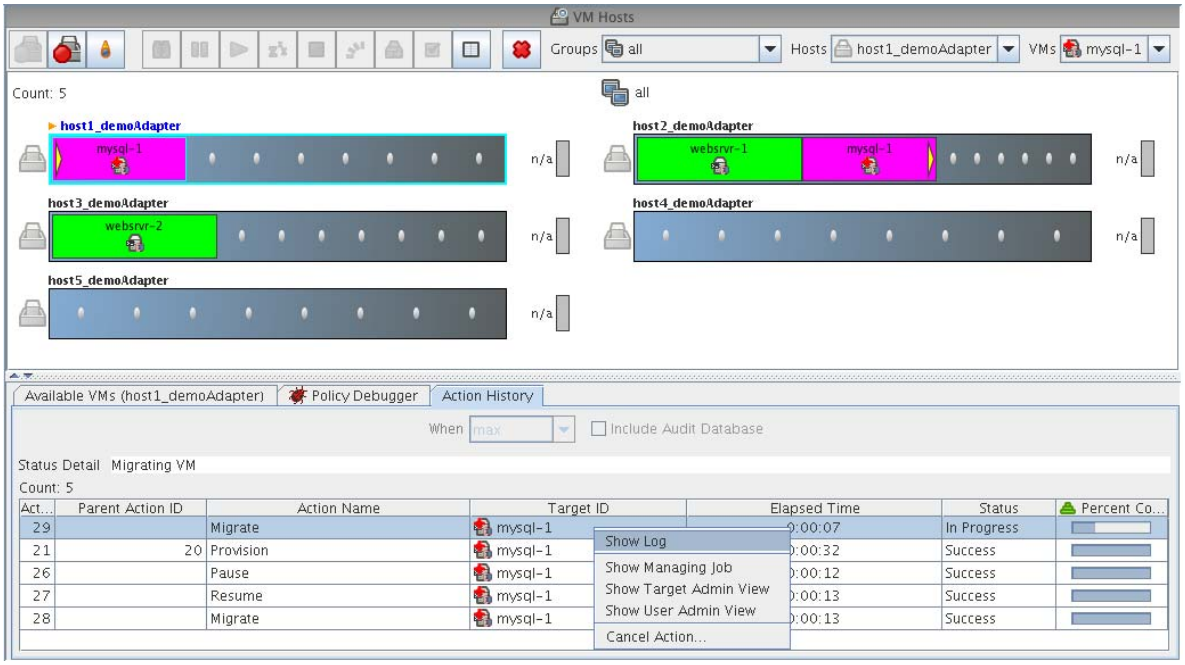
For an example of this job, see sweeper.job (page 172).

VM operations are available on the ResourceInfo (page 273) grid object, and VmHost operations are available on the VMHostInfo (page 283) grid object. In addition, as shown in Section 6.5.3, “Provisioning Example,” on page 124, three provisioner events are fired when a provision action has completed, failed, or cancelled.

The API is equivalent to the actions available within the Development Client. The selection and placement of the VM host is governed by policies, priorities, queues, and ranking, similar to the processes used selecting resources.

Provisioning adapters on the Orchestration Server abstract the VM. These adapters are special provisioning jobs that perform operations for each integration with different VM technologies. The following figure shows the VM host management interface that is using the Development Client.

Figure 6-3 *VM Hosts Management*



6.5.2 VM Placement Policy

To provision virtual machines, a suitable host must be found. The following shows an example of a VM placement policy:

```
<policy>
  <constraint type="vmhost">
    <and>
      <eq fact="vmhost.enabled" value="true"
          reason="VmHost is not enabled" />
      <eq fact="vmhost.online" value="true"
          reason="VmHost is not online" />
      <eq fact="vmhost.shuttingdown" value="false"
          reason="VmHost is shutting down" />
      <lt fact="vmhost.vm.count" factvalue="vmhost.maxvmslots"
          reason="VmHost has reached maximum vmslots" />
     <ge fact="vmhost.virtualmemory.available"
         factvalue="resource.vmimage.virtualmemory"
         reason="VmHost has insufficient virtual memory for guest VM" />
      <contains fact="vmhost.vm.availableids"
                factvalue="resource.id"
                reason="VmImage is not available on this VmHost" />
    </and>
  </constraint>
</policy>
```

6.5.3 Provisioning Example

This job example provisions a virtual machine and monitors whether provisioning completed successfully. The VM name is "webserver" and the job requires a VM to be discovered before it is run. After the provision has started, one of the three provisioner events is called.

```
1  class provision(Job):
2
3      def job_started_event(self):
4          vm = getMatrix().getGridObject(TYPE_RESOURCE,"webserver")
5          vm.provision()
6          self.setFact("job.autoterminate",False)
7
8      def provisioner_completed_event(self,params):
9          print "provision completed successfully"
10         self.setFact("job.autoterminate",True)
11
12     def provisioner_failed_event(self,params):
13         print "provision failed"
14         self.setFact("job.autoterminate",True)
15
16     def provisioner_cancelled_event(self,params):
17         print "provision cancelled"
18         self.setFact("job.autoterminate",True)
```

See additional provisioning examples in Section 6.4, "Manual Management of a VM Lifecycle," on page 118 and Section 6.6, "Automatically Provisioning a VM," on page 124.

6.6 Automatically Provisioning a VM

If you write jobs to automatically provision virtual machines, you set the following facts in the job policy:

```
resource.provision.maxcount
resource.provision.maxpending
resource.provision.hostselection
resource.provision.maxnodefailures
resource.provision.rankby
```

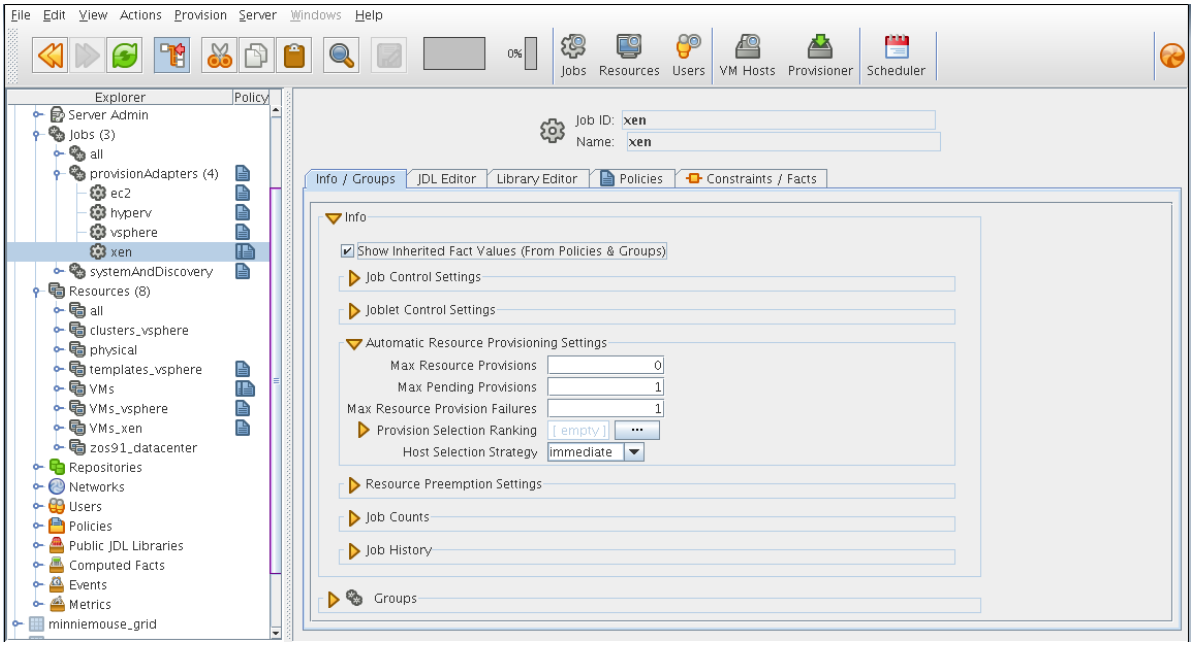
These are the job facts to enable and configure the usage of virtual machines for resource allocation. These facts can be set in a job's policy.

For example, setting the `provision.maxcount` fact to greater than 0 allows for virtual machines to be included in resource allocation:

```
<job>
    <fact name="provision.maxcount" type="Integer" value="1" />
    <fact name="provision.maxpending" type="Integer" value="1" />
</job>
```

The following figure shows the job's Development Client settings that are used to automatically provision VMs:

Figure 6-4 *Job Settings for Automatic VM Provisioning*



When using automatic provisioning, the provisioned resource is reserved for the job requesting the resource. This prevents another job requiring resources from obtaining the provisioned resource.

When the job that reserved the resource has finished its work (joblet has completed) on the provisioned resource, the reservation is relaxed allowing other jobs to use the provisioned resource.

Using JDL, the reservation can be specified to reserve by JobID and also user. This is done using the ProvisionSpec (page 271) class.

7 Job Examples

The following sections demonstrate some practical ways to use NetIQ Cloud Manager Orchestration and should help you better understand how to write your own jobs:

7.1 Simple Job Examples

The following simple examples demonstrate how you can use JDL scripting to manage specific functionality:

To learn about other job examples that are packaged with the Cloud Manager Orchestration Servere, see Chapter 7, “Job Examples,” on page 127.

7.1.1 provisionBuildTestResource.job

The following job example illustrates simple scripting to ensure that each of three desired OS platforms might be available in the grid and, if not, it tries to provision them (provided that a VM image matching the OS type exists). The resource Constraint object is created programmatically, so there is no need for external policies.

```
1  class provisionBuildTestResource(Job):
2
3      def job_started_event(self):
4          oslist = ["Windows XP", "Windows 2000", "Windows 2003 Server"]
5          for os in oslist:
6              constraint = EqConstraint()
7              constraint.setFact("resource.os.name")
8              constraint.setValue(os)
9              resources = getMatrix().getGridObjects("resource",constraint)
10             if len(resources) == 0:
11                 print "No resources were found to match constraint. \
12 os:%s" % (os)
13             else:
14               #
15               # Find an offline vm instance or template.
```

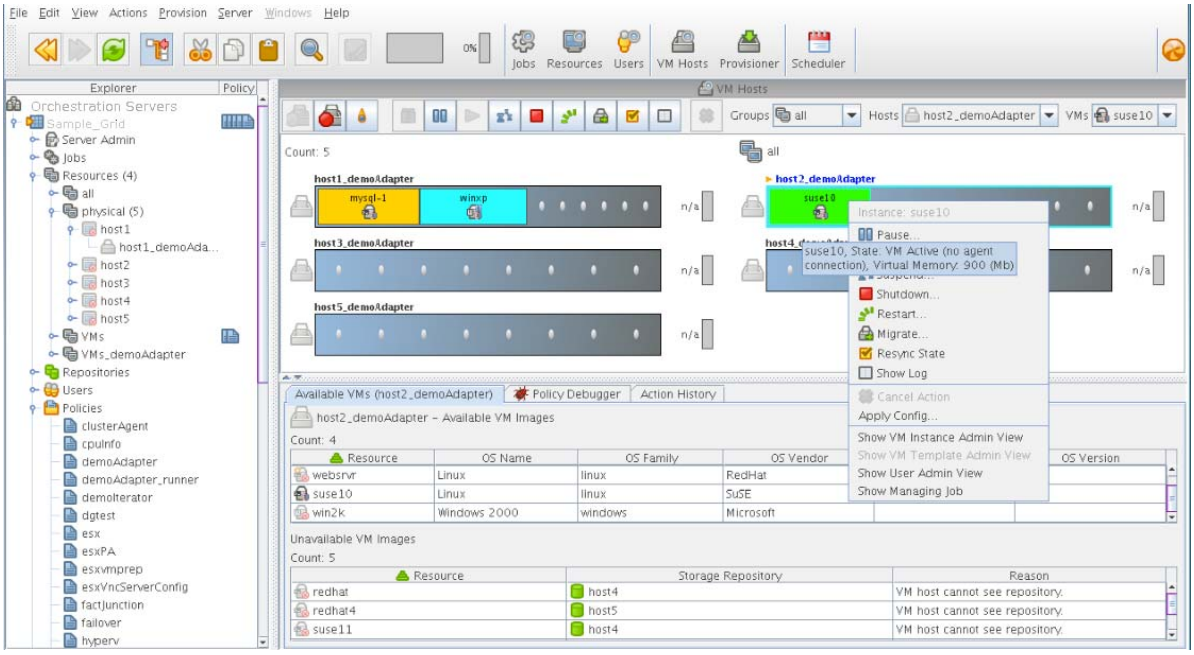
```
16                #
17                instance = None
18                for resource in resources:
19                     if resource.getFact("resource.type") != "Fixed Physical" and
\
20                       resource.getFact("resource.online") == False:
21                       # Found a vm or template. provision it for job.
22                       print "Submitting provisioning request for vm %s." %
(resource)
23                       instance = resource.provision()
24                       print "Provisioning successfully submitted."
25                       break
26                if instance == None:
27                      print "No offline vms or templates found for os: %s" % (os)
```

It is not necessary to always script resource provisioning. Automatic resource provisioning ("on demand") is one of the built-in functions of the Orchestration Server. For example, a job requiring a Windows 2003 Server resource that cannot be satisfied with online resources only needs to have the appropriate facts set in the Orchestration Console; that is, `job.provision.maxcount` is enabled.

This fact could also be set through association with a policy. If it is set up this way, the Orchestration Server detects that a job is in need of a resource and automatically takes the necessary provisioning steps, including reservation of the provisioned resource.

All provisioned virtual machines and the status of the various hosts are visible in the following view of the Orchestration Console.

Figure 7-1 *The Cloud Manager Orchestration Console Showing Virtual Machine Management*



7.1.2 Workflow Job Example

This brief example illustrates a job that does not require resources but simply acts as a coordinator (workflow) for the buildTest and provision jobs discussed in Section 7.2, "BuildTest Job Examples," on page 129.

```
1 class Workflow(Job):
2    def job_started_event(self):
3       self.runJob("provisionBuildTestResource", {})
4       self.runJob("buildTest", { "testlist" : "/QA/testlists/production",
5 "buildId": "2006-updateQ1" } )
```

The job starts in line 1 with the job_started_event, which initiates provisionBuildTestResource.job (page 127) to ensure all the necessary resources are available, and then starts the buildTest.jdl Example (page 131). This workflow job does not complete until the two subjobs are complete, as defined in lines 3 and 4.

If so desired, this workflow could monitor the progress of subjobs by simply defining new event handler methods (by convention, using the `_event` suffix). The system defines many standard events. Every message received by the job executes the corresponding event handler method and can also contain a payload (a Python dictionary).

7.2 BuildTest Job Examples

There are many available facts that you can use in creating your jobs. If you find that you need specific kinds of information about a resource or a job, such as the load average of a user or the ID of a job or joblet, chances are that it is already available.

If a fact is not listed, you can create your own facts by creating a `<fact>` element in the job policy. You can also create a fact directly in the JDL job code.

If you want to remember something from one loop to the next or make something available to other objects in the grid, you can set a fact with your own self-defined name.

This section shows an example of a relatively simple working job that performs a set (100) of regression tests on three different platform types. A number of assumptions have been made to simplify this example:

- Each regression test is atomic and has no dependencies.
- Every resource is preconfigured to run the tests. Typically, the configuration setup is included as part of the job.
- The tests are expressed as line entries in a file. The Orchestration Server has multiple methods to specify parameters. This (`/QA/testlists/nightly.dat`) is just one example:

```
dir c:/windows
dir c:/windows/system32
dir c:/notexist
dir c:/tmp
dir c:/cygwin
```

To demonstrate the possible functionality for this example, here are some policies that might apply to this example:

- Only users running tests can use resources owned by their group.
- To conserve resources, terminate the test after 50 failures.
- Because the system under test requires a license, prevent more than three of these regression tests from running at one time.
- To prevent a job backlog, limit the number of queued jobs in the system.
- To allow the regression test run to tolerate resource failures (for example, unexpected network disconnections, unexpected reboots, and so on), enable automatic failover without affecting the regression run.

The section includes the following information:

- Section 7.2.1, “buildTest.policy Example,” on page 130
- Section 7.2.2, “buildTest.jdl Example,” on page 131

7.2.1 buildTest.policy Example

Policies are typically spread over different objects, entities, and groups on the system. However, to simplify the concept, we have combined all policies into this one example that is directly associated with the job.

The arguments available to the job are specified in the in the <jobargs> section (lines 1-11). When the job is run, job arguments are made available as facts to the job instance. The default values of these arguments can be overridden when the job is invoked.

```
1 <policy>
2    <jobargs>
3        <fact name="buildId"
4            type="String"
5            value="02-24-06 1705"
6            description="Build Id to show in memo field" />
7        <fact name="testlist"            type="String"
9            value="/QA/testlists/nightly.dat"
10           description="Path to testlist to use in tests" />
11    </jobargs>
```

The <job> section (lines 12-25) defines facts that are associated with the job. These facts are used in other policies or by the JDL logic itself. Typically, these facts are aggregated from inherited policies.

```
12    <job>
13        <fact name="max_queue_size"
14            type="Integer"
15            value="10"
16            description="Limit of queued jobs. Any above this limit are not
accepted." />
17        <fact name="max_licenses"
18            type="Integer"
19            value="5"
20             description="License count to limit number of jobs to run
simultaneously. Any above this limit are queued." />
21        <fact name="max_test_failures"
22            type="Integer"
23            value="50"
24            description="To decide to end the job if the number of failures
exceeds a limit" />
25    </job>
```

The <accept> (line 26), <start> (line 31), and <continue> (line 40) constraints control the job life cycle and implement the policy outlined in the example. In addition, allowances are made for “privileged users” (lines 28 and 33) to bypass the accept and start constraints.

```
26    <constraint type="accept" reason="Maximum number of queued jobs has been
reached">
27        <or>
28            <defined fact="user.privileged_user" />
28            <lt fact="job.instances.queued" factvalue="job.max_queue_size" />
29        </or>
30    </constraint>
31    <constraint type="start">
32        <or>
33            <defined fact="user.privileged_user" />
34            <lt fact="job.instances.active" factvalue="job.max_licenses" />
35        </or>
36    </constraint>
```

The <resource> constraint (lines 37 and 38) ensures that only resources that are members of the buildtest group are used by this job.

```
37    <constraint type="resource">
38        <contains fact="resource.groups" value="buildtest" reason="No resources
are in the buildtest group" />
39    </constraint>
40    <constraint type="continue" >
41        <lt fact="jobinstance.test_failures" factvalue="job.max_test_failures"
reason="Reached test failure limit" />
42    </constraint>
</policy>
```

7.2.2 buildTest.jdl Example

The following example shows how additional resource constraints representing the three test platform types are specified in XML format. These also could have been specified in the Cloud Manager Orchestration Console.

This section includes the following information:

- Setting Resource Constraints
- Creating a Memo Field
- Joblet Definition

Setting Resource Constraints

The annotated JDL code represents the job definition, consisting of base Python v2.1 (and libraries) as well as a large number of added Orchestration operations that allow interaction with the Orchestration Server:

```
1  import sys,os,time

2  winxp_platform = "<eq fact=\"resource.os.name\" value=\"Windows XP\" />"
3  win2k_platform = "<eq fact=\"resource.os.name\" value=\"Windows 2000\" />"
4  win2003_platform = "<eq fact=\"resource.os.name\" value=\"Windows 2003 Server\"
/>"
```

Lines 2-4 specify the resource constraints representing the three test platform types (Windows XP, Windows 2000, and Windows 2003) in XML format.

The job_started_event in line 6 is the first event delivered to the job on the server. The logic in this method performs some setup and defines the parameter space used to iterate over the tests.

```
5 class BuildTest(Job):

6    def job_started_event(self):
7       self.total_counts = {"failed":0,"passed":0,"run":0}
8       self.setFact("jobinstance.test_failures",0)

9       self.testlist_fn = self.getFact("jobargs.testlist")
10      self.buildId = self.getFact("jobargs.buildId")
11      self.form_memo(self.total_counts)

12      # Form range of tests based on a testlist file
13      filerange = FileRange(self.testlist_fn)
```

Parameter spaces (lines 14-16) can be multidimensional but, in this example, they schedule three units of work (joblets), one for each platform type, each with a parameter space of the range of lines in the (optionally) supplied test file (lines 21, 24 and 27).

```
14      # Form ParameterSpace defining Joblet Splitting
15      pspace = ParameterSpace()
16      pspace.appendDimension("cmd",filerange)

17      # Form JobletSet defining execution on resources
18      jobletset = JobletSet()
19      jobletset.setCount(1)
20      jobletset.setJobletClass(BuildTestJoblet)
```

Within each platform test, a joblet is scheduled for each test line item on each different platform.

```
21      # Launch tests on Windows XP
22      jobletset.setConstraint(winxp_platform)
23      self.schedule(jobletset)

24      # Launch tests on Windows 2000
25      jobletset.setConstraint(win2k_platform)
26      self.schedule(jobletset)

27      # Launch tests on Windows 2003
28      jobletset.setConstraint(win2003_platform)
29      self.schedule(jobletset)
```

The `test_results_event` in line 32 is a message handler that is called whenever the joblets send test results.

```
30 # Event invoked when a Joblet has completed running tests.
31 #
32 def test_results_event(self,params):
33      self.form_memo(params)
```

Creating a Memo Field

In line 37, the `form_memo` method is called to form an informational string to display the running totals for this test. These totals are displayed in the memo field for the job (visible in the Orchestration Console, and Web interface tools). The memo field is accessed through setting the String fact `jobinstance.memo` in line 55.

```
34 #
35 # Update the totals and write totals to memo field.
36 #
37 def form_memo(self,params):
38     # total_counts will be empty at start
39     m = "Build Test BuildId %s " % (self.buildId)
40     i = 0
41     for key in self.total_counts.keys():
42     if params.has_key(key):
43         total = self.total_counts[key]
44         count = params[key]
45         total += count
46         printable_key = str(key).capitalize()
47        if i > 0:
48            m += ", "
48        else:
49            if len(m) > 0:
50        m+= ", "
51        m += printable_key + ": %d" % (total)
52        i += 1
53        self.total_counts[key] = total
54     self.setFact("jobinstance.test_failures",self.total_counts["failed"])
55     self.setFact("jobinstance.memo",m)
```

Joblet Definition

As previously discussed, a joblet is the logic that is executed on a remote resource employed by a job, as defined in lines 56-80, below. The `joblet_started_event` in line 60 mirrors the `job_started_event` (line 6) but runs on a different resource than the server.

The portion of the parameter space allocated to this joblet in line 65-66 represents some portion of the total test (parameter) space. The exact breakdown of this is under full control of the administrator/job. Essentially, the size of the "work chunk" in line 67 is a compromise between overhead and retry convenience.

In this example, each element of the parameter space (a test) in line 76 is executed and the exit code is used to determine pass or failure. (The exit code is often insufficient and additional logic must be added to analyze generated files, copy results, or to perform other tasks.) A message is then sent back to the job prior to completion with the result counts.

```
56 #
57 # Define test execution on a resource.
58 #

59 class BuildTestJoblet(Joblet):
60    def joblet_started_event(self):
61        passed = 0
62        failed = 0
63        run = 0
64        # Iterate over parameter space assigned to this Joblet
65        pspace = self.getParameterSpace()
66        while pspace.hasNext():
67            chunk = pspace.next()
68            cmd = chunk["cmd"].strip()
69            rslt = self.run_cmd(cmd)
70            print "rslt=%d cmd=%s" % (rslt,cmd)
71            if rslt == 0:
72                passed +=1
73            else:
74                failed +=1
75            run += 1
76
self.sendEvent("test_results_event",{"passed":passed,"failed":failed,"run":run})
77    def run_cmd(self,cmd):
78        e = Exec()
79        e.setCommand(cmd)
80        return e.execute()
```

7.3 Using Deployable Job Examples Included with the Orchestration Server

The basic examples delivered with the Cloud Manager Orchestration Server are located in either of two possible installation directories depending on the type of installation. For server installations, look here:

`/opt/novell/zenworks/zos/server/examples/`

For client installation, look here:

`/opt/novell/zenworks/zos/client/examples/`

When you unjar or unzip examples from the `<path>/examples/<example>.job` file or view jobs using the details panel and the JDL and Policy tabs in the Orchestration Console, you should see the `.jdl` and `.policy` files.

Policy files specify how the job arguments and static attributes are defined. Or, you can use the `zos jobinfo` command to simply display job arguments and their default values.

All of the examples can be opened and modified using a standard code editor, then redeployed and examined using the procedure explained in .

This section includes the following information:

- Section 7.3.1, "Preparing to Deploy Job Examples," on page 134
- Section 7.3.2, "Cloud Manager Orchestration Deployable Job Examples," on page 135

You can find the detailed deployable job example documentation in the following sections:

- Section 7.4, "Deployable Job Examples: Parallel Computing," on page 135
- Section 7.5, "Deployable Job Examples: General Purpose," on page 146
- Section 7.6, "Job Examples: Miscellaneous Code-Only," on page 183

7.3.1 Preparing to Deploy Job Examples

To run the Orchestration jobs described in this section, use the following guidelines:

- Install and configure the Cloud Manager Orchestration Server and the Cloud Manager Orchestration Agent properly (see "Installing Cloud Manager Orchestration Components" and "Configuring Cloud Manager Orchestration Components"in the *NetIQ Cloud Manager 2.1.1 Orchestration Installation Guide*.
- Unless otherwise indicated, install at least one agent on a managed resource and have it running (see "Installing Cloud Manager Orchestration Components" in the *NetIQ Cloud Manager 2.1.1 Orchestration Installation Guide*).
- Before running zosadmin or zos commands, you must log into the Orchestration Server.

  The zosadmin command is required for administrating jobs. This includes deploying and undeploying a job to the server. The zos command is for job control, including starting a job and viewing a job's log. As you learn about the Orchestration job samples, you will use the zosadmin command for deploying a sample job and the zos command for running the sample.

  - For an explanation of the zosadmin commands, see "The zosadmin Command Line Tool" in the *NetIQ Cloud Manager 2.1.1 Orchestration Server Command Line Reference*.

    ```
    > zosadmin login --user psoadmin
    Login to server: skate
    Please enter current password for 'psoadmin':
    Logged into grid on server 'skate'
    ```

  - For an explanation of zos commands, see "The zos Command Line Tool" in the *NetIQ Cloud Manager 2.1.1 Orchestration Server Command Line Reference*.

    ```
    > zos login --user psouser
    Please enter current password for 'psouser':
    Logged into grid as psouser
    ```

  You should create a user (see "Creating a User Account" in the *NetIQ Cloud Manager 2.1.1 Orchestration Administrator Reference*) before logging in.

7.3.2 Cloud Manager Orchestration Deployable Job Examples

The following table provides a high-level explanation of some of the Cloud Manager Orchestration job examples that are delivered with the Orchestration Server and the job developer concepts you might want to understand:

Table 7-1 *Cloud Manager Orchestration Job Development Examples*

Example Name	Job Function Capabilities
demoIterator.job (page 136)	◆ Using policy constraints and job arguments to restrict joblet execution to specific resources. ◆ Scheduling joblets using a ParameterSpace. ◆ Provides an example of executing a command on a resource.
dgtest.job (page 147)	◆ Downloading files stored on grid management servers to networked nodes.
failover.job (page 156)	◆ Managing how joblets failover to enhance the robutsness of your jobs.
instclients.job (page 162)	◆ Installing an Orchestration client on multiple machines. ◆ Provides an example of executing a command on a resource.
jobargs.job (page 184)	◆ Understanding the various argument types that jobs can accept (integer, real, Boolean, string, time, date, list, dictionary, and array, which can contain the types integer, real, Boolean, time, date, and String).
notepad.job (page 168)	◆ Understanding how to launch specific applications on specified resources.
quickie.job (page 142)	◆ Understanding how jobs can start multiple instances of a joblet on one or more resources.
sweeper.job (page 172)	◆ Understanding how poll all resources on the grid.an ordered serialized scheduling of the joblets
whoami.job (page 178)	◆ Sending a command to the operating system's default command interpreter. On Microsoft Windows, this is `cmd.exe`. On POSIX systems, this is `/bin/sh`.

7.4 Deployable Job Examples: Parallel Computing

The following examples demonstrate high performance or parallel computing concepts:

- "demoIterator.job" on page 136
- "quickie.job" on page 142

demoIterator.job

Reference implementation for a simple test iterator. Several concepts are demonstrated: 1) Using policy constraints and job arguments to restrict joblet execution to a specific resource, 2) Scheduling joblets using a ParameterSpace, and 3) An example of executing a command on a resource.

Usage

```
> zos login --user zenuser
Please enter current password for 'zenuser':
Logged into grid as zenuser

> zos jobinfo --detail demoIterator
Jobname/Parameters    Attributes
------------------    ----------
demoIterator       Desc: This example job is a reference for a simple test
                         iterator. It is useful for demonstrating how policies
                         and job args can be used to target the job to a
                         particular resource.

    cpu               Desc: Regular expression match for CPU architecture
                      Type: String
                   Default: .*

    os                Desc: Regular expression match for Operating System
                      Type: String
                   Default: .*

    cmd               Desc: Simple command to execute
                      Type: String
                   Default:

    numJoblets        Desc: joblets to run
                      Type: Integer
                   Default: 100
```

Description

The files that make up the DemoIterator job include:

```
demoIterator                                # Total: 156 lines
|-- demoIterator.jdl                        #   79 lines
`-- demoIterator.policy                     #   77 lines
```

demoIterator.jdl

```
 1 # ---------------------------------------------------------------------------
 2 #  Copyright (C) 2010 Novell, Inc. All Rights Reserved.
 3 #
 4 #  NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 5 #  WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 6 #  FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGMENT.  NOVELL, THE AUTHORS
 7 #  OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 8 #  FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 9 #  TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
10 #  OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
11# ---------------------------------------------------------------------------
12 #  $Id: demoIterator.jdl 10344 2009-11-20 21:46:43Z jastin $
13 # ---------------------------------------------------------------------------
14
15 import time, random
16
17 #
```

```
# Add to the 'examples' group on deployment
#
if __mode__ == "deploy":
   try:
       jobgroupname = "examples"
       jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
       if jobgroup == None:
           jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
       jobgroup.addMember(__jobname__)
   except:
       exc_type, exc_value, exc_traceback = sys.exc_info()
       print "Error adding %s to %s group: %s %s" % (__jobname__, jobgroupname,
exc_type, exc_value)


class demoIteratorJob(Job):

     def job_started_event(self):
          print 'job_started_event'
          self.completed = 0

          # Launch the joblets
          numJoblets = self.getFact("jobargs.numJoblets")
          print 'Launching ', numJoblets, ' joblets'

          pspace = ParameterSpace()
          i = 1
          while i <= numJoblets:
             pspace.appendRow({'name':'joblet'+str(i)})
             i += 1
         pspace.maxJobletSize = 1
         self.schedule(demoIteratorJoblet,pspace,{})

     def joblet_completed_event(self, jobletnumber, node):
         self.completed += 1
         self.setFact("jobinstance.memo", "Tests run: %s" % (self.completed))


class demoIteratorJoblet(Joblet):

     def joblet_started_event(self):
         print "Hi from joblet ", self.getFact("joblet.number")
         time.sleep(random.random() * 15)

         cmd = self.getFact("jobargs.cmd")
         if len(cmd) > 0:
             system(cmd)



             # Example of more sophisticated exec
             # e.g.  e.signal("SIGUSR1")
             """
             e = Exec()
             e.setCommand(cmd)
             #e.setStdoutFile("cmd.out")
             e.writeStdoutToLog()
             e.writeStderrToLog()
             #try:
             e.execute()
             #except:
                 #self.retry("retryable example error")
             """
```

demoIterator.policy

```
<!--
 *=========================================================================
 * Copyright (C) 2010 Novell, Inc. All Rights Reserved.
 *
 * NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE, AND NON INFRINGMENT.  NOVELL, THE AUTHORS
 * OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 * FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 * TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
 * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *=========================================================================
 * $Id: demoIterator.policy 10344 2009-11-20 21:46:43Z jastin $
 *=========================================================================
 -->

<policy>
   <constraint type="accept" reason="Too busy for more work. Try again later!">
      <or>
         <lt fact="job.instances.queued" value="4" />
         <contains fact="user.groups" value="superuser" />
      </or>
   </constraint>

   <constraint type="start" reason="Waiting on queue">
     <or>
       <lt fact="job.instances.active" value="2" />
       <contains fact="user.groups" value="superuser" />
     </or>
   </constraint>

   <jobargs>
     <fact name="numJoblets"
           type="Integer"
           description="joblets to run"
           value="100"
           visible="true" />

     <fact name="cmd"
           type="String"
          description="Simple command to execute"
           value="" />

     <fact name="os"
           type="String"
           description="Regular expression match for Operating System"
           value=".*" />

     <fact name="cpu"
           type="String"
           description="Regular expression match for CPU architecture"
           value=".*" />
   </jobargs>

  <constraint type="resource" reason="Does not match">
    <and>
      <eq fact="resource.os.family" factvalue="jobargs.os" match="regexp" />
      <eq fact="resource.cpu.architecture" factvalue="jobargs.cpu"
match="regexp"/>

      <or>
        <and>
             <defined fact="env.VENDOR" />
           <eq fact="resource.os.vendor" factvalue="env.VENDOR" match="regexp" /
>
        </and>
        <undefined fact="env.VENDOR" />
```

```
66       </or>
67     </and>
68   </constraint>
69
70   <job>
71     <fact name="description"
72           type="String"
73          value="This example job is a reference for a simple test iterator. It
is useful for demonstrating how policies and job args can be used to target the job
to a particular resource." />
74   </job>
75
76 </policy>
77
```

Classes and Methods

Definitions:

Job

A representation of a running job instance.

Joblet

Defines execution on the resource.

MatrixInfo

A representation of the matrix grid object, which provides operations for retrieving and creating grid objects in the system. MatrixInfo is retrieved using the built-in `getMatrix()` function. Write capability is dependent on the context in which `getMatrix()` is called. For example, in a joblet process on a resource, creating new grid objects is not supported.

GroupInfo

A representation of Group grid objects. Operations include retrieving the group member lists and adding/removing from the group member lists, and retrieving and setting facts on the group.

Exec

Used to manage command line execution on resources.

ParameterSpace

Defines a parameter space to be used by the scheduler to create a Joblet set. A parameter space might consist of rows of columns or a list of columns that is expanded and can be turned into a cross product.

Job Details

The following sections describe the DemoIterator job:

zosadmin deploy

The deployment for the DemoIterator job is performed by lines 20-29 of demoIterator.jdl (page 136). When jobs are deployed into the grid, they can optionally be organized for grouping. In this case, the demoIterator job is added to the group named examples, and can be displayed in the Cloud Manager Orchestration Console in the Explorer view at the location:

```
/Orchestration Servers/Grid_Name/Jobs/examples
```

For a general overview of how jobs are added to groups during deployment, see "Deploying a Sample Job" in the "*NetIQ Cloud Manager 2.1.1 Orchestration Administrator Reference*."

job_started_event

When the DemoIterator job receives a job_started_event, it creates a ParameterSpace JDL class and adds the number of rows as indicated by the value of the argument `numJoblets` (see lines 42-46 in demoIterator.jdl (page 136)). A ParameterSpace object is like a spreadsheet, containing rows and columns of information that might all be given to one joblet or sliced up across many joblets at schedule time. In this case, the ParameterSpace is told that maxJobletSize is 1 (see line 47), meaning a joblet instance is created for each row in the ParameterSpace during job scheduling (see line 48).

Not shown in this example is the fact that a joblet can get access to this "spreadsheet" of information by calling `self.getParameterSpace()`, and calling `hasNext()` and `next()` to enumerate through each row of information. To learn more about putting information in a ParameterSpace object from a job and obtaining that information from the JobletParameterSpace object from a joblet, see ParameterSpace (page 266).

The resource that runs the joblet is determined from the resource constraint specified in lines 18-30 and 55-68 of demoIterator.policy (page 138), and from the values specified for the parameters `os` and `cpu` supplied on the command line. If these parameters are not specified on the command line, the default value for both is the regular expression `.*`, which means to include everything.

The constraints at lines 18-30 in demoIterator.policy (page 138) define the work load for the resources. In this case, resources do not accept jobs if there are already four jobs queued up, and are not to run jobs if there are two or more jobs currently in progress.

To learn more about setting `start`, `resource`, or `accept` constraints in a policy file, see "Job Arguments and Parameter Lists in Policies" on page 19.

joblet_started_event

As the DemoIterator joblet is executed on a particular resource, it receives a `joblet_started_event`. When this happens, the DemoIterator joblet simply sleeps for a random amount of time to stagger the execution of the joblets and then sends a command to the operating system, if one was supplied as a job argument. The command is executed on the target operating system using the built-in function `system()`, which is an alternative to using the more feature-rich class Exec.

For more information on sending commands to the operating system using the Exec class, see Exec .

After the joblet is finished running, a joblet_completed_event is sent to demoIteratorJob, which increments the variable `completed`, and posts the updated value to the job fact `jobinstance.memo` (see lines 50-52 in demoIterator.jdl (page 136)). You can see the text for the memo displayed on the Job Log tab in the list of running jobs in the Cloud Manager Orchestration Console.

For more information, see "Stopping and Starting Orchestration Components" in the *NetIQ Cloud Manager 2.1.1 Orchestration Administrator Reference*.

Configure and Run

Execute the following commands to deploy and run `demoIterator.job`:

1. Deploy `demoIterator.job` into the grid:

   ```
   > zosadmin deploy demoIterator.job
   ```

2. Display the list of deployed jobs:

   ```
   > zos joblist
   ```

   demoIterator should appear in this list.

3. Run the job on the first available resource without regard to OS or CPU, and use the default value for number of joblets, which is 100:

   ```
   > zos run demoIterator
   ```

4. Run 10 joblets on Intel Windows resources, and launch the Notepad application on each one:

   ```
   > zos run demoIterator numJoblets=10 cmd=dir os=windows cpu=i386
   ```

NOTE: If a resource with the matching OS is not available, the job remains in the "waiting" state.

Here is an example that runs the `pwd` command on three joblets on the Linux operating system:

```
> zos run demoIterator numJoblets=3 cmd=pwd os=linux
JobID: zenuser.demoIterator.417

zos log zenuser.demoIterator.417
job_started_event
Launching  3  joblets
[freeze] Hi from joblet  1
[freeze] /var/opt/novell/zenworks/zos/agent/node.default/freeze/
zenuser.demoIterator.417.1
[skate] Hi from joblet  0
[skate] /var/opt/novell/zenworks/zos/agent/node.default/skate/
zenuser.demoIterator.417.0
[melt] Hi from joblet  2
[melt] /var/opt/novell/zenworks/zos/agent/node.default/melt/
zenuser.demoIterator.417.2
```

See Also

- Setting Constraints Using Policies (see Section 2.3, "Policies," on page 18 and Chapter 4, "Understanding Grid Object Facts, Computed Facts, and Custom Facts," on page 51).
- quickie.job (page 142) demonstrates how a job starts up multiple instances of a joblet on one or more resources. The Joblet class defines how a joblet is executed on a resource.
- Setting default parameter values using policies
- Configuring constraints in a policy file
- Naming conventions for policy facts (see Section 5.1.1, "Cloud Manager Orchestration Datagrid Filepaths," on page 109)
- Facts provided by the Orchestration system that can be referenced within a JDL file
- Using the zos command line tool (see "The zos Command Line Tool" in the *NetIQ Cloud Manager 2.1.1 Orchestration Server Command Line Reference*.
- Running commands using the Exec class

quickie.job

Demonstrates a job starting up multiple instances of a joblet on one or more resources. Because this job simply launches and returns immediately, it can also be useful for testing network latency.

Usage

```
> zos login --user zenuser
Please enter current password for 'zenuser':
Logged into grid as zenuser

> zos jobinfo --detail quickie
Jobname/Parameters    Attributes
------------------    ----------
quickie            Desc: This example job does absolutely nothing. It just
                         returns immediately. For testing network latency.

    sleeptime      Desc: time to sleep (in seconds)
                   Type: Integer
                Default: 0

    numJoblets     Desc: joblets to run
                   Type: Integer
                Default: 100
```

Description

The files that make up the Quickie job include:

```
quickie                                    # Total: 88 lines
|-- quickie.jdl                            #   48 lines
`-- quickie.policy                         #   40 lines
```

quickie.jdl

```
 1  # ------------------------------------------------------------------------
-
 2  #  Copyright (C) 2010 Novell, Inc. All Rights Reserved.
 3  #
 4  #  NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 5  #  WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 6  #  FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGMENT.  NOVELL, THE AUTHORS
 7  #  OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 8  #  FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 9  #  TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
10  #  OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
11  # ------------------------------------------------------------------------
-
12  #  $Id: quickie.jdl 10344 2009-11-20 21:46:43Z jastin $
13  # ------------------------------------------------------------------------
-
14
15  import time
16
17  #
18  # Add to the 'examples' group on deployment
19  #
20  if __mode__ == "deploy":
21      try:
22          jobgroupname = "examples"
23          jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
24          if jobgroup == None:
25              jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
```

```
26          jobgroup.addMember(__jobname__)
27      except:
28          exc_type, exc_value, exc_traceback = sys.exc_info()
29        print "Error adding %s to %s group: %s %s" % (__jobname__, jobgroupname,
exc_type, exc_value)
30
31
32  class quickieJob(Job):
33
34       def job_started_event(self):
35
36            # Launch the joblets
37            numJoblets = self.getFact("jobargs.numJoblets")
38            print 'Launching ', numJoblets, ' joblets'
39
40            self.schedule(quickieJoblet, numJoblets)
41
42
43  class quickieJoblet(Joblet):
44
45       def joblet_started_event(self):
46            self.setFact("joblet.memo", "quickie's memo - joblet started")
47            sleeptime = self.getFact("jobargs.sleeptime")
48            time.sleep(sleeptime)
```

quickie.policy

```
 1  <!--
 2
*=============================================================================
 3   * Copyright © 2010 Novell, Inc. All Rights Reserved.
 4   *
 5   * NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 6   * WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 7   * FITNESS FOR A PARTICULAR PURPOSE, AND NON INFRINGMENT.  NOVELL, THE AUTHORS
 8   * OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 9   * FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
10   * TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
11   * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
12
*=============================================================================
13   * $Id: quickie.policy 10344 2009-11-20 21:46:43Z jastin $
14
*=============================================================================
15   -->
16
17  <policy>
18
19       <jobargs>
20            <fact name="numJoblets"
21                  type="Integer"
22                  description="joblets to run"
23                  value="100"
24                  visible="true" />
25
26            <fact name="sleeptime"
27                  type="Integer"
```

```
28                  description="time to sleep (in seconds)"
29                  value="0"
30                  visible="true" />
31        </jobargs>
32
33        <job>
34             <fact name="description"
35                   type="String"
36                value="This example job does absolutely nothing. It just returns
immediately. For testing network latency." />
37        </job>
38
39   </policy>
40
```

Classes and Methods

Definitions:

Job

A representation of a running job instance.

Joblet

Defines execution on the resource.

MatrixInfo

A representation of the matrix grid object, which provides operations for retrieving and creating grid objects in the system. MatrixInfo is retrieved using the built-in `getMatrix()` function. Write capability is dependent on the context in which `getMatrix()` is called. For example, in a joblet process on a resource, creating new grid objects is not supported.

GroupInfo

A representation of Group grid objects. Operations include retrieving the group member lists and adding/removing from the group member lists, and retrieving and setting facts on the group.

Job Details

The quickie job can be broken down into the following separate operations:

- "Configure and Run" on page 145
- "See Also" on page 145

zosadmin deploy

The job is first deployed into the grid, as shown in lines 20-29 of quickie.jdl (page 142). When jobs are deployed into the grid, they can optionally be organized for grouping. In this example, the Quickie job is added to the group named `examples` and displays in the Cloud Manager Orchestration Console in the Explorer view at the location:

`/Orchestration Servers/`*`Grid_Name`*`/Jobs/examples`

For a general overview of how jobs are added to groups during deployment, see "Deploying a Sample Job" in the *NetIQ Cloud Manager 2.1.1 Orchestration Administrator Reference*.

job_started_event

As shown in line 37 of quickie.jdl (page 142), scheduling one or more instances of the Quickie joblet to run immediately is the second operation performed by the Quickie job. When the Quickie job class receives a `job_started_event()` notification, it schedules the number of QuickieJoblet instances as indicated by the value of the setting `numJoblets`, whose value might have been supplied on the command line or from the quickie.policy file (see line 20-24 in quickie.policy (page 143)).

joblet_started_event

The final operation performed by the Quickie job is for the joblet to sleep an amount of time as specified by the value of the setting `sleeptime` (see line 48 in quickie.jdl (page 142)), and then exit.

Configure and Run

1. Deploy quickie.job into the grid:

   ```
   > zosadmin deploy quickie.job
   ```

2. Display the list of deployed jobs:

   ```
   > zos joblist
   ```

   quickie should appear in this list.

3. Run the job on one or more resources using the default values for numJoblets and sleeptime:

   ```
   > zos run quickie
   ```

4. Run the job on one or more resources using supplied values for `numJoblets` and `sleeptime`:

   ```
   > zos run quickie numJoblets=10 sleeptime=3
   JobID: zenuser.quickie.418

   > zos status zenuser.quickie.418
   Completed

   > zos log zenuser.quickie.418
   Launching  10  joblets
   ```

   Ten joblets will be run simultaneously, depending on the number of resources available in the grid and how many simultaneous jobs each resource is configured to run. After the job runs, each quickie joblet instance simply starts up, sleeps for 3 seconds, and then exits.

See Also

- Setting Constraints Using Policies (see Section 2.3, “Policies,” on page 18 and Chapter 4, “Understanding Grid Object Facts, Computed Facts, and Custom Facts,” on page 51).
- Scheduling multiple instances of a joblet

7.5 Deployable Job Examples: General Purpose

The following examples demonstrate general purpose job concepts:

dgtest.job

This job demonstrates downloading a file from the datagrid.

Usage

```
> zos login --user zenuser
Please enter current password for 'zenuser':
Logged into grid as zenuser

> zos jobinfo --detail dgtest
Jobname/Parameters    Attributes
------------------    ----------
dgtest                 Desc: This job demonstrates downloading from the Datagrid

    multicast        Desc: Whether to download using multicast or unicast
                     Type: Boolean
                  Default: false

    filename         Desc: The filename to download from the Datagrid
                     Type: String
                 Default: None! Value must be specified
```

Description

Demonstrates usage of the datagrid to download a file stored on the Orchestration Server to a node. For additional background information, see Section 5.1, “Defining the Datagrid,” on page 109.

Because it typically grows quite large, the physical location of the Orchestration Server root directory is important. Use the following procedure to determine the location of the datagrid in the Orchestration Console:

1. Select the grid id on the left in the Orchestration Console Explorer tree >
2. Click the *Constraints*/*Facts* tab.

   The read-only fact name (`matrix.datagrid.root`) is located here by default:

   ```
   /var/opt/novell/zenworks/zos/server
   ```

   The top level directory name is dataGrid.

   Contents of the Orchestration Server can be seen with the command:

   ```
   > zos dir grid:///
        <DIR>        Feb-17-2010 15:10 installs
        <DIR>        Feb-17-2010 15:10 jobs
        <DIR>        Feb-17-2010 15:10 lib
        <DIR>        Feb-24-2010 15:59 users
        <DIR>        Feb-17-2010 15:10 vms
   ```

Job Filesdg

The files that make up the Dgtest job include:

```
dgtest                                          # Total: 238 lines
|-- dgtest.jdl                                  #  172 lines
`-- dgtest.policy                               #   66 lines
```

dgtest.jdl

```
  1  # ------------------------------------------------------------------- ----
---
  2  #  Copyright © 2010 Novell, Inc. All Rights Reserved.
  3  #
  4  #  NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
  5  #  WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY ,
  6  #  FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGMENT.  NOVELL, THE AUTHORS
  7  #  OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
  8  #  FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF
CONTRACT,
  9  #  TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
 10  #  OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 11  # ------------------------------------------------------------------------
--
 12  #  $Id: dgtest.jdl 10344 2009-11-20 21:46:43Z jastin $
 13  # ------------------------------------------------------------------------
--
 14
 15  """
 16  Example usage of DataGrid to download a file stored on the Server to a node.
 17
 18  Setup:
 19      Before running the job, you must:
 20          (1) Create a dgtest resource group using the management console.
 21          (2) Copy a suitable file into the Server DataGrid
 22          (3) Modify the dgtest policy with the filename to download
 23                  (to not use the default test file).
 24
 25      For example, use the following command to copy the file 'suse-10-fla
t.vmdk'
 26      into the deployment area for the job 'dgtest'
 27          >zos mkdir grid:///images
 28
 29          >zos copy suse-10-flat.vmdk grid:///images/
 30
 31      To verify the file is there:
 32          >zos dir grid:///images
 33
 34
 35  To start the job after the above setup steps are complete:
 36          >zos run dgtest filename=suse-10-flat.vmdk
 37
 38  """
 39  import os,time
 40
 41  #
 42  # Add to the 'examples' group on deployment
 43  #
 44  if __mode__ == "deploy":
 45      try:
 46          jobgroupname = "examples"
 47          jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
 48          if jobgroup == None:
 49              jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
 50          jobgroup.addMember(__jobname__)
 51      except:
 52          exc_type, exc_value, exc_traceback = sys.exc_info()
 53          print "Error adding %s to %s group: %s %s" % (__jobname__, jobgr
oupname, exc_type, exc_value)
 54
 55
 56  class test(Job):
 57
 58       def job_started_event(self):
 59            filename = self.getFact("jobargs.filename")
 60            print "Starting Datagrid Test Job."
 61            print "Filename: %s" % (filename)
```

```
 62
 63            rg = None
 64            try:
 65               rg = getMatrix().getGroup("resource","dgtest")
 66            except:
 67               # no such group
 68               pass
 69
 70            if rg == None:
 71               self.fail("The resource group 'dgtest' was not found. It is
required for this job.")
 72               return
 73
 74            members = rg.getMembers()
 75            count = 0
 76            for resource in members:
 77                if resource.getFact("resource.online") == True and \
 78                      resource.getFact("resource.enabled") == True:
 79                     count += 1
 80
 81            memo = "Scheduling Datagrid Test on %d Joblets" % (count)
 82            self.setFact("jobinstance.memo",memo)
 83            print memo
 84            self.schedule(testnode,count)
 85
 86
 87  class testnode(Joblet):
 88
 89       def joblet_started_event(self):
 90            jobletnum = self.getFact("joblet.number")
 91            print "Running datagrid test joblet #%d" % (jobletnum)
 92            filename = self.getFact("jobargs.filename")
 93            multicast = self.getFact("jobargs.multicast")
 94
 95            # Test download a file from server job directory
 96            dg_url = "grid:///images/" + filename
 97
 98            # Create an intance of the JDL DataGrid object
 99            # This object is used to manage DataGrid operations
100            dg = DataGrid()
101
102            # Set to always force a download.
103            dg.setCache(False)
104
105            # Set whether to use multicast or unicast
106            # If set to True, then the following  4 multicast
107            # options are applicable
108            dg.setMulticast(multicast)
109
110            # how long to wait for a quorom (milliseconds)
111            #dg.setMulticastWait( 10000 )
112
113            # Number of receivers that constitute a quorum
114            #dg.setMulticastQuorum(4)
115
116            # Requested data rate in bytes per second. 0 means use default
117            #dg.setMulticastRate(0)
118
119            # Min number of receivers
120            #dg.setMulticastMin(1)
121
122            if multicast:
123                mode = "multicast"
124            else:
125                mode = "unicast"
126
127            memo = "Starting %s download of file: %s" % (mode,dg_url)
128            self.setFact("joblet.memo",memo)
129            print memo
130
```

```
131             # Destination defaults to Node's Joblet dir.
132             # Change this path to go to any other local filesystem.
133             # e.g. to store in /tmp:
134             #     dest = "/tmp/" + filename
135             dest = filename
136             try:
137                 dg.copy(dg_url,dest)
138             except:
139                 exc_type, exc_value, exc_traceback = sys.exc_info()
140                 retryUnicast = False
141                 if multicast == True:
142                     # If node's OS and/or NIC does not fully support multi cast,
143                     # then the node will timeout waiting for broadcasts.
144                     # Note the error and fallback to unicast
145                     if exc_type != None and len(str(exc_type)) > 0:
146                         msg = str(exc_type)
147                         index = msg.find("Multicast receive timed out")
148                         retryUnicast = index != -1
149 
150                 if retryUnicast:
151                     memo = "Multicast timeout. Fallback to unicast"
152                     self.setFact("joblet.memo",memo)
153                     print memo
154                     dg.setMulticast(False)
155                     dg.copy(dg_url,dest)
156                 else:
157                     raise exc_type,exc_value
158 
159             if os.path.exists(dest):
160                  print dg_url + " downloaded successfully."
161 
162                  # Show directory listing of downloaded file to job log
163                  if self.getFact("resource.os.family") == "windows":
164                      cmd = "dir %s" % (dest)
165                  else:
166                      cmd = "ls -lsart %s" % (dest)
167 
168                  system(cmd)
169             else:
170                  raise RuntimeError, "Datagrid copy() failed"
171 
172             print "Datagrid test completed"
```

dgtest.policy

```
 1  <!--
 2 
*================================================================================
 3   * Copyright © 2010 Novell, Inc. All Rights Reserved.
 4   *
 5   * NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 6   * WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 7   * FITNESS FOR A PARTICULAR PURPOSE, AND NON INFRINGMENT.  NOVELL, THE AUTHORS
 8   * OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 9   * FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
10   * TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
11   * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
12 
*================================================================================
13   * $Id: dgtest.policy 10344 2009-11-20 21:46:43Z jastin $
14 
*================================================================================
15   -->
16 
17  <policy>
18 
19       <jobargs>
20 
```

```
21            <!--
22               Name of file that is stored in the Datagrid area to
23               download to the resource.
24
25               A value for this fact the 'zos run' is assigned when
26               using the 'zos run' command.
27            -->
28            <fact name="filename"
29                  type="String"
30                  description="The filename to download from the Datagrid"
31                  />
32
33            <fact name="multicast"
34                  type="Boolean"
35                  description="Whether to download using multicast or unicast"
36                  value="false" />
37
38       </jobargs>
39
40       <job>
41            <fact name="description"
42                  type="String"
43                  value="This job demonstrates downloading from the Datagrid" />
44
45            <!-- limit to one per host -->
46            <fact name="joblet.maxperresource"
47                  type="Integer"
48                  value="1" />
49       </job>
50
51
52       <!--
53          This job will only run on resources in the "dgtest" resource group.
54
55          You must create a Resource Group named 'dgtest' using the management
56          console and populate the new group with resources that you wish to have
57          participate in the datagrid test.
58       -->
59       <constraint type="resource" reason="No resources are in the dgtest group"
>
60
61            <contains fact="resource.groups" value="dgtest"
62              reason="Resource is not in the dgtest group" />
63
64       </constraint>
65
66  </policy>
```

Classes and Methods

Definitions:

Job

A representation of a running job instance.

Joblet

Defines execution on the resource.

MatrixInfo

A representation of the matrix grid object, which provides operations for retrieving and creating grid objects in the system. MatrixInfo is retrieved using the built-in `getMatrix()` function. Write capability is dependent on the context in which `getMatrix()` is called. For example, in a joblet process on a resource, creating new grid objects is not supported.

GroupInfo

A representation of Group grid objects. Operations include retrieving the group member lists and adding/removing from the group member lists, and retrieving and setting facts on the group.

test

Class test (line 56 in dgtest.jdl (page 148) is derived from the Job class.

testnode

Class testnode (line 87 in dgtest.jdl (page 148) is derived from the Joblet (page 254) class.

Job Details

dgtest.job can be broken down into the following parts:

- "Configure and Run" on page 153

Policy

In addition to describing the `filename` and `multicast` jobargs and the default settings for `multicast` (lines 19-38) in the dgtest.policy (page 150) file, there is the <job/> section (lines 40-49), which describes static facts. You must assign the `filename` argument when executing this example. This is only the name of the file in the "images" area of the Orchestration Server. For example, for `grid:///images/disk.img`, just assign `disk.img` to the argument. This file must be in the Orchestration file system for fetching and delivering to remote nodes used in this example.

To populate the Orchestration Server, use the `zos copy` command. For example, for a file named `suse-10-flat.vmd` in the current directory, use the following command:

```
> zos mkdir grid:///images
> zos copy suse-10-flat.vmd grid:///images/
```

The `multicast` jobarg is a Boolean, defaulted to `false` so that unicast is used for transport. Set this value to `true` to use multicast transport for delivery of the file.

The policy also describes a `resource.groups` constraint.This requires a resource group named `dgtest` (lines 52-64 in dgtest.policy (page 150)) and that group should have member nodes. Consequently, you must create this resource group using the Orchestration Console and assign it some nodes to run this example successfully.

zosadmin deploy

When the Orchestration Server deploys a job for the first time (see Section 3.5, "Deploying Jobs," on page 37), the job JDL files are executed in a special deploy mode. Looking at dgtest.jdl (page 148), you might notice that when the job is deployed (line 44), either through the Orchestration Console or the zosadmin deploy command, that it attempts to find the `examples` jobgroup (lines 46-47), create it if missing (lines 48-49), and add the dgtest job to the group (line 50).

If this deployment fails for some reason, an exception is thrown (line 51), which prints (line 53) the job name, group name, exception type, and value.

job_started_event

In dgtest.jdl (page 148), the `test` class (line 56) defines only the required job_started_event (line 58) method. This method runs on the Orchestration server when the job is run to launch the joblets.

When `job_started_event` is executed, it gets the name of the file assigned to the `jobargs.filename` variable and prints useful tracing information (lines 59-61). It then tries to find the resource group named `dgtest`. If the resource group does not exist, the member `fail` string is set to inform the user and returns without scheduling the joblet(s) (lines 63-72).

After finding the `dgtest` group, the job gets the member list and determines how many nodes are online and enabled. The total `count` is stored in lines 74-79. After setting the memo line in the Orchestration Console (81-82), the job schedules `count` number of `testnode` joblets (line 84).

joblet_started_event

In dgtest.jdl (page 148), the `testnode` class (line 87) defines only the required joblet_started_event (line 89) method. This method runs on the Orchestration Agent nodes when scheduled by a Job (page 252) class.

The joblet_started_event prints some trace information (lines 90-91), gets the name of the file to transfer (line 92) and the mode of transfer (line 93), and creates the grid URL for the file (line 96).

A DataGrid (page 242) is instantiated (line 100), set not to cache (line 103), and set to use the multicast jobarg (line 108). The next four settings control multicast behavior are commented out (lines 111, 114, 117, and 120).

The joblet prints a memo line for the Orchestration Console (lines 122-128), sets the location for the file on the local node (line 135), and tries to transfer the file from the datagrid (line 137).

If the datagrid copy at line 137 fails for some reason, we have a retry mechanism in the exception handler (lines 138-157). The information for why the exception occurred is fetched (line 139).

The variable `retryUnicast` (line 140) is set `False` and will only be set `True` if the failed download attempt was using multicast transport and the exception type has the string "Multicast receive timed out" (lines 140-148). If the `timed out` string is not found, the triad assigns the `retryUnicast` a value of `-1`. With this logic, either multicast timeout or not, a unicast attempt is made if multicast fails.

If you get to line 150 from a failed multicast copy, a memo for the Orchestration Console is set and printed to the log (151-152), `setMulticast` is set to `false` (154), and another copy from the datagrid is attempted.

If we get to line 150 from a failed unicast copy, an exception is raised (line 157) and we're done.

Configure and Run

```
> zos run dgtest filename=suse-10-flat.vmd
JobID: zenuser.dgtest.323
```

Looks like it ran successfully; let's see what the log says:

```
> zos log zenuser.dgtest.323
Starting Datagrid Test Job.
Filename: suse-10-flat.vmd
Job 'zenuser.dgtest.323' terminated because of failure. Reason: The resource group
'dgtest' was not found. It is required for this job.
```

There is no resource group. Using the Orchestration Console, create the resource group `dgtest`:

```
> zos run dgtest filename=suse-10-flat.vmd
JobID: zenuser.dgtest.324

> zos log zenuser.dgtest.324
Starting Datagrid Test Job.
Filename: suse-10-flat.vmd
Scheduling Datagrid Test on 0 Joblets
```

NOTE: No joblets were scheduled because we have no active nodes in the group.

Using the Orchestration Console, populate the `dgtest` group with nodes that are both online and enabled:

```
> zos run dgtest filename=suse-10-flat.vmd
JobID: zenuser.dgtest.325

> zos log zenuser.dgtest.325
Starting Datagrid Test Job.
Filename: suse-10-flat.vmd
Scheduling Datagrid Test on 2 Joblets
[freeze] Running datagrid test joblet #0
[freeze] Starting unicast download of file: grid:///images/suse-10-flat.vmd
[freeze] Traceback (innermost last):
[freeze]   File "dgtest.jdl", line 170, in joblet_started_event
[freeze] copy() failed: DataGrid file "/images/suse-10-flat.vmd" does not exist.
Job 'zenuser.dgtest.325' terminated because of failure. Reason: Job failed because
of too many joblet failures (job.joblet.maxfailures = 0)
[melt] Running datagrid test joblet #1
[melt] Starting unicast download of file: grid:///images/suse-10-flat.vmd
[melt] Traceback (innermost last):
[melt]   File "dgtest.jdl", line 170, in joblet_started_event
[melt] copy() failed: DataGrid file "/images/suse-10-flat.vmd" does not exist.
```

Because the path and the file in the datagrid are missing, we need to create and populate them:

```
> zos mkdir grid:///images
Directory created.

> zos copy suse-10-flat.vmd grid:///images/
suse-10-flat.vmd copied.

> zos run dgtest filename=suse-10-flat.vmd
JobID: zenuser.dgtest.326

> zos log zenuser.dgtest.326
Starting Datagrid Test Job.
Filename: suse-10-flat.vmd
Scheduling Datagrid Test on 2 Joblets
[melt] Running datagrid test joblet #1
[melt] Starting unicast download of file: grid:///images/suse-10-flat.vmd
[melt] grid:///images/suse-10-flat.vmd downloaded successfully.
[melt] 16732 -rw-r--r-- 1 root root 17108462 Dec 21 21:32 suse-10-flat.vmd
[melt] Datagrid test completed
[freeze] Running datagrid test joblet #0
[freeze] Starting unicast download of file: grid:///images/suse-10-flat.vmd
[freeze] grid:///images/suse-10-flat.vmd downloaded successfully.
[freeze] 16732 -rw-r--r-- 1 root root 17108462 Dec 21 21:31 suse-10-flat.vmd
[freeze] Datagrid test completed
```

Finally, the file is deployed from the datagrid and copied successfully. However, you will not find it if you look for it on the agent after the joblet is finished. By default, the file is deployed only for the joblet's lifetime into a directory for the joblet, like the following:

```
/var/opt/novell/zenworks/zos/agent/node.default/melt/zenuser.dgtest.326.0
```

So, for a more permanent demonstration, see lines 132-134 in dgtest.jdl (page 148). Uncomment line 134 and comment out line 135 to store your file in the `/tmp` directory and have it continue to exist on the agent after the joblet executes completely.

failover.job

A test job that demonstrates handling of joblet failover.

Usage

```
> zos login --user zenuser
Please enter current password for 'zenuser':
 Logged into grid as zenuser

> zos jobinfo --detail failover
Jobname/Parameters    Attributes
------------------    ----------
failover              Desc: This test jobs can be used to demonstrate joblet
                            failover handling.

    sleeptime         Desc: specify the execute length of joblet before failure in
                            seconds
                      Type: Integer
                   Default: 7

    numJoblets        Desc: joblets to run
                      Type: Integer
                   Default: 1
```

Description

Schedules one joblet, which fails, then re-instantiates in a repeating cycle until a specified retry limit is reached and the Orchestration Server does not create another instance. This example demonstrates how the orchestration server can be made more robust, as described in Section 3.11, “Improving Job and Joblet Robustness,” on page 45.

The files that make up the Failover job include:

```
failover                                    # Total: 94 lines
|-- failover.jdl                            #   64 lines
`-- failover.policy                         #   30 lines
```

failover.jdl

```
 1  # ------------------------------------------------------------------------
-
 2  #  Copyright © 2010 Novell, Inc. All Rights Reserved.
 3  #
 4  #  NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 5  #  WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 6  #  FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGMENT.  NOVELL, THE AUTHORS
 7  #  OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 8  #  FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 9  #  TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
10  #  OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
11  # ------------------------------------------------------------------------
-
12  #  $Id: failover.jdl 10344 2009-11-20 21:46:43Z jastin $
13  # ------------------------------------------------------------------------
-
14
15  # Test job to illustrate joblet failover and max retry limits
16  #
17  # Job args:
18  #    numJoblets - specify number of Joblets to run
19  #    sleeptime -- specify the execute length of joblet before failure in seconds
```

```
#

import sys,os,time

#
# Add to the 'examples' group on deployment
#
if __mode__ == "deploy":
    try:
        jobgroupname = "examples"
        jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
        if jobgroup == None:
            jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
        jobgroup.addMember(__jobname__)
    except:
        exc_type, exc_value, exc_traceback = sys.exc_info()
       print "Error adding %s to %s group: %s %s" % (__jobname__, jobgroupname,
exc_type, exc_value)


class failover(Job):

     def job_started_event(self):
          numJoblets = self.getFact("jobargs.numJoblets")
          print 'Launching ', numJoblets, ' joblets'
          self.schedule(failoverjoblet,numJoblets)


class failoverjoblet(Joblet):

     def joblet_started_event(self):
          print "------------------ joblet_started_event"
          print "node=%s joblet=%d" % (self.getFact("resource.id"),
self.getFact("joblet.number"))
          print "self.getFact(joblet.retrynumber)=%d" %
(self.getFact("joblet.retrynumber"))
          print "self.getFact(job.joblet.maxretry)=%d" %
(self.getFact("job.joblet.maxretry"))

          sleeptime = self.getFact("jobargs.sleeptime")
          print "sleeping for %d seconds" % (sleeptime)
          time.sleep(sleeptime)

          # This will cause joblet failure and thus retry
          raise RuntimeError, "Artifical error in joblet. node=%s" %
(self.getFact("resource.id"))
```

failover.policy

```
<!--
*================================================================================
  * Copyright © 2010 Novell, Inc. All Rights Reserved.
  *
  * NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
  * WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
  * FITNESS FOR A PARTICULAR PURPOSE, AND NON INFRINGMENT.  NOVELL, THE AUTHORS
  * OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 * FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
  * TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
  * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

*================================================================================
  * $Id: failover.policy 10344 2009-11-20 21:46:43Z jastin $
```

```
14
*=========================================================================
15   -->
16
17  <policy>
18      <jobargs>
19            <fact name="sleeptime" description="specify the execute length of
joblet before failure in seconds" value="7" type="Integer"  />
20            <fact name="numJoblets" description="joblets to run" value="1"
type="Integer" />
21      </jobargs>
22
23      <job>
24            <fact name="description" value="This test jobs can be used to
demonstrate joblet failover handling." type="String" />
25
26           <!-- Number of times to retry joblet on failure -->
27           <fact name="joblet.maxretry" type="Integer" value="3" />
28      </job>
29  </policy>
30
```

Classes and Methods

Definitions:

Class failover in line 25 of failover.jdl (page 156) is derived from the Job (page 252) class; and the class failoverjoblet in line 33 of failover.jdl (page 156) is derived from the Joblet (page 254) class.

Job

A representation of a running job instance.

Joblet

Defines execution on the resource.

MatrixInfo

A representation of the matrix grid object, which provides operations for retrieving and creating grid objects in the system. MatrixInfo is retrieved using the built-in `getMatrix()` function. Write capability is dependent on the context in which `getMatrix()` is called. For example, in a joblet process on a resource, creating new grid objects is not supported.

GroupInfo

A representation of Group grid objects. Operations include retrieving the group member lists and adding/removing from the group member lists, and retrieving and setting facts on the group.

failover

Class failover (line 39 in dgtest.jdl (page 148) is derived from the Job class.

failoverjoblet

Class failoverjoblet (line 47 in dgtest.jdl (page 148) is derived from the Joblet (page 254) class.

Job Details

The following sections describe the Failover job:

- "Configure and Run" on page 160
- "See Also" on page 161

zosadmin deploy

In failover.policy (page 157), in addition to describing the jobargs and default settings for `sleeptime` and `numJoblets` (lines 18-21), the <job/> section (lines 23-28) describes static facts. Note that the `joblet.maxretry` attribute in line 27 has a default setting of 0 but is set here to 3. This attribute can also be modified in the failover.jdl (page 156) file by inserting a line between line 41 and 42, as shown in the following example:

```
41        def job_started_event(self):
++             self.setFact("job.joblet.maxretries", 3)
42             numJoblets = self.getFact("jobargs.numJoblets")
```

job_started Event

After the Orchestration Server deploys a job for the first time (see Section 3.5, "Deploying Jobs," on page 37), the job JDL files are executed in a special "deploy" mode. When the job is deployed (line 27, failover.jdl (page 156), it attempts to find the `examples` jobgroup (lines 29-30), creates it if is missing (lines 31-32), and adds the failover job to the group (line 33).

Jobs can be deployed using either the Orchestration Console or the `zosadmin deploy` command. If the deployment fails for some reason, an exception is thrown (line 34), which prints the job name (line 36), group name, exception type, and value.

job_started Event

In failover.jdl (page 156), the failover class (line 39) defines only the required `job_started_event` (line 41) method. This method runs on the Orchestration Server when the job is run to launch the joblets.

On execution, the `job_started_event` simply gets the number of joblets to create (numJoblets in line 42), then schedules that specified number of instances (line 44) of the `failoverjoblet class.failoverjoblet`. The failoverjoblet class (lines 47-60) defines only the required `joblet_started_event` (line 49) method.

When executed on an agent node, the `joblet_started_event` prints some helpful information for tracking execution (lines 50-53). The first output is where the joblet is running and which instance is running (line 51). The current joblet retry number (line 52) is displayed, followed by the job's static `joblet.maxretry` (line 53) that was specified in the policy file.

The joblet then sleeps for `jobargs.sleeptime` seconds (lines 55-57) and on waking raises an exception of type `RuntimeError` (line 60).

This is the point of this example. After a `RuntimeError` exception is thrown, the zos server attempts to run the same instance of the joblet again if `job.joblet.maxretry` (default is 0) is less than or equal to `joblet.retrynumber`.

Configure and Run

You must be logged into the Orchestration Server before you run `zosadmin` or `zos` commands.

1. Deploy failover.job into the grid:

   ```
   > zosadmin deploy failover.job
   JobID: zenuser.failover.269
   ```

   The job appears to have run successfully, now take a look at the log and see the joblet failure and being relaunched until finally the "maxretry" count is exceeded and the job exits with a failure status:

2. Display the list of deployed jobs:

   ```
   > zos joblist
   ```

   failover should appear in this list.

3. Run the job on one or more resources using the default values for numJoblets and sleeptime, specified in the `failover.policy` file:

   ```
   > zos run failover sleeptime=1 numJoblets=2
   JobID: zenuser.failover.269
   ```

The job appears to have run successfully, now take a look at the log and see the joblet failure and being relaunched until finally the `maxretry` count is exceeded and the job exits with a failure status:

```
> zos log zenuser.failover.269Launching  2  joblets
[melt] ------------------ joblet_started_event
[melt] node=melt joblet=1
[melt] self.getFact(joblet.retrynumber)=0
[melt] self.getFact(job.joblet.maxretry)=3
[melt] sleeping for 1 seconds
[melt] Traceback (innermost last):
[melt]   File "failover.jdl", line 60, in joblet_started_event
[melt] RuntimeError: Artifical error in joblet. node=melt
[freeze] ------------------ joblet_started_event
[freeze] node=freeze joblet=0
[freeze] self.getFact(joblet.retrynumber)=0
[freeze] self.getFact(job.joblet.maxretry)=3
[freeze] sleeping for 1 seconds
[freeze] Traceback (innermost last):
[freeze]   File "failover.jdl", line 60, in joblet_started_event
[freeze] RuntimeError: Artifical error in joblet. node=freeze
[melt] ------------------ joblet_started_event
[melt] node=melt joblet=0
[melt] self.getFact(joblet.retrynumber)=1
[melt] self.getFact(job.joblet.maxretry)=3
[melt] sleeping for 1 seconds
[melt] Traceback (innermost last):
[melt]   File "failover.jdl", line 60, in joblet_started_event
[melt] RuntimeError: Artifical error in joblet. node=melt
[freeze] ------------------ joblet_started_event
[freeze] node=freeze joblet=1
[freeze] self.getFact(joblet.retrynumber)=1
[freeze] self.getFact(job.joblet.maxretry)=3
[freeze] sleeping for 1 seconds
[freeze] Traceback (innermost last):
[freeze]   File "failover.jdl", line 60, in joblet_started_event
[freeze] RuntimeError: Artifical error in joblet. node=freeze
[melt] ------------------ joblet_started_event
[melt] node=melt joblet=1
```

```
[melt] self.getFact(joblet.retrynumber)=2
[melt] self.getFact(job.joblet.maxretry)=3
[melt] sleeping for 1 seconds
[melt] Traceback (innermost last):
[melt]   File "failover.jdl", line 60, in joblet_started_event
[melt] RuntimeError: Artifical error in joblet. node=melt
[freeze] ------------------ joblet_started_event
[freeze] node=freeze joblet=0
[freeze] self.getFact(joblet.retrynumber)=2
[freeze] self.getFact(job.joblet.maxretry)=3
[freeze] sleeping for 1 seconds
[freeze] Traceback (innermost last):
[freeze]   File "failover.jdl", line 60, in joblet_started_event
[freeze] RuntimeError: Artifical error in joblet. node=freeze
```

See Also

- Setting Constraints Using Policies (Section 2.3, “Policies,” on page 18 and Chapter 4, “Understanding Grid Object Facts, Computed Facts, and Custom Facts,” on page 51).
- Executing Commands Using Exec (page 245)

instclients.job

Installs the Cloud Manager Orchestration client applications to the specified resource machine. Note that while most of the other examples are deployed by default, this example is not.

Detail

The following concepts are demonstrated:

- Using constraints to restrict joblet execution to a specific resource.
- Adding files to a job's directory in the datagrid, and retrieving them during joblet execution.
- Using the Exec class to send a command to the operating system. The system command is invoked directly without using the system command interpreter (either `cmd.exe` or `/bin/sh`).

Usage

```
> zosadmin login --user zosadmin Login to server: skate
Please enter current password for 'zosadmin':
Logged into grid on server 'skate'

> cd /opt/novell/zenworks/zos/server/examples
> zosadmin deploy instclients.job
instclients successfully deployed

> zos login --user zenuser
Please enter current password for 'zenuser':
 Logged into grid as zenuser

> zos jobinfo --detail instclients
Jobname/Parameters    Attributes
------------------    ----------
instclients         Desc: This job installs the clients on a resource

    host            Desc: The host name of resource to install on
                    Type: String
                 Default: None! Value must be specified
```

Description

The files that make up the instclients job include:

```
instclients                                 # Total: 138 lines
|-- instclients.jdl                         #   97 lines
`-- instclients.policy                      #   41 lines
```

instclients.jdl

```
 1  # ------------------------------------------------------------------------
-
 2  #  Copyright © 2010 Novell, Inc. All Rights Reserved.
 3  #
 4  #  NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 5  #  WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 6  #  FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGMENT.  NOVELL, THE AUTHORS
 7  #  OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 8  #  FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 9  #  TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
10  #  OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
11  # ------------------------------------------------------------------------
-
12  #  $Id: instclients.jdl 10344 2009-11-20 21:46:43Z jastin $
13  # ------------------------------------------------------------------------
-
14
15  """
16
17  Run install clients on a resource
18
19  Setup:
20      Before running the job, you must copy installers into DataGrid of
21      server.
22
23      >zos copy zosclients_windows_1_3_0_with_jre.exe grid:///\!instclients/
24
25  """
26  import os,time
27
28  #
29  # Add to the 'examples' group on deployment
30  #
31  if __mode__ == "deploy":
32      try:
33          jobgroupname = "examples"
34          jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
35          if jobgroup == None:
36              jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
37          jobgroup.addMember(__jobname__)
38      except:
39          exc_type, exc_value, exc_traceback = sys.exc_info()
40         print "Error adding %s to %s group: %s %s" % (__jobname__, jobgroupname,
exc_type, exc_value)
41
42
43  class InstClients(Job):
44
45       def job_started_event(self):
46            print "Scheduling joblet"
47            self.schedule(InstClientsJoblet)
48
49
50  class InstClientsJoblet(Joblet):
51
52       def joblet_started_event(self):
53            print "Launching Installer"
54            windowsInstaller = "zosclients_windows_2_0_0_with_jre.exe"
55            linuxInstaller = "zosclients_linux_2_0_0_with_jre.sh"
56            if self.getFact("resource.os.family") == "windows":
57                 print "Downloading Windows install"
58                 dg = DataGrid()
59                 dg.copy("grid:///!instclients/" +
windowsInstaller,windowsInstaller)
60
61                 print "Starting install"
62                 cmd = self.getcwd() + "/" + windowsInstaller + " -q "
```

```
63                e = Exec()
64                e.setCommand(cmd)
65                e.setRunAsJobUser(False)
66                e.writeStdoutToLog()
67                e.writeStderrToLog()
68                result = e.execute()
69           else:
70                print "Downloading Linux install"
71                dg = DataGrid()
72               dg.copy("grid:///!instclients/" + linuxInstaller,linuxInstaller)
73
74                print "Starting install"
75                cmd = "chmod +x " + self.getcwd() + "/" + linuxInstaller
76                print "cmd=%s" % (cmd)
77                e = Exec()
78                e.setCommand(cmd)
79                e.setRunAsJobUser(False)
80                e.writeStdoutToLog()
81                e.writeStderrToLog()
82                result = e.execute()
83
84                cmd = self.getcwd() + "/" + linuxInstaller + " -q"
85                print "cmd=%s" % (cmd)
86                e = Exec()
87                e.setRunAsJobUser(False)
88                e.setCommand(cmd)
89                e.writeStdoutToLog()
90                e.writeStderrToLog()
91                result = e.execute()
92
93           if result == 0:
94                print "Install complete"
95           else:
96                print "result=%d" % (result)
97
```

instclients.policy

```
 1  <!--
 2
*=============================================================================
 3   * Copyright © 2010 Novell, Inc. All Rights Reserved.
 4   *
 5   * NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 6   * WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY ,
 7   * FITNESS FOR A PARTICULAR PURPOSE, AND NON INFRINGMENT.  NOVELL, THE AUTHORS
 8   * OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 9   * FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
10   * TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
11   * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
12
*=============================================================================
13   * $Id: instclients.policy 10344 2009-11-20 21:46:43Z jastin $
14
*=============================================================================
15   -->
16
17  <policy>
18
19       <jobargs>
20
21            <fact name="host"
22                  type="String"
23                  description="The host name of resource to install on"
24                   />
25
26       </jobargs>
27
```

```
28       <job>
29            <fact name="description"
30                  type="String"
31                  value="This job installs the clients on a resource" />
32       </job>
33
34       <constraint type="resource" >
35
36            <eq fact="resource.id" factvalue="jobargs.host" />
37
38       </constraint>
39
40  </policy>
41
```

Classes and Methods

Definitions:

Job

A representation of a running job instance.

Joblet

Defines execution on the resource.

MatrixInfo

A representation of the matrix grid object, which provides operations for retrieving and creating grid objects in the system. MatrixInfo is retrieved using the built-in `getMatrix()` function. Write capability is dependent on the context in which `getMatrix()` is called. For example, in a joblet process on a resource, creating new grid objects is not supported.

GroupInfo

A representation of Group grid objects. Operations include retrieving the group member lists and adding/removing from the group member lists, and retrieving and setting facts on the group.

Exec

Used to manage command line execution on resources.

DataGrid

Provides a way to interact with the datagrid. Operations include copying files from the datagrid down to the resource for joblet usage and uploading files from a resource to the datagrid.

Job Details

The following sections describe the instclients job:

- "Configure and Run" on page 167
- "See Also" on page 167

zosadmin deploy

When jobs are deployed into the grid, they can optionally be placed in groups for organization and easy reference. In this case, the instclients job will be added to the group named Examples (lines 31-40), and will show up in the Cloud Manager Orchestration Console in the Explorer view at the location:

```
/Orchestration Servers/Grid_Name/Jobs/examples
```

For a general overview of how jobs are added to groups during deployment, see "Deploying a Sample Job" in the *NetIQ Cloud Manager 2.1.1 Orchestration Administrator Reference*.

job_started_event

When the instclients job receives a job_started_event, it schedules a single instance of the Instclients joblet to be run (see line 45 of instclients.jdl (page 163)). The resource that runs the joblet is determined from the resource constraint specified in instclients.policy (page 164), lines 21-24, and from the value for the parameter `host` supplied on the command line.

joblet_started_event

After the Instclients joblet is executed on a particular resource, it receives a joblet_started_event. When this happens, the Instclients joblet decides which Orchestration Client installation file to download, and the commands to execute on the operating system by checking the value of `resource.os.family` (see line 56 of instclients.jdl (page 163)). The `resource.os.family` fact does not exist in the `instclients.policy` file, but is instead provided by the Cloud Manager Orchestration system.

After deciding which operating system the joblet is being run on, the `Instclients` joblet uses the DataGrid class to download the appropriate client installation file to the current working directory of the running joblet (see lines 58-59 and 71-72 in instclients.jdl (page 163)). The URL `grid://!instclients/` points to a directory reserved for the joblet in the datagrid on the server.

After the client installation file has been downloaded from the server, the `Instclients` joblet uses the Exec class to begin the installation (see lines 63-68 and 86-91 in instclients.jdl (page 163)). As indicated by lines 66, 67, 80, 81, 89 and 90, all standard out and standard err are written to the job's log file.

To view the log file for the `Instclients` job after it has been run, you can execute the command

```
zos log instclients
```

For more information about using zos, see Section 3.5.2, "Using the zosadmin Command Line Tool to Deploy Jobs," on page 38. See the Exec class in Cloud Manager Orchestration Job Classes and JDL Syntax for more information on running commands.

NOTE: The `Instclients` job uses the Exec class twice when running on a Linux resource. The first command changes the mode of the installation file to be an executable, and the second runs the installation file.

Configure and Run

Execute the following commands to deploy and run `instclients.job`:

1. Copy client installation files into the directory reserved for the `Instclients` joblet in the datagrid of the Orchestration Server:

   ```
   zos copy zosclients_linux_2_1_0_with_jre.sh grid:///\!instclients/
   ```

   NOTE: Replace "linux" with windows, linux, solaris, etc. for your given operating system, and replace 2_1_0 with your version of the product.

   This command copies the file `zosclients_linux_2_1_0_with_jre.sh` into the datagrid job directory for instclients.

   For more information about using the Orchestration Console to copy files, type `zos copy -help`.

   NOTE: Replace windows with *linux*, *solaris*, etc. for your given operating system.

   ---

2. Deploy instclients.job into the grid by entering:

   ```
   zosadmin deploy instclients.job
   ```

3. Display the list of deployed jobs by entering:

   ```
   zos joblist
   ```

   `instclients` should appear in this list.

4. Run the job on the resource with the given host:

   ```
   zos run instclients host=my_resource_host
   ```

   Installs the Orchestration clients onto the resource with the host: my_resource_host.

See Also

- Setting Constraints Using Policies (Section 2.3, "Policies," on page 18 and Chapter 4, "Understanding Grid Object Facts, Computed Facts, and Custom Facts," on page 51).
- Scheduling multiple instances of a joblet
- Setting default parameter values using policies
- Configuring constraints in a policy file
- Naming conventions for policy facts (Section 5.1.1, "Cloud Manager Orchestration Datagrid Filepaths," on page 109.Section 5.1.1, "Cloud Manager Orchestration Datagrid Filepaths," on page 109)
- Facts provided by the Cloud Manager Orchestration system that can be referenced within a JDL file
- Using the Orchestration Console ("How Do I Interact with the Orchestration Server?")
- Running commands using the Exec class.

notepad.job

Launches the Notepad application on a Windows resource.

Usage

```
> zos login --user zenuser
Please enter current password for 'zenuser':
 Logged into grid as zenuser

> zos jobinfo --detail notepad
Jobname/Parameters    Attributes
------------------    ----------
notepad            Desc: No description available.
```

Description

The files that make up the Notepad job include:

```
notepad                                    # Total: 86 lines
|-- notepad.jdl                            #   54 lines
`-- notepad.policy                         #   32 lines
```

notepad.jdl

```
 1  # ------------------------------------------------------------------------
-
 2  #  Copyright © 2010 Novell, Inc. All Rights Reserved.
 3  #
 4  #  NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 5  #  WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 6  #  FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGMENT.  NOVELL, THE AUTHORS
 7  #  OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 8  #  FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 9  #  TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
10  #  OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
11  # ------------------------------------------------------------------------
-
12  #  $Id:  notepad.jdl 10344 2009-11-20 21:46:43Z jastin $
13  # ------------------------------------------------------------------------
-
14
15  """
16
17  Run Notepad Application on windows resoure
18
19  """
20  import os,time
21
22  #
23  # Add to the 'examples' group on deployment
24  #
25  if __mode__ == "deploy":
26      try:
27          jobgroupname = "examples"
28          jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
29          if jobgroup == None:
30              jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
31          jobgroup.addMember(__jobname__)
32      except:
33          exc_type, exc_value, exc_traceback = sys.exc_info()
34          print "Error adding %s to %s group: %s %s" % (__jobname__, jobgroupname,
exc_type, exc_value)
35
```

```
36
37  class Notepad(Job):
38
39       def job_started_event(self):
40            print "Scheduling joblet"
41            self.schedule(NotepadJoblet)
42
43
44  class NotepadJoblet(Joblet):
45
46       def joblet_started_event(self):
47            print "Starting Notepad"
48            cmd = "notepad"
49            e = Exec()
50            e.setCommand(cmd)
51            e.writeStdoutToLog()
52            e.writeStderrToLog()
53            result = e.execute()
54
```

notepad.policy

```
 1  <!--
 2
*=============================================================================
 3   * Copyright © 2010 Novell, Inc. All Rights Reserved.
 4   *
 5   * NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 6   * WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 7   * FITNESS FOR A PARTICULAR PURPOSE, AND NON INFRINGMENT.  NOVELL, THE AUTHORS
 8   * OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 9   * FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
10   * TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
11   * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
12
*=============================================================================
13   * $Id: notepad.policy 10344 2009-11-20 21:46:43Z jastin $
14
*=============================================================================
15   -->
16
17  <policy>
18
19       <constraint type="accept" >
20
21            <gt fact="jobinstance.matchingresources" value="0" reason="No
Windows's resources are available to run Notepad" />
22
23       </constraint>
24
25       <constraint type="resource" >
26
27            <eq fact="resource.os.family" value="windows" reason="Notepad only
runs on Windows OS" />
28
29       </constraint>
30
31  </policy>
32
```

Classes and Methods

Definitions:

Job

A representation of a running job instance.

Joblet

Defines execution on the resource.

MatrixInfo

A representation of the matrix grid object, which provides operations for retrieving and creating grid objects in the system. MatrixInfo is retrieved using the built-in `getMatrix()` function. Write capability is dependent on the context in which `getMatrix()` is called. For example, in a joblet process on a resource, creating new grid objects is not supported.

GroupInfo

A representation of Group grid objects. Operations include retrieving the group member lists and adding/removing from the group member lists, and retrieving and setting facts on the group.

Exec

Used to manage command line execution on resources.

Job Details

The Notepad job is broken down into three separate operations:

- "Configure and Run" on page 171
- "See Also" on page 171

zosadmin deploy

In notepad.jdl (page 168), lines 25-34 places the job into the "examples" job group. After jobs are deployed into the grid, they can optionally be placed in groups for organization and easy reference. In this case, the Notepad job is added to the group named Examples and appears in the Orchestration Console in the Explorer view at the location:

```
/Orchestration Servers/Grid_Name/Jobs/examples
```

For a general overview of how jobs are added to groups during deployment, see "Deploying a Sample Job" in the *NetIQ Cloud Manager 2.1.1 Orchestration Administrator Reference*.

job_started_event

Scheduling the Notepad joblet to run immediately is the second operation performed by the Notepad job in line 41 of notepad.jdl (page 168). When the Notepad job class receives a `job_started_event()` notification, it simply schedules the NotepadJoblet class to be run on any target device that meets the restrictions identified in the `notepad.policy` file.

As specified in lines 21 and 27 of notepad.policy (page 169), there must be at least one Windows machine available in the grid for the Notepad job to run. The `accept` constraint in lines 19-23 prevents the Notepad job from being accepted for running if there are no Windows resources available.

The `resource` constraint in lines 25-29 constrain the Orchestration Job Scheduler to choose a resource that is running a Windows OS only.

For more information on setting constraints using policies, see Section 2.3, "Policies," on page 18 and Chapter 4, "Understanding Grid Object Facts, Computed Facts, and Custom Facts," on page 51.

joblet_started_event

As specified in lines 49-53 in notepad.jdl (page 168), the joblet executing a command on the target machine is the last operation performed by the Notepad job.

In this example, after the `joblet_started_event()` method of the NotepadJoblet class gets called, the Orchestration API class named Exec is used to run the command `notepad` on is captured and written to the log file for the Notepad job.

Configure and Run

Execute the following commands to deploy and run `notepad.job`:

1. Deploy notepad.job into the grid:

   ```
   > zosadmin deploy notepad.job
   ```

2. Display the list of deployed jobs:

   ```
   > zos joblist
   ```

   notepad should appear in this list.

3. Run the job on the first available Windows resource.

   ```
   > zos run notepad
   ```

   You should now see the Windows Notepad application appear on the screen of the target Windows system. You will see the following error if there are no Windows resources.

   ```
   No Windows resources available to run Notepad
   ```

See Also

- Setting Constraints Using Policies see Section 2.3, "Policies," on page 18 and Chapter 4, "Understanding Grid Object Facts, Computed Facts, and Custom Facts," on page 51.
- Executing Commands Using Exec (page 245)

sweeper.job

This example job illustrates how to schedule a "sweep," which is an ordered, serialized scheduling of the joblets across all matching resources.

Usage

```
> zos login --user zenuser
Please enter current password for 'zenuser':
 Logged into grid as zenuser

> zos jobinfo --detail sweeper
Jobname/Parameters    Attributes
------------------    ----------
sweeper            Desc: This example job ilustrates how to schedule a 'sweep'
                         accross all matching resources.

    sleeptime      Desc: time to sleep (in seconds)
                   Type: Integer
                Default: 1
```

Options

Job

A representation of a running job instance.

Joblet

Defines execution on the resource.

MatrixInfo

A representation of the matrix grid object, which provides operations for retrieving and creating grid objects in the system. MatrixInfo is retrieved using the built-in `getMatrix()` function. Write capability is dependent on the context in which `getMatrix()` is called. For example, in a joblet process on a resource, creating new grid objects is not supported.

GroupInfo

A representation of Group grid objects. Operations include retrieving the group member lists and adding/removing from the group member lists, and retrieving and setting facts on the group.

Exec

Used to manage command line execution on resources.

sleeptime

Specifies the time in seconds that the job remains dormant before running (default 1).

Description

The files that make up the Sweeper job include:

```
sweeper                                          # Total: 140 lines
|-- sweeper.jdl                                  #   66 lines
`-- sweeper.policy                               #   74 lines
```

The ScheduleSpec (page 275) utility class is also related to this example.

sweeper.jdl

```
 1  # ------------------------------------------------------------------------
-
 2  #  Copyright © 2010 Novell, Inc. All Rights Reserved.
 3  #
 4  #  NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 5  #  WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 6  #  FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGMENT.  NOVELL, THE AUTHORS
 7  #  OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 8  #  FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 9  #  TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
10  #  OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
11  # ------------------------------------------------------------------------
-
12  #  $Id: sweeper.jdl 10344 2009-11-20 21:46:43Z jastin $
13  # ------------------------------------------------------------------------
-
14
15  import time
16
17  #
18  # Add to the 'examples' group on deployment
19  #
20  if __mode__ == "deploy":
21      try:
22          jobgroupname = "examples"
23          jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
24          if jobgroup == None:
25              jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
26          jobgroup.addMember(__jobname__)
27      except:
28          exc_type, exc_value, exc_traceback = sys.exc_info()
29          print "Error adding %s to %s group: %s %s" % (__jobname__, jobgroupname,
exc_type, exc_value)
30
31
32  class sweeperJob(Job):
33
34       def job_started_event(self):
35            self.setFact("jobinstance.memo", self.getFact("job.description"))
36
37            sp = ScheduleSpec()
38
39            # Optionally a constraint can be specified to further limit matching
40            # resources from the job's default 'resource' constraint. Could also
41            # compose an object Constraint.
42            # For example, uncomment to restrict to resource group 'sweeper'
43           #sp.setConstraint("<contains fact='resource.groups' value='sweeper' /
>")
44
45            # Specify the joblet to run on each resource
46            sp.setJobletClass(sweeperJoblet)
47
48            # Specify the sweep across active nodes
49            sp.setUseNodeSet(sp.ACTIVE_NODE_SET)
50
51            # Schedule a sweep (creates preassigned joblets)
52            self.scheduleSweep(sp)
```

```
53
54            # Now the ScheduleSpec contains the number of joblets created
55            print 'Launched', sp.getCount(), 'joblets'
56
57
58  class sweeperJoblet(Joblet):
59
60        def joblet_started_event(self):
61            msg = "run on resource %s" % (self.getFact("resource.id"))
62            self.setFact("joblet.memo", msg)
63            print "Sweep", msg
64            sleeptime = self.getFact("jobargs.sleeptime")
65            time.sleep(sleeptime)
66
```

sweeper.policy

```
 1  <!--
 2
*=============================================================================
 3   * Copyright (c) 2010 Novell, Inc. All Rights Reserved.
 4   *
 5   * NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 6   * WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 7   * FITNESS FOR A PARTICULAR PURPOSE, AND NON INFRINGMENT.  NOVELL, THE AUTHORS
 8   * OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 9   * FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
10   * TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
11   * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
12
*=============================================================================
13   * $Id: sweeper.policy 10344 2009-11-20 21:46:43Z jastin $
*=============================================================================
15   -->
16
17  <policy>
18
19        <jobargs>
20            <!--
21              -  Defines and sets the length of time the joblet should pretend
22              -  it is doing something important
23              -->
24            <fact name="sleeptime"
25                  type="Integer"
26                  description="time to sleep (in seconds)"
27                  value="1"
28                  visible="true" />
29        </jobargs>
30
31
32        <job>
33            <!--
34              -  Give the job a description for GUI's
35              -->
36            <fact name="description"
37                  type="String"
38                  value="This example job ilustrates how to schedule a 'sweep'
accross all matching resources." />
39
40            <!--
41              -  This activates a built in throttle to limit the number of
42              -  resources this job will run on at a time
43              -->
44            <fact name="maxresources"
45                  type="Integer"
46                  value="3" />
47
48            <!--
```

```
49              -  Rank resources from least loaded to the highest loaded. The
50              -  idea is to run the joblets on the least loaded node first
51              -  and hopefully by the time we get to the higher loaded machines
52              -  their load may have gone down
53              -->
54            <!--
55            <fact name="resources.rankby">
56              <array>
57                <string>resource.loadaverage/a</string>
58              </array>
59            </fact>
60             -->
61
62            <!--
63              -  Alternative ranking that is easier to see:
64              -  decending alphabetic of node name
65              -->
66            <fact name="resources.rankby">
67              <array>
68                <string>resource.id/d</string>
69              </array>
70            </fact>
71        </job>
72
73    </policy>
74
```

Classes and Methods

The class sweeperJob (see line 32, sweeper.jdl (page 173)) is derived from the Job Class.

The class sweeperJoblet (see line 58, sweeper.jdl (page 173)) is derived from the Joblet Class.

Definitions:

Job

A representation of a running job instance.

Joblet

Defines execution on the resource.

Job Details

The `sweeper.job` can be broken down into four separate parts:

- "Configure and Run" on page 176
- "See Also" on page 177

Policy

In addition to specifying the jobarg and default settings for `sleeptime` in lines 24-28, sweeper.policy (page 174)), there also is the `<job/>` section in lines 32-71, which describes static facts..

The `resources.rankby` array has two notable setting in this example:

- **resource.loadaverage:** This is the first string assignment (lines 55-59), which is commented out, that causes joblets to run on the least loaded nodes first.This is the default value and the default launch order for scheduleSweep.

- **resource.id:** This is the second string assignment (lines 66-70), which is actually used, and assigns the string to the rank by array so that joblets run on nodes in reverse alphabetical order.

zosadmin deploy

When the Orchestration Server deploys a job for the first time (see Section 3.5, "Deploying Jobs," on page 37), the job JDL files are executed in a special deploy mode. When `sweeper.jdl` is run in this way (either through the Development Client or the zosadmin deploy command), lines 20-29 are executed. This attempts to locate the examples jobgroup (lines 22-23), creates the group if it is not found (lines 24-25), and adds the sweeper job to the group (line 26).

If the deployment fails for any reason, then an exception is thrown (line 27), which prints the job name, group name, exception type and value (line 29).

job_started_event

The sweeperJob class (line 32) defines only the required `job_started_event` (line 34) method. This method runs on the Orchestration Server when the job is run to launch the joblets.

When executed, `job_started_event` displays a message on the memo line of the Job Log tab within the Jobs view in the Orchestration Console (line 35), via `jobinstance.memo` (see Section 3.10.1, "Creating a Job Memo," on page 43).

Jumping ahead for a moment, instead of calling `self.schedule()` as most the other examples do to instantiate joblets, sweeperJob calls `self.scheduleSweep()` (line 52). scheduleSweep requires an instance of ScheduleSpec (page 275), so one is created (line 37).

The ScheduleSpec method setConstraint can be used to constrain the available resources to a particular group, as shown with a comment (line 43). If this setConstraint line is uncommented, joblets will only run on members of the sweeper `resource.group` instead of using the default resource group `all`.

NOTE: The sweeper group must already be created and have computing nodes assigned to it (see "Creating a Resource Account" in the *NetIQ Cloud Manager 2.1.1 Orchestration Installation Guide*). This constraint would also be ANDed to any existing constraint, including any aggregated policies.

The sweeperJoblet is set to be scheduled (line 52), and setUseNodeSet(intnodeSet) is assigned (line 49) the value sp.ACTIVE_NODE_SET. So, the joblet set is constructed after applying resource constraints to the active/online resources. This in contrast to the other possible value of sp.PROVISIONABLE_NODE_SET, where constraints are applied to all provisionable resources.

joblet_started_event

The sweeperJoblet class (lines 58-65) defines only the required `joblet_started_event` (line 60) method. After this method is executed, it displays a message on the memo line of the Joblet tab within the Jobs view in the Orchestration Console (lines 61-62). It also prints a similar log message (line 63), and then just sleeps for `jobargs.sleeptime` seconds (lines 64-65) before completion.

Configure and Run

Execute the following commands to deploy and run `sweeper.job`:

1 Deploy notepad.job into the grid:

```
> zosadmin deploy sweeper.job
```

2 Display the list of deployed jobs:

```
> zos joblist
```

sweeper should appear in this list.

3 Run the job on one or more resources using the default values for numJoblets and resource, specified in the `sweeper.policy` file:

```
> zos run sweeper sleeptime=30
JobID: zenuser.sweeper.420

> zos status zenuser.sweeper.420
Completed

> zos log zenuser.sweeper.420
Launched 3 joblets
[melt] Sweep run on resource melt
[freeze] Sweep run on resource freeze
[skate] Sweep run on resource skate
```

See Also

- Setting Constraints Using Policies, see Section 2.3, “Policies,” on page 18 and Chapter 4, “Understanding Grid Object Facts, Computed Facts, and Custom Facts,” on page 51.

whoami.job

Demonstrates using the Exec class to send a command to the operating system's default command interpreter. On Microsoft Windows, this is `cmd.exe`. On POSIX systems, this is `/bin/sh`.

Usage

```
> zos login --user zenuser
Please enter current password for 'zenuser':
 Logged into grid as zenuser

~> zos jobinfo --detail whoami
Jobname/Parameters    Attributes
------------------    ----------
whoami              Desc: This is a demo example of enhanced exec

    numJoblets        Desc: The number of joblets to schedule
                      Type: Integer
                   Default: 1

    resource          Desc: The resource id to run on
                      Type: String
                   Default: .*
```

Description

The files that make up the Whoami job include:

```
whoami                                                  # Total: 118 lines
|-- whoami.jdl                                          #   69 lines
`-- whoami.policy                                       #   49 lines
```

whoami.jdl

```
 1  # ------------------------------------------------------------------------
-
 2  #  Copyright (c) 2010 Novell, Inc. All Rights Reserved.
 3  #
 4  #  NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 5  #  WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 6  #  FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGMENT.  NOVELL, THE AUTHORS
 7  #  OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 8  #  FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 9  #  TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
10  #  OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
11  # ------------------------------------------------------------------------
-
12  #  $Id: whoami.jdl 10344 2009-11-20 21:46:43Z jastin $
13  # ------------------------------------------------------------------------
-
14
15  """
16
17  Demonstrate running setuid exec.
18
19  """
20  import os,time
21
22  #
23  # Add to the 'examples' group on deployment
24  #
25  if __mode__ == "deploy":
26      try:
```

```
27          jobgroupname = "examples"
28          jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
29          if jobgroup == None:
30              jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
31          jobgroup.addMember(__jobname__)
32      except:
33          exc_type, exc_value, exc_traceback = sys.exc_info()
34        print "Error adding %s to %s group: %s %s" % (__jobname__, jobgroupname,
exc_type, exc_value)
35
36
37  class Whoami(Job):
38
39       def job_started_event(self):
40            # Launch the joblets
41            numJoblets = self.getFact("jobargs.numJoblets")
42            user = self.getFact("user.id")
43            print "Launching %d joblets for user '%s'" % (numJoblets,user)
44            self.schedule(WhoamiJoblet,numJoblets)
45
46
47  class WhoamiJoblet(Joblet):
48
49       def joblet_started_event(self):
50            if self.getFact("resource.os.family") == "windows":
51                 cmd = "echo %USERNAME%"
52            elif self.getFact("resource.os.family") == "solaris":
53                 cmd = "echo $USER"
54            else:
55                 cmd = "whoami"
56            print "cmd=%s" % (cmd)
57
58            # example using built-in system()
59            #result = system(cmd)
60
61            # example using Exec class
62            e = Exec()
63            e.setShellCommand(cmd)
64            e.writeStdoutToLog()
65            e.writeStderrToLog()
66            result = e.execute()
67
68            print "result=%d" % (result)
69
```

whoami.policy

```
 1  <!--
 2
*================================================================================
 3   * Copyright (c) 2010 Novell, Inc. All Rights Reserved.
 4   *
 5   * NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 6   * WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY ,
 7   * FITNESS FOR A PARTICULAR PURPOSE, AND NON INFRINGMENT.  NOVELL, THE AUTHORS
 8   * OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 9   * FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
10   * TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
11   * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
12
*================================================================================
13   * $Id: whoami.policy 10344 2009-11-20 21:46:43Z jastin $
14
*================================================================================
15   -->
16
17  <policy>
18
```

```
19       <jobargs>
20
21            <fact name="numJoblets"
22                  type="Integer"
23                  description="The number of joblets to schedule"
24                  value="1" />
25
26            <fact name="resource"
27                  type="String"
28                  description="The resource id to run on"
29                  value=".*" />
30
31       </jobargs>
32
33       <job>
34            <fact name="description"
35                  type="String"
36                  value="This is a demo example of enhanced exec" />
37
38          <!-- only allow one run resource at a time so that multiple re sources
can be visited -->
39            <fact name="joblet.maxperresource"
40                  type="Integer"
41                  value="1" />
42       </job>
43
44       <constraint type="resource" >
45          <eq fact="resource.id" factvalue="jobargs.resource" match="regex p" />
46       </constraint>
47
48  </policy>
49
```

Classes and Methods

Definitions:

Job

A representation of a running job instance.

Joblet

Defines execution on the resource.

MatrixInfo

A representation of the matrix grid object, which provides operations for retrieving and creating grid objects in the system. MatrixInfo is retrieved using the built-in `getMatrix()` function. Write capability is dependent on the context in which `getMatrix()` is called. For example, in a joblet process on a resource, creating new grid objects is not supported.

GroupInfo

A representation of Group grid objects. Operations include retrieving the group member lists and adding/removing from the group member lists, and retrieving and setting facts on the group.

Exec

Used to manage command line execution on resources.

Job Details

The following sections describe the Whoami job:

- "Configure and Run" on page 182
- "See Also" on page 182

zosadmin deploy

When jobs are deployed into the grid, they can optionally be placed in groups for organization and easy reference. In this case, the Whoami job is added to the group named "examples" (see lines 25-34 of `whoami.jdl`) and is displayed in the Orchestration Console in the Explorer view at the location:

`/Orchestration Servers/Grid_Name/Jobs/examples`

For a general overview of how jobs are added to groups during deployment, see "Deploying a Sample Job" in the *NetIQ Cloud Manager 2.1.1 Orchestration Administrator Reference*.

job_started_event

When the Whoami job receives a job_started_event, it schedules one or more instances of the Whoami joblet to be run (see line 44 in whoami.jdl (page 178)). The number of WhoamiJoblet instances is indicated by the value of the `numJoblets` fact, whose value might have been supplied on the command-line, or referenced from what's been supplied in the whoami.policy file by default (see lines 21-24 in whoami.policy (page 179)).

In addition to supplying a default value for `numJoblets`, the `whoami.policy` file also supplies a default value for the ID of the resource on which the joblet runs. The default value is .*, which means all resources are included (see lines 26-29 in whoami.policy (page 179)).

Note that the setting for resource is not used in the JDL code but is used to affect which resources on which the joblet run. This occurs because a constraint is specified in `whoami.policy` that restricts the resources that can run this joblet to the current value of the `resource` fact (see line 45 in whoami.jdl (page 178)).

`maxperresource` is another job setting that affects scheduling of the Whoami joblet. The system uses `maxperresource` to determine how many instances of the joblet can run simultaneously on the same resource. In this case, only one instance of the Whoami job can be run on a machine at a time, as specified in lines 39-42 in whoami.policy (page 179).

When facts are referenced in the JDL file, they are prepended with `jobargs.` or `job.` However, when supplied on the command line, this prefix is omitted. JDL files must use an explicit naming convention when it references facts from the different sections of the policy files. For more information on naming conventions for policy facts, see Section 5.1.1, "Cloud Manager Orchestration Datagrid Filepaths," on page 109.

joblet_started_event

When the Whoami joblet is executed on a particular resource it receives a joblet_started_event. After this happens, the Whoami joblet decides which command to use to get the current username by checking the value of `resource.os.family` (see lines 50-55 in whoami.jdl (page 178)). This setting is

not set in the `whoami.policy`, but instead is available from the Cloud Manager Orchestration system.

After the command to get the current username has been decided, the Orchestration API class named Exec is used to execute the command on the resource where the joblet is running (see lines 62-66 in whoami.jdl (page 178)).

By passing the command to the Exec setShellCommand method, the command will be executed by the operating system's default command interpreter. On Microsoft Windows this `cmd.exe`. On POSIX systems, this is `/bin/sh`. As indicated by lines 64-65 in whoami.jdl (page 178), all standard out and standard errors are written to the job's log file.

To view the log file for the whoami job after it has been run, execute the command `> zos log whoami`.

For more information about using the zos command line, see "The zosadmin Command Line Tool" in the *NetIQ Cloud Manager 2.1.1 Orchestration Server Command Line Reference*. For more information on running commands using the Exec class, see Exec (page 245).

Configure and Run

Execute the following commands to deploy and run `whoami.job`:

1. Deploy notepad.job into the grid:

   ```
   > zosadmin deploy whoami.job
   ```

2. Display the list of deployed jobs:

   ```
   > zos joblist
   ```

   whoami should appear in this list.

3. Run the job on one or more resources using the default values for numJoblets and resource, specified in the `whoami.policy` file:

   ```
   > zos run whoami
   ```

4. Run the job on one or more resources using supplied values for numJoblets and resource:

   ```
   > zos run whoami numJoblets=10 resource=my_resource_.*
   ```

   Run 10 joblets simultaneously, but only on resources beginning with the name "my_resource_".

   NOTE: The value for "resource" is specified using regular expression syntax.

See Also

- Setting Constraints Using Policies (Section 2.3, "Policies," on page 18 and Chapter 4, "Understanding Grid Object Facts, Computed Facts, and Custom Facts," on page 51).
- Scheduling multiple instances of a joblet
- Setting default parameter values using policies
- Configuring constraints in a policy file
- Naming conventions for policy facts (Section 5.1.1, "Cloud Manager Orchestration Datagrid Filepaths," on page 109.Section 5.1.1, "Cloud Manager Orchestration Datagrid Filepaths," on page 109)
- Facts provided by the Orchestration system that can be referenced within a JDL file

- Using Cloud Manager Orchestration ("How Do I Interact with the Orchestration Server?")
- Running commands using the Exec class.

7.6 Job Examples: Miscellaneous Code-Only

The following examples demonstrate useful, miscellaneous code-only job concepts:

jobargs.job

Demonstrates the usage of the various argument types that jobs can accept. These types are integer, Real, Boolean, String, Time, Date, List, Dictionary, and Array (which can contain the types Integer, Real, Boolean, Time, Date, String). For more information about how to define job arguments, and specify their values on the command line, see Chapter 4, “Understanding Grid Object Facts, Computed Facts, and Custom Facts,” on page 51.

Usage

```
> zosadmin login --user zosadmin Login to server: skate
Please enter current password for 'zosadmin':
Logged into grid on server 'skate'

> cd /opt/novell/zenworks/zos/server/examples
> zosadmin deploy jobargs.job
jobargs successfully deployed

> zos login --user zenuser Please enter current password for 'zenuser':
 Logged into grid as zenuser

> zos jobinfo --detail jobargs
Jobname/Parameters    Attributes
------------------    ----------
jobargs                          Desc: This example job tests all fact types.

    OptionalRealArray            Desc: No description available.
                                 Type: Real[]
                              Default: [1.1,2.2]

    RequiredRealArg              Desc: No description available.
                                 Type: Real
                              Default: None! Value must be specified

    RequiredDateArg              Desc: No description available.
                                 Type: Date
                              Default: None! Value must be specified

    OptionalListArg              Desc: No description available.
                                 Type: List
                              Default: [hi, mom, 42]

    OptionalIntegerArg           Desc: Optional Integer Arg
                                 Type: Integer
                              Default: 123

    OptionalStringArg            Desc: Optional String Arg
                                 Type: String
                              Default: foo

    OptionalDateArray            Desc: No description available.
                                 Type: Date[]
                              Default: [Mon Jan 02 12:01:00 MST 2006,Tue Jan 03
                                       12:02:00 MST 2006,Wed Jan 04 00:00:00
                                       MST 2006]

    OptionalStringArray          Desc: No description available.
                                 Type: String[]
                              Default: [string1,string2]

    RequiredBooleanArg           Desc: No description available.
                                 Type: Boolean
                              Default: None! Value must be specified

    OptionalString2ArgAsTag      Desc: Optional String Arg as tag
```

```
                              Type: String
                           Default: bar

    RequiredTimeArg           Desc: No description available.
                              Type: Time
                           Default: None! Value must be specified

    OptionalBooleanArg        Desc: Optional Boolean Arg
                              Type: Boolean
                           Default: true

    OptionalTimeArg           Desc: Optional Time Arg
                              Type: Time
                           Default: 43260000

    RequiredStringArg         Desc: No description available.
                              Type: String
                           Default: None! Value must be specified

    OptionalRealArg           Desc: Optional Real Arg
                              Type: Real
                           Default: 3.14

    OptionalDateArg           Desc: Optional Date Arg
                              Type: Date
                           Default: Mon Jan 02 12:01:00 MST 2006

    RequiredIntegerArg        Desc: No description available.
                              Type: Integer
                           Default: None! Value must be specified

    OptionalDictArg           Desc: No description available.
                              Type: Dictionary
                           Default: {time=12600000, date=Sat Apr 15 00:00:00
                                    MDT 2006, age=12, name=joe}

    OptionalString3ArgAsCDATA Desc: Optional String Arg as CDATA
                              Type: String
                           Default: this text is part of | a multi-line
                                    cdata section containing | xml
                                    <html>test</html> | <eq fact="foo.bar"
                                    value="qwerty" /> | cool!

    OptionalTimeArray         Desc: No description available.
                              Type: Time[]
                           Default: [43260000,43320000]

    OptionalIntegerArray      Desc: No description available.
                              Type: Integer[]
                           Default: [1,2]
```

Description

The files that make up the Jobargs job include:

```
jobargs.job                                  # Total: 254 lines
|-- jobargs.jdl                              #   77 lines
`-- jobargs.policy                           #  177 lines
```

jobargs.jdl

```
 1  # ------------------------------------------------------------------------
-
 2  #  Copyright © 2010 Novell, Inc. All Rights Reserved.
 3  #
 4  #  NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
 5  #  WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY ,
 6  #  FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGMENT.  NOVELL, THE AUTHORS
 7  #  OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
 8  #  FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 9  #  TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
10  #  OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
11  # ------------------------------------------------------------------------
-
12  #  $Id: jobargs.jdl 10344 2009-11-20 21:46:43Z jastin $
13  # ------------------------------------------------------------------------
-
14
15  """
16  Example job showing all available job argument types.
17
18  Example cmd line to run job:
19
20    zos run jobargs RequiredTimeArg="12:01 AM" RequiredRealArg="3.14"
RequiredIntegerArg="123" RequiredStringArg="foo" RequiredBooleanArg="true"
RequiredDateArg="6/10/08 11:35 AM"
21
22  """
23
24  import time
25
26  #
27  # Add to the 'examples' group on deployment
28  #
29  if __mode__ == "deploy":
30      try:
31          jobgroupname = "examples"
32          jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
33          if jobgroup == None:
34              jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
35          jobgroup.addMember(__jobname__)
36      except:
37          exc_type, exc_value, exc_traceback = sys.exc_info()
38          print "Error adding %s to %s group: %s %s" % (__jobname__, jobgroupname,
exc_type, exc_value)
39
40
41  class jobargs(Job):
42
43       def job_started_event(self):
44
45            jobid = self.getFact("jobinstance.id")
46            print "*****Dumping %s JobInstance jobargs facts*****" % (jobid)
47            keys = self.getFactNames()
48            keys.sort()
49            for s in keys:
50               if s.startswith("jobargs"):
51                   v = self.getFact(s)
52                   ty = type(v)
53
54                   if str(ty).endswith("Dictionary"):
55                       self.dump_dict(s,v)
56                   else:
57                        if s.endswith("TimeArg") or s.endswith("TimeArgReq"):
58                            hrs = v/3600
59                            min = (v % 3600)/60
60                            sec = (v % 3600) % 60
61                            print "%s %s %s  hrs:%d min:%d sec:%d" %
```

```
(s,type(v),v,hrs,min,sec)
62
63                          elif s.endswith("DateArg") or s.endswith("DateArgReq"):
64                              sv = time.ctime(v)
65                              print "%s %s %s" % (s,type(v),sv)
66
67                          else:
68                              print "%s %s %s" % (s,type(v),str(v))
69
70              print "*****End %s dump*****" % (jobid)
71
72              #self.schedule(jobargsJoblet)
73
74          def dump_dict(self,name,dict):
75              print "Dict: %s" % (name)
76              keys = dict.keys()
77              for k in keys:
78                  v = dict[k]
79                  ty = type(v)
80                  if k == "dob":
81                      v = time.ctime(v/1000)
82                  print "   %s %s %s" % (k,ty,str(v))
83
84
85  class jobargsJoblet(Joblet):
86
87          def joblet_started_event(self):
88              pass
89
```

jobargs.policy

```
  1  <!--
  2
*=============================================================================
  3   * Copyright © 2010 Novell, Inc. All Rights Reserved.
  4   *
  5   * NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
  6   * WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
  7   * FITNESS FOR A PARTICULAR PURPOSE, AND NON INFRINGMENT.  NOVELL, THE AUTHORS
  8   * OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
  9   * FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF
CONTRACT,
 10   * TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
 11   * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 12
*=============================================================================
 13   * $Id: jobargs.policy 10344 2009-11-20 21:46:43Z jastin $
 14
*=============================================================================
 15   -->
 16
 17  <policy>
 18
 19    <jobargs>
 20
 21      <!-- Optional job args -->
 22      <fact name="OptionalDateArg"
 23            description="Optional Date Arg"
 24            type="Date"
 25            value="1/2/06 12:01 PM"/>
 26
 27      <fact name="OptionalTimeArg"
 28            description="Optional Time Arg"
 29            type="Time"
 30            value="12:01 PM"/>
 31
 32      <fact name="OptionalRealArg"
```

```
            description="Optional Real Arg"
            type="Real"
            value="3.14" />

      <fact name="OptionalIntegerArg"
            description="Optional Integer Arg"
            type="Integer"
            value="123" />

      <fact name="OptionalStringArg"
            description="Optional String Arg"
            type="String"
            value="foo" />

      <fact name="OptionalString2ArgAsTag"
            description="Optional String Arg as tag">
        <string>bar</string>
      </fact>

      <fact name="OptionalString3ArgAsCDATA"
            description="Optional String Arg as CDATA">
        <string>
          <![CDATA[this text is part of
a multi-line cdata section containing
xml <html>test</html>
<eq fact="foo.bar" value="qwerty" />
cool!
]]>
        </string>
      </fact>

      <fact name="OptionalBooleanArg"
            description="Optional Boolean Arg"
            type="Boolean"
            value="true" />

      <fact name="OptionalListArg">
        <list>
          <listelement value="hi" type="String" />
          <listelement value="mom" />
          <listelement value="42" type="Integer" />
        </list>
      </fact>

      <fact name="OptionalDictArg">
        <dictionary>
          <dictelement key="name" type="String"  value="joe" />
          <dictelement key="date" type="Date"    value="4/15/06" />
          <dictelement key="time" type="Time"    value="3:30 AM" />
          <dictelement key="age"  type="Integer" value="12" />
        </dictionary>
      </fact>

      <fact name="OptionalDateArray">
        <array>
          <date>1/2/06 12:01 PM</date>
          <date>1/3/06 12:02 PM</date>
          <date>1/4/06</date>
        </array>
      </fact>
      <fact name="OptionalTimeArray">
        <array>
          <time>12:01 PM</time>
          <time>12:02 PM</time>
        </array>
      </fact>
      <fact name="OptionalRealArray">
        <array>
          <real>1.1</real>
          <real>2.2</real>
```

```
103       </array>
104     </fact>
105     <fact name="OptionalIntegerArray">
106       <array>
107         <integer>1</integer>
108         <integer>2</integer>
109       </array>
110     </fact>
111     <fact name="OptionalStringArray">
112       <array>
113         <string>string1</string>
114         <string>string2</string>
115       </array>
116     </fact>
117  <!-- Arrays of dictionary or list not currently supported
118     <fact name="OptionalDictionaryArray">
119       <array>
120         <dictionary>
121           <dictelement key="name" type="String"  value="joe" />
122         </dictionary>
123       </array>
124     </fact>
125  -->
126
127     <!-- Required job args -->
128     <fact name="RequiredDateArg" type="Date" />
129     <fact name="RequiredTimeArg" type="Time" />
130     <fact name="RequiredRealArg" type="Real" />
131     <fact name="RequiredIntegerArg" type="Integer" />
132     <fact name="RequiredStringArg" type="String" />
133     <fact name="RequiredBooleanArg" type="Boolean" />
134    <!-- XXX Ooops, not currently supported without value!
135     <fact name="RequiredListArg" type="list" />
136     <fact name="RequiredDictArg" type="dictionary" />
137     <fact name="RequiredStringArray" type="string">
138       <array />
139     </fact>
140    -->
141
142     <!-- Invisible job args -->
143     <fact name="InvisibleDateArg" type="Date" value="1/2/06 12:01 PM"
visible="False" />
144     <fact name="InvisibleTimeArg" type="Time" value="12:01 PM" visible="False"
/>
145     <fact name="InvisibleRealArg" type="Real" value="3.14" visible="False" />
146     <fact name="InvisibleIntegerArg" type="Integer" value="123"
visible="False" />
147     <fact name="InvisibleStringArg" type="String" value="foo" visible="False"
/>
148     <fact name="InvisibleString2Arg" visible="False" >
149       <string>bar</string>
150     </fact>
151     <fact name="InvisibleBooleanArg" type="Boolean" value="true"
visible="False" />
152     <fact name="InvisibleListArg" visible="False">
153       <list>
154         <listelement value="hi" type="String" />
155         <listelement value="mom" />
156         <listelement value="42" type="integer" />
157       </list>
158     </fact>
159     <fact name="InvisibleDictArg" visible="False">
160       <dictionary>
161         <dictelement key="name" type="String"  value="joe" />
162         <dictelement key="date" type="Date"    value="4/15/06" />
163         <dictelement key="time" type="Time"    value="3:30 AM" />
```

```
164           <dictelement key="age"  type="Integer" value="12" />
165         </dictionary>
166       </fact>
167
168     </jobargs>
169
170     <job>
171       <fact name="description"
172             type="String"
173             value="This example job tests all fact types." />
174     </job>
175
176   </policy>
177
```

Schedule File (optional)

jobargs.sched

```
1 <schedule name="jobargs" description="Run jobargs" active="true">
2 <runjob job="jobargs" user="labuser" priority="medium" />
3 <triggers>
4 <trigger name="trigger1" />
5 <trigger name="trigger2" />
6 </triggers>
7 </schedule>
```

Classes and Methods

Definitions:

Job

A representation of a running job instance.

Joblet

Defines execution on the resource.

MatrixInfo

A representation of the matrix grid object, which provides operations for retrieving and creating grid objects in the system. MatrixInfo is retrieved using the built-in `getMatrix()` function. Write capability is dependent on the context in which `getMatrix()` is called. For example, in a joblet process on a resource, creating new grid objects is not supported.

GroupInfo

A representation of Group grid objects. Operations include retrieving the group member lists and adding/removing from the group member lists, and retrieving and setting facts on the group.

Job Details

The Jobargs job performs its work by handling the following events:

- "Configure and Run" on page 191
- "See Also" on page 192

zosadmin deploy

In jobargs.jdl (page 186), lines 27-36 deploy the job into the grid. After jobs are deployed into the grid, they can optionally be placed in groups for organization and easy reference. In this case, the jobargs job will be added to the group named "examples", and will show up in the Orchestration Console in the Explorer view at the location:

`/Orchestration Servers/Grid_Name/Jobs/examples`

For a general overview of how jobs are added to groups during deployment, see "Deploying a Sample Job" in the *NetIQ Cloud Manager 2.1.1 Orchestration Administrator Reference*.

job_started_event

After the Jobargs job receives a job_started_event, it gets a list of all the facts available to it, as shown in line 45 of jobargs.jdl (page 186). This list is sorted, filtered according to whether or not it's a jobarg fact, and then enumerated (lines 46-58). Each jobarg fact is printed in a "name type value" format. When the complex Dictionary type is encountered (line 52), a separate method is used to print the values for all the key-value pairs (lines 63-71).

The list of optional and required arguments for this Jobargs example are available as facts within the <jobargs> section (see lines 19-168 in jobargs.policy (page 187)).

For more information about defining job arguments and their types, see Chapter 4, "Understanding Grid Object Facts, Computed Facts, and Custom Facts," on page 51 and Section 2.3, "Policies," on page 18.

joblet_started_event

The Jobargs job only illustrates passing job arguments to a job. Therefore, no work is performed on the resource by the jobargsJoblet.

Configure and Run

To run this example, you must have the Cloud Manager Orchestration Server and the Orchestration Agenet installed and configured properly. No agents on separate resources are required. You also must be logged into your Orchestration Server before you run `zosadmin` or `zos` commands.

Execute the following commands to deploy and run `jobargs.job`:

1. Deploy jobargs.job into the grid:

   ```
   > zosadmin deploy jobarg.job
   ```

   NOTE: Run zosadmin login to log in for zos administration.

2. Display the list of deployed jobs:

   ```
   > zos joblist
   ```

   jobargs should appear in this list.

   NOTE: Run `zos login` to run zos client jobs.

3. Display the list of optional and required arguments for this job:

   ```
   > zos jobinfo jobargs
   ```

4 Run the jobargs job and view the results.

NOTE: You must supply values for TimeArgReq, RealArgReq, StringArgReq, BooleanArgReq, IntegerArgReq, and DateArgReq as follows (see jobargs.policy (page 187) for the full list of arguments that can be specified):

```
> zos run jobargs TimeArgReq=12:01:02 RealArgReq=3.14 StringArgReq=Hello
BooleanArgReq=True IntegerArgReq=42 DateArgReq="04/05/07 7:45 AM"

> zos log jobargs
```

See Also

- Defining job arguments and their types
- Using the Cloud Manager Orchestration Server and the Orchestration Agent ("How Do I Interact with the Orchestration Server?")

8 Job Scheduling

The Cloud Manager Orchestration Server schedules jobs either start manually using the Job Scheduler or to start programatically using the Job Description Language (JDL). This section contains the following topics:

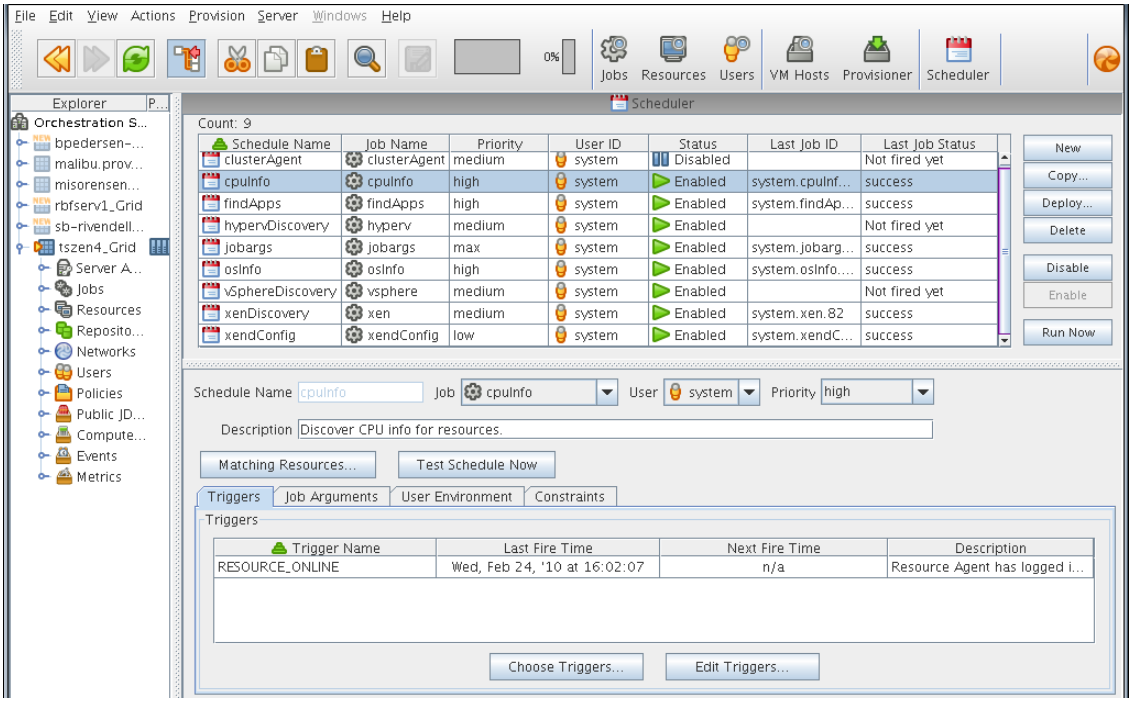
Job scheduling in JDL occurs in the sense that jobs are allocated in priority order (sometimes referred to as “scheduling”) by the Orchestration Resource Allocation Broker. For more information, see “Scheduling with Constraints” on page 26.

8.1 The Cloud Manager Orchestration Job Scheduler Interface

After the Cloud Manager Orchestration Server is enabled with a license, users have access to a built-in job Scheduler. This GUI interface allows jobs to be started periodically based upon user scheduling or when various system or user-defined events occur.

The following figure illustrates the Job Scheduler, with nine jobs staged in the main Scheduler panel.

Figure 8-1 *The Orchestration Job Scheduler GUI*



Jobs are individually submitted and managed using the Job Scheduler as discussed in "The Orchestration Server Job Scheduler" in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*.

8.2 Schedule and Trigger Files

In addition to using the Job Scheduler GUI, developers can also write XML files to schedule and trigger jobs to run when triggered by specific events. These files are designated using the `.sched` and `.trig` extensions. and can be included as part of the job archive file (`.job`) or deployed separately.

Everything that you do manually in the Job Scheduler can be automated by creating a `.sched` or `.trig` XML script as part of a job. The XML files enable you to package system and job scheduling logic without using the GUI. This includes setting up cron triggers (for example, running a job at specified intervals) and other triggers that respond to built-in system events, such as resource startup, user startup (that is, login), or user-defined events that trigger on a rule.

For example, the `osInfo` discovery job, which probes a resource for its operating system information, is packaged with a schedule file, as shown in the "Schedule File Examples" on page 194. See also Section 8.2.2, "Trigger File XML Examples," on page 195.

This section includes the following information:

8.2.1 Schedule File Examples

A schedule file (`.sched`) can be packaged either within a `.job` archive alongside the `.jdl` file or independently deployed using the zosadmin command line utility. Because the XML file defines the job schedule programmatically outside of the Orchestration Console, packaging these scripts into jobs is typically a developer task.

This section includes the following information:

Schedule File Example: osInfo.sched

The `osinfo.sched` file is packaged with the `osInfo` discovery job, which is deployed as part of the base server. Its purpose is to trigger a run of the `osInfo` job on a resource when the resource comes on line as it logs into the server.

The following shows the syntax of the schedule file that wraps the job:

```
1  <schedule name="osInfo" description="Discover OS info on resources."
active="true" reourcesRequired="false">
2    <runjob job="osInfo" user="system" priority="high" />
3    <triggers>
4      <trigger name="RESOURCE_ONLINE" />
5    </triggers>
6  </schedule>
```

Line 1: Defines a new schedule named `osinfo`, which is used to schedule a run of the job `osInfo`. If the job (in this case, `osinfo`) is not deployed, the deployment returns a "Job is not deployed" error.

Line 2: Instructs the schedule to run the named job (`osinfo`) by the named user (`system`) using a defined priority (high). If the user (in this case, `system`) does not exist, the deployment returns a `User does not exist` error.

NOTE: Only the Orchestration Server users belonging to the `administrators` group can assign priorities higher than `medium`. Assigning a higher priority than specified by the `user.priority.max` fact defaults to a priority equal to `user.priority.max` when the job runs.

Line 3-5: Defines the triggers (in this case only one trigger, an event, `RESOURCE_ONLINE`) that initiates the job.

Schedule File Example: Multiple Triggers

A schedule can include one or more triggers. The following example shows the syntax of a schedule file that has two cron triggers for scheduling a job:

```
1  <schedule name="ReportTwice" active="true">
2    <runjob job="jobargs" user="JohnD" priority="medium" />
3    <triggers>
4      <trigger name="DailyReportTrigger" />
5      <trigger name="NightlyReportTrigger" />
6    </triggers>
7  </schedule>
```

Line 1: Defines the schedule name, deployed condition, and description (if any).

Line 2: Instructs the schedule to run the named job (`jobargs`) by the named user (`JohnD`) using a defined priority (`medium`).

Lines 3-6: Defines the triggers (in this case two occurring time triggers) that initiate the job.

8.2.2 Trigger File XML Examples

Trigger files define when a job and how often a schedule fires. This can happen when a defined event occurs, when a defined amount of time passes, or when a given point in time is reached. so that one or more triggers can be associated with a job schedule (`.sched`). You can create these triggers yourself in XML format and deploy them, or you can edit and choose them in the Job Scheduler, which automatically deploys them. This section includes examples to show you the syntax of different trigger files.

- "XML Example: Event Trigger" on page 195
- "XML Example: Interval Time Trigger" on page 196
- "XML Example: Cron Expression Trigger" on page 196

XML Example: Event Trigger

An event trigger starts a job when a defined event occurs. Several built-in event triggers (such as events that occur when a managed object comes online or offline or has a health status change) are available in the Trigger chooser of the Job Scheduler along with any user-defined Events:

- `RESOURCE_ONLINE`
- `RESOURCE_OFFLINE`
- `USER_ONLINE`

- USER_OFFLINE
- RESOURCE_HEALTH
- USER_HEALTH
- VMHOST_HEALTH
- REPOSITORY_HEALTH

When deployed as triggers in a schedule, built-in events do not generate a `.trig` file, as other triggers do.

You can also associate an Event object with a job schedule (see "Event Triggers" in "The Orchestration Server Job Scheduler" in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*). You can define an Event object in an XML document, deploy it to a server, and then manage it with the Orchestration Console.

The following example, `PowerOutage.trig` shows the XML format for a trigger that references an event object.

```
1  <trigger name="PowerOutage" description="Fires when UPS starts at power outage">
2    <event value="UPS_interrupt"/>
3  </trigger>
```

Line 1: Defines the trigger name and description.

Line 2: Defines the event object chosen for the trigger.

For more information about events, see Section 3.12, "Using an Event Notification in a Job," on page 45.

XML Example: Interval Time Trigger

The following example, `EveryMin1Hr.trig` shows the XML format for a trigger that uses the system clock to define (in seconds) how soon the schedule is to start, how often the schedule is to repeat, and how many times the schedule is to be repeated:

```
1  <trigger name="EveryMin1Hr" description="Fires every minute for one hour">
2    <interval startin="600" interval="60" repeat="60"/>
3  </trigger>
```

Line 1: Defines the trigger name and description.

Lines 2-3: Defines how soon the schedule is to start, how often the schedule is to repeat, and how many times the schedule is to be repeated

XML Example: Cron Expression Trigger

The following example, `NoonDaily.trig` shows the XML format for a trigger that uses a Quartz Cron expression to precisely define when an event is to fire.

```
1  <trigger name="NoonDaily" description="Fires every day at noon">
2    <cron value="0 0 12 * * ?"/>
3  </trigger>
```

Line 1: Defines the trigger name and description.

Lines 2-3: Defines the cron expression to be used by the schedule. Cron expressions are used to precisely define the future point in time when the schedule is to fires. For more information, see "Understanding Cron Syntax in the Job Scheduler" in "The Orchestration Server Job Scheduler" in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*.

9 Provisioning Adapter Hooks

The Cloud Manager Orchestration Server includes a hooks facility that lets you make grid events more extensible in the provisioning adapters by adding pre-event or post-event jobs to prepare for and then to clean up after the event.

This section includes the following information:

9.1 Grid Events for VMs That Are Implemented by Provisioning Adapters

The grid events for VMs that are implemented by the respective Orchestration Server provisioning adapters are shown in the following table. The actual event names use a "pre_" or "post_" suffix, depending on whether it occurs as a pre-event or post-event.

Table 9-1 *Grid Events Implemented by Provision Adapters*

Event Name	Parameters Passed with the Event	Provisioning Adapters That Support the Event
build_event	◆ vm_host ◆ vm	◆ xen
clone_event	◆ vm_host? ◆ vm ◆ new_vm	◆ xen ◆ vsphere
move_event	◆ vm_host? ◆ vm	◆ xen ◆ vsphere ◆ hyperv
start_event	◆ vm_host? ◆ vm	◆ xen ◆ vsphere ◆ hyperv
shutdown_event	◆ vm	◆ xen ◆ vsphere ◆ hyperv

Event Name	Parameters Passed with the Event	Provisioning Adapters That Support the Event
restart_event	◆ vm	◆ xen ◆ vsphere ◆ hyperv
destroy_event	◆ vm	◆ xen ◆ vsphere ◆ hyperv
suspend_event	◆ vm	◆ xen ◆ vsphere ◆ hyperv
pause_event	◆ vm	◆ xen ◆ hyperv
resume_event	◆ vm	◆ xen ◆ hyperv
personalize_event	◆ vm	◆ xen ◆ vsphere
saveConfig_event	◆ vm	◆ xen ◆ vsphere ◆ hyperv
applyConfig_event	◆ vm	◆ xen
createTemplate_event	◆ vm_host? ◆ vm ◆ new_vm	◆ xen ◆ vsphere
migrate_event	◆ vm ◆ new_vm_host	◆ xen ◆ vsphere ◆ hyperv
checkpoint_event	◆ vm	◆ vsphere ◆ hyperv
restore_event	◆ vm	◆ vsphere ◆ hyperv
installAgent_event	◆ vm	◆ xen
checkStatus_event	◆ vm	◆ xen ◆ vsphere ◆ hyperv

Event Name	Parameters Passed with the Event	Provisioning Adapters That Support the Event
discoverVmHost_event	◆ vm_hosts	◆ xen ◆ vsphere ◆ hyperv
discoverRepository_event	◆ repository	◆ xen ◆ vsphere ◆ hyperv
makeStandalone_event	◆ vm	◆ xen ◆ vsphere

NOTE: A `vm_host` parameter with the (?) annotation signifies that it is passed by the event only if the engine actually has the parameter when the event occurs. If the engine has only a VM but not a VM host, the hooks framework performs an internal lookup for the VM host.

9.2 Customizing Jobs for the Provisioning Adapter Hooks

This section includes the following information:

9.2.1 Adding Hooks Jobs to Customize a VM Event

If you want to augment what happens when a VM event occurs, you can do so by setting the `job.paHooksVmJob` fact in the Orchestration Console and adding your custom hooks jobs to run in conjunction with the provisioning adapter.

NOTE: The `job.paHooksVmJob` standard fact is available for implementation only in provision adapter jobs (xen, vsphere, and hyperv). For information about the location of this fact in the Development Client, see Provision Adapter Hook Jobs in "Job Control Settings" in the *NetIQ Cloud Manager 2.1.1 Orchestration Console Reference*.

The `job.paHooksVmJob` fact is a String array that needs to contain the names of the hooks job or jobs that you create and customize for use with the provisioning adapter. By default, the fact specifies that the listed jobs run sequentially as pre-event jobs.Then, after the provision adapter runs, these same hooks jobs execute in reverse sequence.

NOTE: Pre- and Post-VM event jobs are executed synchronously, defined by default sequence, or by custom sequence. The actual VM event is allowed to run synchronously or asynchronously.

The Cloud Manager Orchestration Server includes two example jobs (`paHooks_mount` and `paHooks_simple`) that were written for use by the Xen provisioning adapter. They illustrate some of the implemented Xen provision adapter pre and post hooks. You can use these example jobs as a model to write similar jobs for performing tasks prior to and following a VM event executed by the provisioning adapter.

Hyper-V hooks jobs you create can use a job structure similar to the example Xen jobs. For the vSphere provisioning adapter, however, we recommend that you confer with NetQ Consulting for assistance in creating your own hooks jobs.

You can create hooks jobs that implement only one kind of event or many events. The jobs can be configured to be triggered by a single type of event, or that include all of the VM events. Customization options allow flexibility in the role of the job.

NOTE: To increase the amount of detail available for hooks job implementation and monitoring, go to the hooks job and set the `job.debug` fact or the `job.tracing` fact to True.

9.2.2 Customizing Pre- and Post-Job Execution Order

If you don't want to use the default execution order for the Pre- and Post-event jobs, you can customize the order of their execution by adding custom facts:

- **job.paHooksPreVmJob:** Use this fact to hard code the implementation order of the hooks jobs prior to a VM event. Make sure the fact is defined as a String array.
- **job.paHooksPostVmJob:** Use this fact to hard code the implementation order of the hooks jobs following a VM event. Make sure the fact is defined as a String array.

IMPORTANT: Make sure you create these custom facts using the exact naming syntax shown above.

For example, if the default order for Pre-event VM jobs was Job 1, Job 2, Job 3 (specified using the `job.paHooksVmJob` fact), you could add the `job.paHooksPostVmJob` custom fact to specify the Post-event VM job execution order as Job 2, Job 1, Job 3.

9.2.3 Customizing Hooks Job Execution Based on Event Type

If you want to make sure that a provisioning adapter is invoked only when certain events occur, you have two options:

- Manually add hooks jobs to the `job.paHooksVmJob` fact that contain only the event types you want (this is always an option).
- Create a custom fact for the hooks jobs that works only for a specific event.

  The naming syntax for a custom fact like this follows the pattern of `job.paHooks<`*`Pre or Post`*`><`*`event_name`*`>VmJob`. For example, if you wanted to invoke a job to execute exclusively prior to a START event, you would add the custom fact `job.paHooksPreStartVmJob`, with the hooks job name or names specified in its String array.

  Each hooks job that you define is called only if it implements the event named, so you can set up a job for the provisioning adapter that implements only the desired event. For example, if you set up a Pre-Start VM hooks job for the Xen provisioning adapter, any other event except Start that occurs in that hooks job is never invoked.

A The Cloud Manager Orchestration Client SDK

The Cloud Manager Orchestration installation pattern includes a Java Client SDK in which you can write Java applications to remotely manage jobs. The zos command line tool is written using the Client SDK. This SDK application can perform the following operations:

- Login and logout to an Orchestration Server.
- Manage the life cycle of a job (run/cancel).
- Monitor running jobs (get job status).
- Communicate to a running job using events.
- Receive events from a running job.
- Search for grid objects using constraints.
- Retrieve and modify grid object facts.
- Datagrid operations (such as copying files to the server and downloading files from the server).

This section includes the following information:

A.1 SDK Requirements

Before you can run the Cloud Manager Orchestration Client SDK, you must perform the following tasks:

1. Install the Cloud Manager Orchestration Client package. For instructions, see "Launching the Orchestration Console and Logging in to the Orchestration Server" in the *NetIQ Cloud Manager 2.1.1 Orchestration Installation Guide*.
2. Install JDK 1.6.*x*.
3. Examine the two example Cloud Manager Orchestration SDK client applications that are included in the examples directory:
   - **extranetDemo:** Provides a sophisticated example of launching multiple jobs and listening and sending events to running jobs.
   - **cracker:** Demonstrates a simple example how to launch a job and listen for events sent from the job to the client application..

A.2 Creating an SDK Client

Use the following procedure to create an SDK client in conjunction with the sample Java code (see "Interface ClientAgent" on page 213):

1 Create ClientAgent instance:

```
// example zos server host is "myserver"

ClientAgent clientAgent = ClientAgentFactory.newClientAgent("myserver");
```

2 Use the following user and password example to log in to the Orchestration Server (see "Launching the Orchestration Console and Logging in to the Orchestration Server" in the *NetIQ Cloud Manager 2.1.1 Orchestration Installation Guide*):

```
  Credential credential =
CredentialFactory.newPasswordCredential(username,password);

  clientAgent.login(credential);
```

3 Run the server operations. In this case, it is the `quickie.jdl` example job (which must have been previously deployed) with no job arguments:

```
String jobID = clientAgent.runJob("quickie",null)
```

4 (Optional) Listen for server events using the AgentListener interface:

```
clientAgent.addAgentListener(this);
```

4a Register with the Orchestration Server to receive job events for the job you started.

```
clientAgent.getMessages(jobID);
```

5 Log out of the server:

```
clientAgent.logout()
```

A.3 Client SDK Reference information

This section provides the reference information for the Java classes used by the Cloud Manager Orchestration Client SDK

A.3.1 Constraint Package

The Java classes included in the Constraint package form the basis of the Cloud Manager Orchestration infrastructure. For complete documentation of each class, click on the links to access the online documentation javadoc.

Interfaces

The following Java files form the interfaces for the Cloud Manager Orchestration constraint grid structure:

Interface AndConstraint

Perform a logical and-ing of all child constraints.This is a no-op if this constraint contains no children.

For complete documentation of the class, see *AndConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/AndConstraint.html).

Interface BetweenConstraint

Binary Operator Constraints that have both a left and right side.

For complete documentation of the class, see *BetweenConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/BetweenConstraint.html).

Interface BinaryConstraint

Binary Operator Constraints that have both a left and right side.

For complete documentation of the class, see *BinaryConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/BinaryConstraint.html).

Interface Constraint

Basic Constraint interface that allows traversal and evaluation of a constraint tree.

For complete documentation of the class, see *Constraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/Constraint.html).

Interface ContainerConstraint

Container constraints that perform logical aggregation operations on contained constraints.

For complete documentation of the class, see *ContainerConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/ContainerConstraint.html).

Interface ContainsConstraint

Performs a simple set operation that returns true if the right side of the operation is found in the value set of the left side.

For complete documentation of the class, see *ContainsConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/ContainsConstraint.html).

Interface DefinedConstraint

Evaluates to true only if the left side fact is defined in the match context. If the left side is not defined, this will evaluate to false.

For complete documentation of the class, see *DefinedConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/DefinedConstraint.html).

Interface EqConstraint

Performs a equality constraint operation. Missing arguments will always result in this constraint evaluating to false.

Supported match modes:

- Strings — MATCH_MODE_REGEXP & MATCH_MODE_GLOB

For complete documentation of the class, see *EqConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/EqConstraint.html).

Interface ForEachIdConstraint

Representation of the ForEachId Constraint. Evaluate the child constraints for each object id contained in the specified list. This is a no-op if this constraint contains no children. Constraints are added to this constraint using `add()`.

For complete documentation of the class, see *ForEachIdConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/ForEachIdConstraint.html).

Interface GeConstraint

Performs a 'greater than or equal to' constraint operation. Missing arguments always result in this constraint evaluating to false. The standard lexicographical ordering of values is used to determine result.

For complete documentation of the class, see *GeConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/GeConstraint.html).

Interface GtConstraint

Performs a 'greater than' constraint operation. Missing arguments will always result in this constraint evaluating to false. The standard lexicographical ordering of values is used to determine result.

For complete documentation of the class, see *GtConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/GtConstraint.html).

Interface IfConstraint

Perform a conditional if,then,else block. If conditional passes, the pass block is run as it would be in an AND constraint. If the the conditional fails, the fail block is run as it would be in an AND constraint.

For complete documentation of the class, see *IfConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/IfConstraint.html).

Interface IncludeConstraint

Extends the class Constraint.

For complete documentation of the class, see *IncludeConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/IncludeConstraint.html).

Interface LeConstraint

Performs a "less than or equal to" constraint operation. Missing arguments always result in this constraint evaluating to false. The standard lexicographical ordering of values is used to determine result.

For complete documentation of the class, see *LeConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/LeConstraint.html).

Interface LtConstraint

Performs a "'less than" constraint operation. Missing arguments always result in this constraint evaluating to false. The standard lexicographical ordering of values is used to determine result.

For complete documentation of the class, see *LtConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/LtConstraint.html).

Interface NeConstraint

Performs a not equal constraint operation. Missing arguments always result in this constraint evaluating to false.

Supported match modes:

- Strings — MATCH_MODE_REGEXP & MATCH_MODE_GLOB

For complete documentation of the class, see *NeConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/NeConstraint.html).

Interface NotConstraint

Perform a logical not operation of all the child constraints. This is a no-op if this constraint contains no children.

For complete documentation of the class, see *NotConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/NotConstraint.html).

Interface OperatorConstraint

Operator constraints that perform comparison operation on facts.

For complete documentation of the class, see *OperatorConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/OperatorConstraint.html).

Interface OrConstraint

Perform a logical or-ing operation of all the child constraints. This is a no-op if this constraint contains no children.

For complete documentation of the class, see *OrConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/OrConstraint.html).

Interface TypedConstraint

Typed constraint must only be used as the outermost wrapper when it is necessary to override the default constraint type of 'resource.' It provides the necessary context about where to add the contained constraints.

For complete documentation of the class, see *TypedConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/TypedConstraint.html).

Interface UndefinedConstraint

Evaluates to true only if the left side fact is not defined in the match context. If the left side is not defined, this will evaluate to false.

For complete documentation of the class, see *UndefinedConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/UndefinedConstraint.html).

Classes

The following Java files form the classes for the Cloud Manager Orchestration constraint grid structure:

- "Class ContainsConstraint.ContainsMode" on page 209
- "Class ForEachIdConstraint.ForEachIdMode" on page 209

Class ContainsConstraint.ContainsMode

For complete documentation of the class, see *ContainsConstraint.ContainsMode* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/ContainsConstraint.ContainsMode.html).

Class ForEachIdConstraint.ForEachIdMode

For complete documentation of the class, see ForEachIdConstraint.ForEachIdMode (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/ForEachIdConstraint.ForEachIdMode.html).

Exceptions

The following Java files form the exceptions for the Cloud Manager Orchestration constraint grid structure:

- "Class ConstraintException" on page 209

Class ConstraintException

For exceptions that occur in parsing or executing constraints.

For complete documentation of the class, see *ConstraintException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/constraint/ConstraintException.html).

A.3.2 Datagrid Package

The Java classes included in the Datagrid package form the basis of the Cloud Manager Orchestration infrastructure. For complete documentation of each class, click on the links to access the online documentation javadoc.

- "Interfaces" on page 210
- "Classes" on page 211
- "Exceptions" on page 211

Interfaces

The following Java files form the interfaces for the Cloud Manager Orchestration datagrid structure:

Interface GridFile

Specifies the Cloud Manager Orchestration datagrid interface for individual files and directories.

This interface rather closely mirrors `java.io.File`. It does not, however, extend that class, since the standard Java I/O classes would not understand the semantics of this extended version. In particular, path names specified by this class refer to remote files that might not be directly accessible via the file system, as expected by standard Java file I/O classes.

The mirroring of `java.io.File` is done strictly for consistency and familiarity. There are a few methods in `java.io.File` that don't make sense in the context of the datagrid, and have thus been omitted. However, the commonly used methods such as `canWrite`, `mkdir()`, and so on are implemented and provide functionality for datagrid paths that is analogous to that provided by `java.io.File` for local file system paths.

For complete documentation of the class, see *GridFile* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/dataGrid/GridFile.html).

Interface GridFileFilter

Filter for accepting/rejecting file names in a directory list.

Filtering is done by fully qualified `GridFile` objects representing the files and directories contained directly under the parent.

For complete documentation of the class, see *GridFileFilter* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/dataGrid/GridFileFilter.html).

Interface GridFileNameFilter

Filter for accepting/rejecting file names in a directory list.

Filtering is done by simple string path component names relative to the parent.

For complete documentation of the class, see *GridFileNameFilter* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/dataGrid/GridFileNameFilter.html).

Classes

The following Java files form the classes for the Cloud Manager Orchestration datagrid structure:

Class DGLogger

Definitions of the DataGrid Logger options used for multicast.

For complete documentation of the class, see *DGLogger* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/dataGrid/DGLogger.html).

Exceptions

The following Java files form the exceptions for the Cloud Manager Orchestration datagrid structure:

Class DataGridException

General exception class for datagrid errors.

For complete documentation of the class, see *DataGridException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/dataGrid/DataGridException.html).

Class DataGridNotAvailableException

Exception thrown if the datagrid cannot be reached due to a network error.

For complete documentation of the class, see *DataGridNotAvailableException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/dataGrid/DataGridNotAvailableException.html).

Class GridFile.CancelException

Exception thrown by cancelled requests.

For complete documentation of the class, see *GridFile.CancelException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/dataGrid/GridFile.CancelException.html).

A.3.3 Grid Package

The Java classes included in the Grid package form the basis of the Cloud Manager Orchestration infrastructure. For complete documentation of each class, click on the links to access the online documentation javadoc.

Interfaces

The following Java files form the interfaces for the Cloud Manager Orchestration grid structure:

Interface AgentListener

Provides the interface necessary for processing messages sent from the Orchestration Server.

The implementation of this interface is registered with the agent using `ClientAgent.addAgentListener()`.

For complete documentation, see *AgentListener* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/AgentListener.html).

Interface ClientAgent

API for client communication with server for job and datagrid operations. This includes retrieving information about available jobs, to start jobs and to manage running jobs.

For complete documentation, see *ClientAgent* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/ClientAgent.html).

Interface Credential

A credential used for identity on the Cloud Manager Orchestration system. Use the `CredentialFactory` to create `Credential` instances.

For complete documentation, see *Credential* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Credential.html).

Interface Fact

The Fact object. This contains accessors for setting and getting fact values and for describing a Fact.

For complete documentation, see *Fact* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Credential.html).

Interface FactSet

Definition of a set of facts. Typically, this represents all facts associated with a particular Grid object.

NOTE: There is also a `FactSetSnapshot` that can hold a read-only, non-dynamic version of the facts.

For complete documentation, see *FactSet* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/FactSet.html).

Interface GridObjectInfo

Client interface to any Grid object. All “Info” objects are serializable.

For complete documentation, see *GridObjectInfo* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/GridObjectInfo.html).

Interface ID

A unique identifier for an engine or a facility or Grid object.

The default identifiers for the broker, facilities, and an unknown ID are defined here.

For complete documentation, see *ID* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/ID.html).

Interface JobInfo

A client representation of a deployed job..

The interface is for describing details about a deployed job. This is a simplified interface that is likely to be a subset of Job. It is used in the client API and for management.

For complete documentation, see *JobInfo* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/JobInfo.html).

Interface Message

A base interface for all the messages in the system. It defines the basic methods that must be implemented by a message class, and also defines sub-interfaces for each of the actual messages in the system.

All of these message interfaces are intended to be viewed from the perspective of the consumer, not the producer. Producers are responsible for implementing the concrete classes which will underly these interfaces, and have complete freedom of choice as to how to implement constructors and set methods.

For complete documentation, see *Message* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.html).

Interface Message.Ack

A general acknowledgement of "action" message.

For complete documentation, see *Message.Ack* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.Ack.html).

Interface Message.AuthFailure

Authentication failure messages indicating that processing of a client message will not occur because client credentials are invalid.

For complete documentation, see *Message.AuthFailure* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.AuthFailure.html).

Interface Message.ClientResponseMessage

All messages that can optionally carry an error string back to the client extend this.

For complete documentation, see *Message.ClientResponseMessage* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.ClientResponseMessage.html).

Interface Message.ConnectionID

Messages assigned a connection ID by the session manager.

For complete documentation, see *Message.ConnectionID* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.ConnectionID.html).

Interface Message.Event

An Event is used to signal clients and workflows. A client can receive an Event sent from a workflow. The `ClientAgent.sendEvent()` constructs an Event to send to workflows.

For complete documentation, see *Message.Event* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.Event.html).

Interface Message.GetGridObjects

Client request to retrieve an (optionally ordered) set of grid objects that match a search criteria (constraint).

For complete documentation, see *Message.GetGridObjects* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.GetGridObjects.html).

Interface Message.GridObjects

Server response to client request to retrieve a grid object set.

For complete documentation, see *Message.GridObjects* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.GridObjects.html).

Interface Message.JobAccepted

A JobAccepted message is sent in response to a RunJob message when a job is successfully accepted into the system.

For complete documentation, see *Message.JobAccepted* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.JobAccepted.html).

Interface Message.JobError

A JobError message is sent when an unrecoverable error occurs in a job.

For complete documentation, see *Message.JobError* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.JobError.html).

Interface Message.JobFinished

A JobFinished message is sent when processing of a job completes.

For complete documentation, see *Message.JobFinished* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.JobFinished.html).

Interface Message.JobIdEvent

Base Event interface for retrieving JobID used for jobid messages.

For complete documentation, see *Message.JobIdEvent* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.JobIdEvent.html).

Interface Message.JobInfo

A JobInfo message contains information describing a deployed job.

For complete documentation, see *Message.JobInfo* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.JobInfo.html).

Interface Message.Jobs

A Jobs message contains a list of deployed job names.

For complete documentation, see *Message.Jobs* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.Jobs.html).

Interface Message.JobStarted

A JobStarted message is sent when a job is successfully started.

For complete documentation, see *Message.JobStarted* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.JobStarted.html).

Interface Message.JobStatus

A JobStatus message contains the state of the specified job. This is used in lieu of JobStatusDetail to get simple state info.

For complete documentation, see *Message.JobStatus* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.JobStatus.html).

Interface Message.LoginFailed

Response message for an unsuccessful login.

For complete documentation, see *Message.LoginFailed* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.LoginFailed.html).

Interface Message.LoginSuccess

Response message for a successful login.

For complete documentation, see *Message.LoginSuccess* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.LoginSuccess.html).

Interface Message.LogoutAck

A LogoutAck indicates success or failure of logout operation. It is a specific message type so error filtering can be applied when the message can't be delivered because the transport has already been closed.

For complete documentation, see *Message.LogoutAck* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.LogoutAck.html).

Interface Message.RunningJobs

A RunningJobs message contains the list of running jobs.

For complete documentation, see *Message.RunningJobs* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.RunningJobs.html).

Interface Message.ServerStatus

A ServerStatus message.

ServerStatus is different from a normal status message from the server because it normally requires an action on the part of the receiver whereas a Status is more informational. ServerStatus can be used for server shutdown, restart, version upgrade, migration to new host, and so on.

For complete documentation, see *Message.ServerStatus* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.ServerStatus.html).

Interface Node

Internal interface for Node (Resource) Grid object.

For complete documentation, see *Node* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Node.html).

Interface Priority

Priority information.

For complete documentation, see *Priority* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Priority.html).

Interface WorkflowInfo

A WorkflowInfo can represent either a snapshot of a running instance or an historical record of an instance. It can be thought of as the client view of a Workflow which is its big sister that is active and runs in the server.

NOTE: `Workflow` extends `WorkflowInfo. LiteWorkflowInfo` implements `WorkflowInfo`.

Setter methods should be put in `Workflow` and not in `WorkflowInfo`.

For complete documentation, see *WorkflowInfo* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/WorkflowInfo.html).

Classes

The following Java files form the classess for the Cloud Manager Orchestration grid structure:

- "Class Message.NetConfigUpdate.ConfigSource" on page 217

Class Message.NetConfigUpdate.ConfigSource

Source enum for IP discovery.

For complete documentation, see *Message.NetConfigUpdate.ConfigSource* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/Message.NetConfigUpdate.ConfigSource.html).

Exceptions

The following Java files form the exceptions for the Cloud Manager Orchestration grid structure:

- "Class ClientOutOfDateException" on page 218
- "Class FactException" on page 218
- "Class GridAuthenticationException" on page 218
- "Class GridAuthorizationException" on page 218
- "Class GridConfigurationException" on page 218
- "Class GridDeploymentException" on page 218

Class ClientOutOfDateException

Grid exception indicating the client is not compatible with the server.

This exception is thrown if a connection cannot be established with the Cloud Manager Orchestration Sserver because the current client software is either too old or too new to be compatible with the server.

For complete documentation, see *ClientOutOfDateException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/ClientOutOfDateException.html).

Class FactException

For exceptions that occur in accessing or setting facts.

For complete documentation, see *FactException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/FactException.html).

Class GridAuthenticationException

Thrown when authentication is denied by an Orchestration Server.

For complete documentation, see *GridAuthenticationException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/GridAuthenticationException.html).

Class GridAuthorizationException

Thrown when credentials are insufficient for the desired grid operation.

For complete documentation, see *GridAuthorizationException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/GridAuthorizationException.html).

Class GridConfigurationException

Grid exception thrown to indicate a grid configuration error.

This exception is thrown to indicate a severe error in the grid's configuration that prevents it or one of its major components from operating correctly.

For complete documentation, see *GridConfigurationException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/GridConfigurationException.html).

Class GridDeploymentException

Thrown when credentials are insufficient for the desired grid operation.

For complete documentation, see *GridDeploymentException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/GridDeploymentException.html).

Class GridException

The base exception for all grid exceptions. This provides an easy way to catch multiple types of related exceptions in the system without needing to explicitly list every one.

For complete documentation, see *GridException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/GridException.html).

Class GridObjectNotFoundException

Thrown when a Grid object lookup does not find the requested object.

For complete documentation, see *GridObjectNotFoundException* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/grid/GridObjectNotFoundException.html).

A.3.4 TLS Package

The Java classes included in the TLS package form the basis of the Cloud Manager Orchestration infrastructure. For complete documentation of each class, click on the links to access the online documentation javadoc.

- "Interfaces" on page 219
- "Classes" on page 220

Interfaces

Interfaces used for secure authentication to Orchestration Server include the following:

- "Interface TlsCallbacks" on page 219

Interface TlsCallbacks

Callback interface for TLS certificate exceptions.

An instance of this interface may be passed to `TlsConfiguration.setCallbacks(TlsCallbacks)` or to `TlsConfiguration.setDefaultCallbacks(TlsCallbacks)` to provide customized handling of missing or mismatched TLS server certicates encountered while attempting to make TLS connections to an Orchestration Server.

The `TlsCallbacks.onCertificateNotFound(SocketAddress,PemCertificate)` method is invoked when the Orchestration Server returns a server certificate and there is currently no certificate found for that server. If this method returns false, a certificate exception is thrown on the client; otherwise, the code for this method can "accept" the certificate, possibly with user warnings and a confirmation dialog before returning true to indicate that the certificate is "OK."

The `TlsCallbacks.onCertificateMismatch(SocketAddress,PemCertificate,PemCertificate)` method is invoked when the Orchestration Server returns a server certificate that does not correctly match the current certificate held by the client. This exception is a more severe error than "not found" because it indicates a possible spoofing attempt by a "man in the middle." We recommend that implementations of this callback method default to returning false unless the user or administrator very specifically indicates a willingness to accept the new certificate.

For complete documentation of the class, see *TlsCallbacks* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/tls/TlsCallbacks.html).

Classes

Classes used for secure authentication to Orchestration Server include the following:

- "Class PemCertificate" on page 220
- "Class TlsConfiguration" on page 220

Class PemCertificate

PEM certificate wrapper for X.509 certificates.

This convenience class wraps up an X.509 certificate in an object that allows the certificate to be read from and stored to a standard PEM encoded X.509 certificate file. This allows the Orchestration Clients to make use of the Sun TLS provider without requiring that the Orchestration Server certificate be manually installed in the JRE's keystore. The use of standardized PEM certificates allows more portable handling and offline generation of certificates (if desired for security purposes) and enables simplified management of certificates.

For complete documentation of the class, see *PemCertificate* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/tls/PemCertificate.html).

Class TlsConfiguration

TLS Configuration parameters for Orchestration Clients.

This class holds various TLS-related configuration parameters used by Orchestration Clients to connect to the server. By passing an instance of this class to the Development Client API upon login, or by setting global defaults, the behavior and usage of Transport Layer Security (TLS) can be modified by client writers.

The factory default behavior for Orchestration Clients and agents is to enable TLS, with the level of encryption specified by the server upon client connection. By default, "client" (as opposed to "agent") connection mode is assumed.

This class is fully cloneable and serializable. It is recommended that the type-safe `TlsConfiguration.copy()` method be used for cloning.

The following is a simple example of the usage of this class to configure TLS support on a client connection to an Orchestration Server:

```
    import com.novell.zos.tls.TlsConfiguration;
    import com.novell.zos.grid.ClientAgent;
    import com.novell.zos.grid.Credential;
    import com.novell.zos.toolkit.CredentialFactory;
    ...
    TlsConfiguration tlsConfig = new TlsConfiguration();
    tlsConfig.setCertificatePath("/tmp");
    Credential cred = CredentialFactory.newPasswordCredential("user",
"pass".toCharArray());
    ClientAgent client = ClientAgentFactory.newClientAgent("127.0.0.1");
    client.setTlsConfiguration(tlsConfig);
    client.login(cred);
    System.out.println("Logged In");
```

If custom handling of unknown or mismatched server certificates is required by the client, then add a call to `TlsConfiguration.setCallbacks(TlsCallbacks)` with an instance of `TlsCallbacks` providing methods for handling each of those cases.

If certain TLS parameters will always be the same for all instances (that is, they were specified on a global command line at JVM launch), those parameters can be specified as "global defaults" using the "setDefault*" versions of the various methods of this class. This can be used to avoid passing global configuration parameters among many different objects.

For complete documentation of the class, see *TlsConfiguration* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/tls/TlsConfiguration.html).

A.3.5 Toolkit Package

The Java classes included in the Toolkit package form the basis of the Cloud Manager Orchestration infrastructure. For complete documentation of each class, click on the links to access the online documentation javadoc.

- "Classes" on page 221

Classes

The following Java files form the classes for the Cloud Manager Orchestration toolkit structure:

- "Class ClientAgentFactory" on page 221
- "Class ConstraintFactory" on page 221
- "Class CredentialFactory" on page 221

Class ClientAgentFactory

Factory pattern used to create new clients for connection to an Orchestration Server. This is the starting point for clients to communicate with the server.

For complete documentation of the class, see *ClientAgentFactory* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/toolkit/ClientAgentFactory.html).

Class ConstraintFactory

Factory pattern used to create constraint objects that can be combined into larger constraint hierarchies for use in searches or other constraint-based matching.

For complete documentation of the class, see *ConstraintFactory* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/toolkit/ConstraintFactory.html).

Class CredentialFactory

Factory pattern used to create a `Credential` used for connection to an Orchestration Server.

For complete documentation of the class, see *CredentialFactory* (http://www.novell.com/documentation/cloudmanager2/resources/sdkjavadoc_2/com/novell/zos/toolkit/CredentialFactory.html).

B Cloud Manager Orchestration Job Classes and JDL Syntax

B.1 Job Class

To review the detailed JDL structure of the joblet class, see Job (page 252).

B.2 Joblet Class

To review the detailed JDL structure of the joblet class, see Joblet (page 254).

B.3 Utility Classes

The following are some of the main utility JDL classes you can use to customize your Cloud Manager Orchestration jobs:

- DataGrid (page 242)
- Exec (page 245)
- MatrixInfo (page 261)
- ResourceInfo (page 273)
- RunJobSpec (page 274)
- ScheduleSpec (page 275)

B.4 Built-in JDL Functions and Variables

The information in this section defines the built-in Cloud Manager Orchestration JDL functions and variables.

B.4.1 getMatrix()

This function returns the matrix grid object. For more information, see MatrixInfo (page 261).

Purpose: The matrix object is used to retrieve other grid objects in the system.

B.4.2 system(cmd)

This executes a system command in a shell on the resource. The command is passed to the operating system’s default command interpreter. On Microsoft Windows systems this is `cmd.exe`, while on POSIX systems, this is `/bin/sh`. Stdout and stderr are directed to the job log. No access to stdin is provided.

Returns: Returns an exit code result of the command execution.

B.4.3 Grid Object TYPE_* Variables

The list of variables are constants for grid object type. For more information, see MatrixInfo (page 261).

Variable Names:

```
TYPE_USER
TYPE_JOB
TYPE_RESOURCE
TYPE_VMHOST
TYPE_REPOSITORY
TYPE_USERGROUP
TYPE_JOBGROUP
TYPE_RESOURCEGROUP
TYPE_REPOSITORYGROUP
TYPE_VNIC
TYPE_VDISK
TYPE_VBRIDGE
TYPE_VBRIDGEGROUP
```

Type: String.

Purpose: Use these in JDL functions for retrieving and creating grid objects.

B.4.4 The __agent__ Variable

Variable Name: `__agent__`

Type: Boolean.

Purpose: Defines whether the JDL is executing on the agent.

B.4.5 The __jobname__ Variable

Variable Name: `__jobname__`

Type: String.

Purpose: Defines the name of the deployed job.

B.4.6 The __mode__ Variable

Variable Name: `__mode__`

Type: String.

Purpose: Defines the execution mode.

Values:

`parse` - JDL is being parsed.

`deploy` - JDL is being deployed.

`undeploy` - JDL is being undeployed.

`runtime` - JDL is being executed.

B.5 Job State Field Values

Here are the job state field values for the Job (page 252) class:

Constant	Value	Description
int CANCELLED_STATE	9	Cancelled end state.
int CANCELLING_STATE	6	Cancelling. Transitions to: Cancelled.
int COMPLETED_STATE	8	Completed end state.
int COMPLETING_STATE	5	Completing. Transitions to: Completing.
int FAILED_STATE	10	Failed end state.
int FAILING_STATE	7	Failing. Transitions to: Failed.
int PAUSED_STATE	4	Paused. Transitions to: Running/Completing/ Failing/Cancelling.
int QUEUED_STATE	1	Queued. Transitions to: Starting/Failing/ Cancelling.

Constant	Value	Description
int RUNNING_STATE	3	Running. Transitions to: Paused/Completing/ Failing/Cancelling.
int STARTING_STATE	2	Starting. Transitions to: Running/Failing/ Cancelling.
int SUBMITTED_STATE	0	Submitted. Transitions to: Queued/Failing.
String TERMINATION_TYPE_ADMIN	“Admin"	Indicates Job was cancelled by the admin and only applies if Job is in CANCELLED_STATE. Value is obtained from `jobinstance.terminationtype fact`.
String TERMINATION_TYPE_JOB	“Job”	Indicates Job was cancelled due to exceeding the job timeout value and only applies if Job is in CANCELLED_STATE. The value is obtained from `jobinstance.terminationtype fact`.
String TERMINATION_TYPE_TIMEOUT	“Timeout”	Indicates Job was cancelled due to exceeding the job timeout value and only applies if Job is in CANCELLED_STATE. Value is obtained from `jobinstance.terminationtype fact`.
String TERMINATION_TYPE_USER	“User”	Indicate Job was cancelled by client user and only applies if Job is in CANCELLED_STATE. The value is obtained from `jobinstance.terminationtype fact`.

B.6 Repository Information String Values

Constant	Value	Description
SAN_TYPE_FibreChannel	Fibre Channel	Specifies a fibre channel SAN repository.
SAN_TYPE_ISCSI	iSCSI	Specifies an iSCSI SAN repository.
SAN_VENDOR_IQN	iqn	Specifies an IQN SAN repository.
SAN_VENDOR_NPIV	npiv	Specifies a N_Port ID Virtualization SAN repository.
TYPE_DATAGRID	datagrid	Specifies a datagrid repository.
TYPE_LOCAL	local	Specifies a local repository.
TYPE_NAS	NAS	Specifies a NAS repository.
TYPE_SAN	SAN	Specifies a SAN repository.
TYPE_VIRTUAL	virtual	Specifies a virtual repository.

B.7 Joblet State Values

The following values are defined for the various states that the joblet can be in:

Constant	Value	Description
INITIAL_STATE	0	Joblet initial state.
WAITING_STATE	1	Joblet waiting for a resource
WAITING_RETRY_STATE	2	Joblet waiting for a resource for retry.
CONTRACTED_STATE	3	Joblet waiting for a resource for retry.
STARTED_STATE	4	Joblet started on a resource.
PRE_CANCEL_STATE	5	Joblet starting cancellation.
CANCELLING_STATE	6	Joblet cancelling.
POST_CANCEL_STATE	7	Joblet finishing cancellation.
COMPLETING_STATE	8	Joblet completing state.
FAILING_STATE	9	Joblet failing state.
FAILED_STATE	11	Joblet failed end state.
CANCELLED_STATE	12	Joblet cancelled end state.
COMPLETED_STATE	13	Joblet completed end state.

See Joblet (page 254).

B.8 Resource Information Values

Use the following values to specify a resource type:

Constant	Value Type	Resource Description
TYPE_BM_INSTANCE	String	Blade server.
TYPE_BM_TEMPLATE	String	Blade server template.
TYPE_FIXED_PHYSICAL	String	Fixed physical server.
TYPE_VM_INSTANCE	String	VM server.
TYPE_VM_TEMPLATE	String	VM template.

For full class descriptions, see ResourceInfo (page 273).

B.9 JDL Class Definitions

The following Cloud Manager Orchestration JDL classes can be implemented in the custom jobs that you create. Because JDL is implemented in Java, we have provided direct links to detailed Javadoc for each of the "Pythonized" JDL classes below:

- "AndConstraint" on page 230
- "BetweenConstraint" on page 231

AndConstraint

Representation of the And Constraint. Perform a logical ANDing of all child constraints. If this constraint contains no children, no operation is performed. Constraints are added to this constraint using `add()`.

See Also

- ContainerConstraint (page 238)
- Javadoc: *AndConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/AndConstraint.html)

BetweenConstraint

Representation of the Between Constraint. Evaluates to true only if the left side fact is between the values specified in the right side. Contains is typically used to check membership of a value in a group fact.

Description

Evaluates to true only if the left side fact is between the values specified in the right side. Contains is typically used to check membership of a value in a group fact.

Example of building a `ContainsConstraint` to constrain that a resource belongs to a group:

```
c = BetweenConstraint()
c.setFact("matrix.time")
c.setBeginValue("9:00 AM")
c.setEndFactValue("user.home.time")
```

This constraint can be used independently or added to a And, Or, Not constraint to combine with other constraints.

See Also

- Constraint (page 237)
- Javadoc: *BetweenConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/BetweenConstraint.html)

BinaryConstraint

Representation of a Constraint operating on the left and right operands. This is a base class and is not directly constructed.

See Also

- Subclasses: ContainsConstraint (page 239), EqConstraint (page 244), GeConstraint (page 248), GtConstraint (page 251), LeConstraint (page 257), LtConstraint (page 258), NeConstraint (page 263).
- Javadoc: *BinaryConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/BinaryConstraint.html)

BuildSpec

Defines the attributes for building a new VM. An instance of this class is passed to `resource.build()`.

See Also

- Javadoc: *BuildSpec* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/BuildSpec.html)

CharRange

Define lexical character string range of values for ParameterSpace scheduling.

See Also

- Javadoc: *CharRange* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/CharRange.html)

ComputedFact

Defines the base class for creating custom computed facts. Computed facts provide the ability to create custom calculations that extend the built-in factsets for a Grid object. The computed fact can be in constraints. User defined computed facts are required to subclass this class. In order to use `ComputedFact`, you must deploy a subclass of `ComputedFact` and then create a linked fact referencing the deployed `ComputedFact`. The linked fact is then used in constraints.

See Also

- ComputedFactContext (page 236)
- Javadoc: *ComputedFact* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ComputedFact.html)

ComputedFactContext

Provides access to the evaluation context.

Description

The context contains the grid objects that the constraint engine uses to evaluate constraints.If they are available in the current context, the ComputedFactContext provides access to the current job instance, deployed job, User, Resource, vBridge, and Repository grid objects.

The VM host, vBridge and Repository grid objects are only in the context for the evaluation of the provisioning constraints such as vmHost. The Job and Job Instance objects are only in the context for a resource or allocation constraint evaluation.

See Also

- ComputedFact (page 235)
- Javadoc: *ComputedFactContext* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ComputedFactContext.html)

Constraint

Defines the base class for all constraint classes.

See Also

- BinaryConstraint (page 232), ContainerConstraint (page 238), DefinedConstraint (page 243), UndefinedConstraint (page 278).
- Javadoc: *Constraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/Constraint.html)

ContainerConstraint

Representation of a Constraint that contains other Constraints. This is a base class and is not directly constructed.

See Also

- Subclasses: AndConstraint (page 230), NotConstraint (page 264), OrConstraint (page 265)
- Javadoc: *ContainerConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ContainerConstraint.html)

ContainsConstraint

Representation of the Contains Constraint. Evaluates to true only if the left side fact is defined in the match context. If the left side is not defined, this will evaluate to False. Contains is typically used to check membership of a value in a group fact.

See Also

Javadoc: *ContainsConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ContainsConstraint.html)

Credential

Representation of a credential stored in the CredentialManager.

Description

Fields contained in this object are:

- **name:** The ID of this credential.
- **type:** A string type used to group related credentials (for example, amazon-ec2).
- **user:** The user string.
- **secret:** The secret associated with this user.

See Also

Javadoc: *Credential* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/Credential.html)

CredentialManager

Interface into the CredentialManager.

Description

This class is available only in a joblet context. Only users in the administrator group or the system user are allowed access to these methods.

From this class you can add, get, update and delete credentials that are stored in an encrypted store on the Orchestration Server.

See Also

Javadoc: *CredentialManager* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/CredentialManager.html)

DataGrid

General interface to the datagrid. See Chapter 5, "The Cloud Manager Orchestration Datagrid," on page 109.

See Also

- GridObjectInfo (page 249)
- Javadoc: *DataGrid* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/DataGrid.html)

DefinedConstraint

Representation of the Defined Constraint. Evaluates to true only if the left side fact is defined in the match context. If the left side is not defined, this will evaluate to False. This constraint can be used independently or added to a And, Or, Not constraint to combine with other constraints.

See Also

- Constraint (page 237)
- Javadoc: *DefinedConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/DefinedConstraint.html)

EqConstraint

Representation of the Equals Constraint. This constraint can be used independently or added to a And, Or, Not constraint to combine with other constraints. Extends BinaryConstraint (page 232).

See Also

- BinaryConstraint (page 232)
- Javadoc: *EqConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/EqConstraint.html)

Exec

The Exec class is used to manage command line execution on resources. This class defines options for input, output and error stream handling, and process management including signaling, error and timeout control.

Description

A command's standard output and error can be redirected to a file, to a stream, to write to the job log, or be discarded. By default, the output is discarded. A command's standard input can be directed from a file or a stream can be written to. By default, the input is not used.

By default, command line execution is done in behalf of the job user. Exec instances are only allowed during the running of the Joblet class on a resource. The built-in function system() can also be used for simple execution of command lines.

See Also

- Javadoc: *Exec* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/Exec.html)

ExecError

`ExecError` is raised for errors in executing a command line using the Exec (page 245) class or system(). Normal raising of this error causes the joblet to fail. Put this Error in an try except block to handle the error.

See Also

- Exec (page 245)
- Javadoc: *ExecError* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ExecError.html)

FileRange

Define a range of values for a ParameterSpace (page 266) based on the lines of a text file. An instance of this class is used as a dimension in a `ParameterSpace` definition.The file name must either refer to a file that is readable from the server and resources (on a shared file system) or must be a DataGrid (page 242) file URL.

See Also

- DataGrid (page 242) and ParameterSpace (page 266)
- Javadoc: *FileRange* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/FileRange.html)

GeConstraint

Representation of the Greater than or Equals constraint. Performs a ‘greater than or equal to’ constraint operation. Missing arguments will always result in this constraint evaluating to false. The standard lexographical ordering of values is used to determine result. This constraint can be used independently or added to a And, Or, Not constraint to combine with other constraints. Extends BinaryConstraint (page 232).

See Also

- Constraint (page 237) and BinaryConstraint (page 232).
- Javadoc: *GeConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/GeConstraint.html)

GridObjectInfo

The `GridObjectInfo` class is the base class representation of all grid objects in the system. This provides accessors and setters to a grid object's fact set.

See Also

Javadoc: *GridObjectInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/GridObjectInfo.html)

GroupInfo

he GroupInfo class is a representation of Group grid objects. Operations include retrieving the group member lists and adding/removing from the group member lists, and retrieving and setting facts on the group. Extends GridObjectInfo (page 249).

See Also

- GridObjectInfo (page 249)
- Javadoc: *GroupInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/GroupInfo.html)

GtConstraint

Representation of the Greater than Constraint. Performs a 'greater than' constraint operation. Missing arguments will always result in this constraint evaluating to false. The standard lexographical ordering of values is used to determine result. This constraint can be used independently or added to a And, Or, Not constraint to combine with other constraints.

See Also

- Constraint (page 237)
- Javadoc: *GtConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/GtConstraint.html)

Job

The `Job` class represents a running job instance. This class defines functions for interacting with the server including handling notification of job state transitions, child job submission, managing joblets and for receiving and sending events from resources and from clients. A job writer defines a subclass of the Job class and uses the methods available on the Job class for scheduling joblets and event processing.

Issues with this Class

- "Calling terminate() from within a Job Class Allows the JDL Thread Execution to Continue" on page 252

Calling terminate() from within a Job Class Allows the JDL Thread Execution to Continue

It is possible that when you attempt to deploy a component such as .job, sjob, jdl, cfact, event, metric, policy, eaf, sar, sched, trig, python, pylib; where prepackaged components are located in the `/opt/novell/zenworks/zos/server/components` directory, the Orchestration might intermittently fail the deployment, displaying a message similar to the following:

```
  class foo(Job):
      def job_started_event(self):
          self.userID = "foo"
```

results in the following job failure:

```
  JobID: aspiers.jobIDtestjob.118426
  Traceback (most recent call last):
    File "jobIDtestjob", line 10, in job_started_event
  AttributeError: read-only attr: userID
  Job 'aspiers.jobIDtestjob.118426' terminated because of failure. Reason:
AttributeError: read-only attr: userID
```

The following identifiers are known to cause problems:

- jobID
- name
- type
- userID

To work around this issue, rename any of these attributes in your JDL code.

See Also

- JobInfo (page 253), Joblet (page 254), JobletInfo (page 255), JobletParameterSpace (page 256)
- Javadoc: *Job* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/Job.html)

JobInfo

The JobInfo class is a representation of a deployed job. The factset available on the JobInfo class is the aggregation of the job's policy and policies on the groups the job is a member of. This includes the "job.*" and "jobargs.*" fact namespaces.

See Also

- Job (page 252)
- Javadoc: *JobInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/JobInfo.html)

Joblet

Defines the execution on the resource. A Job writer constructs a subclass of Joblet to define code to run on a resource. The method `joblet_started_event` is required to be implemented. `joblet_started_event` is invoked on the resource. The Job writer invokes the `schedule()` function in the Job subclass to define when and which resource the Joblet is executed. Each Joblet instance has the Job instance (`jobinstance.*`, `job.*`, `jobargs.*`, `user.*`), Resource (`resource.*`) and Joblet (`joblet.*`, `jobletargs.*`) fact sets available using the base `GridObjectInfo` fact functions. For example, you can use `self.getFact()` to retrieve a Joblet fact. Use the `getMatrix()` built-in function to retrieve facts for other Grid Objects that are not in the context of this Joblet instance's fact set.

See Also

- Job, JobInfo
- Javadoc: *Joblet* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/Joblet.html)

JobletInfo

JobletInfo is a representation of the Joblet grid object created when a job calls `schedule()` to create joblets. This class provides access to a joblet's factset and operations on a joblet such as cancellation and sending events to a joblet that is running on a resource. The separate Joblet class defines execution on a resource.

See Also

Javadoc: *JobletInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/JobletInfo.html)

JobletParameterSpace

JobletParameterSpace is a slice of the ParameterSpace allocated to a joblet. As the scheduler defines slices of the parameter space for a given `schedule()`, JobletParameterSpace instances are created for each joblet. This slice of the parameter space is delivered to the resource on joblet execution. The JobletParameterSpace can also be retrieved from the Joblet object.

See Also

Javadoc: *JobletParameterSpace* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/JobletParameterSpace.html)

LeConstraint

Representation of the Less than or equals Constraint. Performs a "less than or equal to" constraint operation. Missing arguments will always result in this constraint evaluating to false. The standard lexographical ordering of values is used to determine result. This constraint can be used independently or added to a And, Or, Not constraint to combine with other constraints.

See Also

Javadoc: *LeConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/LeConstraint.html)

LtConstraint

Representation of the Less than Constraint. Performs a "less than" constraint operation. Missing arguments always result in this constraint evaluating to false. The standard lexographical ordering of values is used to determine result. This constraint can be used independently or added to a And, Or, Not constraint to combine with other constraints.

See Also

Javadoc: *LtConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/LtConstraint.html)

MatchContext

The MatchContext defines a context for evaluating a constraint. An instance of this class is supplied to `match()` for evaluating constraints. The `MatchContext` provides a way to setup an evaluation context that the constraint engine is using to evaluate constraints. The `MatchContext` is filled out with the context that is required for evaluating your constraints. This includes assigning a deployed Job, User, Resource, VM Host, vBridge, vNIC, and Repository Grid objects.

See Also

- Javadoc: *MatchContext* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/MatchContext.html)

MatchResult

The MatchResult class defines results of a Constraint `match()`. Instance of `MatchResult` is returned from `match()` operations. From the `MatchResult` you can retrieve a list of the IDs of the matching Grid objects and the non-matching Grid objects.

See Also

- Javadoc: *MatchResult* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/MatchResult.html)

MatrixInfo

The MatrixInfo class is a representation of the matrix grid object (see GridObjectInfo (page 249)). This provides operations for retrieving and creating grid objects in the system. MatrixInfo is retrieved using the built-in `getMatrix()` function. Write capability is dependent on the context in which `getMatrix()` is called. For example, in a joblet process on a resource, creating new grid objects is not supported.

See Also

- Javadoc: *MatrixInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/MatrixInfo.html)
- Section B.4, “Built-in JDL Functions and Variables,” on page 224.

MigrateSpec

The MigrateSpec class defines the options for the migrate action. An instance of this class is passed to the `resource.migrate()` method.

Example

The following is an example of using `MigrateSpec` to define a migrate action for a Virtual Machine named “sles10” to a VM host named “host2:”

```
vm = getMatrix().getGridObject(TYPE_RESOURCE,"sles10")
spec = MigrateSpec()
spec.setHost('host2')
vm.migrate(spec)
```

See Also

Javadoc: *MigrateSpec* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/MigrateSpec.html)

NeConstraint

Representation of the Not Equals Constraint. Performs a not equal constraint operation. Missing arguments will always result in this constraint evaluating to false. This constraint can be used independently or added to a And, Or, Not constraint to combine with other constraints.

See Also

Javadoc: *NeConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/NeConstraint.html)

NotConstraint

Representation of a Not Constraint Object. Performs a logical not operation of all the child constraints. This is a no-op if this constraint contains no children. Constraints are added to this constraint using `add()`.

See Also

- See Constraint (page 237) and ContainerConstraint (page 238).
- Javadoc: *NotConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/NotConstraint.html)

OrConstraint

Representation of Or Constraint Object. Perform a logical or-ing operation of all the child constraints. This is a no-op if this constraint contains no children. Constraints are added to this constraint using `add()`.

See Also

- See Constraint (page 237) and ContainerConstraint (page 238).
- Javadoc: *OrConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/OrConstraint.html)

ParameterSpace

Defines a parameter space to be used by the scheduler to create a joblet set. A parameter space may consist of rows of columns or a list of columns that is expanded and can be turned into a cross product. Use `appendRow` to create a `rowMajor` parameter space or `appendCol` to define a column expansion. You cannot use both `appendRow()` and `appendCol()` in the same `ParameterSpace`. Once the scheduler defines a slice of the parameter space for a given Joblet, the scheduler creates `JobletParameterSpace` instances for each Joblet. This slice of the parameter space is delivered to the resource. To limit how many Joblets or the number of rows in a Joblet, use `setMaxJobletSize` or use the `jobletCount` argument to `schedule()` or `SchedulSpec`.

See Also

Javadoc: *ParameterSpace* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ParameterSpace.html)

PdiskInfo

Representation of a pDisk Grid Object. This class provides accessors and setters for pDdisk facts. See `RepositoryInfo.createPdisk()` for how to script creation of pDisk objects.

See Also

Javadoc: *PdiskInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/PdiskInfo.html)

PolicyInfo

Representation of a Policy Object. This class allows for associating and unassociation of Grid objects using this policy

See Also

Javadoc: *PolicyInfo* (http://www.novell.com/documentation/pso_orchestrate25/resources/jdldoc/com/novell/zos/jdl/PolicyInfo.html)

ProvisionJob

This class extends `ProvisionJobBase`. It is a superclass for provisioning Adapter (PA) jobs desiring to use hooks (pre- and post-event) infrastructure. To enable this infrastructure, PA jobs should declare the job class in jdl as `ProvisionJob`.

See Also

Javadoc: *ProvisionJob* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ProvisionJob.html)

ProvisionJoblet

This class extends `Joblet`. Superclass for provisioning adapter (PA) joblets desiring to use hooks (pre- and post-event) infrastructure. To enable this infrastructure, PA jobs should declare the joblet class in jdl as `ProvisionJoblet`.

See Also

Javadoc: *ProvisionJoblet* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ProvisionJoblet.html)

ProvisionSpec

Defines the attributes for starting a provision. An instance of this class is passed to the `self.provision()` method. If no reservations are specifed the lifecycle mode is MODE_MANUAL.

See Also

Javadoc: *ProvisionSpec* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ProvisionSpec.html)

RepositoryInfo

`RepositoryInfo` is a representation of a Repository grid object. This class provides accessors and setters for Repository facts. See MatrixInfo (page 261) for how to script creation of Repository objects.

See Also

- See GridObjectInfo (page 249) and MatrixInfo (page 261).
- Javadoc: *RepositoryInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/RepositoryInfo.html)

ResourceInfo

ResourceInfo is a representation of a Resource grid object. This class inherits the base fact operations from GridObjectInfo (page 249) and adds the provisioning operations for provisionable resources such as VMs. See MatrixInfo (page 261) for how to script creation of Resource objects.

See Also

- GridObjectInfo (page 249) and MatrixInfo (page 261).
- Javadoc: *ResourceInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ResourceInfo.html)

RunJobSpec

Defines the attributes for starting a child job or a standalone job. An instance of this class is passed to `self.runJob()`.

See Also

Javadoc: *RunJobSpec* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/RunJobSpec.html)

ScheduleSpec

Defines one or more joblets to be scheduled and run on resources. A `ScheduleSpec` instance is passed to the job's `schedule()` or `scheduleSweep()`. `schedule()` creates the joblets and schedules joblets to run on resources.

See Also

- Joblet (page 254)
- Javadoc: *ScheduleSpec* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ScheduleSpec.html)

SparseFiles

Extends `java.lang.Object`.

See Also

- Javadoc: *SparseFiles* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/SparseFiles.html)

Timer

Timer schedules a callback to a job or joblet method. Timers can schedule a one time or a repeated callback on an interval basis. An active Timer keeps the job or joblet running. You must manually cancel or terminate the job or joblet or invoke the Timer's `cancel()` method.

See Also

Javadoc: *Timer* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/Timer.html)

UndefinedConstraint

Representation of the Undefined Constraint. Evaluates to true only if the left side fact is *not* defined in the match context. If the left side is not defined, this will evaluate to `false`. This constraint can be used independently or added to a And, Or, Not constraint to combine with other constraints.

See Also

- Constraint (page 237)
- Javadoc: *UndefinedConstraint* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/UndefinedConstraint.html)

UserInfo

UserInfo is a representation of a user grid object. This class provides accessors and setters for User facts.

See Also

Javadoc: *UserInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/UserInfo.html)

VbridgeInfo

VbridgeInfo is a representation of a vBridge grid object. This class provides accessors and setters for vBridge facts. See `VMHostInfo.createVbridge()` (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/VMHostInfo.html#createVbridge(java.lang.String)) for how to script creation of vBridge objects.

See Also

Javadoc: *VbridgeInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/VbridgeInfo.html)

VdiskInfo

VdiskInfo is a representation of a vDisk grid object. This class provides accessors and setters for vDisk facts. See `ResourceInfo.createVdisk()` (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ResourceInfo.html#createVdisk()) for how to script creation of vDisk objects.

See Also

Javadoc: *VdiskInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/VdiskInfo.html)

VmHostClusterInfo

Extens the VMHostInfo class. VmHostCluster is a representation of a clustered VM host Grid Objec. This class provides accessors and setters for vmHostClouster facts.

See Also

Javadoc: *VmHostClusterInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/VmHostClusterInfo.html)

VMHostInfo

The VmHostInfo class is a representation of a VM host grid object. This class provides accessors and setters to the VM host facts and operations to control the state of the VM host object.

See Also

Javadoc: *VMHostInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/VMHostInfo.html)

VmSpec

Defines the attributes for creating a virtual machine. An instance of this class is passed to `resource.createInstance()`, `resource.createTemplate()`, `resource.clone()`.

Example

Example of using `VmSpec` for creating a clone on a named host from a template resource:

```
template = getMatrix().getGridObject(TYPE_RESOURCE,"myTemplate")
spec = VmSpec()
spec.setNewName("newvm")
spec.setHost('vmhost-qa')
template.clone(spec)
```

If the host and repository is not set, the default is to use the source resource object's repository as the destination repository.

See Also

- VMHostInfo (page 283)
- Javadoc: *VmSpec* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/VmSpec.html)

VnicInfo

VnicInfo is a representation of a vNIC grid object. This class provides accessors and setters for vNIC facts. See `ResourceInfo.createVnic()` (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/ResourceInfo.html#createVnic()) for how to script creation of vNIC objects.

See Also

Javadoc: *VnicInfo* (http://www.novell.com/documentation/cloudmanager2/resources/jdljavadoc_2/com/novell/zos/jdl/VnicInfo.html)