

Developer Guide and Reference

Novell® ZENworks® Orchestrator

1.3

August 3, 2008

www.novell.com



Legal Notices

Novell, Inc. makes no representations or warranties with respect to the contents or use of this documentation, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Further, Novell, Inc. makes no representations or warranties with respect to any software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to make changes to any and all parts of Novell software, at any time, without any obligation to notify any person or entity of such changes.

Any products or technical information provided under this Agreement may be subject to U.S. export controls and the trade laws of other countries. You agree to comply with all export control regulations and to obtain any required licenses or classification to export, re-export or import deliverables. You agree not to export or re-export to entities on the current U.S. export exclusion lists or to any embargoed or terrorist countries as specified in the U.S. export laws. You agree to not use deliverables for prohibited nuclear, missile, or chemical biological weaponry end uses. See the [Novell International Trade Services Web page \(http://www.novell.com/info/exports/\)](http://www.novell.com/info/exports/) for more information on exporting Novell software. Novell assumes no responsibility for your failure to obtain any necessary export approvals.

Copyright © 2008 Novell, Inc. All rights reserved. No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without the express written consent of the publisher.

Novell, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed on the [Novell Legal Patents Web page \(http://www.novell.com/company/legal/patents/\)](http://www.novell.com/company/legal/patents/) and one or more additional patents or pending patent applications in the U.S. and in other countries.

Novell, Inc.
404 Wyman Street, Suite 500
Waltham, MA 02451
U.S.A.
www.novell.com

Online Documentation: To access the latest online documentation for this and other Novell products, see [the Novell Documentation Web page \(http://www.novell.com/documentation\)](http://www.novell.com/documentation).

Novell Trademarks

For Novell trademarks, see [the Novell Trademark and Service Mark list \(http://www.novell.com/company/legal/trademarks/tmlist.html\)](http://www.novell.com/company/legal/trademarks/tmlist.html).

Third-Party Materials

All third-party trademarks are the property of their respective owners.

Contents

About This Guide	11
1 Getting Started	15
1.1 What You Should Know	15
1.1.1 Prerequisite Knowledge	15
1.1.2 Setting Up Your Development Environment	16
1.2 Orchestrator Documentation Set	16
1.2.1 Novell ZENworks Orchestrator Getting Started Guide	17
1.2.2 Novell ZENworks Orchestrator Administration Guide	17
1.2.3 Novell ZENworks Orchestrator Virtual Machine Management Guide	17
1.2.4 Novell ZENworks Orchestrator Job Management Guide	17
1.3 Prerequisites for the Development Environment	17
2 Job Development Concepts	19
2.1 Orchestrator Development Architecture	19
2.1.1 Orchestrator Agents	20
2.1.2 Orchestrator Resource Monitor	21
2.1.3 Orchestrator Entity Types and Managers	21
2.1.4 Jobs	24
2.1.5 Constraint-Based Job Scheduling	27
2.1.6 Understanding Orchestrator API Interfaces	28
2.2 Understanding ZENworks Orchestrator Functionality	30
2.2.1 Resource Virtualization	30
2.2.2 Policy-Based Management	31
2.2.3 Global Resource Visualization	32
2.2.4 Understanding Job Semantics	34
2.2.5 Distributed Messaging and Failover	35
2.2.6 Web-Based User Interaction	36
2.3 JDL Job Scripts	37
2.3.1 Principles of Job Operation	38
2.4 Understanding TLS Encryption	39
2.5 Understanding Job Examples	40
2.5.1 provisionBuildTestResource.job	40
2.5.2 Workflow Job Example	42
3 The ZENworks Orchestrator Datagrid	43
3.1 Defining the Datagrid	43
3.1.1 Naming Orchestrator Job Files	43
3.1.2 Distributing Files	44
3.1.3 Simultaneous Multicasting to Multiple Receivers	45
3.1.4 Orchestrator Datagrid Commands	45
3.2 Datagrid Communications	45
3.2.1 Multicast Example	46
3.2.2 Grid Performance Factors	46
3.2.3 Plan for Datagrid Expansion	47
3.3 datagrid.copy Example	47

4	Orchestrator Job Classifications	49
4.1	Resource Discovery	49
4.1.1	Provisioning Jobs	50
4.1.2	Resource Targeting	50
4.1.3	Resource Discovery Jobs	50
4.2	Dynamic Scheduling	51
4.3	Workload Management	52
4.4	Policy Management	53
4.5	Auditing and Accounting Jobs	55
5	Developing Policies	57
5.1	Policy Elements	57
5.1.1	Constraints	57
5.1.2	Facts	57
5.1.3	Computed Facts	58
5.2	BuildTest Job Examples	59
5.2.1	buildTest.policy Example	60
5.2.2	buildTest.jdl Example	61
5.2.3	Packaging Job Files	64
5.2.4	Deploying Packaged Job Files	64
5.2.5	Running Your Jobs	64
5.2.6	Monitoring Job Results	65
5.2.7	Debugging Jobs	66
6	Using the Orchestrator Client SDK	69
6.1	SDK Requirements	69
6.2	Creating an SDK Client	69
7	Job Architecture	71
7.1	Understanding JDL	71
7.2	JDL Package	72
7.2.1	.sched Files	72
7.3	Job Class	73
7.3.1	Job State Transition Events	73
7.3.2	Handling Custom Events	74
7.4	Job Invocation	75
7.5	Deploying Jobs	75
7.5.1	Using the Orchestrator Console	75
7.5.2	Using the ZOSADMIN Command Line Tool	75
7.6	Starting Orchestrator Jobs	76
7.7	Working with Facts and Constraints	77
7.7.1	Grid Objects and Facts	77
7.7.2	Defining Job Elements	77
7.7.3	Job Arguments and Parameter Lists	78
7.8	Using Facts in Job Scripts	79
7.9	Using Other Grid Objects	80
7.10	Communicating Through Job Events	80
7.10.1	Sending and Receiving Events	81
7.10.2	Synchronization	81
7.11	Executing Local Programs	81
7.11.1	Output Handling	82
7.11.2	Local Users	82

7.11.3	Safety and Failure Handling	83
7.12	Logging and Debugging	84
7.12.1	Creating a Job Memo	84
7.12.2	Tracing	85
7.13	Improving Job and Joblet Robustness	85
8	Job Scheduling	87
8.1	Job Scheduler GUI	87
8.2	Schedule Files	88
8.2.1	osInfo.sched Example	88
8.2.2	Cron Trigger Example	89
8.3	Scheduling with Constraints	89
9	Virtual Machine Job Development	91
9.1	VM Job Best Practices	91
9.1.1	Plan Robust Application Starts and Stops	91
9.1.2	Managing VM Systems	92
9.1.3	Managing VM Images	92
9.1.4	Managing VM Hypervisors	92
9.1.5	VM Job Considerations	92
9.2	Virtual Machine Management	92
9.3	VM Life Cycle Management	93
9.4	Manual Management of a VM Server's Lifecycle	94
9.4.1	Manually Using the ZOS Command Line	95
9.4.2	Automatically Using the Orchestrator Console Job Scheduler	95
9.4.3	Provision Job JDL	95
9.5	Provisioning Virtual Machines	96
9.5.1	Provisioning VMs Using Jobs	98
9.5.2	VM Placement Policy	99
9.5.3	Provisioning Example	100
9.6	Automatically Provisioning a VM Server	100
9.7	Defining Values for Grid Objects	101
9.7.1	Orchestrator Grid Objects	102
9.7.2	Repository Objects and Facts	102
9.7.3	VmHost Objects and Facts	109
9.7.4	VM Resource Objects and Other Base Resource Facts	113
9.7.5	Physical Resource Objects and Additional Facts	120
10	Complete Job Examples	123
10.1	Accessing Job Examples	123
10.2	Installation and Getting Started	123
10.3	Orchestrator Sample Job Summary	124
10.4	Parallel Computing Examples	125
demolterator.job		126
quickie.job		133
10.5	General Purpose Jobs	137
dgtest.job		138
failover.job		148
instclients.job		154
notepad.job		161
sweeper.job		166
whoami.job		173
10.6	Miscellaneous Code-Only Jobs	178

factJunction.job	179
jobargs.job	188

A Orchestrator Job Classes and JDL Syntax 197

A.1	Job Class	197
A.2	Joblet Class	197
A.3	Utility Classes	197
A.4	Built-in JDL Functions and Variables	197
	A.4.1 getMatrix()	198
	A.4.2 system(cmd)	198
	A.4.3 Grid Object TYPE_* Variables	198
	A.4.4 The __agent__ Variable	198
	A.4.5 The __jobname__ Variable	198
	A.4.6 The __mode__ Variable	199
A.5	Job State Field Values	199
A.6	Repository Information String Values	200
A.7	Joblet State Values	200
A.8	Resource Information Values	201
A.9	JDL Class Definitions	201
	AndConstraint()	203
	BinaryConstraint	204
	BuildSpec	205
	CharRange	206
	ComputedFact	207
	ComputedFactContext	208
	Constraint	209
	ContainerConstraint	210
	ContainsConstraint	211
	DataGrid	212
	DefinedConstraint	213
	EqConstraint	214
	Exec	215
	ExecError	216
	FileRange	217
	GeConstraint	218
	GridObjectInfo	219
	GroupInfo	220
	GtConstraint	221
	Job	222
	JobInfo	223
	Joblet	224
	JobletInfo	225
	JobletParameterSpace	226
	LeConstraint	227
	LtConstraint	228
	MatrixInfo	229
	NeConstraint	230
	NotConstraint	231
	OrConstraint	232
	ParameterSpace	233
	PolicyInfo	234
	ProvisionSpec	235
	RepositoryInfo	236
	ResourceInfo	237
	RunJobSpec	238
	ScheduleSpec	239

Timer	240
UndefinedConstraint	241
UserInfo	242
VMHostInfo	243
VmSpec	244

B Orchestrator Client SDK 245

B.1	Constraint Package	245
B.1.1	AndConstraint	245
B.1.2	BetweenConstraint	246
B.1.3	BinaryConstraint	246
B.1.4	Constraint	246
B.1.5	ContainerConstraint	246
B.1.6	ContainsConstraint	246
B.1.7	ConstraintException	246
B.1.8	DefinedConstraint	247
B.1.9	EqConstraint	247
B.1.10	GeConstraint	247
B.1.11	GtConstraint	247
B.1.12	IfConstraint	247
B.1.13	LeConstraint	247
B.1.14	LtConstraint	247
B.1.15	NeConstraint	248
B.1.16	NotConstraint	248
B.1.17	OperatorConstraint	248
B.1.18	OrConstraint	248
B.1.19	TypedConstraint	248
B.1.20	UndefinedConstraint	248
B.2	Datagrid Package	249
B.2.1	DGLogger	249
B.2.2	GridFile	249
B.2.3	GridFileFilter	249
B.2.4	GridFileNameFilter	249
B.2.5	DataGridException	249
B.2.6	DataGridNotAvailableException	250
B.2.7	GridFile.CancelException	250
B.3	Grid Package	250
B.3.1	AgentListener	251
B.3.2	ClientAgent	251
B.3.3	Credential	251
B.3.4	Fact	252
B.3.5	FactSet	252
B.3.6	GridObjectInfo	252
B.3.7	ID	252
B.3.8	JobInfo	252
B.3.9	Message	252
B.3.10	Message.Ack	252
B.3.11	Message.AuthFailure	253
B.3.12	Message.ClientResponse	253
B.3.13	Message.Event	253
B.3.14	Message.GetGridObjects	253
B.3.15	Message.GridObjects	253
B.3.16	Message.JobAccepted	253
B.3.17	Message.JobError	253
B.3.18	Message.JobFinished	254
B.3.19	Message.JobIdEvent	254
B.3.20	Message.JobInfo	254

B.3.21	Message.Jobs	254
B.3.22	Message.JobStarted	254
B.3.23	Message.JobStatus	254
B.3.24	Message.LoginFailed	254
B.3.25	Message.LoginSuccess	255
B.3.26	Message.LogoutAck	255
B.3.27	Message.NeedUpgrade	255
B.3.28	Message.RunningJobs	255
B.3.29	Message.ServerStatus	255
B.3.30	Message.SessionAck	255
B.3.31	Message.SessionChallenge	255
B.3.32	Message.SessionResponse	256
B.3.33	Message.SessionStatus	256
B.3.34	Node	256
B.3.35	Priority	256
B.3.36	WorkflowInfo	256
B.3.37	ClientOutOfDateException	256
B.3.38	FactException	256
B.3.39	GridAuthenticationException	257
B.3.40	GridAuthorizationException	257
B.3.41	GridConfigurationException	257
B.3.42	GridDeploymentException	257
B.3.43	GridException	257
B.3.44	GridObjectNotFoundException	257
B.4	Toolkit Package	258
B.4.1	ClientAgentFactory	258
B.4.2	ConstraintFactory	258
B.4.3	CredentialFactory	258

C Documentation Updates 259

C.1	Aug 3, 2008	259
C.1.1	Documentation Updates	259

About This Guide

The Novell® ZENworks Orchestrator Job Development Guide is a component of the Novell ZENworks Orchestrator documentation library. While Orchestrator 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 policies that form the basis of Orchestrator functionality. This guide provides developer information to create and run custom Orchestrator jobs. It also helps provides the basis to build, debug, and maintain policies using Orchestrator.

This guide contains the following sections:

- ◆ Chapter 1, “Getting Started,” on page 15
- ◆ Chapter 2, “Job Development Concepts,” on page 19
- ◆ Chapter 3, “The ZENworks Orchestrator Datagrid,” on page 43
- ◆ Chapter 4, “Orchestrator Job Classifications,” on page 49
- ◆ Chapter 5, “Developing Policies,” on page 57
- ◆ Chapter 6, “Using the Orchestrator Client SDK,” on page 69
- ◆ Chapter 7, “Job Architecture,” on page 71
- ◆ Chapter 8, “Job Scheduling,” on page 87
- ◆ Chapter 9, “Virtual Machine Job Development,” on page 91
- ◆ Chapter 10, “Complete Job Examples,” on page 123
- ◆ Appendix A, “Orchestrator Job Classes and JDL Syntax,” on page 197
- ◆ Appendix B, “Orchestrator Client SDK,” on page 245

Audience

This guide is intended for use by application developers and technically advanced datacenter technicians assigned to write Job Description Language (JDL) jobs to manage all resources in a Orchestrator-enabled environment. It assumes that users have the following background:

- ◆ Thorough understanding of concepts related to Novell ZENworks Orchestrator.
- ◆ Experience with the Python programming language.
- ◆ General understanding of network, operating environments, and systems architecture.
- ◆ Knowledge of basic UNIX* shell commands, Windows* command line tools, and text editors.
- ◆ An understanding of parallel computing and applications running on grid network infrastructures.

Documentation Updates

For the most recent version of this *Installation and Getting Started Guide*, visit the [ZENworks Orchestrator 1.3 Web site \(http://www.novell.com/documentation/zen_orchestrator13/\)](http://www.novell.com/documentation/zen_orchestrator13/).

Additional Documentation

For additional documentation that might assist you in developing Orchestrator jobs, see the following guides:

- ♦ *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*
- ♦ *Novell ZENworks Orchestrator 1.3 Job Management Guide*
- ♦ *Novell ZENworks Orchestrator 1.3 Virtual Machine Management Guide*

Documentation Conventions

In Novell documentation, a greater-than symbol (>) is used to separate actions within a step and items in a cross-reference path.

A trademark symbol (® ,™, etc.) denotes a Novell trademark. An asterisk (*) denotes a third-party trademark.

When a single pathname can be written with a backslash for some platforms or a forward slash for other platforms, the pathname is presented with a backslash. Users of platforms that require a forward slash, such as Linux or UNIX, should use forward slashes as required by your software.

Other typographical conventions used in this guide include the following:

Convention	Description
Italics	Indicates variables, new terms and concepts, and book titles. For example, a job is a piece of work that describes how an application can be run in Grid Management on multiple computers.
Boldface	Used for advisory terms such as Note, Tip, Important, Caution, and Warning.
Keycaps	Used to indicate keys on the keyboard that you press to implement an action. If you must press two or more keys simultaneously, keycaps are joined with a hyphen. For example, Ctrl-C. Indicates that you must press two or more keys to implement an action. Simultaneous keystrokes (in which you press down the first key while you type the second character) are joined with a hyphen; for example, press Stop-a. Consecutive keystrokes (in which you press down the first key, then type the second character) are joined with a plus sign; for example, press F4+q.

Convention	Description
Fixed-width	<p>Used to indicate various types of items. These include:</p> <p>Commands that you enter directly, code examples, user type-ins in body text, and options. For example,</p> <pre>cd mydir</pre> <pre>System.out.println("Hello World");</pre> <p>Enter abc123 in the Password box, then click Next.</p> <pre>-keep option</pre> <p>Jobs and policy keywords and identifiers. For example,</p> <pre><run></pre> <pre></run></pre> <p>File and directory names. For example,</p> <pre>/usr/local/bin</pre> <p>Note: UNIX path names are used throughout and are indicated with a forward slash (/). If you are using the Windows platform, substitute backslashes (\) for the forward slashes (/).</p>
Fixed-width italic and <Fixed-width italic>	<p>Indicates variables in commands and code. For example,</p> <pre>zos login <servername> [--user=] [--passwd=] [--port=]</pre> <p>Note: Angle brackets (< >) are used to indicate variables in directory paths and command options.</p>
(pipe)	<p>Used as a separator in menu commands that you select in a graphical user interface (GUI), and to separate choices in a syntax line. For example,</p> <pre>File New</pre> <pre>{a b c}</pre> <pre>[a b c]</pre>
{ } (braces)	<p>Indicates a set of required choices in a syntax line. For example,</p> <pre>{a b c}</pre> <p>means you must choose a, b, or c.</p>
[] (brackets)	<p>Indicates optional items in a syntax line. For example,</p> <pre>[a b c]</pre> <p>means you can choose a, b, c, or nothing.</p>
< > (angle brackets)	<p>Used for XML content elements and tags, and to indicate variables in directory paths and command options. For example,</p> <pre><template></pre> <pre><DIR></pre> <pre>-class <class></pre>

Convention	Description
. . . (horizontal ellipses)	Used to indicate that portions of a code example have been omitted to simplify the discussion, and to indicate that an argument can be repeated several times in a command line. For example, zosadmin [options]optfile.xmlc [...] docfile
plain text	Used for URLs, generic references to objects, and all items that do not require special typography. For example, http://www.novell.com/documentation/index.html The presentation object is in the presentation layer.
ALL CAPS	Used for SQL statements and HTML elements. For example, CREATE statement <INPUT>
lowercase	Used for XML elements. For example, <onevent> Note: XML is case-sensitive. If an existing XML element uses mixed-case or uppercase, it is shown in that case. Otherwise, XML elements are in lowercase.
ZENworks Orchestrator Server root directory	Where the ZENworks Orchestrator Server is installed. The Orchestrator executables and libraries are in a directory. This directory is referred to as the ZENworks Orchestrator Server root directory or <ZENworks Orchestrator Server_root>.
Paths	UNIX path names are used throughout and are indicated with a forward slash (/). If you are using the Windows platform, substitute backslashes (\) for the forward slashes (/). For example, UNIX: /usr/local/bin Windows: \usr\local\bin
URLs	URLs are indicated in plain text and are generally fully qualified. For example, http://www.novell.com/documentation/index.html
Screen shots	Most screen shots reflect the Microsoft Windows look and feel.

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.

Novell Support

Novell offers a support program designed to assist with technical support and consulting needs. The Novell support team can help with installing and using the Novell product, developing and debugging code, maintaining the deployed applications, providing onsite consulting services, and delivering enterprise-level support.

Getting Started

1

This *Developer Guide* for Novell™ ZENworks™ Orchestrator 1.3 provides information on how to assemble, deploy, and manage grid applications—called “jobs”—on the ZENworks Orchestrator Server. This guide also explains how to build, debug, and maintain policies that manage jobs running on the Orchestrator Server.

This section includes the following information:

- ♦ [Section 1.1, “What You Should Know,” on page 15](#)
- ♦ [Section 1.2, “Orchestrator Documentation Set,” on page 16](#)
- ♦ [Section 1.3, “Prerequisites for the Development Environment,” on page 17](#)

1.1 What You Should Know

This section includes the following information:

- ♦ [Section 1.1.1, “Prerequisite Knowledge,” on page 15](#)
- ♦ [Section 1.1.2, “Setting Up Your Development Environment,” on page 16](#)

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 \(http://pydev.sourceforge.net/\)](http://pydev.sourceforge.net/) enables developers to use Eclipse* for Python and Jython development. The plug-in makes Eclipse a more robust Python IDE and comes with tools such as code completion, syntax highlighting, syntax analysis, refactor, debug and many others.
 - ♦ **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, \(http://python.org/doc/2.1/lib/lib.html\)](http://python.org/doc/2.1/lib/lib.html) which is distributed with the language and assists in development.
 - ♦ **Python Tutorial:** This [online tutorial \(http://python.org/doc/2.1/ref/ref.html\)](http://python.org/doc/2.1/ref/ref.html) helps developers get started with Python.
 - ♦ **Extending and Embedding the Python Interpreter:** This [resource \(http://python.org/doc/2.1/ext/ext.html\)](http://python.org/doc/2.1/ext/ext.html) describes how to add new extensions to Python and how to embed it in other applications.
- ♦ Sound understanding of the ZENworks Orchestrator Job Development Language (JDL).
JDL integrates compact Python scripts to create jobs to manage nearly every aspect of the Orchestrator grid. For more information, see [Appendix A, “Orchestrator Job Classes and JDL Syntax,” on page 197](#).

- ◆ 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 ZENworks Orchestrator infrastructure.
- ◆ Familiarity with on-line ZENworks Orchestrator API Javadoc as you build custom client applications. For more information see [Appendix B, “Orchestrator Client SDK,” on page 245](#).
- ◆ Developer must assume both Orchestrator 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 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 [“Independent Installation of the Agent and Clients”](#)).
- 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:
 - ◆ A server deployed on one computer.
 - ◆ An agent installed on every managed server.
 - ◆ A client installed on your client machine.

From your client machine you can build jobs/policies, and then remotely deploy them using *zosadmin Command Line* tool. You can then remotely modify the jobs and other grid object through the Orchestrator 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 at make time. This will help you keep your job versions organized.
- 5 Create make or Ant controls for bundling and deploying your jobs.
- 6 After you are familiar with more simple jobs, you will want to move to more complex “meta jobs,” which enable you to tie together several applications in a service. These jobs will include startup dependencies and other logic to enable one application job code with different parameters and policies to be used in more than one service.
- 7 To enhance the robustness of your jobs, we suggest you deploy all jobs to more than one site, with different policies activated at each site.
- 8 Ideally, to leverage the flexibility of the Orchestrator environment, you should not have to write jobs targeted specifically for one container technology (Xen, VMWare, etc.).

1.2 Orchestrator Documentation Set

Before developing, deploying, and managing the Orchestrator jobs explained in this document, you should have a thorough understanding of how to deploy and manage all product components. These administrative tasks are explained in the following documents in both PDF and HTML formats:

- ◆ [Section 1.2.1, “Novell ZENworks Orchestrator Getting Started Guide,” on page 17](#)
- ◆ [Section 1.2.2, “Novell ZENworks Orchestrator Administration Guide,” on page 17](#)

- ♦ [Section 1.2.3, “Novell ZENworks Orchestrator Virtual Machine Management Guide,” on page 17](#)
- ♦ [Section 1.2.4, “Novell ZENworks Orchestrator Job Management Guide,” on page 17](#)

1.2.1 Novell ZENworks Orchestrator Getting Started Guide

The *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide* introduces ZENworks Orchestrator 1.3, including its basic administration environment, which is accessed either through the Orchestrator console or the zos command line. It provides an introductory overview of Orchestrator components and explains how to install, monitor, and manage applications running on the Orchestrator. The guide provides basic startup and management instructions for system administrators.

1.2.2 Novell ZENworks Orchestrator Administration Guide

The *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide* provides basic information on how to deploy and manage specific, basic jobs on the ZENworks Orchestrator Grid Management Server using the resources available in the data center. Job managers typically have limited rights and responsibilities and are not expected to know the intricacies of the Orchestrator or to understand how to create the jobs themselves.

1.2.3 Novell ZENworks Orchestrator Virtual Machine Management Guide

The *Novell ZENworks Orchestrator 1.3 Virtual Machine Management Guide* introduces Novell ZENworks Virtual Machine Management, including its basic administration environment, which is accessed through an Eclipse interface console, and its interface for developers, which is accessed through the developer portal. The guide provides an introductory overview of virtual machine management (VMM), explains how to install, monitor, and manage VMs and coordinate their management with applications running on the Orchestrator.

1.2.4 Novell ZENworks Orchestrator Job Management Guide

The *Novell ZENworks Orchestrator 1.3 Job Management Guide* provides in-depth information on the ZENworks Orchestrator Console, a thin-client browser console, and the ZENworks Orchestrator user grid command-line management and deployment tool. It is anticipated that developers will typically develop jobs using the zos command line tool, while higher level system administrators will use the ZENworks GUI-based interfaces to submit, deploy and run jobs and manage network resources.

1.3 Prerequisites for the Development Environment

- ♦ Install the [Java* Development Kit \(https://sdlc3d.sun.com/ECom/EComActionServlet;jsessionid=DCA955A842E56492B469230CC680B2E1\)](https://sdlc3d.sun.com/ECom/EComActionServlet;jsessionid=DCA955A842E56492B469230CC680B2E1), version 1.5 or later, to create jobs and to compile a Java SDK client in the Orchestrator environment. The Orchestrator installer ships with a Java Runtime Environment (JRE) suitable for running Orchestrator jobs.

- ◆ **Components to write Python-based Job Description Language (JDL) scripts:**
 - ◆ **Eclipse version 3.2.1 or later.** (<http://www.eclipse.org/>) is the interface console accessed through the developer portal.
- ◆ **Product License:** To expose full product functionality and access to all job examples, you should be provisioned with a 90-day trial license SKU. If not, you need to add additional example jobs to customize your setup profile to mirror a fully enabled Novell ZENworks Server. If you need to customize setup profile, go to “**Tested Platforms and Installation Methods for the Orchestrator Server, Agent, and Clients**” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.
- ◆ **Development Environment:** Set up your environment according to the guidelines outlined in “**Planning the Installation.**” In general, the deployed Orchestrator Server requires 2 (minimum for 100 or fewer managed resources) to 4 gigabytes (recommended for more than 100 managed resources) of RAM. By default, the Orchestrator Server is configured to use 1 gigabyte of memory.
- ◆ **Network Capabilities:** For Virtual Machine Management, you need a high-speed Gigabit Ethernet. For more information about network requirements, see “**Tested Platforms and Installation Methods for Virtual Machine Management Components**”.
- ◆ **Initial Configuration:** After you install and configure Orchestrator, start in the auto registration mode as described in “**First Use of Basic ZENworks Orchestrator Components.**” 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, be certain that promiscuous mode is deactivated.

Job Development Concepts

2

This document is written for individuals who assume the role of Novell® ZENworks® Orchestrator job developers, rather than network administrators or system technicians who later deploy the jobs that developers create. This document discusses the tools and technology required to create discrete programming scripts—called “jobs”—that control nearly every aspect of the Orchestrator product.

As a job developer, you need your own self-contained, standalone system with full access to your network environment. At one point or another, you will assume all system roles: job creator, job deployer, system administrator, tester, etc. For more information about jobs, see “Jobs” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.

This section provides conceptual information to help you create your own Novell ZENworks Orchestrator jobs:

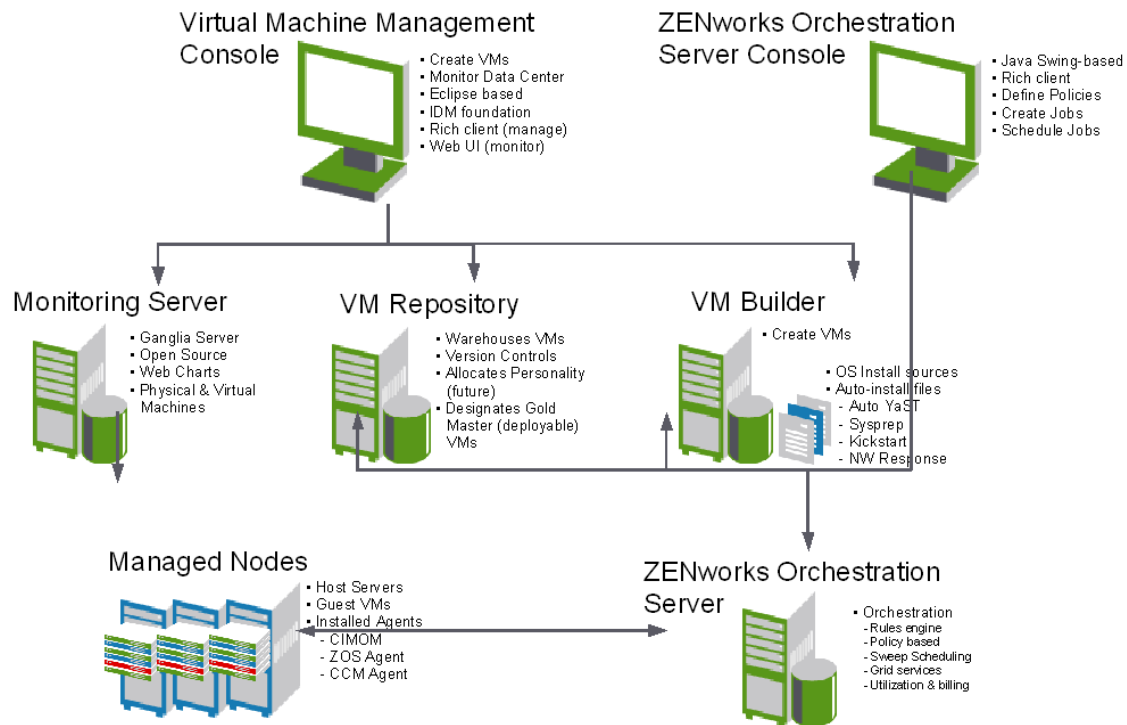
- ♦ [Section 2.1, “Orchestrator Development Architecture,” on page 19](#)
- ♦ [Section 2.2, “Understanding ZENworks Orchestrator Functionality,” on page 30](#)
- ♦ [Section 2.3, “JDL Job Scripts,” on page 37](#)
- ♦ [Section 2.4, “Understanding TLS Encryption,” on page 39](#)
- ♦ [Section 2.5, “Understanding Job Examples,” on page 40](#)

2.1 Orchestrator Development Architecture

Novell ZENworks Orchestrator is an advanced datacenter management solution designed to manage all network resources. It provides the infrastructure that manages a group of ten, one hundred, or thousands of physical or virtual resources.

Orchestrator is equally apt at performing a number of distributed processing problems. From high performance computing, the breaking down of work into lots of small chunks that can be processed in parallel through distributed job scheduling. The following figure shows the product’s high-level architecture:

Figure 2-1 ZENworks Orchestrator Architecture



This section contains information about the following topics:

- ◆ [Section 2.1.1, “Orchestrator Agents,” on page 20](#)
- ◆ [Section 2.1.2, “Orchestrator Resource Monitor,” on page 21](#)
- ◆ [Section 2.1.3, “Orchestrator Entity Types and Managers,” on page 21](#)
- ◆ [Section 2.1.4, “Jobs,” on page 24](#)
- ◆ [Section 2.1.5, “Constraint-Based Job Scheduling,” on page 27](#)
- ◆ [Section 2.1.6, “Understanding Orchestrator API Interfaces,” on page 28](#)

2.1.1 Orchestrator Agents

Agents are installed on all managed resources as part of the product deployment. For more detailed information about these components, see “[Software Architecture](#)” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.

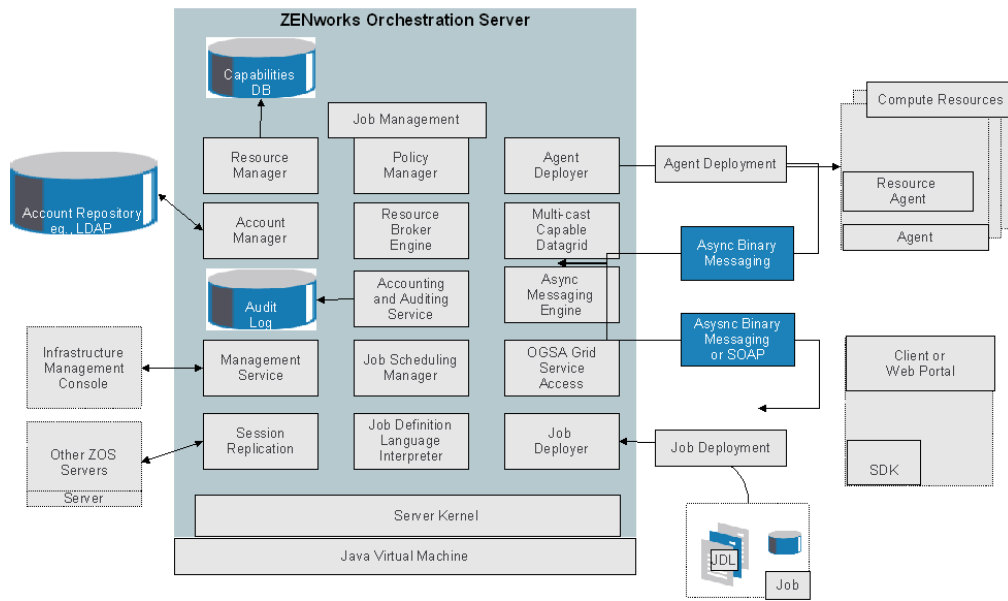
The agent connects every managed resource to its configured server and advertises to the ZENworks Orchestrator Server that the resource is available for tasks. This persistent and auto-reestablishing connection is important because it provides a message bus for the distribution of work, collection of information about the resource, per-job messaging, health checks, and resource failover control.

After resources are enabled, Orchestrator can discover, access, and store detailed abstracted information—called “facts”—about every resource. Managed resources, referred to as “nodes,” are addressable members of the of the Orchestrator Server “grid” (also sometimes called the “matrix”).

When integrated into the grid, nodes can be deployed, monitored, and managed by the Orchestrator Server, as discussed in [Section 2.2, “Understanding ZENworks Orchestrator Functionality,”](#) on [page 30](#).

An overview of the Orchestrator grid architecture is illustrated in the figure below, much of which is explained in this developer’s guide:

Figure 2-2 Orchestrator Server Architecture



For additional information about job architecture, see [Chapter 7, “Job Architecture,”](#) on [page 71](#).

2.1.2 Orchestrator Resource Monitor

ZENworks Orchestrator enables you to monitor your system computing resources using the built-in Resource Monitor. To open the Resource Monitor in the console, see “Monitoring Server Resources” in the *ZENworks Orchestrator Administration Guide*.

2.1.3 Orchestrator Entity Types and Managers

The following entities are some of key components involved in the Orchestrator Server:

- ♦ “Resources” on [page 22](#)
- ♦ “Users” on [page 22](#)
- ♦ “Job Definitions” on [page 22](#)
- ♦ “Job Instances” on [page 22](#)
- ♦ “Policies” on [page 22](#)
- ♦ “Facts” on [page 23](#)
- ♦ “Constraints” on [page 23](#)
- ♦ “Groups” on [page 24](#)

- ◆ [“VM: Hosts, Images, and Instances” on page 24](#)
- ◆ [“Templates” on page 24](#)

Resources

All managed resources, which are called nodes, have an agent with a socket connection to the Orchestrator Server. All resource use is metered, controlled, and audited by the Orchestrator Server. Policies govern the use of resources.

Orchestrator allocates resources by reacting as load is increased on a resource. As soon as we go above a threshold that was set in a policy, a new resource is allocated and consequently the load on that resource drops to an acceptable rate.

You can also write and jobs that perform cost accounting to account for the cost of a resource up through the job hierarchy, periodically, about every 20 seconds. For more information, see [Section 4.5, “Auditing and Accounting Jobs,” on page 55](#).

A collection of jobs, all under the same hierarchy, can cooperate with each other so that when one job offers to give up a resource it is reallocated to another similar priority job. Similarly, when a higher priority job becomes overloaded and is waiting on a resource, the system “steals” a resource from a lower priority job, thus increasing load on the low priority job and allocating it to the higher priority job. This process satisfies the policy, which specifies that a higher priority job must complete at the expense of a low priority job.

Users

Orchestrator users must authenticate to access the system. Access and use of system resources are governed by policies.

Job Definitions

A job definition is described in the embedded enhanced Python script that you create as a job developer. Each job instance runs a job that is defined by the Job Definition Language (JDL). Job definitions might also contain usage policies. For more information, see the [Job \(page 222\)](#) class.

Job Instances

Jobs are instantiated at runtime from job definitions that inherit policies from the entire context of the job (such as users, job definitions, resources, or groups). For more information, see [JobInfo \(page 223\)](#).

Policies

Policies are XML documents that contain various **constraints** and static **fact** assignments that govern how jobs run in the Orchestrator environment.

Policies are used to enforce quotas, job queuing, resource restrictions, permissions, and other job parameters. Policies can be associated with any Orchestrator object. For more information, see [Section 2.2.2, “Policy-Based Management,” on page 31](#).

Facts

Facts represent the state of any object in the Orchestrator grid. They can be discovered through a job or they can be explicitly set.

Facts control the behavior a job (or joblet) when it's executing. Facts also detect and return information about that job in various UIs and server functions. For example, a job description that is set through its policy and has a specified value might do absolutely nothing except return immediately after network latency.

There are three basic types of facts:

- ♦ **Static:** Facts that require you to set a value. For example, in a policy, you might set a value to be False. Static facts can be modified through policies.
- ♦ **Dynamic:** Facts produced by the Orchestrator system itself. Policies cannot override dynamic facts. They are read only and their value is determined by the orchestrator itself.
- ♦ **Computed:** Facts derived from a value, like that generated from the cell of a spreadsheet. Computed facts have some kind of logic behind them which derive their values.

For example, you might have two numeric facts that you want expressed in another fact as an average of the two. You could compose a computed fact which averages two other facts and express it as an average value under a certain fact name. This enables you to create facts that represent other metrics on the system that aren't necessarily available in the default set, or are not static to anything that might impact other dynamic facts.

For more information about facts, see [Section 5.1.2, "Facts," on page 57](#).

Constraints

In order for the Orchestrator to choose resources for a job, it uses resource constraints. A resource constraint is some Boolean logic that executes against facts in the system. Based upon this evaluation, it will only consider resources that match the criteria that have been set up by use of constraints.

For more detailed information, see [Section 7.7, "Working with Facts and Constraints," on page 77](#) and the following JDL constraint definitions:

- ♦ [AndConstraint\(\) \(page 203\)](#)
- ♦ [BinaryConstraint \(page 204\)](#)
- ♦ [Constraint \(page 209\)](#)
- ♦ [ContainerConstraint \(page 210\)](#)
- ♦ [ContainsConstraint \(page 211\)](#)
- ♦ [DefinedConstraint \(page 213\)](#)
- ♦ [EqConstraint \(page 214\)](#)
- ♦ [GeConstraint \(page 218\)](#)
- ♦ [GtConstraint \(page 221\)](#)
- ♦ [LeConstraint \(page 227\)](#)
- ♦ [LtConstraint \(page 228\)](#)
- ♦ [NeConstraint \(page 230\)](#)
- ♦ [NotConstraint \(page 231\)](#)

- ♦ [OrConstraint \(page 232\)](#)
- ♦ [UndefinedConstraint \(page 241\)](#)

Groups

Resources, users, job definitions and virtual machines (VM) are managed in groups with group policies that are inherited by members of the group.

VM: Hosts, Images, and Instances

A virtual machine host is a resource that is able to run guest operating systems. Attributes (facts) associated with the VM host control its limitations and functionality within the Orchestrator Server. A VM image is a resource image that can be cloned and/or provisioned. A VM instance represents a running copy of a VM image.

Templates

Templates are images that are meant to be cloned (copied) prior to provisioning the new copy. For more information, see “[Managing Virtual Machine Templates](#)” in the *Novell ZENworks Orchestrator 1.3 Virtual Machine Management Guide*.

2.1.4 Jobs

The Orchestrator Server manages all nodes by administering jobs (and the functional control of jobs at the resource level by using joblets), which control the properties (facts) associated with every resource. In other words, jobs are units of functionality that dispatch data center tasks to resources on the network such as management, migration, monitoring, load balancing, etc.

Orchestrator provides a unique job development, debugging, and deployment environment that expands with the demands of growing data centers.

As a job developer, your task is to develop jobs to perform a wide array of work that can be deployed and managed by ZENworks Orchestrator.

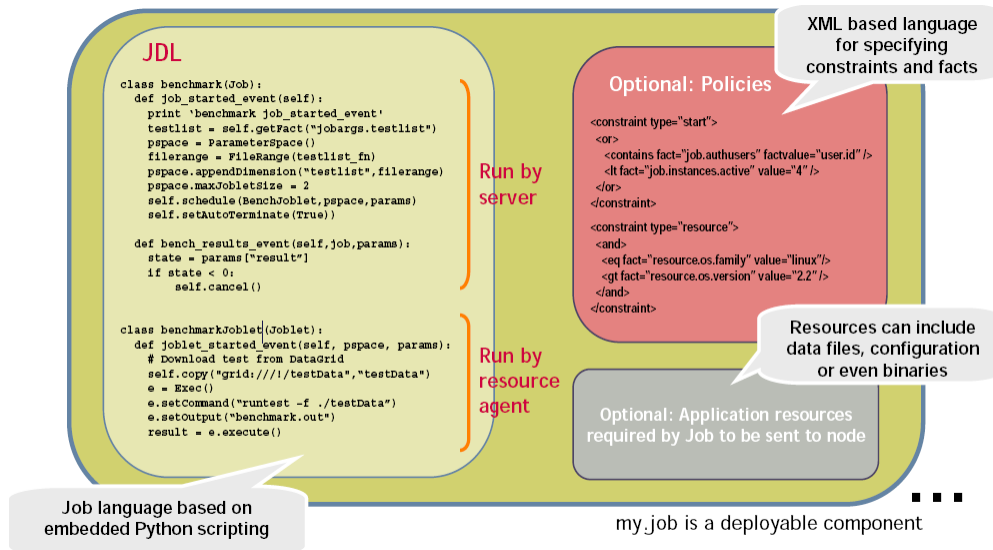
Jobs, which run on the Orchestrator server, can provide functions within the Orchestrator environment that might last from seconds to months. Job and joblet code exist in the same script file and are identified by the `.jdl` extension. The `.jdl` script contains only one job definition and zero or more joblet definitions. A `.jdl` script can have only one Job subclass. As for naming conventions, the Job subclass name does not have to match the `.jdl` filename; however, the `.jdl` filename is the defined job name, so the `.jdl` filename must match the `.job` filename that contains the `.jdl` script. For example, the job files (`demoIterator.jdl` and `demoIterator.policy`) included in the `demoIterator` example job are packaged into the archive file named `demoIterator.job`, so in this case, the name of the job is `demoIterator`.

A job file also might have policies associated with it to define and control the job’s behavior and to define certain constraints to restrict its execution. A `.jdl` script that is accompanied by a policy file is typically packaged in a job archive file (`.job`). Because a `.job` file is physically equivalent to a Java archive file (`.jar`), you can use the JDK JAR tool to create the job archive.

Multiple job archives can be delivered as a management pack in a service archive file (SAR) identified with the `.sar` extension. Typically, a group of related files are delivered this way. For example, the Xen30 management pack is a SAR.

As shown in the following illustration, jobs include all of the code, policy, and data elements necessary to execute specific, predetermined tasks administered either through the ZENworks Orchestrator console, or from the zos command line tool.

Figure 2-3 Components of an Orchestrator Job (*my.job*,)



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. Jobs provide the following functionality:

- ◆ “Controlling Process Flow” on page 25
- ◆ “Parallel Processing” on page 25
- ◆ “Managing the Cluster Life Cycle” on page 26
- ◆ “Discovery Jobs” on page 26
- ◆ “System Jobs” on page 26
- ◆ “Provisioning Jobs” on page 27

For more information, see [Chapter 4, “Orchestrator Job Classifications,”](#) on page 49 and the JDL job class definitions:

- ◆ [Job](#) (page 222)
- ◆ [JobInfo](#) (page 223)

Controlling Process Flow

Jobs can be written to control all operations and processes of managed resources. Through jobs, the Orchestrator Server manages resources to perform work. Automated jobs (written in JDL), are broken down into joblets, which are distributed among multiple resources.

Parallel Processing

By managing many small joblets, the Orchestrator server can enhance system performance and maximize resource use.

Managing the Cluster Life Cycle

Jobs can detect demand and monitor health of system resources, then modify clusters automatically to maximize system performance and provide failover services.

Discovery Jobs

Some jobs provide inspection of resources to more effectively management assets. These jobs enable all agents to periodically report basic resource facts and performance metrics. In essence, these metrics are stored as facts consisting of a key word and typed-value pairs like the following example:

```
resource.loadaverage=4.563, type=float
```

Jobs can poll resources and automatically trigger other jobs if resource performance values reach certain levels.

The system job scheduler is used to run resource discovery jobs to augment resource facts as demands change on resources. This can be done on a routine, scheduled basis or whenever new resources are provisioned, new software is installed, bandwidth changes occur, OS patches are deployed, or other events occur that might impact the system.

Consequently, resource facts form a capabilities database for the entire system. Jobs can be written that apply constraints to facts in policies, thus providing very granular control of all resources as required. All active resources are searchable and records are retained for all off-line resources.

The following `osInfo.job` example shows how a job sets operating system facts for specific resources:

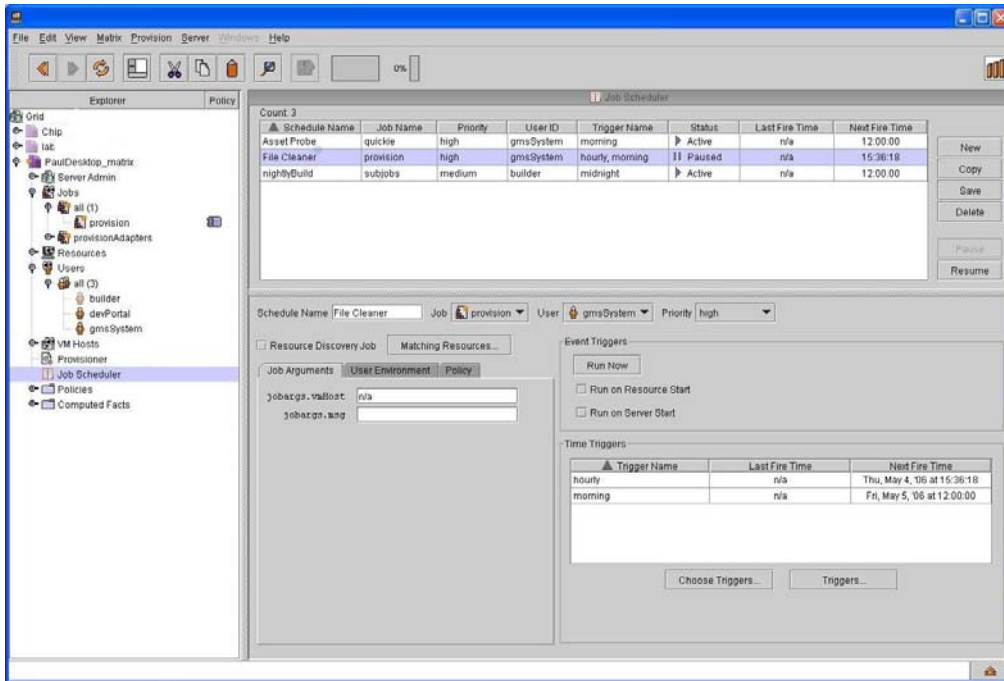
```
resource.cpu.mhz (integer) e.g., "800" (in Mhz)
resource.cpy.vendor (string) e.g. "GenuineIntel"
resource.cpu.model (string) e.g. "Pentium III"
resource.cpu.family (string) e.g. "i686"
```

`osInfo.job` is packaged as a single cross-platform job and includes the Python-based JDL and a policy to set the timeout. It is run each time a new resource appears and once every 24 hours to ensure validity of the resources. For a more detailed review of this example, see [osInfo.job \(page 51\)](#).

System Jobs

Jobs can be scheduled to periodically trigger specific system resources based on specific time constraints or events. As shown in the following figure, Orchestrator provides a built-in job scheduler that enables you or system administrators to flexibly deploy and run jobs.

Figure 2-4 Orchestrator Job Scheduler



For more information, see [Section 4.2, “Dynamic Scheduling,”](#) on page 51, [Chapter 8, “Job Scheduling,”](#) on page 87, and [Job](#) (page 222).

Provisioning Jobs

Jobs also drive provisioning for virtual machines and blade servers. Provisioning adapter jobs are deployed and organized into appropriate job groups for management convenience. Provisioning adapters are deployed as part of your VMM license.

For more information, see [Section 9.2, “Virtual Machine Management,”](#) on page 92 and [“Virtual Machine Technologies and Actions”](#) in the *Novell ZENworks Orchestrator 1.3 Virtual Machine Management Guide*.

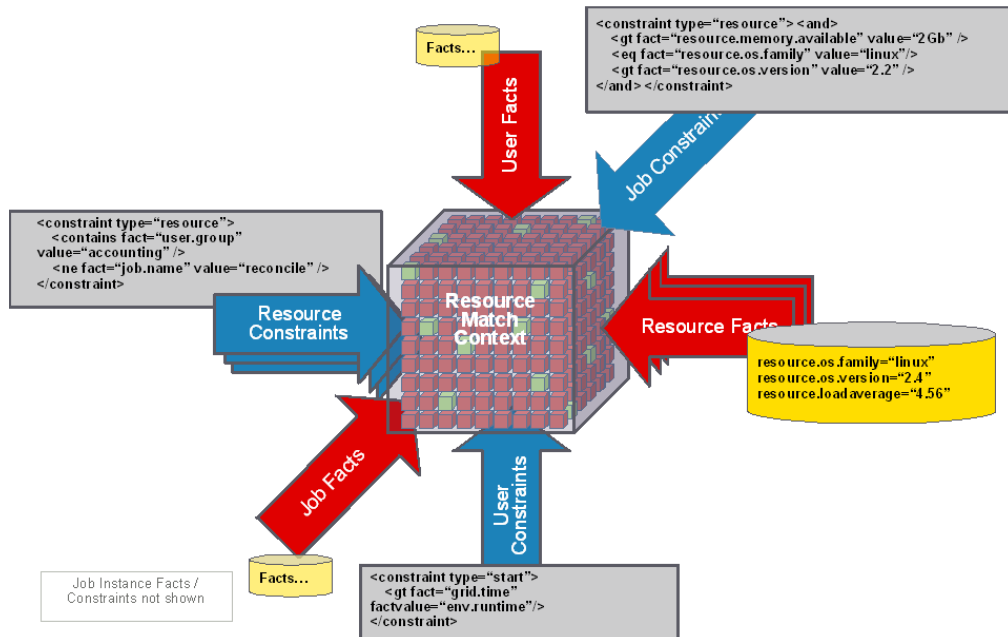
2.1.5 Constraint-Based Job Scheduling

The Orchestrator Server is a “broker” that can distribute jobs to every “partner” agent on the grid. Based on assigned policies, jobs have priorities and are executed based on the following contexts:

- ◆ User Constraints
- ◆ User Facts
- ◆ Job Constraints
- ◆ Job Facts
- ◆ Job Instance
- ◆ Resource User Constraints
- ◆ Resource Facts
- ◆ Groups

Each object in a job context contains the following elements:

Figure 2-5 Constraint-Based Resource Brokering



For more information, see [Section 7.7, “Working with Facts and Constraints,”](#) on page 77.

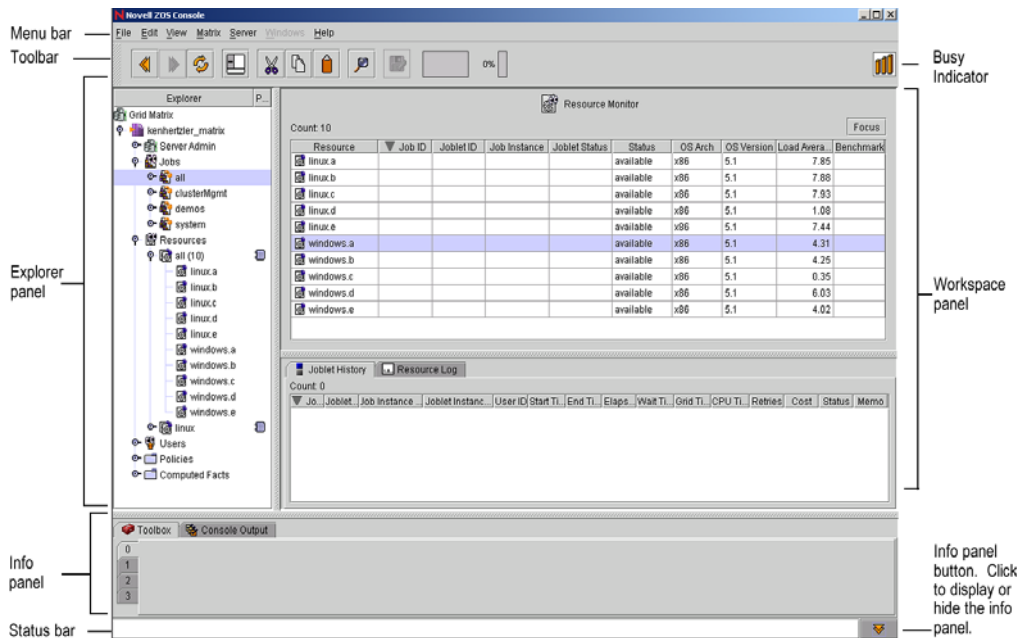
2.1.6 Understanding Orchestrator API Interfaces

There are three API interfaces available to the Orchestrator Server:

- ♦ **Orchestrator Server Management Interface:** The ZENworks Orchestrator server, written entirely in Java using the JMX (Java MBean) interface for management, leverages this API for the ZENworks Orchestrator Console. The console is a robust desktop GUI designed for administrators to apply, manage, and monitor usage-based policies on all infrastructure resources. The console also provides at-a-glance grid health and capacity checks.

For more information, see “Using the ZENworks Orchestrator Console” in the *ZENworks Orchestrator Administration Guide*.

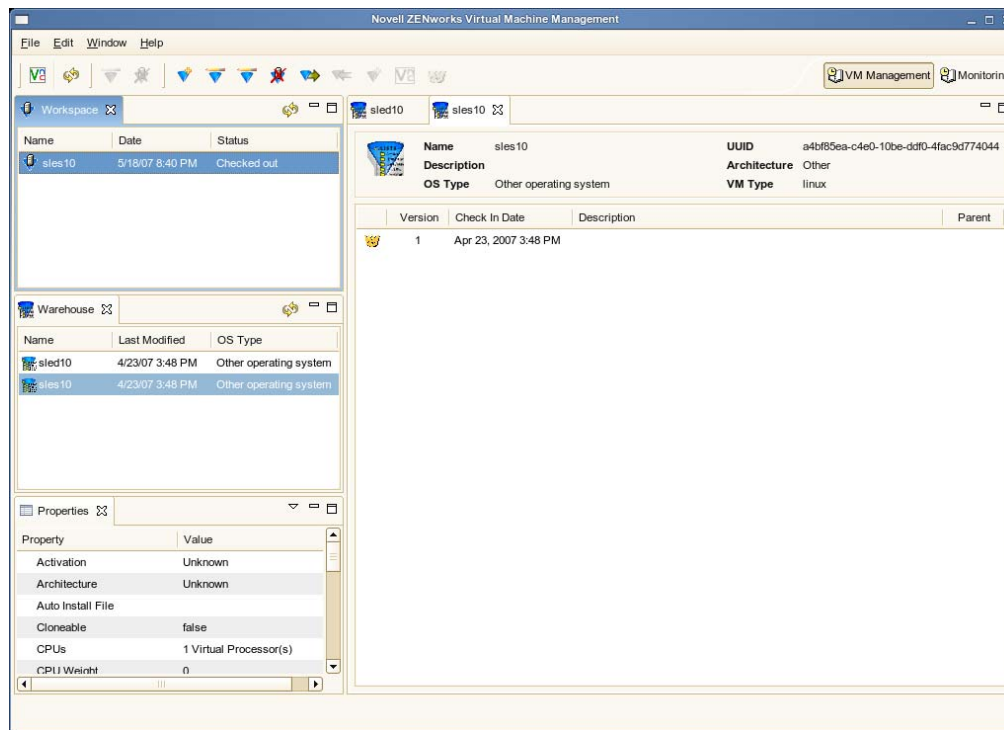
Figure 2-6 Novell ZENworks Orchestrator Console



- ♦ **Job Interface:** Includes a customizable/replaceable Web application and the zosadmin command line tool. The Web-based User Portal built with this API provides a universal job viewer from which job logs and progress can be monitored. The job interface is accessible via a Java API or CLI. A subset is also available as a Web Service. The default Orchestrator Developer Portal web application leverages this API. It can be customized or alternative J2EE* application can be written.
- ♦ **ZENworks Monitoring System:** Monitors all aspects of the data center through an open source, Eclipse*-based interface. This interface operates in conjunction with the Orchestrator Server and monitors the following objects:
 - ♦ Deployed jobs that teach Orchestrator and provide the control logic that Orchestrator runs when performing its management tasks.
 - ♦ Users and Groups
 - ♦ Virtual Machines

For more information, see the *Novell ZENworks Orchestrator 1.3 Virtual Machine Management Guide*.

Figure 2-7 Novell ZENworks Orchestrator Virtual Machine Manager interface.



2.2 Understanding ZENworks Orchestrator Functionality

- ♦ [Section 2.2.1, “Resource Virtualization,”](#) on page 30
- ♦ [Section 2.2.2, “Policy-Based Management,”](#) on page 31
- ♦ [Section 2.2.3, “Global Resource Visualization,”](#) on page 32
- ♦ [Section 2.2.4, “Understanding Job Semantics,”](#) on page 34
- ♦ [Section 2.2.5, “Distributed Messaging and Failover,”](#) on page 35
- ♦ [Section 2.2.6, “Web-Based User Interaction,”](#) on page 36

2.2.1 Resource Virtualization

Host machines or test targets managed by the Orchestrator Server form nodes on the grid (sometimes referred to as the matrix). All resources are virtualized for access by maintaining a capabilities database containing extensive information (facts) for each managed resource.

This information is automatically polled and obtained from each resource periodically or when it first comes online. The extent of the resource information the system can gather is customizable and highly extensible, controlled by the jobs you create and deploy.

The ZENworks Virtual Machine Builder is a service of VM Management that allows you to build a VM to precise specifications required for your data center. You designate the parameters required: processor, memory, hard drive space, operating system, virtualization type, if it’s based of an auto-

install file, and any additional parameters. When you launch the build job, VM Builder sends the build request to a machine that meets the hardware requirements of the defined VM and builds the VM there.

For more information, see “Building a Virtual Machine” in the *ZENworks Orchestrator Virtual Machine Management Guide* (http://www.novell.com/documentation/zen_orchestrator11/zos11_yvm/data/bookinfo.html).

2.2.2 Policy-Based Management

Policies are aggregations of facts and constraints that are used to enforce quotas, job queuing, resource restrictions, permissions, and other user and resource functions. Policies can be set on all objects and are inherited, which facilitates implementation within related resources.

Facts, which might be static, dynamic or computed for complex logic, are used when jobs or test scenarios require resources in order to select a resource that exactly matches the requirements of the test, and to control the access and assignment of resources to particular jobs, users, projects, etc. through policies. This abstraction keeps the infrastructure fluid and allows for easy resource substitution.

Of course, direct named access is also possible. An example of a policy that constrains the selection of a resource for a particular job or test is shown in the figure below. Although resource constraints can be applied at the policy level, they can also be described by the job itself or even dynamically composed at runtime.

Figure 2-8 Resource Selection Policy Example

```
<policy>
  <constraint type="resource">
    <and>
      <eq fact="resource.os.family" value="Linux" />
      <gt fact="resource.os.version" value="2.2" />
    </and>
  </constraint>
</policy>
```

An example of a policy that constrains the start of a job or test because too many tests are already in progress is shown in the following figure:

Figure 2-9 Job/Test Start Policy Example

```
<policy>
  <!-- Constrains the job to limit the number of running jobs to a
  defined value but exempt certain users from this limit. All jobs
  that attempt to exceed the limit are queued until the running jobs
  count decreases and the constraint passes. -->
  <constraint type="start" reason="Too busy">
    <or>
      <lt fact="job.instances.active" value="5" />
      <eq fact="user.name" value="canary" />
    </or>
  </constraint>
</policy>
```

See also:

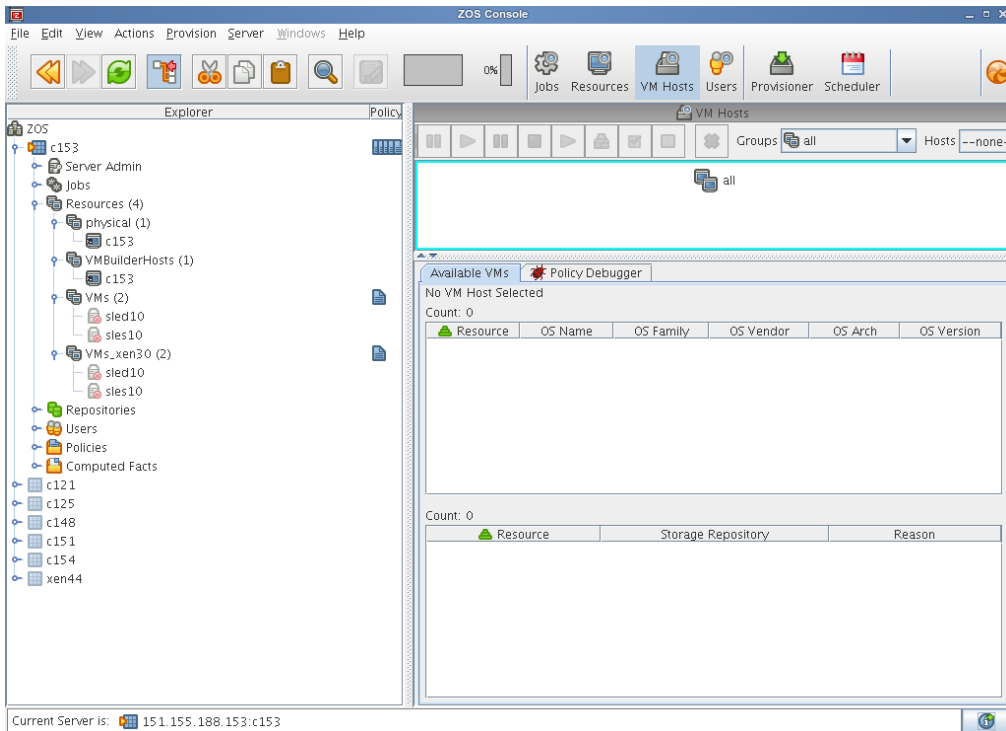
- ♦ [Chapter 5, “Developing Policies,” on page 57.](#)
- ♦ [Chapter 7, “Job Architecture,” on page 71.](#)

2.2.3 Global Resource Visualization

One of the greatest strengths of the Novell ZENworks Orchestrator solution is the ability to manage and visualize the entire grid. This is performed through the ZENworks Orchestrator Console and the ZENworks Monitoring System.

The desktop console is a Java application that has broad platform support and provides job, resource, and user views of activity as well as access to the historical audit database system, cost accounting, and other graphing features.

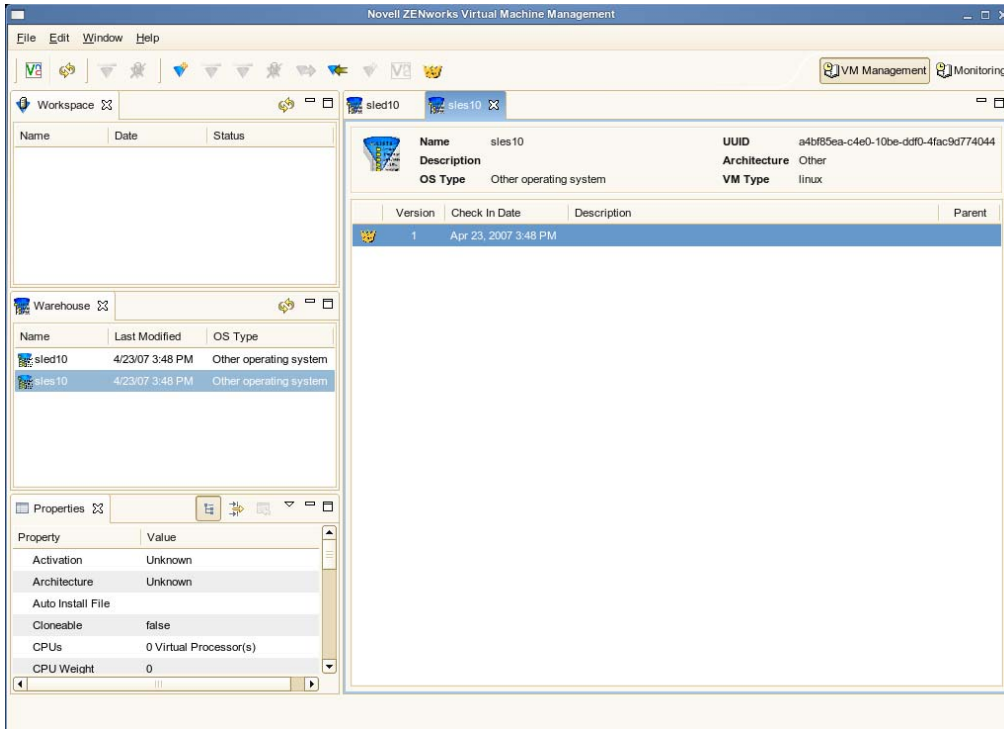
Figure 2-10 The ZENworks Orchestrator Console with Virtual Machine Management Elements



The console also applies policies that govern the use of shared infrastructure or simply create logical grouping of nodes on the grid. For more information about the ZENworks Orchestrator Console, see the *Novell ZENworks Orchestrator 1.3 Administration Guide*.

The ZENworks Monitoring System provides robust graphical monitoring of all managed virtual resources managed on the grid.

Figure 2-11 *The Eclipse ZENworks Monitoring System*

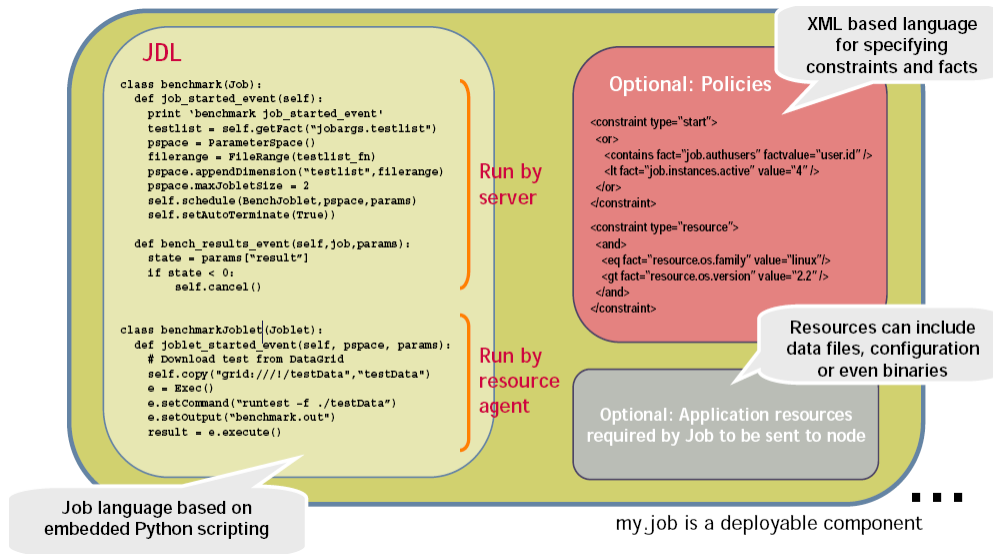


For more information, see the [Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide](#).

2.2.4 Understanding Job Semantics

As mentioned earlier, ZENworks Orchestrator runs jobs. A job is a container that can encapsulate several components including the Python-based logic for controlling the job life cycle (such as a test) through logic that accompanies any remote activity, task-related resources such as configuration files, binaries and any policies that should be associated with the job, as illustrated below.

Figure 2-12 Components of an Orchestrator Job



Workflows

Jobs can also invoke other jobs creating hierarchies. Because of the communication between the job client (either a user/user client application or another job) it is easy to create complex workflows composed of discrete and separately versioned components.

When a job is executed and an instance is created, the class that extends job is run on the server and as that logic requests resources, the class(es) that extend the joblet are automatically shipped to the requested resource to manage the remote task. The communication mechanism between these distributed components manifests itself as event method calls on the corresponding piece.

For more information, see [Section 2.5.2, “Workflow Job Example,” on page 42](#), [Section 7.3.1, “Job State Transition Events,” on page 73](#), and [Section 7.10, “Communicating Through Job Events,” on page 80](#).

2.2.5 Distributed Messaging and Failover

A job has control over all aspects of its failover semantics, which can be specified separately for conditions such as the loss of a resource, failure of an individual joblet, or joblet timeout.

The failover/health check mechanisms leverage the same communications mechanism that is available to job and joblet logic. Specifically, when a job is started and resources are employed, a message interface is established among all the components as shown in [Figure 2-13 on page 36](#).

Optionally, a communication channel can also be kept open to the initiating client. This client communication channel can be closed and reopened later based on jobid. Messages can be sent with the command

```
sendEvent(foo_event, params, ...)
```

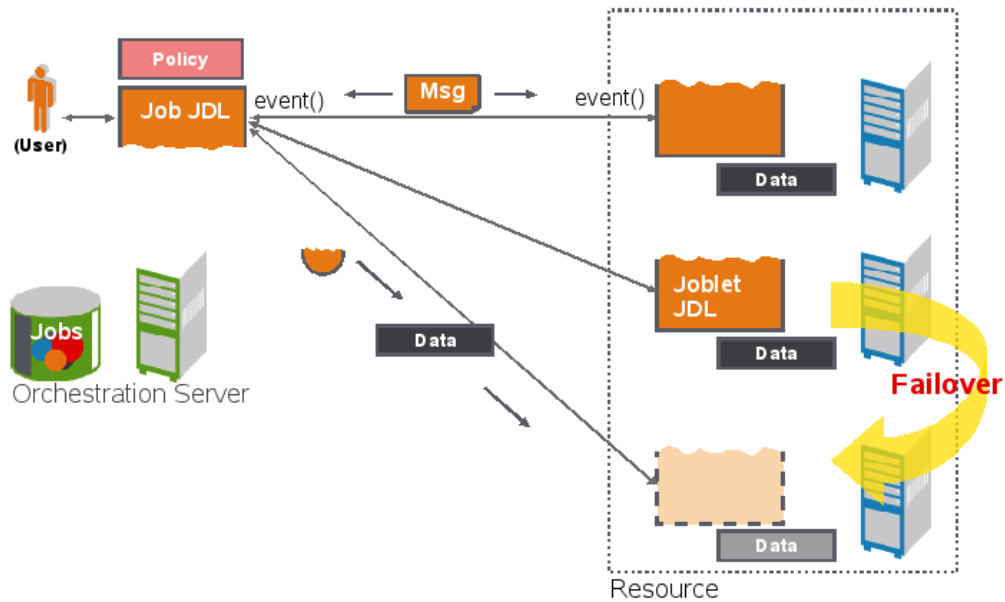
and received at the other end as a method invocation

```
def foo_event(self, params)
```

If a job allows it, a failure in any joblet causes the Orchestrator Server to automatically find an alternative resource, copy over the joblet JDL code, and reestablish the communication connection. A job also can listen for such conditions simply by defining a method for one of the internally generated events, such as `def joblet_failure_event(...)`.

Such failover allows, for example, for a large set of regression tests to be run (perhaps in parallel) and for a resource to die in the middle of the tests without the test run being rendered invalid. The figure below shows how job logic is distributed and failover achieved:

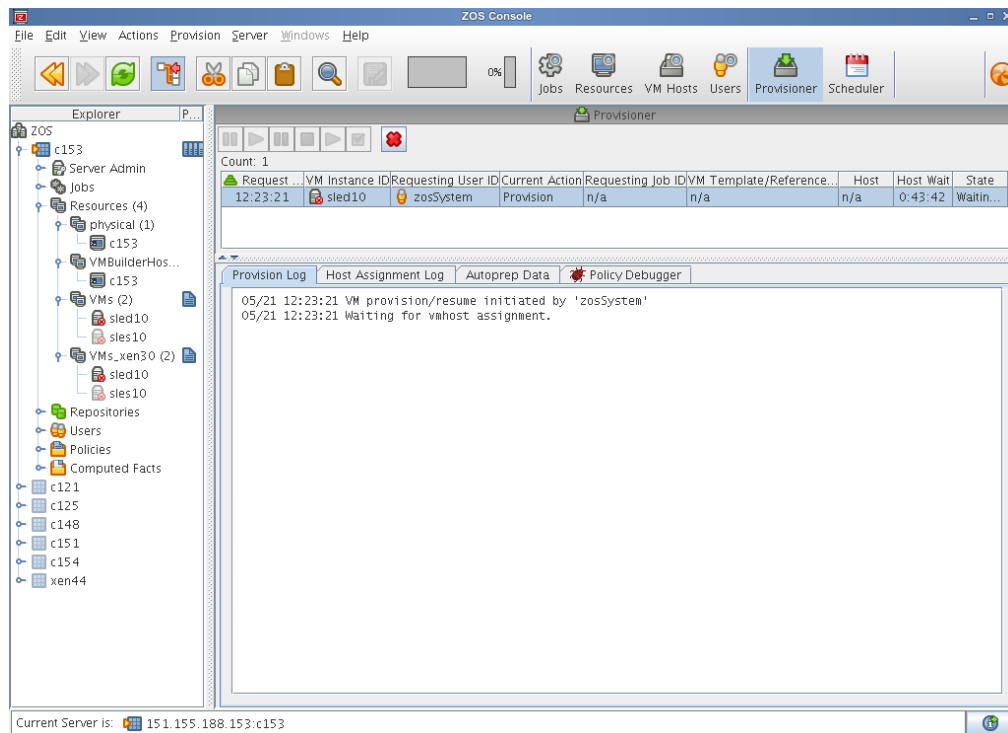
Figure 2-13 A Job in Action



2.2.6 Web-Based User Interaction

ZENworks Orchestrator ships a universal job monitoring and submission interface as a Web application that natively runs on the Orchestrator Server. This application is written to the Orchestrator job management API and can be customized or replaced with alternative rendering as required. The figure belows shows an example of the interface and more details are discussed in [“The ZENworks Orchestrator Console.”](#)

Figure 2-14 The ZENworks Orchestrator Console Management Interface



2.3 JDL Job Scripts

The Orchestrator job definition language (JDL) is an extended and embedded implementation of Python. It is completely multi-threaded. The developer of the job has full access to the Python language and standard extensions, and the ZENworks Orchestrator 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 and cached movement of files across the infrastructure.
- ◆ Automatic breakdown and distribution of parallel operations
- ◆ Failover logic

For more information about the Orchestrator JDL script editor, see [Section 7.2, “JDL Package,” on page 72](#) or “JDL Editor” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.

The JDL language allows for the scripted construction of test cases that can be driven by external parameters and constraints at the time the job instance is executed. In addition, the development of a job with the JDL (Python) language is very straightforward. For a listing of the job, joblet, and utility classes, see [Appendix A, “Orchestrator Job Classes and JDL Syntax,” on page 197](#).

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 (or user and relevant groups). If none are specified, all resources match. The `exampleJoblet` class would typically execute some process or test based on unique parameters.

Finally, the `ParameterSpace` object accesses the built in “grid” ability of the Orchestrator Server, which is the way to describe the parallel inherent in a problem. In this example, the simple addition of `appendRows()` indicates that each joblet can run in parallel. This guide provides information to create much more sophisticated constructs.

2.3.1 Principles of Job Operation

Whenever a job is run on the Orchestrator system it undergoes state transition, as illustrated in [Figure 2-15 on page 39](#). In all, there are 11 states. The following four states are important in understanding how constraints are applied on a job’s life cycle through policies:

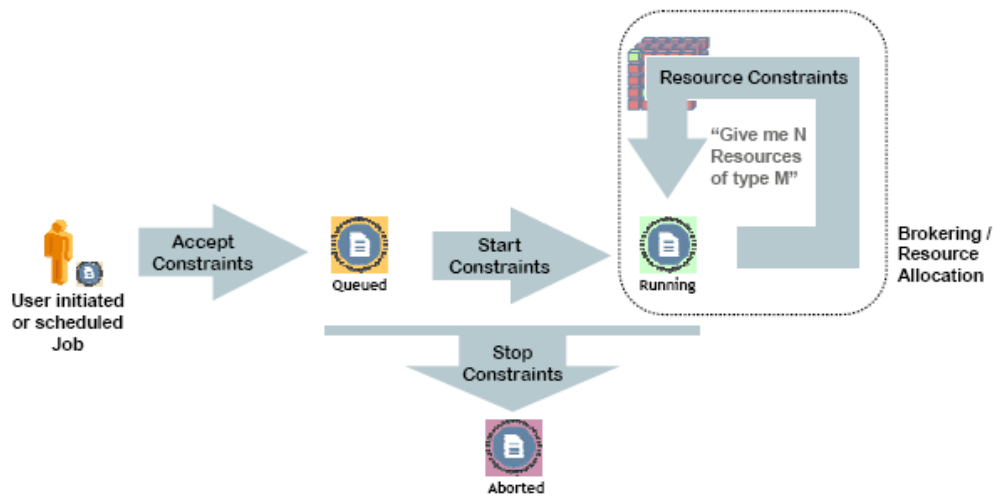
Accept: Used to prevent work from starting; enforces a hard quota on the jobs.

Start: Used to queue up work requests; limits the quantity of jobs or the load on a resource.

Resource: Used to select specific resources.

Stop: Used to abort jobs; provides special timeout or overrun conditions.

Figure 2-15 Constraint-Based Job State Transition



For more information about job life cycle, see [Section 7.3.1, “Job State Transition Events,”](#) on [page 73](#).

2.4 Understanding TLS Encryption

Understanding Transport Layer Security (TLS) encryption is particularly important if you reinstall the server and have an old server certificate in either your agent or client user profile. It's kind of like “ssh.” If you have an old certificate, you need to either manually replace it or delete it and allow the client or agent to download the new one from the server using one of the following procedures:

- ♦ **For the Agent:** The TLS certificate is in `<agentdir>/tls/server.pem`. Deleting this certificate will cause the agent, by default, to log a minor warning message and download the new one the next time it tries to connect to the server. This is technically not secure, since the server could be an impersonator. If security is required for this small window of time, then the real server's `<serverdir>/<instancedir>/tls/cert.pem` can be copied to the above `server.pem` file.
- ♦ **For the Client:** The easiest way to update the certificate from the command line tools is to simply answer "yes" both times when prompted about the out-of-date certificate. This is, again, not 100% secure, but is suitable for most situations. For absolute security, hand copy the server's `cert.pem` (see above) to `~/.novell/zos/client/tls/<serverIPAddr:Port>.pem`.
- ♦ **For Java SDK clients:** Follow the manual copy technique above to replace the certificate. If the local network is fairly trustworthy, you can also delete the above `~/.novell/.../* .pem` files, which will cause the client to auto-download the new certificate on a once-only basis.

2.5 Understanding Job Examples

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

- ♦ [Section 2.5.1, “provisionBuildTestResource.job,” on page 40](#)
- ♦ [Section 2.5.2, “Workflow Job Example,” on page 42](#)

For additional job examples, see [Chapter 10, “Complete Job Examples,” on page 123](#).

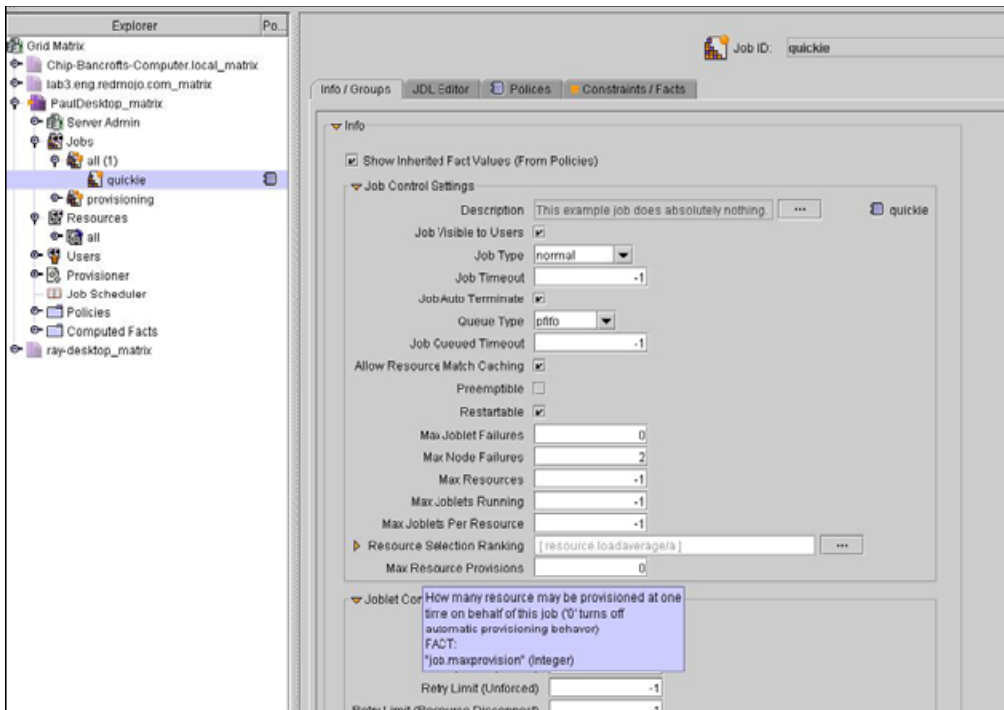
2.5.1 provisionBuildTestResource.job

The following job example illustrates simple scripting to ensure that each of the desired OS platforms are available in the grid and, if not, it tries to provision them. 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 \
21                    resource.getFact("resource.online") == False:
22                        # Found a vm or template. provision it for job.
23                        print "Submitting provisioning request for vm %s." %
24 (resource)
25                        instance = resource.provision()
26                        print "Provisioning successfully submitted."
27                        break
28                if instance == None:
29                    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 Orchestrator 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 Orchestrator console; that is, `job.maxprovision` is enabled as shown in the following figure.

Figure 2-16 Example of a Job to Provision Resources Automatically



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

Figure 2-17 The ZENworks Orchestrator Console Showing Virtual Machine Management



All provisioned virtual machines and the status of the various hosts are visible in this example of the Orchestrator console.

2.5.2 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 5.2, “BuildTest Job Examples,”](#) on page 59.

```
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 40) to ensure all the necessary resources are available, and then starts the [buildTest.jdl Example](#) (page 61). This workflow job does not complete until the two subjobs are complete, as defined in lines 3 and 4.

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

The ZENworks Orchestrator Datagrid

3

This section explains concepts related to the Novell® ZENworks® Orchestrator Server datagrid and specifies many of the objects and facts that are managed in the grid environment:

- ♦ [Section 3.1, “Defining the Datagrid,” on page 43](#)
- ♦ [Section 3.2, “Datagrid Communications,” on page 45](#)
- ♦ [Section 3.3, “datagrid.copy Example,” on page 47](#)

3.1 Defining the Datagrid

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

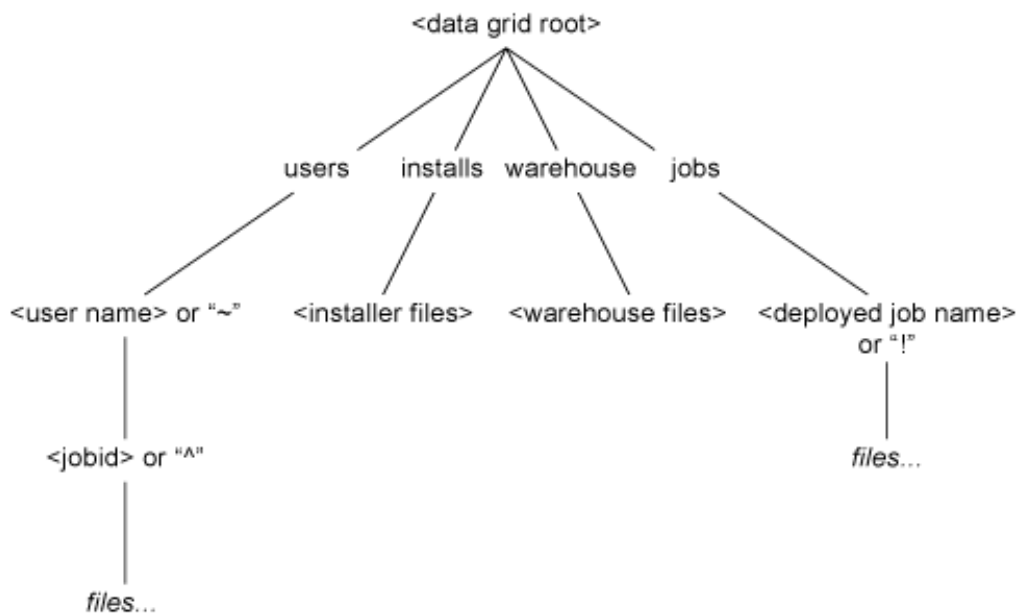
- ♦ [Section 3.1.1, “Naming Orchestrator Job Files,” on page 43](#)
- ♦ [Section 3.1.2, “Distributing Files,” on page 44](#)
- ♦ [Section 3.1.3, “Simultaneous Multicasting to Multiple Receivers,” on page 45](#)

3.1.1 Naming Orchestrator Job Files

The Orchestrator datagrid provides a file naming convention that is used by the job JDL code and by the Orchestrator CLI for accessing files created by the grid job. The naming convention is in the form of a URL. For more information, see [Section 2.1.4, “Jobs,” on page 24](#).

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

Figure 3-1 File Structure of Data Nodes in a Datagrid



The grid URL naming convention is the form `grid://<gridID>/<file path>`. Including the gridID is optional and its absence means the host default grid. When writing jobs and configuring a datagrid, entering the symbol `^` can be used 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 Orchestrator Server you are using:

- ◆ “Directory Listing of the Datagrid Root Example” on page 44
- ◆ “Directory Listing of the Jobs Subdirectory Example” on page 44

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

This example displays all the jobs that are deployed with the Virtual Machine Management (VMM) pack. The jobs listing is different in the High performance Computing (HPC) pack.

```
$ 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
```

3.1.2 Distributing Files

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

3.1.3 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 3.2, “Datagrid Communications,” on page 45](#).

3.1.4 Orchestrator Datagrid Commands

The following datagrid commands can be used when creating job files. To see where these commands are applied in the Novell Orchestrator console, see [“Typical Use of the Grid”](#).

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.

3.2 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 lose 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 Orchestrator 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.

3.2.1 Multicast Example

Multicast transfers are currently only supported through JDL code on the agents. Doing it via the command line client interface UI, would be far too messy. 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 script:

```
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.

3.2.2 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 “[ClientAgent](#)” on page 251.

3.2.3 Plan for Datagrid Expansion

When planning your datagrid, you need to consider where you want the Orchestrator 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`.

3.3 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()
```


Orchestrator Job Classifications

4

This section discusses the core job classifications that can be run by the Novell® ZENworks Orchestrator Server® on grid objects:

- ♦ [Section 4.1, “Resource Discovery,”](#) on page 49
- ♦ [Section 4.2, “Dynamic Scheduling,”](#) on page 51
- ♦ [Section 4.3, “Workload Management,”](#) on page 52
- ♦ [Section 4.4, “Policy Management,”](#) on page 53
- ♦ [Section 4.5, “Auditing and Accounting Jobs,”](#) on page 55

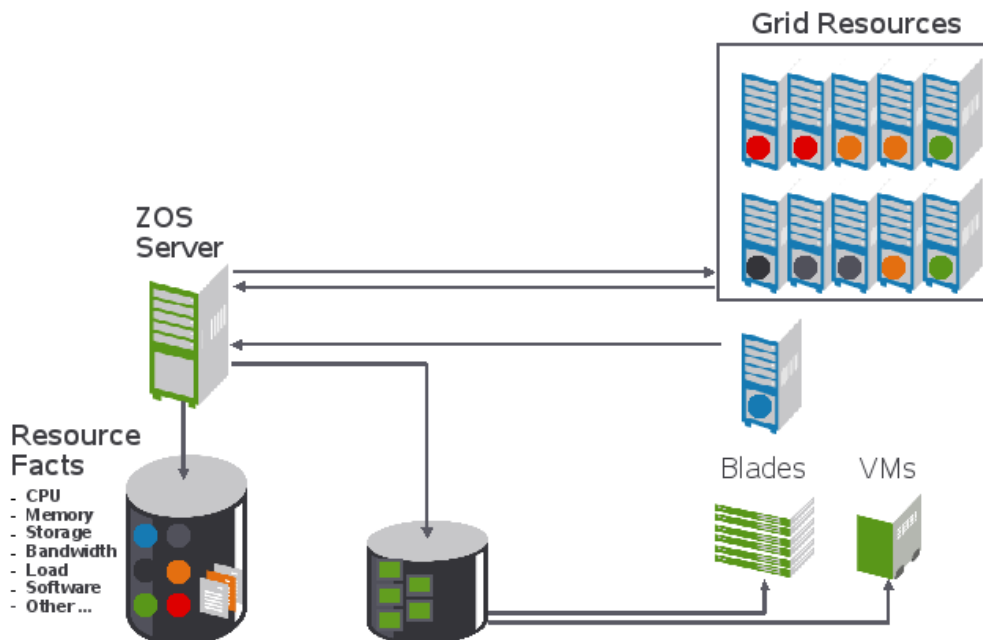
For more information, see [Section 2.1, “Orchestrator Development Architecture,”](#) on page 19.

4.1 Resource Discovery

Resource discovery jobs introspect a resource’s environment to set resource facts stored with the resource grid object.

As shown in the following figure, resource discovery jobs automatically discover the resource attributes (fully extensible facts relating to CPU, memory, storage, bandwidth, load, software inventory, and so forth) of the resources being managed by the ZENworks Orchestrator server. This enables the Orchestrator Server to automatically detect new resources and to integrate resource provisioning of on-line resources. The server can also keep account of these facts for managed resources that are off line.

Figure 4-1 Resource Discovery Overview



For more information, see “[Walkthrough: Observe Discovery Jobs Run](#)” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*, and “[Discovering Virtual Machine Hosts](#)” in the *Novell ZENworks Orchestrator 1.3 Virtual Machine Management Guide*.

4.1.1 Provisioning Jobs

Provisioning jobs are included in the Orchestrator VM Management Pack and are used for interacting with a VM Technology for VM life cycle management and for cloning, moving VMs, and other management tasks. These type of jobs are called Provisioning Adapters and are members of the job group called “provisionAdapters.”

For more information, see [Section 9.2, “Virtual Machine Management,” on page 92](#) and “[Virtual Machine Technologies and Actions](#)” in the *Novell ZENworks Orchestrator 1.3 Virtual Machine Management Guide*.

4.1.2 Resource Targeting

Resource targeting jobs typically match constraints on the client and server, then test for conditions required by specific grid resources.

4.1.3 Resource Discovery Jobs

Some of the commonly used resource discovery jobs include:

- ♦ “[auditCleaner.job](#)” on page 50
- ♦ “[cpuInfo.job](#)” on page 50
- ♦ “[demoInfo.job](#)” on page 50
- ♦ “[findApps.job](#)” on page 50
- ♦ “[logRotator.job](#)” on page 50
- ♦ “[osInfo.job](#)” on page 51

auditCleaner.job

Cleans up the audit database, which frequently becomes cluttered because of massive audit files.

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.

logRotator.job

Rotates server logs.

osInfo.job

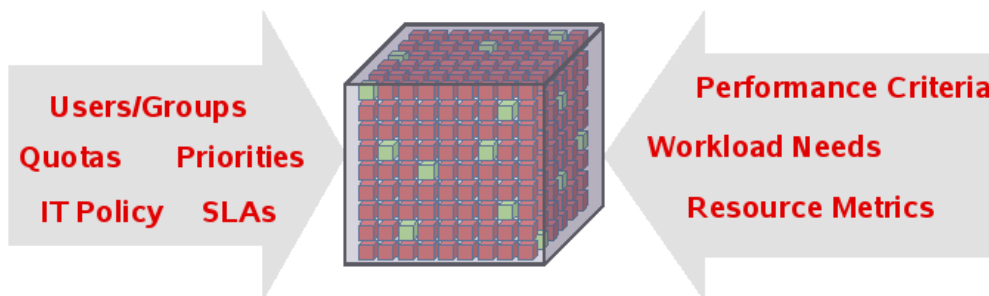
Gets the operating system of a grid resource. On Linux, it reads the `/proc/cpuinfo`; on Windows, it reads the registry; on UNIX, it executes `uname`.

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

4.2 Dynamic Scheduling

ZENworks Orchestrator enables you to create jobs that meet the infrastructure scheduling and resource management requirements of your data center, as illustrated in the following figure.

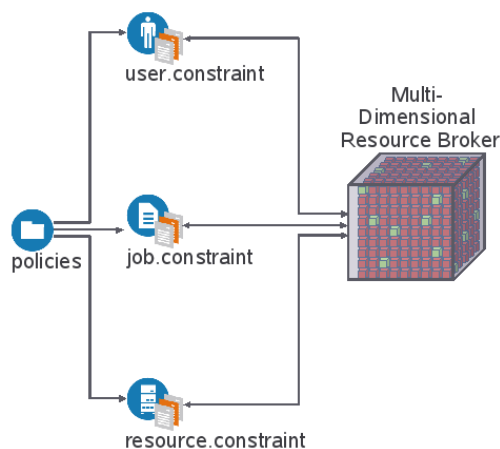
Figure 4-2 Multi-Dimensional Resource Scheduling Broker



Notice that there are many combinations of constraints and scheduling demands on the system that can be managed by the highly flexible Orchestrator resource broker.

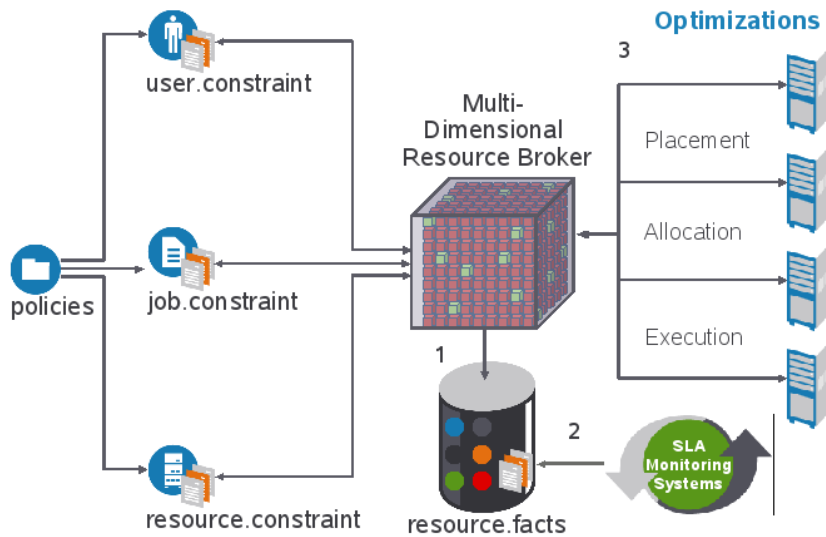
Policies govern how jobs are dynamically scheduled based on the various job constraints that are shown in the following figure. Policies are XML files that specify the requirement for all constraints.

Figure 4-3 Policy-Based Resource Management Relying on Various Constraints



Every resource has abstracted attributes, called facts, that define its operational characteristics.

Figure 4-4 Policy-Based Job Management



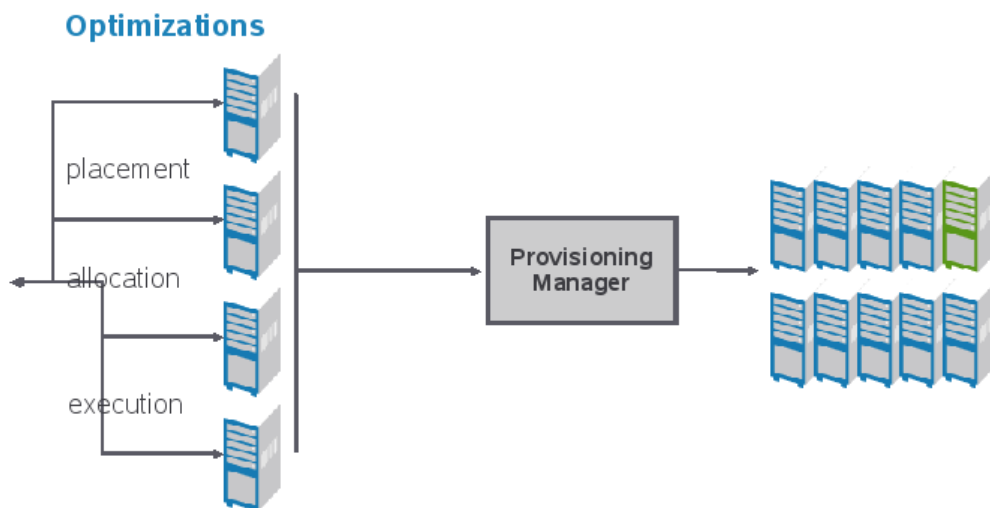
Resources that are discovered by the Orchestrator Server (resource broker) are defined by a resource.facts file (1), which are monitored by service level agreement systems on the broker (2).

See [Chapter 8, “Job Scheduling,” on page 87](#) for examples of scheduling jobs. See also [Section 7.7, “Working with Facts and Constraints,” on page 77](#).

4.3 Workload Management

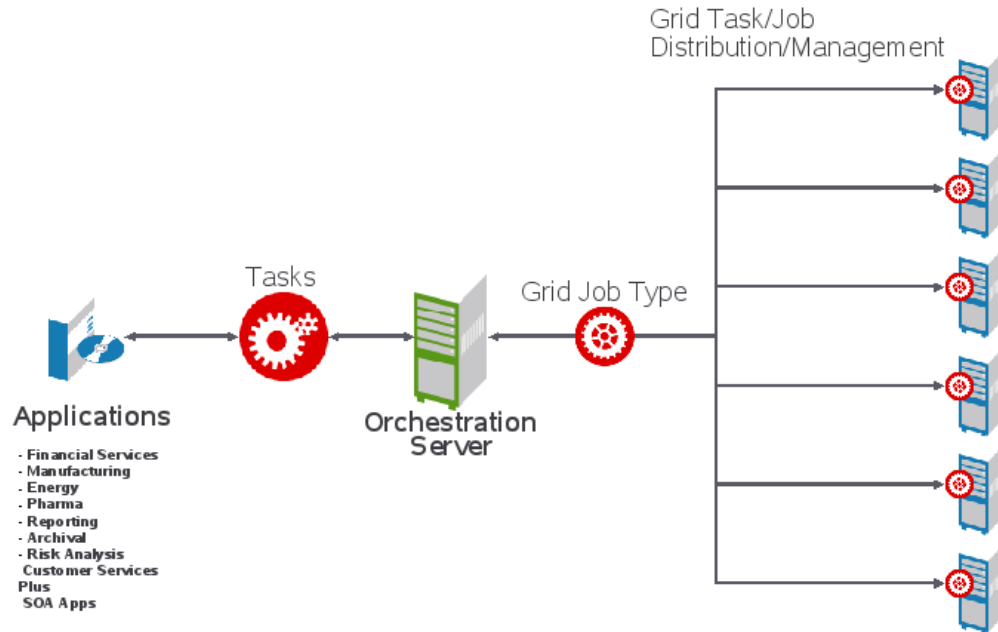
The Orchestrator Server uses a provisioning manager to initiate and monitor changes for both online and offline resources, and also supports physical and virtual servers. It also manages preemption and stealing (reassignment) of resources by monitoring the system and submitting workload management jobs as needed.

Figure 4-5 Workload Management



Depending on the tasks that applications might require, the Orchestrator Server submits the required jobs to one or more of the connected services to perform the specific tasks as shown in the figure below.

Figure 4-6 Grid Scheduling



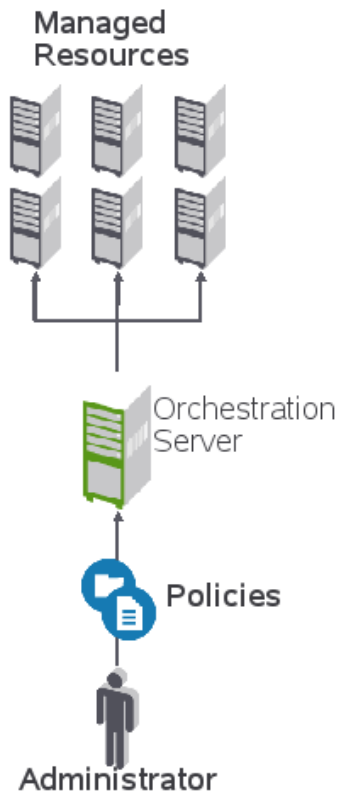
For more information about how grid scheduling works, see the following sections:

- ♦ [Chapter 8, “Job Scheduling,”](#) on page 87
- ♦ [Section 9.6, “Automatically Provisioning a VM Server,”](#) on page 100
- ♦ Examples: [dgtest.job](#) (page 138) and [factJunction.job](#) (page 179).

4.4 Policy Management

Policies are the core control system across the Orchestrator Server. Jobs can be written to change the configuration and behavior of policies, such as pooling of dynamic and virtual resources, managing system loads, monitoring and controlling user quotas, controlling priorities of resources, scheduling jobs and resources, controlling failover operations, and other functions.

Figure 4-7 *Managing Policies*



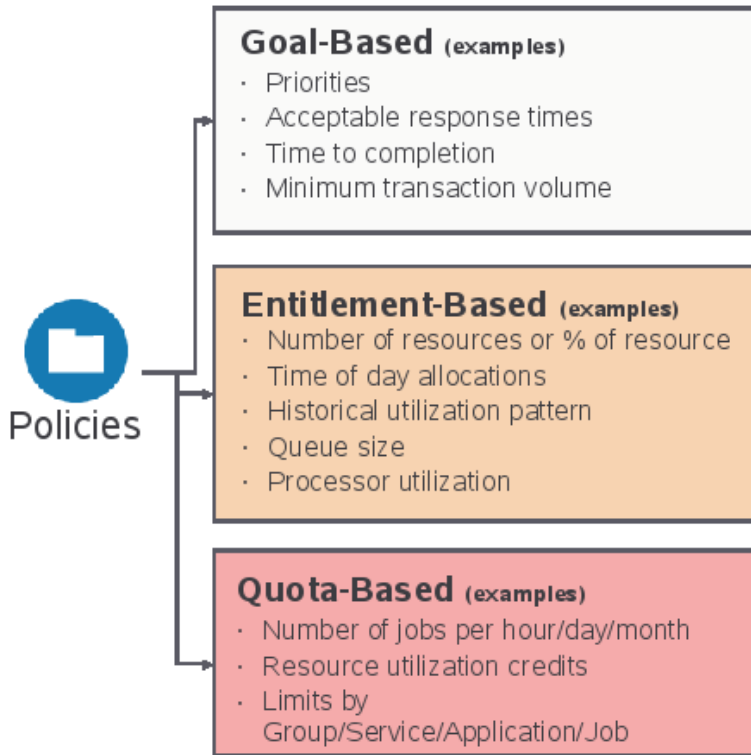
Policies are XML-based files that aggregate resource facts and constraints used to control resources. This provides an extensible and highly flexible system that can covers a wide range of user and resource demands.

Policies are used to enforce quotas, job queuing, resource restrictions, permissions, etc. They can be set on all objects. The policy example below constrains a job to limit the number of running jobs to a defined value, but exempts certain users from this limit. All of the jobs that attempt to 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>
```

As illustrated in the following figure, policies can be based on goals, entitlements, quotas, and other factors, all of which are controlled by jobs.

Figure 4-8 Policy Types and Examples



For more information about policies, see [Section 2.2.2, “Policy-Based Management,”](#) on page 31. See also [Section 7.7, “Working with Facts and Constraints,”](#) on page 77.

4.5 Auditing and Accounting Jobs

Jobs also can be created to control reporting, auditing, and costing. All activity is logged to an ADBMS database and is available for reporting. Your jobs can aggregate cost accounting for assigned resources and perform resource audit trails.

For more detailed information about the structure of jobs, see [“auditCleaner.job”](#) on page 50 and [Chapter 8, “Job Scheduling,”](#) on page 87.

This section explains policy concepts required when developing Novell® ZENworks® Orchestrator jobs:

- ♦ [Section 5.1, “Policy Elements,” on page 57](#)
- ♦ [Section 5.2, “BuildTest Job Examples,” on page 59](#)

5.1 Policy Elements

XML is the representation for Orchestrator Policy elements. A policy can be deployed to the server and associated with any grid object. The policy element is the root element for policies. Policies contain constraints and fact definitions for grid objects:

- ♦ [Section 5.1.1, “Constraints,” on page 57](#)
- ♦ [Section 5.1.2, “Facts,” on page 57](#)
- ♦ [Section 5.1.3, “Computed Facts,” on page 58](#)

5.1.1 Constraints

The constraint element defines the selection of grid objects such as resources. The required type attribute defines the selection type. Supported types are:

- ♦ Resource
- ♦ Provision
- ♦ Allocation
- ♦ Accept
- ♦ Start
- ♦ Continue
- ♦ vmhost
- ♦ Repository

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. See also [Section 7.7, “Working with Facts and Constraints,” on page 77](#).

5.1.2 Facts

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.

5.1.3 Computed Facts

Computed facts are used when you want to run JDL to generate the value for a fact. Although computed facts are not jobs, they use the same JDL syntax.

To create a new computed fact, you subclass the `ComputedFact` class with the `.cfact` extension. An implementation uses the `ComputedFactContext` to get the evaluation context. For more information, see the job structure from the following examples:

- ♦ [ComputedFact \(page 207\)](#)
- ♦ [ComputedFactContext \(page 208\)](#)

After the new computed fact is created, you deploy it using the same procedures required for jobs (using either the `zosadmin` command line tool or the Orchestrator console).

The following example shows a computed fact that returns the number of active job instances for a specific job for the current job instance. This fact can be used in an accept or start constraint to limit how many jobs a user can run in the system. The constraint is added to the job policy in which to have the limit. In this example, the start constraint uses this fact to limit the number of active jobs for a user to one:

```
"""
    <constraint type="start" >
      <lt fact="cfact.activejobs"
        value="1"
        reason="You are only allowed to have 1 job running at a time" />
    </constraint>
```

Change `JOB_TO_CHECK` to define which job is to be limited.

```
"""
JOB_TO_CHECK="quickie"

class activejobs(ComputedFact):

    def compute(self):

        j = self.getContext()
        if j == None:
            # This means computed Fact is executed in a non running

            # job context. e.g., the ZOC fact browser
            print "no job instance"
            return 0
        else:
            # Computed fact is executing in a job context
            user = j.getFact("user.id")
            activejobs = self.getMatrix().getActiveJobs()
            count = 0
            for j in activejobs:
                jobname = j.getFact("job.id")

                # Don't include queued in count !
                state = j.getFact("jobinstance.state.string")
                if jobname == JOB_TO_CHECK \
```

```

        and j.getFact("user.id") == user \
        and (state == "Running" or state == "Starting"):
count+=1

    jobid = j.getFact("jobinstance.id")
    print "jobid=%s count=%d" % (jobid,count)
    return count

```

For another computed fact example, see `activejobs.cfact` (located in the `examples/activejobs.cfact` directory).

5.2 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's policy. In addition, you can create a fact directly in the JDL job code. The fact does not need to be an argument; you can simply create the fact and write it to the fact database for use in specifying parameters for other objects as well.

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 names.

This section explains a relatively simple working job that performs a set (100) 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. Orchestrator has multiple methods to specify parameters. This is just one example (`/QA/testlists/nightly.dat`):

```

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

```

To demonstrate the possible functionality for this example, we have invented some policies that might apply to this example:

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

5.2.1 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"8       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
17 accepted." />
18     <fact name="max_licenses"
19       type="Integer"
20       value="5"
21       description="License count to limit number of jobs to run
22 simultaneously. Any above this limit are queued." />
23     <fact name="max_test_failures"
24       type="Integer"
25       value="50"
26       description="To decide to end the job if the number of failures
27 exceeds a limit" />
28   </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
27 reached">
28     <or>
29       <defined fact="user.privileged_user" />
30       <lt fact="job.instances.queued" factvalue="job.max_queue_size" />
31     </or>
32   </constraint>
33   <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. As noted earlier, this is normally associated with the user or user group.

```

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>

```

NOTE: Typically, resource usage restrictions are specified on the user or user group and not on the job.

5.2.2 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 ZENworks Orchestrator Console.

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 Orchestrator operations that allow interaction with the Orchestrator 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. Alternatively, these constraints could be set programmatically as options in the Orchestrator console.

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. After they are deployed, these joblets can be viewed individually in various Orchestrator UI portals.

```

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 all portal, Orchestrator Console, command line, SDK, 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 += ", "
49         else:
50             if len(m) > 0:
51                 m += ", "
52             m += printable_key + ": %d" % (total)
53             i += 1
54         self.total_counts[key] = total
55     self.setFact("jobinstance.test_failures",self.total_counts["failed"])
56     self.setFact("jobinstance.memo",m)

```

Joblet Definition

As previously discussed, a joblet is the logic that is automatically shipped to any resource employed by this job, as defined in lines 56-80. The `joblet_started_event` in line 60 mirrors the `job_started_event` (line 6) but, of course, 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 locally 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 server prior to completion with the result counts.

```

56 #
57 # Define test execution on a resource.
58 #59 class BuildTestJoblet(Joblet):
59     def joblet_started_event(self):
60         passed = 0
61         failed = 0
62         run = 0
63         # Iterate over parameter space assigned to this Joblet
64         pspace = self.getParameterSpace()
65         while pspace.hasNext():
66             chunk = pspace.next()
67             cmd = chunk["cmd"].strip()
68             rslt = self.run_cmd(cmd)
69             print "rslt=%d cmd=%s" % (rslt,cmd)
70             if rslt == 0:
71                 passed +=1
72             else:
73                 failed +=1
74             run += 1
75
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()

```

5.2.3 Packaging Job Files

A job package might consist of the following elements:

- ♦ Job description language (JDL) code (the Python-based script containing the bits to control jobs).
- ♦ An optional policy XML file, which applies constraints and other job facts to control jobs. Constraints and facts are discussed [Constraints \(page 57\)](#) and [Facts \(page 57\)](#).
- ♦ Any other associated executables or data files that the job requires.

[Section 7.2, “JDL Package,” on page 72](#) provides more information about job elements.

5.2.4 Deploying Packaged Job Files

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

- ♦ Copy job files into the “hot” Orchestrator Server deployment directory. See [“Deploying a Sample System Job”](#) in the *Novell ZENworks Orchestrator 1.3 Administration Guide*.
- ♦ Drag and drop job files using the Orchestrator management console. You can drag browser attributes and job files to the Orchestrator Toolbox. This serves as a shortcut to its browser view in the workspace panel. When you double-click a component in the Toolbox, the console displays that component’s view in the workspace panel.
- ♦ Use the Orchestrator command line (CLI) tools. This process is discussed in [“The Zos Command Line Tool”](#) in the *Novell ZENworks Orchestrator 1.3 Job Management Guide* and in [“The zosadmin Command Line Tool”](#) in the *Novell ZENworks Orchestrator 1.3 Administration Guide*.

5.2.5 Running Your Jobs

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

- ♦ **Command Line Interface:** Nearly all Orchestrator 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.

- ♦ **Web Portal interface:** After Orchestrator is installed, the management console is available online to edit, deploy, and run all jobs you create. This process is discussed in [“Using the Orchestrator User Portal to Run VM Jobs”](#) in the *Novell ZENworks Orchestrator 1.3 Virtual Machine Management Guide*.
- ♦ **Custom Client:** The Orchestrator toolkit provides an SDK that provides a custom client that can invoke your custom jobs. This process is discussed in [Appendix B, “Orchestrator Client SDK,” on page 245](#).

- ♦ **Built-in Job Scheduler:** The Orchestrator Server uses a built-in job scheduler to run deployed jobs. This process is discussed in [Chapter 8, “Job Scheduling,” on page 87](#).
- ♦ **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 87](#).

5.2.6 Monitoring Job Results

ZENworks Orchestrator enables you to monitor jobs by using the same methods outlined in [Section 5.2.5, “Running Your Jobs,” on page 64](#). The following figure shows examples of the status of the job ray.buildtest.18 using different monitoring interfaces:

CLI Job Monitoring

Figure 5-1 CLI Job Monitoring Example

```

> zos status -e ray.buildertest.18
Job Status for ray.buildtest.18
-----
      State: Completed
Resource Count: 0                (0 this job)
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 02-24-06 1705 , Failed: 1, Run: 5,
           Passed: 4
  Priority: medium
Arguments: <none>

Submit Time: 02/24/2006 01:46:12
Delayed Start: n/a
  Start Time: 02/24/2006 01:46:12
    End Time: 02/24/2006 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)

```

Notice in the bottom section that job costing metrics also can be monitored, which are quite minimal in this example. More sophisticated job monitoring is possible. For more information, see [“Verification at the Command Line”](#) and [“Verification at the Jobs Monitor”](#) in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.

Web-Based Control

Figure 5-2 User Portal Job Monitoring Example

The screenshot shows a web-based interface for monitoring jobs. At the top, there are filter options for User (ray), Job (group_all), History (None), and Hours. Below this is a table with columns for Identification, Schedule, Status, and Misc. The table lists several jobs, including workflow jobs (23, 21, 19) and buildtest jobs (18, 17, 16). Each job entry includes its status (Cancelled or Completed), user, name, priority, submit, start, end, and elapsed times, along with cost and progress information. A memo field is visible for buildtest jobs, indicating a failure due to missing resources. At the bottom right, there are buttons for Cancel, Pause, Resume, and Refresh, and a summary line showing Total: 0 | Active: 0 | Pending: 0 | Error: 0.

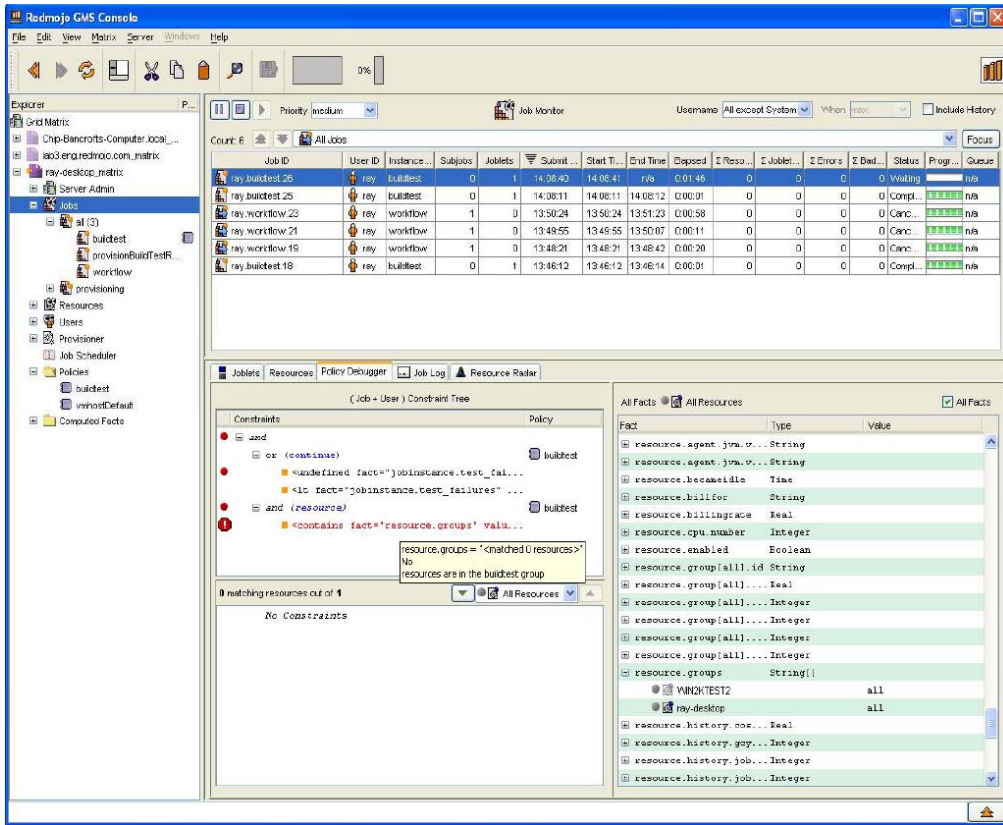
Identification	Schedule	Status	Misc
Cancelled User: ray Job: workflow.23 Name: workflow	Submit: 02/24/06 01:50 PM Start: 02/24/06 01:50 PM End: 02/24/06 01:51 PM Elapsed: 0:00:58	Cost: \$0.00 Progress: 99%	Subjobs: 1 view... Joblets: 0 Tot: 0 Errors: 0
Cancelled User: ray Job: workflow.21 Name: workflow	Submit: 02/24/06 01:49 PM Start: 02/24/06 01:49 PM End: 02/24/06 01:50 PM Elapsed: 0:00:11	Cost: \$0.00 Progress: 99%	Subjobs: 1 view... Joblets: 0 Tot: 0 Errors: 0
Cancelled User: ray Job: workflow.19 Name: workflow	Submit: 02/24/06 01:48 PM Start: 02/24/06 01:48 PM End: 02/24/06 01:48 PM Elapsed: 0:00:20	Cost: \$0.00 Progress: 99%	Subjobs: 1 view... Joblets: 0 Tot: 0 Errors: 0
Completed User: ray Job: buildtest.18 Name: buildtest Priority: medium	Submit: 02/24/06 01:46 PM Start: 02/24/06 01:46 PM End: 02/24/06 01:46 PM Elapsed: 0:00:01	Cost: \$0.00 Progress: 100%	Subjobs: 0 Joblets: 1 Tot: 1 Errors: 0
Memo: Build Test BuildId 02-24-06 1705 , Failed: 1, Run: 5, Passed: 4			
Completed User: ray Job: buildtest.17 Name: buildtest Priority: medium	Submit: 02/24/06 01:43 PM Start: 02/24/06 01:43 PM End: 02/24/06 01:43 PM Elapsed: 0:00:00	Cost: \$0.00 Progress: 100%	Subjobs: 0 Joblets: 1 Tot: 1 Errors: 0
Memo: Build Test BuildId 02-24-06 1705 , Failed: 1, Run: 5, Passed: 4			
Completed User: ray Job: buildtest.16 Name: buildtest Priority: medium	Submit: 02/24/06 01:43 PM Start: 02/24/06 01:43 PM End: 02/24/06 01:43 PM Elapsed: 0:00:01	Cost: \$0.00 Progress: 100%	Subjobs: 0 Joblets: 1 Tot: 1 Errors: 0

In this example, the job memo field is displayed.

5.2.7 Debugging Jobs

As a brief introduction to the built-in debugging capabilities of ZENworks Orchestrator, the following figure shows how to find 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 5-3 Debugging Jobs Using the Orchestrator Console



The policy debugger shows the blocking constraint, and the tooltip gives the reason why. Dragging and dropping the resources to the required group allows the job to quickly continue and no restart was necessary.

Using the Orchestrator Client SDK

6

Novell® ZENworks® Orchestrator 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, as described in [Appendix B, “Orchestrator Client SDK,” on page 245](#). This SDK application can perform the following operations:

- ◆ Login and logout to an Orchestrator 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).

6.1 SDK Requirements

Before you can run the Orchestrator Client SDK, you must perform the following tasks:

1. Install the Orchestrator Client package. For instructions, see [“Installing the Agent and Clients by Using the Product ISO”](#) and [“Walkthrough: Launch the ZENworks Orchestrator Console”](#) in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.
2. Install [JDK 1.4.2](http://java.sun.com/j2se/1.4.2/download.html) (<http://java.sun.com/j2se/1.4.2/download.html>).
3. Examine the two example Orchestrator 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. .

6.2 Creating an SDK Client

Use the following procedure to create an SDK client in conjunction with the sample Java code (see [Section B.3.2, “ClientAgent,” on page 251](#)):

- 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 Orchestrator Server (see [“Walkthrough: Log in to the ZENworks Orchestrator Server”](#)):

```
Credential credential =  
CredentialFactory.newPasswordCredential (username,password) ;  
  
clientAgent.login(credential) ;
```

- 3** Run the server operations; in this case, it is the quickie.jbl 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 Orchestrator Server to receive job events for the job you started.

```
clientAgent.getMessages (jobID) ;
```

- 5** Log out of the server:

```
clientAgent.logout ()
```

The ZENworks® Orchestrator job scheduling manager 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:

- ◆ [Section 7.1, “Understanding JDL,” on page 71](#)
- ◆ [Section 7.2, “JDL Package,” on page 72](#)
- ◆ [Section 7.3, “Job Class,” on page 73](#)
- ◆ [Section 7.4, “Job Invocation,” on page 75](#)
- ◆ [Section 7.5, “Deploying Jobs,” on page 75](#)
- ◆ [Section 7.6, “Starting Orchestrator Jobs,” on page 76](#)
- ◆ [Section 7.7, “Working with Facts and Constraints,” on page 77](#)
- ◆ [Section 7.8, “Using Facts in Job Scripts,” on page 79](#)
- ◆ [Section 7.9, “Using Other Grid Objects,” on page 80](#)
- ◆ [Section 7.10, “Communicating Through Job Events,” on page 80](#)
- ◆ [Section 7.11, “Executing Local Programs,” on page 81](#)
- ◆ [Section 7.12, “Logging and Debugging,” on page 84](#)
- ◆ [Section 7.13, “Improving Job and Joblet Robustness,” on page 85](#)

7.1 Understanding JDL

The Orchestrator 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 Orchestrator 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 \(page 222\)](#) class) defines a method as follows:

```
def my_custom_event(self, job, params):
    print "\u201cGot a my_custom event carrying ", params)
```

And the joblet ([Joblet \(page 224\)](#) class) sends an event/message as follows:

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

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.

7.2 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 ZENworks Orchestrator 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.

7.2.1 .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 `.shed` file. The discovery job “`osInfo.job`” on page 51 has a schedule XML file that specifies to always run a specified job whenever a specific resource is started and becomes available.

7.3 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 7.3.1, “Job State Transition Events,”](#) on page 73.

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.

7.3.1 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.

7.3.2 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 Orchestrator client command line tool:

```
zos event 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", { ... } )
```

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 \(page 219\)](#) 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)

```

7.4 Job Invocation

Jobs can be started using either the zos command line tool, scheduling through a `.shed` file, or manually through the ZENworks® Orchestrator 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.

7.5 Deploying Jobs

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

- ♦ [Section 7.5.1, “Using the Orchestrator Console,” on page 75](#)
- ♦ [Section 7.5.2, “Using the ZOSADMIN Command Line Tool,” on page 75](#)

7.5.1 Using the Orchestrator Console

- 1 In the *Actions* menu, click *Deploy Job*.
- 2 For additional deployment details, see [“Walkthrough: Deploy a Sample Job”](#).

7.5.2 Using the ZOSADMIN Command Line Tool

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 Orchestrator Server to perform.
2. Use the preconfigured JDL files for specific tasks listed in [Appendix A, “Orchestrator Job Classes and JDL Syntax,” on page 197](#).
3. To configure jobs, edit the JDL file with an external text editor.
4. Repackage the job as a `.jar` file.
5. Run the ZOS administration tool to redeploy the packaged job into the Orchestrator Server.
6. Run the job using the `zos` command line tool.
7. Monitor the results of the job in the ZENworks Orchestrator Console.

Another method to deploy jobs is to edit JDL files through the Orchestrator 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 Orchestrator 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 Orchestrator Server. The Orchestrator console has a *Save File* menu option if you want to preserve JDL modifications you made using ZOC.

7.6 Starting Orchestrator Jobs

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

- ♦ Running jobs from the ZOS command line (see [“The zosadmin Command Line Tool”](#)).
- ♦ Running jobs from the Orchestrator scheduler (see [“Understanding the Orchestrator Job Scheduler”](#)).
- ♦ Running jobs from Web applications (see [“Using the ZENworks Orchestrator User Portal”](#)).
- ♦ Running jobs from within jobs (see [“Using Facts in Job Scripts” on page 79](#)).

7.7 Working with Facts and Constraints

You can incorporate facts and constraints into the custom jobs you create to manage your data center resources using Orchestrator. You should already be familiar with the concepts related to controlling jobs using job facts and constraints. For more information, see the following JDL links:

- ♦ [Job \(page 222\)](#)
- ♦ [Joblet \(page 224\)](#)

This section contains the following topics:

- ♦ [Section 7.7.1, “Grid Objects and Facts,” on page 77](#)
- ♦ [Section 7.7.2, “Defining Job Elements,” on page 77](#)
- ♦ [Section 7.7.3, “Job Arguments and Parameter Lists,” on page 78](#)

7.7.1 Grid Objects and Facts

Every resource and service discovered in an Orchestrator-enabled network is identified and abstracted as an object. Within the Orchestrator 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 the properties and characteristics of either physical or virtual resources. In essence, by building, deploying, and running jobs on the Orchestrator 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 resources, users, jobs, repositories, and vmhosts. The grid server assigns default values to each of the component facts, although they can be changed at anytime by the administrator unless they are read-only.

However, the developer desires certain constraints to be used for a job and 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.

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.

7.7.2 Defining Job Elements

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 job `args` 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 XML policy for every given job.

The job argument fact values are passed to a job when the job is run. Consequently, the Orchestrator 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 for the Web 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 \(page 133\)](#).

As noted, when running a job, you can pass in a policy file, which is another method the client can use to control job behavior. Policy files can pass in 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
```

7.7.3 Job Arguments and Parameter Lists

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 named with 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 been 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 from the CLI, 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 Orchestrator scheduler, either with the Orchestrator console or by a `.sched` file.

7.8 Using Facts in Job Scripts

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 [“Job Arguments and Parameter Lists” on page 78](#), 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 \(page 219\)](#).

7.9 Using Other Grid Objects

Grid objects can be 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 \(page 229\)](#) 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")
```

The MatrixInfo grid object also provides functions for creating other grid objects. For more complete information about these functions, see [MatrixInfo \(page 229\)](#).

7.10 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 with each other 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 signalling 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 \(page 222\)](#) and [Section A.7, “Joblet State Values,” on page 200](#).

7.10.1 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 \(page 222\)](#) and [Joblet \(page 224\)](#) documentation.

7.10.2 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.

7.11 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 \(page 215\)](#).

7.11.1 Output Handling

The [Exec \(page 215\)](#) function provides controls for specifying how to handle `stdout` and `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
```

7.11.2 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.runtypes` fact specifies how file and executable operations run in the joblet in behalf of the job user, or not.

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

Table 7-1 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.runttype` fact. The fact `resource.agent.config.exec.asagentuseronly` on the resource grid object can overwrite the `job.joblet.runttype` fact.

This ability to run as the job user is supported by the enhanced `exec` feature of the Orchestrator agent. A resource might not support the Orchestrator 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.

7.11.3 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 7.13, “Improving Job and Joblet Robustness,” on page 85](#).

```
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.

7.12 Logging and Debugging

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

- ♦ [Section 7.12.1, “Creating a Job Memo,” on page 84](#)
- ♦ [Section 7.12.2, “Tracing,” on page 85](#)

7.12.1 Creating a Job Memo

This job example shows how to set a brief memo visible in the Orchestrator console.

In the job section of this example (lines 4-8), the fact name (`jobinstance.memo`) is set by job instance.

```
1 JOB:
2 fact name "jobinstance.memo"
3
4 class MyJob(Job):
5     def job_started_event(self):
6         numJoblets = 2
7         self.setFact("jobinstance.memo", "Running MyJob Scheduling + " +
numJoblets + " joblets")
8         self.schedule(MyJoblet,numJoblets)
```

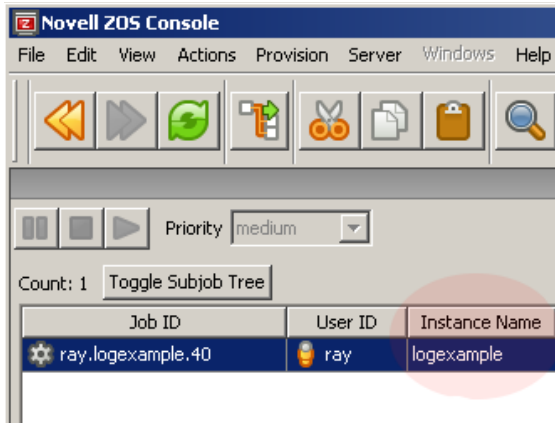
In the joblet section of this example, the fact name (`joblet.memo`, line 10), is set by the joblet instance and consists of a brief memo for each joblet. This typically is used for providing detailed explanations, such as what executable is being run.

```
9 JOBLET:
10 fact name "joblet.memo" (Set by joblet instance)
11
```

In lines 12-19, the name of the joblet is specified. This is typically a simple word, which is displayed in the console joblet column view.

```
12 fact name "jobletinstancename"
13
14
15 class MyJoblet(Joblet):
16     def joblet_started_event(self):
17         self.setFact("joblet.memo", "Running MyJoblet on " +
self.getFact("resource.id"))
18
19         self.setFact("jobletinstancename", "webserver")
```

Figure 7-1 Example of How a 'logexample' Name Appears in the Orchestrator Console



7.12.2 Tracing

There are two facts on the job grid object to turn on /off tracing. 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 via the Orchestrator console or from the `zos run` command tool.

7.13 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 Orchestrator 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.

Job Scheduling

8

Novell® ZENworks® Orchestrator schedules jobs either manually using the Job Scheduler or programmatically using the Job Description Language (JDL). This section contains the following topics:

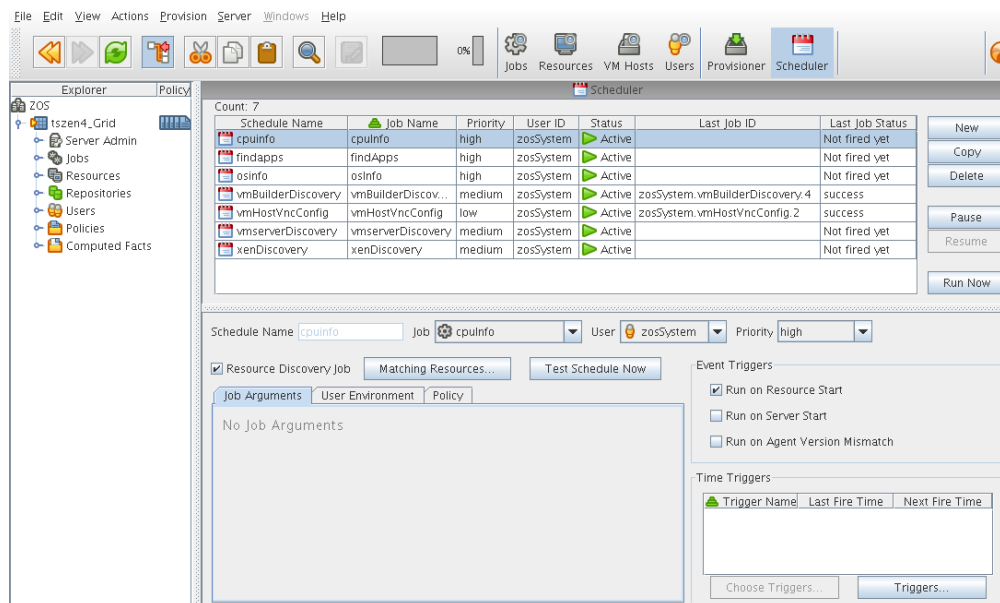
- ♦ [Section 8.1, “Job Scheduler GUI,” on page 87](#)
- ♦ [Section 8.2, “Schedule Files,” on page 88](#)
- ♦ [Section 8.3, “Scheduling with Constraints,” on page 89](#)

8.1 Job Scheduler GUI

After ZENworks Orchestrator 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 events occur.

The following figure illustrates the job Scheduler, with seven jobs staged in the main Scheduler panel.

Figure 8-1 The ZENworks Orchestrator Scheduler GUI



Jobs are individually submitted and managed using the Job Scheduler as discussed in [“Understanding the Orchestrator Job Scheduler”](#) in the *Novell ZENworks Orchestrator 1.3 Administration Guide* and in [“Using the ZENworks Orchestrator User Portal”](#) in the *Novell ZENworks Orchestrator 1.3 Job Management Guide*.

8.2 Schedule Files

In addition to using the Job Scheduler GUI, developers can also write XML files to schedule jobs to run when triggered by specific events. These files are designated using the `.sched` extension and can be included as part of the job archive file.

Everything that you do manually in the Job Scheduler can be automated by creating a `.sched` XML script as part of a job. The XML file enables 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 system events, such as startup (login).

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 “[osInfo.sched Example](#)” on page 88. See also [Section 8.2.2, “Cron Trigger Example,”](#) on page 89.

This section includes the following information:

- ♦ [Section 8.2.1, “osInfo.sched Example,”](#) on page 88
- ♦ [Section 8.2.2, “Cron Trigger Example,”](#) on page 89

8.2.1 osInfo.sched Example

A schedule file can be packaged with a job in a `.job` file or independently deployed to a Orchestrator server. Because the XML file defines the job Schedule programmatically outside of the Orchestrator console, packaging these scripts into jobs is typically a developer task.

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>
2   <job name="osinfo" job="osInfo" user="zosSystem" priority="high" >
3     <resourcediscovery/>
4     <active/>
5     <runonresourcestart/>
6     <runoneachresource/>
7   </job>
8 </schedule>
```

Line 1: Starts the new schedule definition.

Line 2: 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. The `user` parameter identifies what user to run the `osinfo` job, while `priority` specifies a priority at which to run the job.

Line 3: Instructs the schedule to run the job as a resource discovery job.

Line 4: Activates the job after it is deployed. The schedule default is not active.

Line 5: Runs `osInfo` job when the resource logs into the server.

Line 6: Specifies to run a single joblet on each resource.

8.2.2 Cron Trigger Example

The following example shows the syntax of a schedule file that has two cron triggers for scheduling a job:

```
1<schedule>
2  <trigger name="NightlyReportTrigger" description="CRON fire at 4 am every
   day">
3    <cron value="0 0 4 * * ?" />
   </trigger>
4  <trigger name="DailyReportTrigger" description="CRON fire at 4 pm every day">
5    <cron value="0 0 16 * * ?" />
   </trigger>
6  <job name="Report" job="ReportJob" user="manager">
7    <jobargs>
8      <fact name="fullreport" type="Boolean" value="false" />
   </jobargs>
9    <active/>
10   <triggers value="NightlyReportTrigger" />
   <triggers value="DailyReportTrigger" />
11 </job>
12</schedule>
```

Line 1: Starts the new schedule definition.

Lines 2-3: Defines two new triggers to fire at 4 a.m. and 4 p.m. every day.

Line 4: Defines a new scheduled task named *Report* that is used to schedule a run of the job *ReportJob*.

Line 5: Specifies the job parameters for the scheduled job.

Line 6: Activates the job after it is deployed. The schedule default is not active.

Lines 7-8: Defines that the schedule uses the two specified triggers.

A schedule file can be packaged either within a `.job` archive alongside the `.jdl` file or independently deployed using the `zosadmin` command line utility.

8.3 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. However, this is only part of the picture.

Constraints can also be set by an administrator via deployed policies, and additional constraints can be specified by jobs to further restrict a particular job instance. The figure below shows the complete process employed by the Orchestrator 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 in which constraints can be executed. The

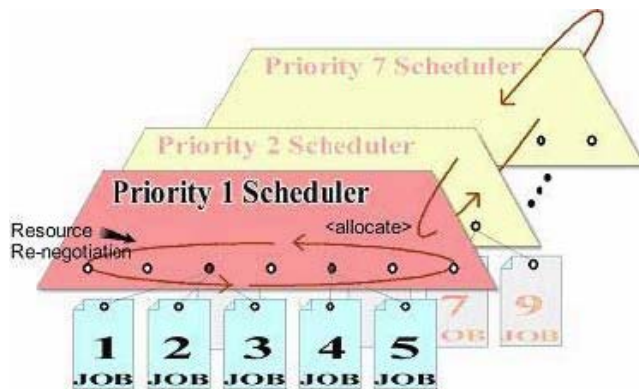
scheduler applies a logic ANDing of job constraints (specified in the policies), grid policy constraints (set on the server), optionally additional user defined constraints specified on job submission, and optional constraints specified by the resources.

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

- ♦ Available resources
- ♦ Preemptable 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 together with information on the minimum acceptable allocation. The scheduler uses this information to appropriate resources for all jobs within the same priority group. Because the scheduler is continually re-evaluating the allocation of resources, the job policies forms part of the schedulers real-time algorithm, thus providing an extremely versatile and powerful scheduling mechanism.

Figure 8-2 Job Scheduling Priority



Although job scheduling might appear complex, it is very easy to use for an end user. For example, a job developer might write just a few lines of policy code to describe a job to require 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>
```

Virtual Machine Job Development

9

This section explains the following concepts related to developing virtual machine (VM) management jobs:

- ◆ [Section 9.1, “VM Job Best Practices,” on page 91](#)
- ◆ [Section 9.2, “Virtual Machine Management,” on page 92](#)
- ◆ [Section 9.3, “VM Life Cycle Management,” on page 93](#)
- ◆ [Section 9.4, “Manual Management of a VM Server’s Lifecycle,” on page 94](#)
- ◆ [Section 9.5, “Provisioning Virtual Machines,” on page 96](#)
- ◆ [Section 9.6, “Automatically Provisioning a VM Server,” on page 100](#)
- ◆ [Section 9.7, “Defining Values for Grid Objects,” on page 101](#)

9.1 VM Job Best Practices

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

- ◆ [Section 9.1.1, “Plan Robust Application Starts and Stops,” on page 91](#)
- ◆ [Section 9.1.2, “Managing VM Systems,” on page 92](#)
- ◆ [Section 9.1.3, “Managing VM Images,” on page 92](#)
- ◆ [Section 9.1.4, “Managing VM Hypervisors,” on page 92](#)
- ◆ [Section 9.1.5, “VM Job Considerations,” on page 92](#)

9.1.1 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 Orchestrator 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.

9.1.2 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.

9.1.3 Managing VM Images

Similar to how the job `instagent` searches for virtual machine grid objects using specified Constraints and runs a VM operation (`installAgent`) on the VMs that are returned, an Orchestrator image must be checked out of the warehouse and mounted, modified, and checked in again. Preferably, this should occur without having to provision the VM itself.

9.1.4 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 management console action.

9.1.5 VM Job Considerations

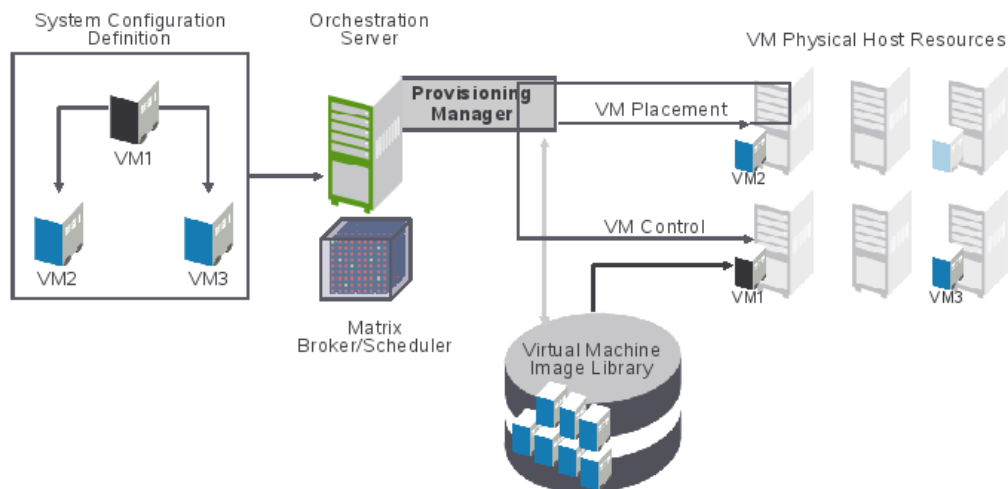
In some instances, some managed resources might host VMs that do not contain an Orchestrator agent. Such VMs can only be controlled by administrators interacting directly with them.

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.

9.2 Virtual Machine Management

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

Figure 9-1 VM Management



For more information about managing virtual machines, see the *Novell ZENworks Orchestrator 1.3 Virtual Machine Management Guide*.

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

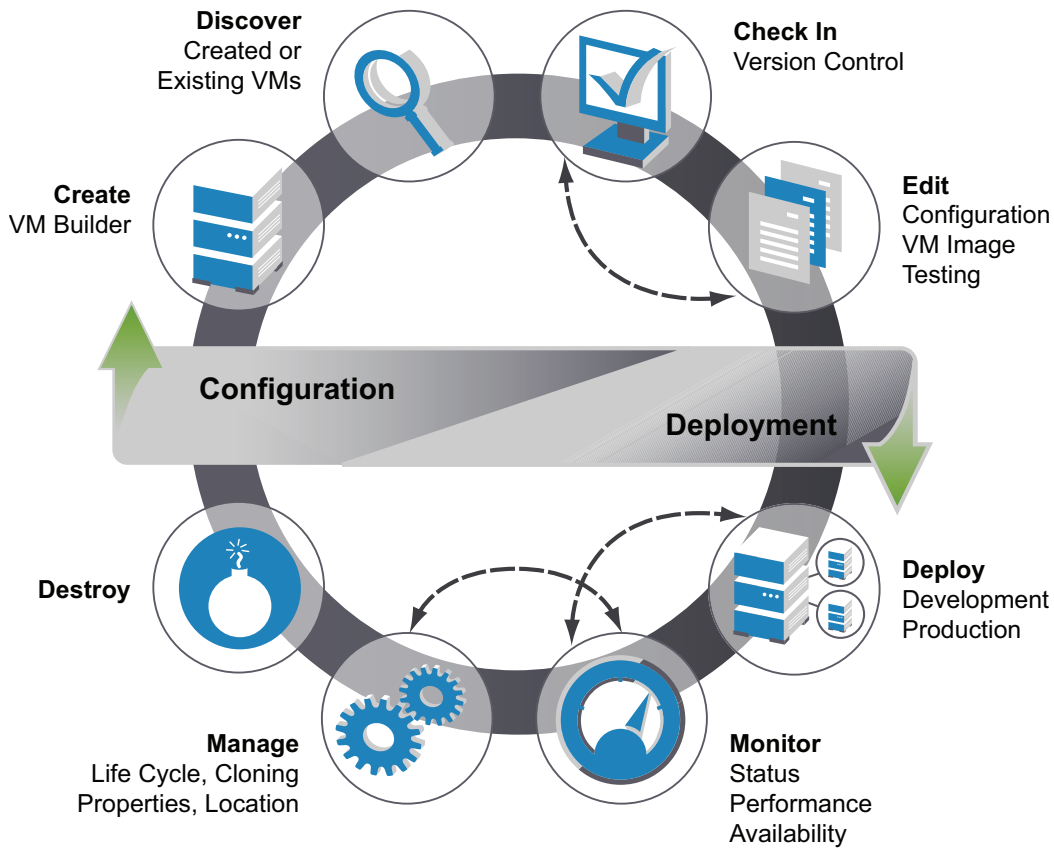
- ♦ **Create and clone VMs:** These jobs Creates virtual machine images to be stored or deployed. They also create templates for building images to be stored or deployed (see “**VM Instance:**” on page 97 and “**VM Template:**” on page 97).
- ♦ **Discover resources that can be used as VM hosts.**
- ♦ **Provision, migrate, and move VMs:** Virtual machine images can be moved from one physical machine to another.
- ♦ **Check in VMs that have the proper versions and system configurations.**
- ♦ **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.
- ♦ **Retire, delete, and destroy VMs:** Jobs can decommission and retire deployed images.
- ♦ **Monitor VM operations:** Jobs can start, shut down, suspend and restart VMs.
- ♦ **Manage on, off, suspend, and restart operations.**

9.3 VM Life Cycle Management

The Novell ZENworks Orchestrator maintains a library of VM images, hosts, and instances. Like physical resources, VMs can be grouped and they have facts that describe their attributes.

ZENworks orchestrator provides a JDL management API for the following tasks you can use jobs to perform on VMs, as illustrated in the following figure:

Figure 9-2 VM Lifecycle Management



The VM life cycle is divided into two parts: configuration and deployment. Configuration is performed on a VM in the security of a development environment which allows for creation, testing, and modification of the VM until you are ready to launch the VM into the production environment. Deployment is performed in the production environment and the changes take place in real time. As the administrator, you might do both the configuration and deployment tasks.

9.4 Manual Management of a VM Server’s 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 Orchestrator 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 9.4.3, “Provision Job JDL,” on page 95](#).

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

- 1 Create your VMs and follow the discovery process in the Orchestrator console so the VMs are in the Orchestrator inventory.
- 2 In Orchestrator console, 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 Orchestrator Server using either the Orchestrator console or the `zosadmin` command line.

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

- ♦ [Section 9.4.1, “Manually Using the ZOS Command Line,” on page 95](#)
- ♦ [Section 9.4.2, “Automatically Using the Orchestrator Console Job Scheduler,” on page 95](#)

9.4.1 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 *Novell ZENworks Orchestrator 1.3 Job Management Guide*.

9.4.2 Automatically Using the Orchestrator Console Job Scheduler

- 1 Create a *New* schedule.
- 2 Fill in the job name (`provision`), user, 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 ZOS Control Scheduler. See also [Section 9.6, “Automatically Provisioning a VM Server,” on page 100](#).

9.4.3 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 222). See also ProvisionSpec (page 235).
```

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>

```

9.5 Provisioning Virtual Machines

VM provisioning adapters run just like regular jobs on the Orchestrator. 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, CPA, 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:

- ♦ **Xen Adapter:** For more information, see [XenSource*](http://www.xen.org/) (<http://www.xen.org/>).
- ♦ **VMware Server 1.0 and GSX 3.2:** For more information, see [VMWare](http://www.vmware.com/) (<http://www.vmware.com/>).
- ♦ **Virtual Center 1.3.x or later, and Virtual Center 2.0.1 or later:** For more information, see [VMware](http://www.vmware.com/) (<http://www.vmware.com/>).

For more information, see “[Virtual Machine Technologies and Actions](#)” in the *Novell ZENworks Orchestrator 1.3 Virtual Machine Management Guide*.

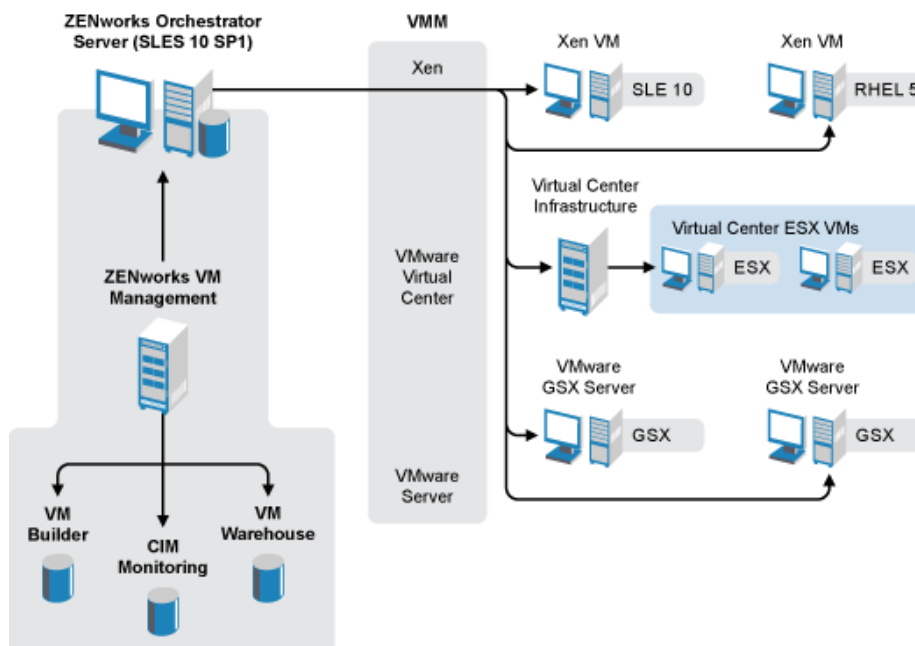
There are two types of VMs that can be provisioned:

- ♦ **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 through 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 of VMware Server, VMware Virtual Center, and Xen 3.0:

Figure 9-3 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/) (<http://www.xen.org/>).

The following sections provide more information on provision of VMs:

- ♦ [Section 9.5.1, “Provisioning VMs Using Jobs,” on page 98](#)
- ♦ [Section 9.5.2, “VM Placement Policy,” on page 99](#)
- ♦ [Section 9.5.3, “Provisioning Example,” on page 100](#)

9.5.1 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 management console), 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.

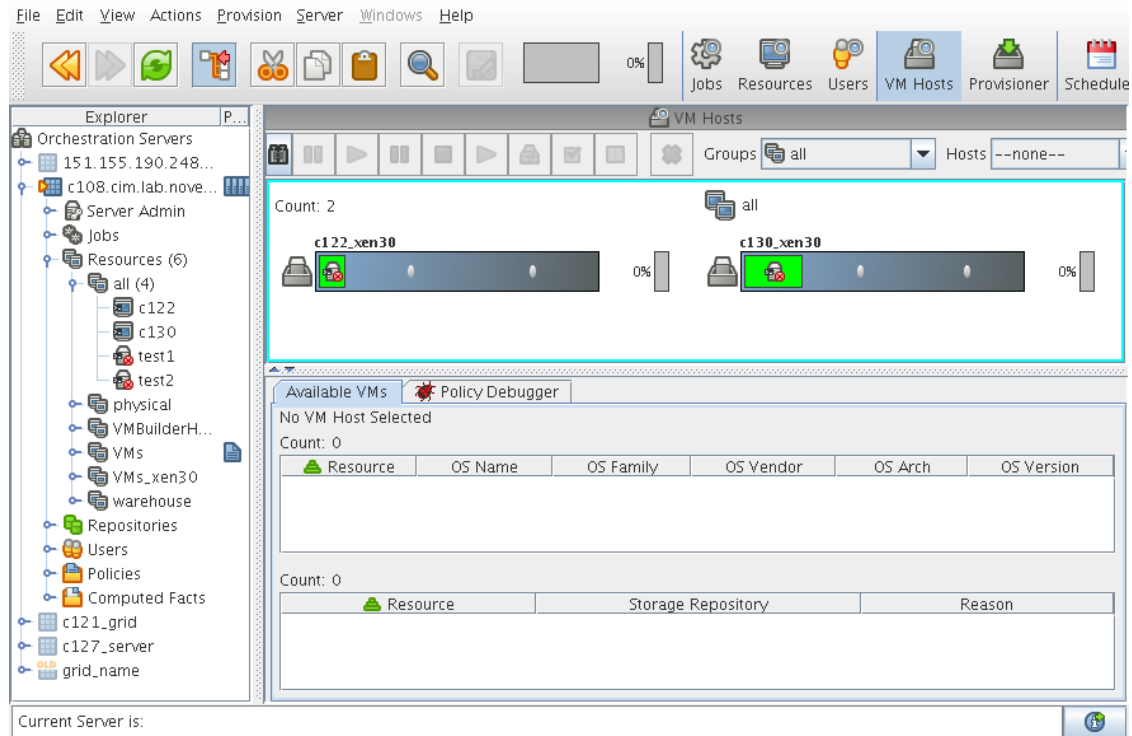
For an example of this job, see [sweeper.job \(page 166\)](#).

VM operations are available on the [ResourceInfo \(page 237\)](#) grid object, and VmHost operations are available on the [VMHostInfo \(page 243\)](#) grid object. In addition, as shown in [Section 9.5.3, “Provisioning Example,” on page 100](#), three provisioner events are fired when a provision action has completed, failed, or cancelled.

The API is equivalent to the actions available within the Orchestrator management console. 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 Orchestrator 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 Orchestrator console.

Figure 9-4 VM Host Management



9.5.2 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>
```

9.5.3 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 9.4, “Manual Management of a VM Server’s Lifecycle,” on page 94](#) and [Section 9.6, “Automatically Provisioning a VM Server,” on page 100](#).

9.6 Automatically Provisioning a VM Server

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
```

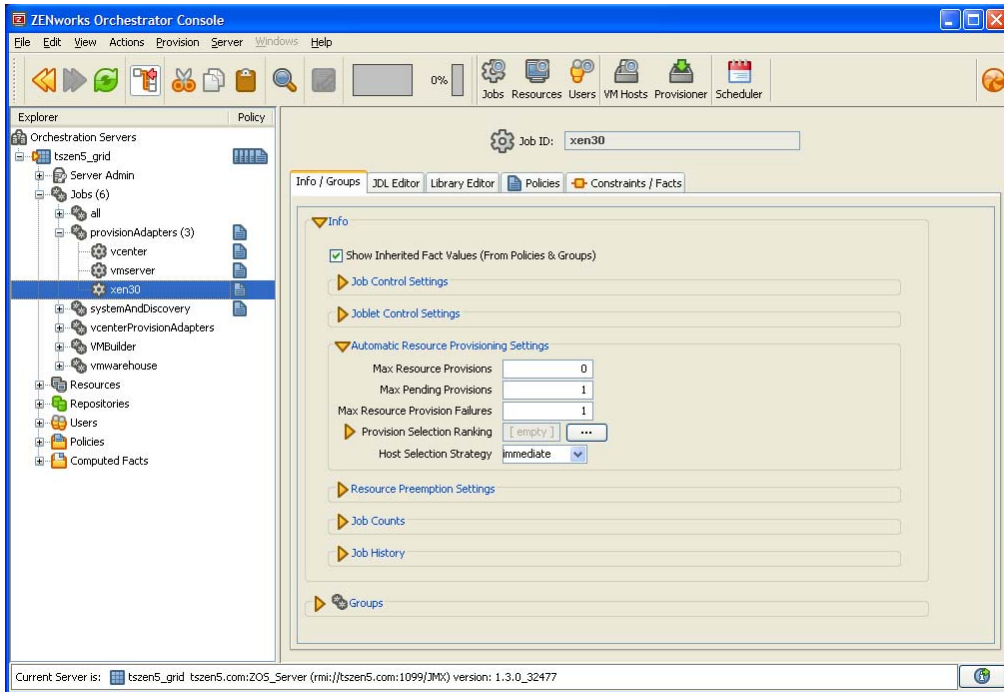
These are the job facts to enable and configure the usage of virtual machines for resource allocation. These facts can be set in a job’s policy.

For example, setting the provision.maxcount 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 ZENworks Orchestrator Console settings to use VMs:

Figure 9-5 Job Settings for VM Provisioning



Specifying Reservations

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, then 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 \(page 235\)](#) class.

9.7 Defining Values for Grid Objects

The following sections describe the ZENworks Orchestrator Server grid objects and facts that are required for provisioning of Orchestrator resource objects: This section highlights the facts that are expected to be set from a virtual machine discovery.

- ◆ [Section 9.7.1, “Orchestrator Grid Objects,”](#) on page 102
- ◆ [Section 9.7.2, “Repository Objects and Facts,”](#) on page 102
- ◆ [Section 9.7.3, “VmHost Objects and Facts,”](#) on page 109
- ◆ [Section 9.7.4, “VM Resource Objects and Other Base Resource Facts,”](#) on page 113
- ◆ [Section 9.7.5, “Physical Resource Objects and Additional Facts,”](#) on page 120

9.7.1 Orchestrator Grid Objects

The following table explains the abbreviated codes used to define the Orchestrator grid objects and facts listed in the following sections:

Table 9-1 *Orchestrator Grid Object Definitions*

Value	Description
Automatic	The fact should be automatically set after the successful discovery of virtual resources (VmHosts 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.
Warehouse	Facts relate to warehouse object types.

9.7.2 Repository Objects and Facts

Facts marked with an X designate that they should be automatically set after the successful discovery of virtual resources (VmHosts and VMs). Unless marked with the ° symbol, all of the following repository objects and facts must be set for the particular provisioning adapter to function. Facts marked with °° indicate the fact is required under certain conditions.

Table 9-2 Repository Objects and Facts

Fact Name	Description	Fact Type	Type: X = automatically set \circ = Not necessary to be set $\circ\circ$ = Required under certain conditions
repository.capacity	The maximum amount of storage space available to virtual machines (in megabytes). The value -1 means unlimited.	Integer	<p>Local: Note: Not auto discovered, but set to a default value of -1 (unlimited size). The Administrator should alter this value.</p> <p>This fact is not currently applicable to SAN because you cannot move file-based disks into a SAN.</p> <p>SAN: Note: Not auto discovered, but set to a default value of -1 (unlimited size). The Administrator should alter this value.</p> <p>nas: Note: Not auto discovered, but set to a default value of -1 (unlimited size). The Administrator should alter this value.</p> <p>Warehouse: Note: Not autodiscovered, but set to a default value of -1' (unlimited size). The Administrator should alter this value.</p> <p>datagrid: Note: Not auto discovered, but set to a default value of -1 (unlimited size). The Administrator should alter this value.</p> <p>virtual: Note: Not auto discovered, but set to a default value of -1 (unlimited size). The Administrator should alter this value.</p>
repository.searchpath	The relative path from the location to search for VM configuration files, which implicitly includes repository.image.preferredpath.	String []	<p>Local: X. [etc/xen/vm, myimages]</p> <hr/> <p>NOTE: The path is relative to repository.location or the leading '/' is ignored.</p> <hr/> <p>SAN: \circ.</p> <p>nas: X. ["my_vms", "saved_vms"] or [""] Specifies to search the whole mount.</p> <hr/> <p>NOTE: The path is either relative to repository.location; the leading '/' ignored.</p> <hr/> <p>Warehouse: N/A</p> <p>datagrid: N/A</p> <p>virtual: N/A</p>

Fact Name	Description	Fact Type	Type: X = automatically set ^o = Not necessary to be set ^{oo} = Required under certain conditions
repository.description	The description of the repository.	String	Local: ^o Default empty. SAN: ^o . nas: ^o . Warehouse: ^o Default empty. datagrid: ^o Default empty. virtual: ^o Default empty.
repository.encyency	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 a score. Thus, thus 0 means no cost and is very efficient).	Real	Local: Defaults to 1, which normalizes the transfer efficiency for moving VM disks. SAN: ^o Defaults to 1, which normalizes the transfer efficiency for moving VM disks. Not currently applicable because file-based disks cannot be moved into a SAN. nas: Defaults to 1, which normalizes the transfer efficiency for moving VM disks. Warehouse: Defaults to 1, which normalizes the transfer efficiency for moving VM disks. datagrid: Defaults to 1, which normalizes the transfer efficiency for moving VM disks. virtual: Defaults to 1, which normalizes the transfer efficiency for moving VM disks.
repository.enabled	True if the Repository is enabled, meaning that new VM instances can be provisioned.	Boolean	Local: Defaults to true. SAN: Defaults to true. nas: Defaults to true. Warehouse: Defaults to true. datagrid: Defaults to true. virtual: Defaults to true.

Fact Name	Description	Fact Type	Type: X = automatically set ^o = Not necessary to be set ^{oo} = Required under certain conditions
repository.freespace	The amount of storage space available to new virtual machines (in megabytes). The value -1 means unlimited.	Integer	Local: Dynamic: (capacity—used space) or -1 if capacity is unlimited. SAN: Dynamic: (capacity—used space) or -1 if capacity is unlimited. nas: Dynamic: (capacity—used space) or -1 if capacity is unlimited. Warehouse: Dynamic: (capacity—used space) or -1 if capacity is unlimited. datagrid: Dynamic: (capacity—used space) or -1 if capacity is unlimited. virtual: Dynamic: (capacity—used space) or -1 if capacity is unlimited.
repository.groups	The groups this Repository is a member of.	String[]	Local: X SAN: X nas: X Warehouse: X virtual: X
repository.id	The repository's unique name.	String	Local: X SAN: X nas: X Warehouse: X. This fact is autogenerated as warehouse, but could add others manually. datagrid: X. Currently one datagrid repository is supported. virtual: X

Fact Name	Description	Fact Type	Type: X = automatically set ° = Not necessary to be set °° = Required under certain conditions
repository.preferredpath	The relative path from the location to search and place VM files for movement and cloning.	String	Local: X. "var/lib/xen/images" <hr/> NOTE: The path is relative to repository.location; the leading '/' is ignored. <hr/> SAN: nas: X. "my_vms" <hr/> NOTE: The path is relative to repository.location; the leading '/' is ignored. <hr/> Warehouse: N/A datagrid: N/A virtual: N/A
repository.location	The Repository's physical location.	String	Local: X. "/" or /var/xen/images. SAN: °. nas: X. /u or /mnt/myshareddisk. <hr/> NOTE: This is the "mount point," which is assumed to be the same mount point on every host that has a connection to this NAS. <hr/> Warehouse: X. grid:///warehouse datagrid: °X. grid:///vms virtual: N/A
repository.provisioner.jobs	The names of the provisioning adapter jobs that can manage VMs on this repository.	String []	Local: X. ["xen30"] SAN: nas: X. ["xen30"] Warehouse: X. ["xen30"] datagrid: X. ["xen30"] virtual: X. ["vcenter"]

Fact Name	Description	Fact Type	Type: X = automatically set ° = Not necessary to be set °° = Required under certain conditions
repository.san.type	The type of SAN (Adapter specific, "iscsi", or "fibrechannel" or "" if not applicable.	String (enum)	Local: N/A, empty. SAN: Administrator must set to "iqn", "npiv", or "emc." nas: N/A, empty. Warehouse: N/A, empty. datagrid: N/A, empty. virtual: N/A, empty.
repository.san.vendor	The vendor of the SAN. Controls which storage bind logic to run (e.g. LUN masking, etc.).	String	Local: N/A, empty. SAN: Administrator must set to "iscsi" or "fibrechannel." nas: N/A, empty. Warehouse: N/A, empty. datagrid: N/A, empty. virtual: N/A, empty.
repository.type	The type of repository: <ul style="list-style-type: none"> ◆ Local; e.g. a local disk. ◆ nas; e.g. a NFS mount. ◆ san , datagrid: A Orchestrator built in datagrid backed store. ◆ warehouse: A ZENworks managed storage. ◆ virtual: An externally managed VM; e.g. VMWare Virtual Center. 	String (enum)	Local: X. Local SAN: nas: Warehouse: X. Warehouse datagrid: X. Datagrid virtual: X. Virtual

Fact Name	Description	Fact Type	Type:
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
repository.usedspace	The amount of storage space used for virtual machines.	Integer	Local: Dynamic: Sum of disk space used by contained VMs. Only includes disks that are stored as local files (not partitions). SAN: Dynamic: Sum of disk space used by contained VMs. Only includes disks that are stored as local files (not partitions). Not currently applicable to SAN because you cannot move file-based disks into SAN. nas: Dynamic: Sum of disk space used by contained VMs. Only includes disks that are stored as local files (not partitions). Warehouse: Dynamic: Sum of disk space used by contained VMs. Only includes disks that are stored as local files (not partitions). datagrid: Dynamic: Sum of disk space used by contained VMs. Only includes disks that are stored as local files (not partitions). virtual: Dynamic: Sum of disk space used by contained VMs. Only includes disks that are stored as local files (not partitions).
repository.vmhosts	The list of VM hosts capable of using this repository (aggregated from the individual VM host fact).	String []	Local: X SAN: nas: X Warehouse: X datagrid: X virtual: X
repository.vmimages	The list of VM images stored in this repository (aggregated from individual VM fact).	String []	Local: X SAN: nas: X Warehouse: X datagrid: X virtual: X

9.7.3 VmHost Objects and Facts

Unless marked with a “°” symbol, all of the following VmHost objects and facts must be set for the particular provisioning adapter to function. The “X” mark designates that the fact should be automatically set after the successful discovery of virtual resources (VmHosts and VMs).

Table 9-3 *VmHost Objects and Facts*

Fact Name	Description	Fact Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
vmhost.accountinggroup	The default vmhost (resource) group which is adjusted for VM statistics.	String	xen30: X. All. vmserver: X. All. vcenter: X. All.
vmhost.enabled	True if the VM host is enabled, which enables new VM instances to be provisioned.	Boolean	xen30: X. True. vmserver: X. True. vcenter: X. True.
vmhost.groups	The groups this VM host is a member of. Alias for 'vmhost.resource.group'.	String []	xen30: X. vmserver: X. vcenter: X.
vmhost.id	The VM host's unique name.	String	xen30: X. <physical host id>_xen30 vmserver: X. <physical host id>_vmserver vcenter: X. <physical host id>_vcenter
vmhost.loadindex.slots	The loading index; the ratio of active hosted VMs to the specified maximum.	Dynamic Real	xen30: X. vmserver: X. vcenter: X.
vmhost.loadindex.virtual memory	The loading index; the ratio of consumed memory to the specified maximum.	Dynamic Real	xen30: X. vmserver: X. vcenter: X.
vmhost.location	The VM host's physical location.	String	xen30: °Defaults to empty string. vmserver: °Defaults to empty string. vcenter: X. Virtual center's 'locator' to the Vmhost; e.g., "/vcenter/eng/esx1".

Fact Name	Description	Fact Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
vmhost.maxvmslots	The maximum number of hosted VM instances.	Integer	xen30: Defaults to 3. Should be reset by Administrator. vmserver: Defaults to 3. Should be reset by Administrator. vcenter: Defaults to 3. Should be reset by Administrator.
vmhost.memory.available	The amount of memory available to new virtual machines.	Dynamic Integer	xen30: X. Calculated to be 'vmhost.memory.max; the memory consumed by running VMs. vmserver: X. Calculated to be 'vmhost.memory.max; the memory consumed by running VMs. vcenter: X. Calculated to be 'vmhost.memory.max; the memory consumed by running VMs.
vmhost.memory.max	The maximum amount of memory available to virtual machines (in megabytes).	Integer	xen30: X. Discovered. vmserver: X. Discovered. vcenter: X. Discovered.
vmhost.migration	True if the VM host can support VM migration; also subject to provision adapter capabilities.	Boolean	xen30: X. Defaults to false. Not discovered. Administrator should enable as appropriate to indicate that the VmHost supports migration. vmserver: X. Defaults to false. Not discovered. Should not be set to true since vmserver/gsx does not support migration. vcenter: X. Discovered.
vmhost.provisioner.job	The name of the provisioning adapter job that manages VM discovery on this host.	String	xen30: X. xen30. vmserver: X. vmserver. vcenter: X. vcenter.
vmhost.provisioner.password	The password required for provisioning on the VM host. This fact is used by the provisioning adapter.	String	xen30: °. vmserver: °. If set, this fact is passed to the vmserver CLI tools to authenticate. Not necessary if Orchestrator agent is run as root. vcenter: °.

			Provision Adapter
Fact Name	Description	Fact Type	X = automatically set ^o = Not necessary to be set ^{oo} = Required under certain conditions
vmhost.provisioner.user name	The username required for provisioning on the VM host. This fact is used by the provisioning adapter.	String	xen30: ^o . vmserver: If set, is passed to vmserver CLI tools to authenticate. Not necessary if Orchestrator agent is run as root. vcenter: ^o .
vmhost.repositories	This list of repositories (VM disk stores) is visible to this VM host.	String []	xen30: X. Discovery only adds the local repository, the datagrid and the warehouse on the first creation of the vmhost. Administrator is required to add SAN/NAS repositories or remove local if desired. vmserver: X. Discovery only adds the local repository, the datagrid and the warehouse on the first creation of the vmhost. Administrator is required to add SAN/NAS repositories or remove local if desired. vcenter: X. Automatically set to VirtualCenter. <hr/> NOTE: This is the only sensible setting.
vmhost.resource	The name of the resource that houses this VM host container.		xen30: ^o . vmserver: X. vcenter: X.
vmhost.shuttingdown	True if the VM host is attempting to shut down and does not need to be provisioned.	Dynamic Boolean	xen30: Initially False, then set to True when the administrator specifies to shut down a host. vmserver: Initially False, then set to True when the administrator specifies to shut down a host. vcenter: Initially False, then set to True when the administrator specifies to shut down a host.

Fact Name	Description	Fact Type	Provision Adapter
			X = automatically set ^o = Not necessary to be set ^{oo} = Required under certain conditions
vmhost.vm.available.groups	The list of resource groups containing VMs that are allowed to run on this host.	String []	xen30: X. Automatically set to VMs_<provisioning_adapter>; e.g., any VM of compatible type can be provisioned. The VMs_<provisioning_adapter> group is automatically created by discovery. The administrator can refine this by creating new groups and editing if further restrictions are required. vmserver: X. Automatically set to VMs_<provisioning_adapter>; e.g., any VM of compatible type can be provisioned. The VMs_<provisioning_adapter> group is automatically created by discovery. The administrator can refine this by creating new groups and editing if further restrictions are required. vcenter: X. Discovery attempts to map Virtual Center grouping to Orchestrator resources groups and sets this fact accordingly. This also includes a special "template_vcenter" group to map to Virtual Center 1.3.x "templates".
vmhost.vm.count	The current number of active VM instances.	Dynamic Integer	xen30: X. vmserver: X. vcenter: X.
vmhost.vm.instanceids	The list of active VM instances.	Dynamic String[]	xen30: X. vmserver: X. vcenter: X.
vmhost.vm.templatecounts	A dictionary of running instance counts for each running VM template.	Dynamic Dictionary	xen30: X. vmserver: X. vcenter: X.
vmhost.xen.bits	xen30 only. Legal values are 32 and 64.	Integer	xen30: X. 64. vmserver: ^o Not defined. vcenter: ^o Not defined.
vmhost.xen.hvm	xen30 only.	Boolean	xen30: X. True. vmserver: ^o Not defined. vcenter: ^o Not defined.

Fact Name	Description	Fact Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
vmhost.xen.version	xen30 only: Major.Minor version of the Xen hypervisor.	Real	xen30: X. 3.00 vmserver: °Not defined. vcenter: °Not defined.
vmhost.vcenter.hostname	vcenter only. The hostname of the resource containing this VM container. NOTE: Deprecated. Use 'vmhost.resource.hostname' instead.	String	xen30: °Not defined. vmserver: °Not defined. vcenter: X. esx1.
vmhost.vcenter.networks	vcenter only. List of network interfaces on the physical host.	List	xen30: °Not defined. vmserver: °Not defined. vcenter: VM network.
vmhost.vcenter.grouppath	vcenter only: Part of the Virtual Center "locator" URL.	List	xen30: °Not defined. vmserver: °Not defined. vcenter: X. /vcenter/eng1.

9.7.4 VM Resource Objects and Other Base Resource Facts

The following virtual machine resource objects and additional base resource facts marked with the “•” symbol must be set for the particular provisioning adapter to function. Facts marked with “••” indicate the fact is required under certain conditions. The “X” character designates that the fact should be automatically set after the successful discovery of virtual resources (VmHosts and VMs).

Table 9-4 Resource Objects (VM only) and Additional Facts to Base Resource Facts
resource.provisioner.warehouse.guidresource.provisioner.warehouse.guid

Fact Name	Description	Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
resource.provision.automatic	Signifies that this resource was cloned/provisioned automatically and thus is shut down/destroyed automatically as well.	Dynamic Boolean	xen30: ° . vmserver: ° . vcenter: ° .

Fact Name	Description	Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
resource.provision.auto prep.*	Fact namespace used to convey configuration information actually used to "personalize" this VM instance.	<various >	xen30: ° X. Can be set when rediscovering the state or as a result of a migration or provision action. vmserver: ° X. Can be set when rediscovering the state or as a result of a migration or provision action. vcenter: ° X. Can be set when rediscovering the state or as a result of a migration or provision action.
resource.provision.currentaction	The current management action in progress on this provisionable resource.c.	Dynamic String	xen30: ° . vmserver: ° . vcenter: ° .
resource.provision.host wait	The time (seconds) this resource has been waiting / waited for a suitable host.	Dynamic Integer	xen30: ° . vmserver: ° . vcenter: ° .
resource.provision.jobid	The current or last job ID that performed a provisioning action on this resource. Useful for viewing the job log.	Dynamic String	xen30: ° . vmserver: ° . vcenter: ° .
resource.provision.resync	Specifies that the provisioned resource's state needs to be re-synced with the underlying provisioning technology at the next opportunity.	Dynamic Boolean	xen30: °X. Can be set on discovery when the Orchestrator state machine mismatches the VM state. This initiates a future VM state recovery action ("Check Status"). May be set for delayed re-discovery by administrator or JDL logic. vmserver: °X. Can be set on discovery when the Orchestrator state machine mismatches the VM state. This initiates a future VM state recovery action ("Check Status"). May be set for delayed re-discovery by administrator or JDL logic. vcenter: °X. Can be set on discovery when the Orchestrator state machine mismatches the VM state. This initiates a future VM state recovery action ("Check Status"). May be set for delayed re-discovery by administrator or JDL logic.

Fact Name	Description	Type	Provision Adapter		
			X = automatically set	° = Not necessary to be set	°° = Required under certain conditions
resource.provision.state	The current state of this provisioned instance (down, suspended, up, paused) or unknown if an admin action is currently being performed.	Dynamic String (enum)	xen30: °	vmserver: °	vcenter: °
resource.provision.status	The current descriptive status of the provisioned resource.	Dynamic String	xen30: °	vmserver: °	vcenter: °
resource.provision.template	The ID of the template resource that this instance was created from (if applicable).	Dynamic String	xen30: °	vmserver: °	vcenter: °
resource.provision.time.request	The time when the last provision (or other administrative action) request was made.	Dynamic Date	xen30: °	vmserver: °	vcenter: °
resource.provision.time.shutdown	The time when the resource was last shut down.	Dynamic Date	xen30: °	vmserver: °	vcenter: °
resource.provision.time.start	The time when the resource was last successfully provisioned.	Dynamic Date	xen30: °	vmserver: °	vcenter: °
resource.provision.vmhost	The ID of the host currently housing this provisioned resource.	Dynamic String	xen30: °	vmserver: °	vcenter: °
resource.provisionable	True if the resources is a provisionable type.	Dynamic Boolean	xen30: °	vmserver: °	vcenter: °
resource.provisioner.autoprep.*	Fact namespace used to convey configuration information actually used to "personalize" this VM instance.	<various >	xen30: ° X (only if set in warehouse).	vmserver: °	vcenter: °

Fact Name	Description	Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
resource.provisioner.count	The total count of operational instances and provisions in progress"	Dynamic Integer	xen30: ° . vmserver: ° . vcenter: ° .
resource.provisioner.debug	Controls the debug log level in the provisioner.	Boolean	xen30: ° . vmserver: ° . vcenter: ° .
resource.provisioner.host.maxwait	The maximum time to wait for a suitable host before timing out (in seconds, '<0' to wait indefinitely).	Integer	xen30: ° . vmserver: ° . vcenter: ° .
resource.provisioner.host.preferredwait	The time after which some VMhost constraints is lifted to increase the available pool by, for example, considering moving the disk image (in seconds, <0 to wait indefinitely).	Integer	xen30: ° . vmserver: ° . vcenter: ° .
resource.provisioner.instances	The list of id's of the instances of this template resource (if applicable).	String[]	xen30: ° . vmserver: ° . vcenter: ° .
resource.provisioner.job	The name of the provisioning job that manages the life cycle of this resource.	String	xen30: X. xen30 vmserver: X. vmserver vcenter: X. vcenter
resource.provisioner.maxinstances	The maximum allowed number of instances of this provisionable resource (applicable only to templates).	Integer	xen30: X. Defaults to 1. Administrator should reset for VM templates to allow multiple clones. vmserver: X. Defaults to 1. Administrator should reset for VM templates to allow multiple clones. vcenter: X. Defaults to 1. Administrator should reset for VM templates to allow multiple clones.

Fact Name	Description	Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
resource.provisioner.recommendedhost	The host on which the image for this resource is associated; e.g., was suspended or is the preferred host for quick startup.	String	xen30: ° X. vmserver: ° X. vcenter: ° X.
resource.provisioner.warehouse.guid	The warehouse ID of this VM.	String	xen30: °° X. Required only if resource.provisioner.repository is 'warehouse' (guid from warehouse). vmserver: N/A vcenter: N/A.
resource.provisioner.warehouse.version	The warehouse version number of this VM.	Integer	xen30: °° X. Required only if resource.provisioner.repository is 'warehouse' {version from warehouse}. vmserver: N/A. vcenter: N/A.
resource.vcenter.groupname	Locator for the Virtual Center group that the VM resides in.	String	xen30: ° Not defined. vmserver: ° Not defined. vcenter: X. /vcenter/eng
resource.vcenter.guestOS	VMWare's name for the guest OS.	String	xen30: ° Not defined. vmserver: ° Not defined. vcenter: X. winNetEnterprise.
resource.vcenter.imagepath	Locator for the VM in Virtual Center.	String	xen30: ° Not defined. vmserver: ° Not defined. vcenter: X. /vcenter/eng/windows2003ent.
resource.vm.basepath	The file system location of the VM files either absolute or relative to the 'repository.location' fact.	String	xen30: X. Example: "var/lib/xen/images/sles10". vmserver: X. For example, "/var/lib/vmware/Virtual-Machines/sles9". Location in the repository of the directory containing VM disks, configuration file and other related files. vcenter: ° N/A.

Fact Name	Description	Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
resource.vm.configfile	The location of the VM's configuration file inside of the default repository (resource.provisioner.repository).	String	xen30: X. /etc/xen/vm/sles10. vmserver: ° Not currently used. vcenter: ° N/A.
resource.vm.cpu.architecture	The required cpu architecture e.g. x86, x86_64, sparc.	String	xen30: ° X (only if set in warehouse). vmserver: ° . vcenter: ° .
resource.vm.cpu.weight	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.	Real	xen30: ° . vmserver: ° . vcenter: ° .
resource.vm.files	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'.	Dictionary	xen30: X. { "mof": /var/lib/xen/images/sles10/mof", "suspendcheckpoint": "/var/lib/xend/domain/checkpoint", "config": "/var/lib/xen/images/sles10/config.xen" }. vmserver: ° X { "config": "/var/lib/vmware/Virtual Machines/sles10/sles10.vmx" }. vcenter: ° N/A.
resource.vm.maxinstancespervmhost	The maximum allowed number of instances of this VM image per vmhost.	Integer	xen30: Defaults to 1. Administrator should increase if more than one instance of the same VM template is allowed to be run on one host.
resource.vm.memory	The configured virtual memory requirement of this VM image (megabytes).	Integer	xen30: X. vmserver: X. vcenter: X.
resource.vm.preventmove	Set by the administrator to prevent relocation of a VM (disk moves) even if possible.	Boolean Default: False	
resource.vm.type	The required system type of a virtual machine ('full' or 'para').	String	xen30: X. vmserver: X. vcenter: X.

Fact Name	Description	Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
resource.vm.uuid	The UUID of a virtual machine (vendor/adaptor specific).	String	xen30: X. {vm uuid} vmserver: Not currently used. vcenter: Not currently used.
resource.vm.vcpu.number	The number of virtual CPUs for this VM.	Integer	xen30: ° X. vmserver: ° . vcenter: ° .
resource.vm.vdisks	The specification of virtual disks that make up this VM. The dictionary keys are name (String), repository (String), location (String), size (Integer), fixed (Boolean).	List of Dictionaries	xen30: X. [{ "location": "/var/lib/xen/images/sles10/disk1", "moveable": True, "repository": "vmhost1" ... }]. vmserver: ° Not currently used. vcenter: ° N/A.
resource.vm.vdisksize	The total size of all the moveable virtual desks for this VM image (megabytes).	Integer	xen30: X. vmserver: X. vcenter: X.
resource.vm.vendor	The vendor of a virtual machine.	String	xen30: ° X. vmserver: ° X. vcenter: ° X.
resource.vm.version	The version number for this VM.	Integer	xen30: X (only if set in warehouse). Required only if resource.provisioner.repository is 'warehouse'. vmserver: ° N/A. vcenter: ° N/A.
resource.vm.vmhost.ranking	The ranking specification used to select suitable vm hosts. Element syntax is <fact>/<order> where order is either a (ascending) or d (descending).	String[]	xen30: Defaults to vmhost.vm.placement.score/a, vmhost.loadindex.slots/a. vmserver: Defaults to vmhost.vm.placement.score/a, vmhost.loadindex.slots/a. vcenter: Defaults to vmhost.vm.placement.score/a, vmhost.loadindex.slots/a.

Fact Name	Description	Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
resource.vnc.ip	The host IP address for a VNC session running on the resource. NOTE: Technically, this fact is available on all resources both VMs and physical.	String	xen30: ° X. 192.168.0.4 vmserver: Not used. vcenter: ° .
resource.vnc.port	The port number for a VNC session running on the resource. NOTE: Technically, this fact is available on all resources both VMs and physical.	Integer	xen30: ° X. 5900 vmserver: Not used. vcenter: ° .

9.7.5 Physical Resource Objects and Additional Facts

The following physical resource objects and additional base resource facts marked with the “•” symbol must be set for the particular provisioning adapter to function. The physical resources have the potential of creating VmHost containers.

Facts marked with “••” indicate the fact is required under certain conditions. The “X” character designates that the fact should be automatically set after the successful discovery of virtual resources (VmHosts and VMs).

Table 9-5 Resource Object (Physical that have the potential for VmHost containers) / Additional Facts (additional to base resource set) *resource.provisioner.warehouse.guid*

Fact Name	Description	Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
resource.vcenter.client	vcenter only: Marks resources and Virtual Center web services client capable.	Boolean	xen30: ° Not defined. vmserver: ° Not defined. vcenter: Administrator must set through association of 'vcenter_client.policy' with appropriate resources.

Fact Name	Description	Type	Provision Adapter
			X = automatically set ° = Not necessary to be set °° = Required under certain conditions
resource.vmserver.cmdpath	vmserver only: Path to VMWare CLI tools.	String	xen30: ° Not defined. vmserver: X. For example, "/usr/bin/vmware-cmd" vcenter: ° Not defined.
resource.vmserver.localrepositories	vmserver only: Paths to VM storage directories.	List	xen30: ° Not defined. vmserver: For example, "/var/lib/vmware/virtual machines" vcenter: ° Not defined.
resource.vmserver.vmruntimepath	vmserver only: Full path to vmrun CLI tool.	String	xen30: ° Not defined. vmserver: X. For example, "/usr/bin/vmrun". vcenter: ° Not defined.
resource.xen	xen30 only: Xen enabled.	Boolean	xen30: X. True. vmserver: ° Not defined. vcenter: ° Not defined.
resource.xen.bits	xen30 only: (legal values are 32 and 64)	String	xen30: X. 64 bit. vmserver: ° Not defined. vcenter: ° Not defined.
resource.xen.hvm	xen30 only:	Boolean	xen30: X. True. vmserver: ° Not defined. vcenter: ° Not defined.
resource.xen.version	xen30 only: Major.Minor version of the Xen hypervisor.	Real	xen30: X. 3.00. vmserver: ° Not defined. vcenter: ° Not defined.

This section describes specific Job examples that can be deployed using Novell® ZENworks® Orchestrator Server. The following sections demonstrate some practical ways to use Orchestrator and should help you better understand how to write your own jobs:

- ♦ [Section 10.1, “Accessing Job Examples,” on page 123](#)
- ♦ [Section 10.2, “Installation and Getting Started,” on page 123](#)
- ♦ [Section 10.3, “Orchestrator Sample Job Summary,” on page 124](#)
- ♦ [Section 10.4, “Parallel Computing Examples,” on page 125](#)
- ♦ [Section 10.5, “General Purpose Jobs,” on page 137](#)
- ♦ [Section 10.6, “Miscellaneous Code-Only Jobs,” on page 178](#)

10.1 Accessing Job Examples

The basic examples delivered with Novell® ZENworks® Orchestrator 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 ZENworks Orchestrator 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 “[Walkthrough: Deploy a Sample Job](#)” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.

10.2 Installation and Getting Started

To run the ZENworks Orchestrator described in this section, use the following guidelines:

- ♦ Install and configure ZENworks Orchestrator properly (see “[Installing and Configuring ZENworks Orchestrator Components](#)” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*).
- ♦ Unless otherwise indicated, install at least one agent on a managed resource and have it running (see “[Installing the ZENworks Orchestrator Agent on VM Hosts and VMs](#)” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*).
- ♦ Before running `zosadmin` or `zos` commands, you must log into the Orchestrator 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 Orchestrator 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 *Novell ZENworks Orchestrator 1.3 Administration Guide*.

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

- ◆ For an explanation of `zos` commands, see “[The Zos Command Line Tool](#)” in the *Novell ZENworks Orchestrator 1.3 Job Management Guide*.

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

You could create a user (see “[Walkthrough: Create a User Account](#)” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*) but `zos login --user=vmmanger` works with the account created by default during installation.

10.3 Orchestrator Sample Job Summary

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

Table 10-1 ZENworks Orchestrator Job Development Examples

Example Name	Job Function Capabilities
demolterator.job (page 126)	<ul style="list-style-type: none"> ◆ 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 138)	<ul style="list-style-type: none"> ◆ Downloading files stored on grid management servers to networked nodes.
factJunction.job (page 179)	<ul style="list-style-type: none"> ◆ Retrieving information about objects in the grid relative to another object.
failover.job (page 148)	<ul style="list-style-type: none"> ◆ Managing how joblets failover to enhance the robustness of your jobs.
instclients.job (page 154)	<ul style="list-style-type: none"> ◆ Installing a ZOS client on multiple machines. ◆ Provides an example of executing a command on a resource.
jobargs.job (page 188)	<ul style="list-style-type: none"> ◆ 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 161)	<ul style="list-style-type: none"> ◆ Understanding how to launch specific applications on specified resources.

Example Name	Job Function Capabilities
quickie.job (page 133)	<ul style="list-style-type: none">◆ Understanding how jobs can start multiple instances of a joblet on one or more resources.
sweeper.job (page 166)	<ul style="list-style-type: none">◆ Understanding how poll all resources on the grid.an ordered serialized scheduling of the joblets
whoami.job (page 173)	<ul style="list-style-type: none">◆ Sending a command to the operating system's default command interpreter. On Microsoft Windows, this is <code>cmd.exe</code>. On POSIX systems, this is <code>/bin/sh</code>.

10.4 Parallel Computing Examples

The following examples demonstrate high performance or parallel computing concepts:

- ◆ [“demoIterator.job” on page 126](#)
- ◆ [“quickie.job” on page 133](#)

demolterator.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.

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

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

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

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

Description

The files that make up the DemoIterator job include:

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

demolterator.jdl

```
1 # -----
-
2 # Copyright © 2008 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-INFRINGEMENT. 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,v 1.4 2008/03/05 20:05:48 ray Exp $
13 # -----
-
14
15 import time, random
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 demoIteratorJob(Job):
33
34     def job_started_event(self):
35         print 'job_started_event'
36         self.completed = 0
37
38         # Launch the joblets
39         numJoblets = self.getFact("jobargs.numJoblets")
40         print 'Launching ', numJoblets, ' joblets'
41
42         pspace = ParameterSpace()
43         i = 1
44         while i <= numJoblets:
45             pspace.appendRow({'name':'joblet'+str(i)})
46             i += 1
47         pspace.maxJobletSize = 1
48         self.schedule(demoIteratorJoblet,pspace,{})
49
50     def joblet_completed_event(self, jobletnumber, node):
51         self.completed += 1
52         self.setFact("jobinstance.memo", "Tests run: %s" % (self.completed))
53
54
55 class demoIteratorJoblet(Joblet):
56
57     def joblet_started_event(self):
58         print "Hi from joblet ", self.getFact("joblet.number")
59         time.sleep(random.random() * 15)
60
61         cmd = self.getFact("jobargs.cmd")
62         if len(cmd) > 0:
63             system(cmd)
64

```

```

65
66
67     # Example of more sophisticated exec
68     # e.g. e.signal("SIGUSR1")
69     """
70     e = Exec()
71     e.setCommand(cmd)
72     #e.setStdoutFile("cmd.out")
73     e.writeStdoutToLog()
74     e.writeStderrToLog()
75     #try:
76     e.execute()
77     #except:
78         #self.retry("retryable example error")
79     """

```

demolterator.policy

```

1  <!--
2
3  =====
4  * Copyright (c) 2008 Novell, Inc. All Rights Reserved.
5  *
6  * NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED
7  * WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY,
8  * FITNESS FOR A PARTICULAR PURPOSE, AND NON INFRINGEMENT. NOVELL, THE AUTHORS
9  * OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE
10 * FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF
11 CONTRACT,
12 * TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE
13 * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
14
15 =====
16 * $Id: demoIterator.policy,v 1.2 2008/02/27 20:49:34 john Exp $
17
18 =====
19 -->
20
21 <policy>
22   <constraint type="accept" reason="Too busy for more work. Try again later!">
23     <or>
24       <lt fact="job.instances.queued" value="4" />
25       <contains fact="user.groups" value="superuser" />
26     </or>
27   </constraint>
28
29   <constraint type="start" reason="Waiting on queue">
30     <or>
31       <lt fact="job.instances.active" value="2" />
32       <contains fact="user.groups" value="superuser" />
33     </or>
34   </constraint>
35
36   <jobargs>
37     <fact name="numJoblets"
38       type="Integer"
39       description="joblets to run"
40       value="100"

```



```

37         visible="true" />
38
39     <fact name="cmd"
40         type="String"
41         description="Simple command to execute"
42         value="" />
43
44     <fact name="os"
45         type="String"
46         description="Regular expression match for Operating System"
47         value=".*" />
48
49     <fact name="cpu"
50         type="String"
51         description="Regular expression match for CPU architecture"
52         value=".*" />
53 </jobargs>
54
55 <constraint type="resource" reason="Does not match">
56     <and>
57         <eq fact="resource.os.family" factvalue="jobargs.os" match="regexp" />
58         <eq fact="resource.cpu.architecture" factvalue="jobargs.cpu"
match="regexp"/>
59
60     <or>
61         <and>
62             <defined fact="env.VENDOR" />
63             <eq fact="resource.os.vendor" factvalue="env.VENDOR" match="regexp"
/>
64         </and>
65         <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” on page 130](#)
- ♦ [“job_started_event” on page 130](#)
- ♦ [“joblet_started_event” on page 131](#)

zosadmin deploy

The deployment for the DemoIterator job is performed by lines 3-15 of [demoIterator.jdl \(page 126\)](#). 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 ZENworks Orchestrator Console in the Explorer view at the location:

```
/ZOS/YOUR_GRID/Jobs/examples
```

For a general overview of how jobs are added to groups during deployment, see [“Walkthrough: Deploy a Sample Job”](#) in the [“Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide.”](#)

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 27-31 in [demoIterator.jdl \(page 126\)](#)). 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 32), meaning a joblet instance is created for each row in the ParameterSpace during job scheduling (see line 33).

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 \(page 233\)](#).

The resource that runs the joblet is determined from the resource constraint specified in lines 2-14 and 39-52 of [demoIterator.policy \(page 128\)](#), 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 2-14 in [demoIterator.policy \(page 128\)](#) 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 [“Defining Job Elements” on page 77](#).

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 35-37 in [demoIterator.jdl \(page 126\)](#)). You can see the text for the memo displayed on the Job Log tab in the list of running jobs in the ZENworks Orchestrator Console.

For more information, see [“Starting and Stopping the ZENworks Orchestrator Console”](#).

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=notepad 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 4.4, “Policy Management,”](#) on page 53 and [Chapter 5, “Developing Policies,”](#) on page 57).
- ♦ Adding Jobs to Groups During Deployment (see how the JDL code can print the ID of group of jobs in [factJunction.job](#) (page 179)).
- ♦ [quickie.job](#) (page 133) 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 3.1.1, “Naming Orchestrator Job Files,”](#) on page 43)
- ♦ Facts provided by the ZENworks Orchestrator system that can be referenced within a JDL file
- ♦ Using `zos`
- ♦ 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 © 2008 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-INFRINGEMENT. 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,v 1.3 2008/02/27 20:51:13 john Exp $
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 © 2008 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 INFRINGEMENT. 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,v 1.2 2008/02/27 20:51:13 john Exp $
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:

- ◆ “zosadmin deploy” on page 136
- ◆ “job_started_event” on page 136
- ◆ “joblet_started_event” on page 136

zosadmin deploy

The job is first deployed into the grid, as shown in lines 2-14 of [quickie.jdl \(page 133\)](#). 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 ZENworks Orchestrator Console in the Explorer view at the location:

```
/ZOS/YOUR_GRID/Jobs/examples
```

For a general overview of how jobs are added to groups during deployment, see “[Walkthrough: Deploy a Sample Job](#)” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.

job_started_event

As shown in line 25 of [quickie.jdl \(page 133\)](#), 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 3 in [quickie.policy \(page 134\)](#)).

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 31 in [quickie.jdl \(page 133\)](#)), 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 ([Section 4.4, “Policy Management,” on page 53](#) and [Chapter 5, “Developing Policies,” on page 57](#)).
- ♦ Adding jobs to groups during deployment (see how the JDL code can print the ID of group of jobs in [factJunction.job \(page 179\)](#)).
- ♦ Scheduling multiple instances of a joblet

10.5 General Purpose Jobs

The following examples demonstrate general purpose job concepts:

- ♦ [“dgtest.job” on page 138](#)
- ♦ [“failover.job” on page 148](#)
- ♦ [“instclients.job” on page 154](#)
- ♦ [“notepad.job” on page 161](#)
- ♦ [“sweeper.job” on page 166](#)
- ♦ [“whoami.job” on page 173](#)

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 ZENworks Orchestrator Server to a node. For additional background information, see [Section 3.1, “Defining the Datagrid,” on page 43](#).

Because it typically grows quite large, the physical location of the ZENworks Orchestrator root directory is important. Use the following procedure to determine the location of the datagrid in the Orchestrator server console:

- 1 Select the grid id on the left in the Orchestrator Explorer window >
- 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 ZENworks Orchestrator can be seen with the Console command:

```
> zos dir grid:///
<DIR>      Dec-6-2007 6:55 installs
<DIR>      Dec-6-2007 6:55 jobs
<DIR>      Dec-6-2007 22:01 users
<DIR>      Dec-6-2007 6:55 vms
<DIR>      Dec-6-2007 6:56 warehouse
```

Job Files

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 © 2008 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-INFRINGEMENT. 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,v 1.4 2008/03/05 20:05:43 ray Exp $
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 instance 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 quorum (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 © 2008 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 INFRINGEMENT. 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,v 1.2 2008/02/27 20:49:29 john Exp $
14
*=====
15 -->
16
17 <policy>
18     <jobargs>
19         <!--
20             Name of file that is stored in the Datagrid area to
21             download to the resource.
22
23             A value for this fact the 'zos run' is assigned when
24             using the 'zos run' command.
25         -->
26         <fact name="filename"
27             type="String"
28             description="The filename to download from the Datagrid"
29             />
30
31         <fact name="multicast"
32             type="Boolean"
33             description="Whether to download using multicast or unicast"
34             value="false" />
35     </jobargs>
36
37     <job>
38         <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 42 in [dgtest.jdl \(page 139\)](#) is derived from the [Job](#) class.

testnode

Class testnode (line 73 in [dgtest.jdl \(page 139\)](#) is derived from the [Joblet \(page 224\)](#) class.

Job Details

dgtest.job can be broken down into the following parts:

- ♦ “Policy” on page 144
- ♦ “zosadmin deploy” on page 144
- ♦ “job_started_event” on page 144
- ♦ “joblet_started_event” on page 145

Policy

In addition to describing the `filename` and `multicast` jobargs and the default settings for `multicast` (lines 3-22) in the `dgtest.policy` (page 142) file, there is the `<job/>` section (lines 24-33), which describes static facts (Section 5.1.2, “Facts,” on page 57). You must assign the `filename` argument when executing this example. This is only the name of the file in the “images” area of ZENworks Orchestrator. For example, for `grid:///images/disk.img`, just assign `disk.img` to the argument. This file must be in the ZENworks Orchestrator file system for fetching and delivering to remote nodes used in this example.

To populate the ZENworks Orchestrator, use the `zos copy` command. For example, for a file named `suse-9-flat.vmd` in the current directory, use the following command:

```
> zos mkdir grid:///images
> zos copy suse-9-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 in the `<job/>` section also describes a `resource.groups` constraint. (For more information, see “Constraints” on page 23). This requires a resource group named `dgtest` (lines 30-39 in `dgtest.policy` (page 142)) and that group should have member nodes. Consequently, you must create this resource group using the Orchestrator server console and assign it some nodes to run this example successfully.

zosadmin deploy

When the Orchestrator server deploys a job for the first time (see Section 7.5, “Deploying Jobs,” on page 75), the job JDL files are executed in a special deploy mode. Looking at `dgtest.jdl` (page 139), you might notice that when the job is deployed (line 30), either via the Orchestrator console or the `zosadmin deploy` command, that it attempts to find the `examples` jobgroup (lines 32-33), create it if missing (lines 34-45), and add the `dgtest` job to the group (line 36).

If this deployment fails for some reason, an exception is thrown (line 37), which prints (line 39) the job name, group name, exception type, and value.

job_started_event

In `dgtest.jdl` (page 139), the `test` class (line 42) defines only the required `job_started_event` (line 44) 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 45-47). It then tries to find the resource group named `dgtest`. If the resource group doesn't exist, the member `fail` string is set to inform the user and returns without scheduling the joblet(s) (lines 49-58).

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 60-65. After setting the memo line in the Console (67-69), the job schedules count number of `testnode` joblets (line 70).

joblet_started_event

In `dgtest.jdl` (page 139), the `testnode` class (line 73) defines only the required `joblet_started_event` (line 75) method. This method runs on the Orchestrator agent nodes when scheduled by a `Job` (page 222) class.

The `joblet_started_event` prints some trace information (lines 76-77), gets the name of the file to transfer (line 78) and the mode of transfer (line 79), and creates the grid URL for the file (line 82).

A `DataGrid` (page 212) is instantiated (line 86), set not to cache (line 89), and set to use the multicast `jobarg` (line 94). The next four settings control multicast behavior are commented out (lines 97, 100, 103, and 106).

The joblet prints a memo line for the Orchestrator console (lines 108-115), sets the location for the file on the local node (line 121), and tries to transfer the file from the `datagrid` (line 123).

If the `datagrid` copy at line 123 fails for some reason, we have a retry mechanism in the exception handler (lines 125-143). The information for why the exception occurred is fetched (line 125).

The variable `retryUnicast` (line 126) 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 125-134). 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 136 from a failed multicast copy, a memo for the Orchestrator console is set and printed to the log (137-138), `setMulticast` is set to `false` (140), and another copy from the `datagrid` is attempted.

If we get to line 136 from a failed unicast copy, an exception is raised (line 143) and we're done.

Configure and Run

```
> zos run dgtest filename=suse-9-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-9-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-9-flat.vmd
JobID: zenuser.dgtest.324
```

```
> zos log zenuser.dgtest.324
Starting Datagrid Test Job.
Filename: suse-9-flat.vmd
Scheduling Datagrid Test on 0 Joblets
```

NOTE: No joblets were scheduled because we have no active nodes in the group.

Using the Orchestrator Console, populate the `dgtest` group with nodes that are both online and enabled:

```
> zos run dgtest filename=suse-9-flat.vmd
JobID: zenuser.dgtest.325
```

```
> zos log zenuser.dgtest.325
Starting Datagrid Test Job.
Filename: suse-9-flat.vmd
Scheduling Datagrid Test on 2 Joblets
[freeze] Running datagrid test joblet #0
[freeze] Starting unicast download of file: grid:///images/suse-9-flat.vmd
[freeze] Traceback (innermost last):
[freeze] File "dgtest.jdl", line 143, in joblet_started_event
[freeze] copy() failed: DataGrid file "/images/suse-9-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-9-flat.vmd
[melt] Traceback (innermost last):
[melt] File "dgtest.jdl", line 143, in joblet_started_event
[melt] copy() failed: DataGrid file "/images/suse-9-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-9-flat.vmd grid:///images/
suse-9-flat.vmd copied.
```

```
> zos run dgtest filename=suse-9-flat.vmd
JobID: zenuser.dgtest.326
```

```
> zos log zenuser.dgtest.326
Starting Datagrid Test Job.
Filename: suse-9-flat.vmd
Scheduling Datagrid Test on 2 Joblets
[melt] Running datagrid test joblet #1
[melt] Starting unicast download of file: grid:///images/suse-9-flat.vmd
[melt] grid:///images/suse-9-flat.vmd downloaded successfully.
[melt] 16732 -rw-r--r-- 1 root root 17108462 Dec 21 21:32 suse-9-flat.vmd
[melt] Datagrid test completed
[freeze] Running datagrid test joblet #0
[freeze] Starting unicast download of file: grid:///images/suse-9-flat.vmd
[freeze] grid:///images/suse-9-flat.vmd downloaded successfully.
[freeze] 16732 -rw-r--r-- 1 root root 17108462 Dec 21 21:31 suse-9-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 118-120 in [dgtest.jdl \(page 139\)](#). Uncomment line 120 and comment out line 121 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 7.13, “Improving Job and Joblet Robustness,”](#) on page 85.

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 © 2008 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-INFRINGEMENT. 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,v 1.3 2008/02/27 20:50:00 john Exp $
```

```

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
20 #
21
22 import sys,os,time
23
24 #
25 # Add to the 'examples' group on deployment
26 #
27 if __mode__ == "deploy":
28     try:
29         jobgroupname = "examples"
30         jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
31         if jobgroup == None:
32             jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
33         jobgroup.addMember(__jobname__)
34     except:
35         exc_type, exc_value, exc_traceback = sys.exc_info()
36         print "Error adding %s to %s group: %s %s" % (__jobname__, jobgroupname,
exc_type, exc_value)
37
38
39 class failover(Job):
40
41     def job_started_event(self):
42         numJoblets = self.getFact("jobargs.numJoblets")
43         print 'Launching ', numJoblets, ' joblets'
44         self.schedule(failoverjoblet,numJoblets)
45
46
47 class failoverjoblet(Joblet):
48
49     def joblet_started_event(self):
50         print "----- joblet_started_event"
51         print "node=%s joblet=%d" % (self.getFact("resource.id"),
self.getFact("joblet.number"))
52         print "self.getFact(joblet.retrynumber)=%d" %
(self.getFact("joblet.retrynumber"))
53         print "self.getFact(job.joblet.maxretry)=%d" %
(self.getFact("job.joblet.maxretry"))
54
55         sleeptime = self.getFact("jobargs.sleeptime")
56         print "sleeping for %d seconds" % (sleeptime)
57         time.sleep(sleeptime)
58
59         # This will cause joblet failure and thus retry
60         raise RuntimeError, "Artificial error in joblet. node=%s" %
(self.getFact("resource.id"))
61
62
63
64

```

failover.policy

```
1  <!--
2
3  * Copyright © 2008 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 INFRINGEMENT. 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: failover.policy,v 1.2 2008/02/27 20:50:00 john Exp $
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 \(page 148\)](#) is derived from the [Job \(page 222\)](#) class; and the class failoverjoblet in line 33 of [failover.jdl \(page 148\)](#) is derived from the [Joblet \(page 224\)](#) 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.

test

Class test (line 42 in [dgtest.jdl \(page 139\)](#) is derived from the `Job` class.

testnode

Class testnode (line 73 in [dgtest.jdl \(page 139\)](#) is derived from the `Joblet (page 224)` class.

Job Details

The following sections describe the Failover job:

- ♦ “zosadmin deploy” on page 151
- ♦ “job_started Event” on page 151
- ♦ “job_started Event” on page 152

zosadmin deploy

In [failover.policy \(page 150\)](#), in addition to describing the jobargs and default settings for `sleeptime` and `numJoblets` (lines 2-5), the `<job/>` section (lines 7-12) describes static facts (see [Section 5.1.2, “Facts,” on page 57](#)). Note that the `joblet.maxretries` attribute in line 11 has a default setting of 0 but is set here to 3. This attribute can also be modified in the [failover.jdl \(page 148\)](#) file by inserting a line between line 27 and 28, as shown in the following example:

```
27         def job_started_event(self):
++             self.setFact("job.joblet.maxretries", 3)
28             numJoblets = self.getFact("jobargs.numJoblets")
```

job_started Event

After the Orchestrator server deploys a job for the first time (see [Section 7.5, “Deploying Jobs,” on page 75](#)), the job JDL files are executed in a special “deploy” mode. When the job is deployed (line 13, [failover.jdl \(page 148\)](#)), it attempts to find the `examples` jobgroup (lines 15-16), creates it if it is missing (lines 17-18), and adds the failover job to the group (line 19).

Jobs can be deployed using either the Orchestrator console (`zoc`) or the `zosadmin deploy` command. If the deployment fails for some reason, an exception is thrown (line 20), which prints the job name (line 22), group name, exception type, and value.

job_started Event

In [failover.jdl \(page 148\)](#), the failover class (line 25) defines only the required `job_started_event` (line 27) method. This method runs on the Orchestrator 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 28), then schedules that specified number of instances (line 30) of the `failoverjoblet` class. `failoverjoblet` class (lines 33-46) defines only the required `joblet_started_event` (line 35) method.

When executed on an agent node, the `joblet_started_event` prints some helpful information for tracking execution (lines 36-39). The first output is where the joblet is running and which instance is running (line 37). The current joblet retry number (line 38) is displayed, followed by the job's static `joblet.maxretry` (line 39) that was specified in the policy file.

The joblet then sleeps for `jobargs.sleeptime` seconds (lines 41-43) and on waking raises an exception of type `RuntimeError` (line 46).

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 Orchestrator 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 46, 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 46, 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 46, 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 46, 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 46, 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 46, in joblet_started_event
[freeze] RuntimeError: Artifical error in joblet. node=freeze

```

See Also

- ◆ [Setting Constraints Using Policies \(Section 4.4, “Policy Management,” on page 53 and Chapter 5, “Developing Policies,” on page 57\)](#)
- ◆ [Adding Jobs to Groups During Deployment \(see how the JDL code can print the ID of group of jobs in `factJunction.job` \(page 179\)\).](#)
- ◆ [Executing Commands Using `Exec` \(page 215\)](#)

instclients.job

Installs the ZENworks Orchestrator 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 ZOS 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 © 2008 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-INFRINGEMENT.  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,v 1.4 2008/03/05 20:05:54 ray Exp $
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_1_3_0_with_jre.exe"
55         linuxInstaller = "zosclients_linux_1_3_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  *=====  

4  * Copyright © 2008 Novell, Inc. All Rights Reserved.  

5  *  

6  * NOVELL PROVIDES THE SOFTWARE "AS IS," WITHOUT ANY EXPRESS OR IMPLIED  

7  * WARRANTY, INCLUDING WITHOUT THE IMPLIED WARRANTIES OF MERCHANTABILITY ,  

8  * FITNESS FOR A PARTICULAR PURPOSE, AND NON INFRINGEMENT. NOVELL, THE AUTHORS  

9  * OF THE SOFTWARE, AND THE OWNERS OF COPYRIGHT IN THE SOFTWARE ARE NOT LIABLE  

10 * FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF  

11 CONTRACT,  

12 * TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE  

13 * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.  

14 *=====  

15 * $Id: instclients.policy,v 1.2 2008/02/27 20:50:26 john Exp $
16

```

```

*=====
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 ZOS 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:

- ♦ “zosadmin deploy” on page 158
- ♦ “job_started_event” on page 158
- ♦ “joblet_started_event” on page 158

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, and will show up in the ZENworks Orchestrator Console in the Explorer view at the location:

```
/ZOS/YOUR_GRID/Jobs/examples.
```

For a general overview of how jobs are added to groups during deployment, see “[Deploying a Sample System Job](#)” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.

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 32 of [instclients.jdl \(page 154\)](#)). The resource that runs the joblet is determined from the resource constraint specified in [instclients.policy \(page 156\)](#), lines 12-14, 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 Orchestrator Client installation file to download, and the commands to execute on the operating system by checking the value of `resource.os.family` (see line 40 of [instclients.jdl \(page 154\)](#)). The `resource.os.family` fact does not exist in the `instclients.policy` file, but is instead provided by the ZENworks Orchestrator 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 41-43 and 54-56 in [instclients.jdl \(page 154\)](#)). 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 46-52 and 58-75 in [instclients.jdl \(page 154\)](#)). As indicated by lines 50, 51, 64, 65, 73 and 74, 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 7.5.2, “Using the ZOSADMIN Command Line Tool,” on page 75](#). See the `Exec` class in [Orchestrator 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 Orchestrator Server (note: replace `windows` with `Linux*`, `Solaris*`, etc. for your given operating system):

```
zos copy zosclients_linux_1_3_0_with_jre.sh grid:///!instclients/
```

This command copies the file `zosclients_linux_1_3_0_with_jre.sh` into the datagrid job directory for `instclients`.

For more information about using ZENworks Orchestrator 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 Orchestrator client onto the resource with the host: `my_resource_host`.

See Also

- ◆ [Setting Constraints Using Policies \(Section 4.4, “Policy Management,” on page 53 and Chapter 5, “Developing Policies,” on page 57\)](#)
- ◆ [Adding Jobs to Groups During Deployment \(see how the JDL code can print the ID of group of jobs in `factJunction.job` \(page 179\)\).](#)
- ◆ [Scheduling multiple instances of a joblet](#)
- ◆ [Setting default parameter values using policies](#)
- ◆ [Configuring constraints in a policy file](#)
- ◆ [Naming conventions for policy facts \(Section 3.1.1, “Naming Orchestrator Job Files,” on page 43. Section 3.1.1, “Naming Orchestrator Job Files,” on page 43\)](#)
- ◆ [Facts provided by the ZENworks Orchestrator system that can be referenced within a JDL file](#)

- ◆ Using ZENworks Orchestrator Console (“[How Do I Interact with ZENworks Orchestrator?](#)”)
- ◆ 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 © 2008 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-INFRINGEMENT. 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,v 1.3 2008/02/27 20:50:47 john Exp $
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 © 2008 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 INFRINGEMENT. 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,v 1.2 2008/02/27 20:50:47 john Exp $
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:

- ◆ “zosadmin deploy” on page 163
- ◆ “job_started_event” on page 164
- ◆ “joblet_started_event” on page 164

zosadmin deploy

In `notepad.jdl` (page 161), lines 7-19 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 ZENworks Orchestrator Console in the Explorer view at the location:

/ZOS/YOUR_GRID/Jobs/examples

For a general overview of how jobs are added to groups during deployment, see “[Walkthrough: Deploy a Sample Job](#)” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.

job_started_event

Scheduling the Notepad joblet to run immediately is the second operation performed by the Notepad job in line 26 of [notepad.jdl \(page 161\)](#). 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 2 and 5 of [notepad.policy \(page 162\)](#), there must be at least one Windows machine available in the grid for the Notepad job to run. The `accept` constraint in lines 1-3 prevents the Notepad job from being accepted for running if there are no Windows resources available.

The `resource` constraint in lines 4-7 constrain the Orchestrator scheduler to only choose a resource that is running a Windows OS.

For more information on setting constraints using policies, see [Section 4.4, “Policy Management,” on page 53](#) and [Chapter 5, “Developing Policies,” on page 57](#).

joblet_started_event

As specified in lines 33-38 in [notepad.jdl \(page 161\)](#), 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 ZENworks Orchestrator 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 4.4, “Policy Management,” on page 53](#) and [Chapter 5, “Developing Policies,” on page 57](#).
- ♦ Adding Jobs to Groups During Deployment (see how the JDL code can print the ID of group of jobs in [factJunction.job \(page 179\)](#)).
- ♦ Executing Commands Using [Exec \(page 215\)](#)

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 illustrates 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 \(page 239\)](#) utility class is also related to this example.

sweeper.jdl

```
1 # -----
-
2 # Copyright © 2008 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-INFRINGEMENT. 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,v 1.3 2008/02/27 20:51:24 john Exp $
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) 2008 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 INFRINGEMENT. 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,v 1.2 2008/02/27 20:51:24 john Exp $
14
=====
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 illustrates how to schedule a 'sweep'
across 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 18, [sweeper.jdl \(page 167\)](#)) is derived from the [Job Class](#).

The class `sweeperJoblet` (see line 44, [sweeper.jdl \(page 167\)](#)) 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:

- ♦ “Policy” on page 170
- ♦ “zosadmin deploy” on page 170
- ♦ “job_started_event” on page 170
- ♦ “joblet_started_event” on page 171

Policy

In addition to specifying the jobarg and default settings for `sleeptime` in lines 8-12, [sweeper.policy](#) (page 168), there also is the `<job/>` section in lines 16-55, which describes static facts (see “Facts” on page 57).

The `resources.rankby` array has two notable setting in this example:

- ♦ **resource.loadaverage:** This is the first string assignment (lines 38-44), 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 50-54), 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 Orchestrator server deploys a job for the first time (see [Section 7.5, “Deploying Jobs,”](#) on page 75), the job JDL files are executed in a special deploy mode. When `sweeper.jdl` is run in this way (either via the `zoc` or the `zosadmin deploy` command), lines 6-15 are executed. This attempts to locate the examples jobgroup (lines 8-9), creates the group if it is not found (lines 10-11), and adds the sweeper job to the group (line 12).

If the deployment fails for any reason, then an exception is thrown (line 13), which prints the job name, group name, exception type and value (line 15).

job_started_event

The `sweeperJob` class (line 18) defines only the required `job_started_event` (line 20) method. This method runs on the Orchestrator 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 Orchestrator console (line 21), via `jobinstance.memo` (see [Section 7.12.1, “Creating a Job Memo,”](#) on page 84).

Jumping ahead for a moment, instead of calling `self.schedule()` as most the other examples do to instantiate joblets, `sweeperJob` calls `self.scheduleSweep()` (line 38). `scheduleSweep` requires an instance of [ScheduleSpec](#) (page 239), so one is created (line 23).

The `ScheduleSpec` method `setConstraint` can be used to constrain the available resources to a particular group, as shown with a comment (line 29). 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 “[Walkthrough: Create a Resource Account](#)”). This constraint would also be ANDed to any existing constraint, including any aggregated policies.

The `sweeperJoblet` is set to be scheduled (line 32), and `setUseNodeSet(intnodeSet)` is assigned (line 35) 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 44-51) defines only the required `joblet_started_event` (line 46) method. After this method is executed, it displays a message on the memo line of the Joblet tab within the Jobs view in the Orchestrator console (lines 47-48). It also prints a similar log message (line 49), and then just sleeps for `jobargs.sleepTime` seconds (lines 50-51) 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 4.4, “Policy Management,”](#) on page 53 and [Chapter 5, “Developing Policies,”](#) on page 57

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) 2008 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-INFRINGEMENT. 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,v 1.3 2008/02/27 20:51:34 john Exp $
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) 2008 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 INFRINGEMENT. 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,v 1.2 2008/02/27 20:51:34 john Exp $
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:

- ♦ [“zosadmin deploy” on page 176](#)
- ♦ [“job_started_event” on page 177](#)
- ♦ [“joblet_started_event” on page 177](#)

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 22-34 of `whoami.jdl`) and is displayed in the ZENworks Orchestrator Console in the Explorer view at the location:

```
/ZOS/YOUR_GRID/Jobs/examples.
```

For a general overview of how jobs are added to groups during deployment, see [“Walkthrough: Deploy a Sample Job”](#) in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.

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 30 in [whoami.jdl \(page 173\)](#)). 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 3-6 in [whoami.policy \(page 175\)](#)).

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 7-10 in [whoami.policy \(page 175\)](#)).

Note that the setting for `resource` isn't 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 22 in [whoami.jdl \(page 173\)](#)).

`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 17-19 in [whoami.policy \(page 175\)](#).

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 3.1.1, "Naming Orchestrator Job Files," on page 43](#).

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 36-42 in [whoami.jdl \(page 173\)](#)). This setting is not set in the `whoami.policy`, but instead is available from the ZENworks Orchestrator system.

After the command to get the current username has been decided, the ZENworks Orchestrator API class named `Exec` is used to execute the command on the resource where the joblet is running (see lines 48-54 in [whoami.jdl \(page 173\)](#)).

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 is `cmd.exe`. On POSIX systems, this is `/bin/sh`. As indicated by lines 50-51 in [whoami.jdl \(page 173\)](#), 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](#)". For more information on running commands using the `Exec` class, see [Exec \(page 215\)](#).

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 4.4, "Policy Management," on page 53](#) and [Chapter 5, "Developing Policies," on page 57](#)).
- ♦ Adding Jobs to Groups During Deployment (see how the JDL code can print the ID of group of jobs in [factJunction.job \(page 179\)](#)).
- ♦ Scheduling multiple instances of a joblet
- ♦ Setting default parameter values using policies
- ♦ Configuring constraints in a policy file
- ♦ Naming conventions for policy facts ([Section 3.1.1, "Naming Orchestrator Job Files," on page 43](#).[Section 3.1.1, "Naming Orchestrator Job Files," on page 43](#))
- ♦ Facts provided by the ZENworks Orchestrator system that can be referenced within a JDL file
- ♦ Using ZENworkd Orchestrator ("[How Do I Interact with ZENworks Orchestrator?](#)")
- ♦ Running commands using the [Exec](#) class.

10.6 Miscellaneous Code-Only Jobs

The following examples demonstrate useful, miscellaneous code-only job concepts:

- ♦ "[factJunction.job](#)" on [page 179](#)
- ♦ "[jobargs.job](#)" on [page 188](#)

factJunction.job

Demonstrates using fact junctions to retrieve information about objects in the grid relative to another object.

Detail

Each object in the grid has a set of facts which can be read and modified. Some of these facts are special in the sense that their value contains the name of another object that must exist in the grid. These special facts are called fact junctions.

Fact junctions provide a way to reference the facts of one object, using another object as a starting point. For example, all jobs in the grid have a fact named `job.accountinggroup`. The value for `job.accountinggroup` must be the name of a job group currently existing in the grid (the default being the group named `all`). The following JDL code prints the ID of the accounting group for the job named `myJob` without using fact junctions:

```
job = getMatrix().getGridObject(TYPE_JOB, "myJob")
groupName = job.getFact("job.accountinggroup")
group = getMatrix().getGridObject(TYPE_JOBGROUP, groupName)
print "Group ID: " + group.getFact("group.id")
```

Using fact junctions, you can obtain the ID of the accounting group without having to retrieve a reference to the group object first, as follows:

```
job = getMatrix().getGridObject(TYPE_JOB, "myJob")
print "Group ID: " + job.getFact("job.accountinggroup.id")
```

Notice the job `myJob` does not have a fact named `job.accountinggroup.id`. However, it does have a fact named `job.accountinggroup`, which contains the name of an existing job group. This job group has the fact `group.id`, and using fact junctions you can obtain the value of this fact without explicitly reading it off of the job group object itself.

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 factJunction.job
factJunction successfully deployed

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

> zos jobinfo --detail factJunction
Jobname/Parameters      Attributes
-----
factJunction           Desc: This is a test job to exercise fact junctions.

No parameters defined for this job.
```

Description

The files that make up the factJunction job include:

```
factJunction # Total: 205 lines
|-- factJunction.jdl # 179 lines
`-- factJunction.policy # 26 lines
```

factJunction.jdl

```
1 # -----
--
2 # Copyright 2008 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-INFRINGEMENT. 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: factJunction.jdl,v 1.3 2008/02/27 20:49:39 john Exp $
13 # -----
--
14
15 #
16 # This is a test job, but also illustrates all the implemented fact junctions.
17 # A fact junction is a way to access facts on a 'referenced' object.
18 # E.g. vmhost.resource.*** redirects from the vmhost object through the
19 # junction onto the underlying physical resource object.
20 #
21 # To setup for test, copy job into the 'provisionAdapter' job group.
22 #
23
24 #
25 # Add to the 'examples' group on deployment
26 #
27 if __mode__ == "deploy":
28     try:
29         jobgroupname = "examples"
30         jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
31         if jobgroup == None:
32             jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
33         jobgroup.addMember(__jobname__)
34     except:
35         exc_type, exc_value, exc_traceback = sys.exc_info()
36         print "Error adding %s to %s group: %s %s" % (__jobname__,
jobgroupname, exc_type, exc_value)
37
38
39 class factJunctionJob(Job):
40
41     def job_started_event(self):
42         m = getMatrix()
```

```

43     nptt = "<Not Possible To Test>"
44
45     # Setup test environment
46     user = m.getGridObject(TYPE_USER, "test")
47     if (user == None):
48         user = m.createGridObject(TYPE_USER, "test")
49     user.setArrayFact("user.privilegedjobgroups", ["all"])
50
51     repository = m.getGridObject(TYPE_REPOSITORY, "test")
52     if (repository == None):
53         repository = m.createGridObject(TYPE_REPOSITORY, "test")
54     repository.setArrayFact("repository.provisioner.jobs",
["factJunction"])
55
56     node = m.getGridObject(TYPE_RESOURCE, "test")
57     if (node == None):
58         node = m.createGridObject(TYPE_RESOURCE, "test")
59
60     vm = m.getGridObject(TYPE_RESOURCE, "vmttest")
61     if (vm == None):
62         vm = m.createResource("vmttest", ResourceInfo.TYPE_VM_INSTANCE)
63     vm.setFact("resource.provisioner.job", "factJunction")
64     vm.setFact("resource.vm.repository", "test")
65     vm.setFact("resource.provisioner.recommendedhost", "test_test")
66
67     vmt = m.getGridObject(TYPE_RESOURCE, "vmttest")
68     if (vmt == None):
69         vmt = m.createResource("vmttest", ResourceInfo.TYPE_VM_TEMPLATE)
70     vmt.setFact("resource.provisioner.job", "factJunction")
71     vmt.setFact("resource.vm.repository", "test")
72
73     try:
74         vmhost = node.getVmHost("test")
75     except:
76         vmhost = node.createVmHost("test")
77     vmhost.setFact("vmhost.provisioner.job", "factJunction")
78     vmhost.setArrayFact("vmhost.repositories", ["test"])
79     vmhost.setArrayFact("vmhost.vm.available.groups", ["all"])
80
81     job = m.getGridObject(TYPE_JOB, "factJunction")
82
83     # Test junctions
84
85     print
86     print "Testing User fact junctions (3):"
87     r = user.getFact("user.accountinggroup.id")
88     print "1. user.accountinggroup.id = %s" % r
89     # Array junctions
90     r = user.getFact("user.privilegedjobgroups[all].id")
91     print "2. user.privilegedjobgroups[all].id = %s" % r
92     r = user.getFact("user.groups[all].jobcount")
93     print "3. user.groups[all].jobcount = %s" % r
94
95
96     print
97     print "Testing Job fact junctions (3):"
98     r = job.getFact("job.accountinggroup.id")
99     print "1. job.accountinggroup.id = %s" % r
100    r = job.getFact("job.resourcegroup.id")

```

```

101     print "2. job.resourcegroup.id = %s" % r
102     # Array junctions
103     r = job.getFact("job.groups[all].jobinstances.total")
104     print "3. job.groups[all].jobinstances.total = %s" % r
105
106
107     print
108     print "Testing VmHost fact junctions (7):"
109     r = vmhost.getFact("vmhost.resource.id")
110     print "1. vmhost.resource.id = %s" % r
111     r = vmhost.getFact("vmhost.accountinggroup.id")
112     print "2. vmhost.accountinggroup.id = %s" % r
113     r = vmhost.getFact("vmhost.provisioner.job.id")
114     print "3. vmhost.provisioner.job.id = %s" % r
115     # Array junctions
116     r = vmhost.getFact("vmhost.groups[all].vmcount")
117     print "4. vmhost.groups[all].vmcount = %s" % r
118     r = vmhost.getFact("vmhost.repositories[test].id")
119     print "5. vmhost.repositories[test].id = %s" % r
120     r = vmhost.getFact("vmhost.vm.available.groups[all].id")
121     print "6. vmhost.vm.available.groups[all].id = %s" % r
122     #r = vmhost.getFact("vmhost.vm.instanceids[vmtest].id")
123     r = npptt
124     print "7. vmhost.vm.instanceids.[vmtest].id = %s" % r
125
126
127     print
128     print "Testing Resource fact junctions (9):"
129     r = vm.getFact("resource.provisioner.job.id")
130     print "1. resource.provisioner.job.id = %s" % r
131     r = vm.getFact("resource.vm.repository.id")
132     print "2. resource.vm.repository.id = %s" % r
133     r = vm.getFact("resource.provisioner.recommendedhost.id")
134     print "3. resource.provisioner.recommendedhost.id = %s" % r
135     #r = vm.getFact("resource.provision.vmhost.id")
136     r = npptt
137     print "4. resource.provision.vmhost.id = %s" % r
138     #r = vm.getFact("resource.provision.template.id")
139     r = npptt
140     print "5. resource.provision.template.id = %s" % r
141     # Array junctions
142     r = vm.getFact("resource.groups[all].loadaverage")
143     print "6. resource.groups[all].loadaverage = %s" % r
144     r = node.getFact("resource.vmhosts[test_test].id")
145     print "7. resource.vmhosts[test_test].id = %s" % r
146     r = node.getFact("resource.repositories[test].id")
147     print "8. resource.repositories[test].id = %s" % r
148     #r = vmt.getFact("resource.provisioner.instances[vmtest_2].id")
149     r = npptt
150     print "9. resource.provisioner.instances[vmtest_2].id = %s" % r
151
152
153     print
154     print "Testing Repository fact junctions (4):"
155     r = repository.getFact("repository.groups[all].id")
156     print "1. repository.groups[all].id = %s" % r
157     r = repository.getFact("repository.vmimages[vmtest].id")
158     print "2. repository.vmimages[vmtest].id = %s" % r
159     r = repository.getFact("repository.vmhosts[test_test].id")

```

```

160     print "3. repository.vmhosts[test_test].id = %s" % r
161     r = repository.getFact("repository.provisioner.jobs[factJunction].id")
162     print "4. repository.provisioner.jobs[factJunction].id = %s" % r
163
164
165     print
166     print "Testing multiple junctions (1):"
167     r =
repository.getFact("repository.vmhosts[test_test].resource.repositories[test].vmho
sts[test_test].groups[all].id")
168     print "1.
repository.vmhosts[test_test].resource.repositories[test].vmhosts[test_test].group
s[all].id = %s" % r
169
170     # Now make sure they are all accessible by the joblet...
171     #self.schedule(factJunctionJoblet, {})
172
173
174 class factJunctionJoblet(Joblet):
175
176     def joblet_started_event(self):
177         # TODO
178         time.sleep(sleeptime)
179

```

factJunction.policy

The description fact displays the commands x, y, z ...

```

1  <!--
2
=====
3  * Copyright © 2008 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 INFRINGEMENT. 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: factJunction.policy,v 1.2 2008/02/27 20:49:39 john Exp $
14
=====
15 -->
16
17 <policy>
18
19   <job>
20     <fact name="description"
21       type="String"

```

```
22         value="This is a test job to exercise fact junctions." />
23     </job>
24
25 </policy>
26
```

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.

UserInfo

A representation of a User grid object. This class provides accessors and setters for User facts.

RepositoryInfo

A representation of a Repository grid object. This class provides accessors and setters for Repository facts. To script the creation of Repository objects, see [MatrixInfo \(page 229\)](#).

ResourceInfo

A representation of a Resource Grid Object. This class inherits the base fact operations from `GridObjectInfo` and adds the provisioning operations for provisionable resources such as virtual machines. See [MatrixInfo \(page 229\)](#) for how to script creation of Resource objects.

JobInfo

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.

Job Details

The FactJunction job performs its work by handling the following events:

- ♦ “zosadmin deploy” on page 185

- ♦ [“job_started_event” on page 185](#)
- ♦ [“joblet_started_event” on page 185](#)

zosadmin deploy

Deploying FactJunction job is performed by lines 10-22 of factJunction.jdl. When jobs are deployed into the grid, they can optionally be placed in groups for organization and easy reference. In this case, the FactJunction job will be added to the group named “examples”, and will show up in the ZENworks Orchestrator Console in the Explorer view at the location:

```
/ZOS/YOUR_GRID/Jobs/examples
```

For a general overview of how jobs are added to groups during deployment, see [“Walkthrough: Deploy a Sample Job”](#) in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.

job_started_event

When the FactJunction job receives a `job_started_event`, it gets a reference to the `MatrixInfo` object, which allows it to obtain references to other objects in the grid, such as Users, Resources, Jobs, etc. (see lines 28, 32, 37, 42, 46, and 53 in [“factJunction.jdl” on page 180](#)). If these objects don't exist in the grid, they are immediately created so they can be used later on (see lines 34, 39, 44, 48, 55, and 62).

After references exist for the various objects in the grid, values for other objects are printed out using the fact junctions that exist on each object (see lines 69-154 in [factJunction.jdl \(page 180\)](#)).

There are several instances where the FactJunction job uses “array notation” to handle fact junctions that contain multiple values (see lines 76, 78, 89, 102, 104, 106, 108, 128, 130, 132, 142, 144, 146, and 153 in [factJunction.jdl \(page 180\)](#)). As previously explained, fact junctions are special facts because their value contains the name of another object that must exist in the grid. However, fact junctions don't always contain a single name. Some fact junctions allow for an array of names to be specified. For example, the value for the fact “`job.groups`” is supplied as a String array.

In this case, the fact junction can be refined using array notation, which allows for the selection of one of the values. For example, the following code retrieves the ID of the group named “`myGroup`”, which is one of the groups the given job is a member of:

```
job.getFact("job.groups[myGroup].id")
```

joblet_started_event

The FactJunction job only illustrates using fact junctions to retrieve information about objects in the grid. Therefore, no work is performed on the resource by the FactJunction joblet.

Configure and Run

To run this example, you must have ZENworks Orchestrator installed and configured properly. No agents on separate resources are required. You also must be logged into your Orchestrator server before you run `zosadmin` or `zos` commands.

Execute the following commands to deploy and run factJunction.job:

- 1 Deploy factJunction.job into the grid:

```
> zosadmin deploy factJunction.job
```

2 Display the list of deployed jobs:

```
> zos joblist
```

factJunction should appear in this list.

3 Run the FactJunction job, and view the results:

```
> zos run factJunction
JobID: zenuser.factJunction.421
> zos log factJunction
> zos status zenuser.factJunction.421
Completed
```

```
> zos log factJunction
```

4 Testing User fact junctions:

1. user.accountinggroup.id = all
2. user.privilegedjobgroups[all].id = all
3. user.groups[all].jobcount = 147

5 Testing Job fact junctions:

1. job.accountinggroup.id = all
2. job.resourcegroup.id = all
3. job.groups[all].jobinstances.total = 1

6 Testing VmHost fact junctions:

1. vmhost.resource.id = test
2. vmhost.accountinggroup.id = all
3. vmhost.provisioner.job.id = factJunction
4. vmhost.groups[all].vmcount = 0
5. vmhost.repositories[test].id = test
6. vmhost.vm.available.groups[all].id = all
7. vmhost.vm.instanceids.[vmtest].id = <Not Possible To Test>

7 Testing Resource fact junctions:

1. resource.provisioner.job.id = factJunction
2. resource.vm.repository.id = test
3. resource.provisioner.recommendedhost.id = test_test
4. resource.provision.vmhost.id = <Not Possible To Test>
5. resource.provision.template.id = <Not Possible To Test>
6. resource.groups[all].loadaverage = 0.036666666666666667
7. resource.vmhosts[test_test].id = test_test
8. resource.repositories[test].id = test
9. resource.provisioner.instances[vmtest_2].id = <Not Possible To Test>

8 Testing Repository fact junctions:

1. `repository.groups[all].id = all`
2. `repository.vmimages[vmtest].id = vmtest`
3. `repository.vmhosts[test_test].id = test_test`
4. `repository.provisioner.jobs[factJunction].id = factJunction`

9 Testing multiple junctions

1. `repository.vmhosts[test_test].resource.repositories[test].vmhosts[test_test].groups[all].id = all`

See Also

- ♦ Adding jobs to groups during deployment (see how the JDL code can print the ID of group of jobs in [factJunction.job \(page 179\)](#)).
- ♦ View the list of fact junctions available for each object type in a ZENworks Orchestrator grid
- ♦ Using array notation to refine multi-valued fact junctions
- ♦ Using ZENworks Orchestrator (“[How Do I Interact with ZENworks Orchestrator?](#)”)

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 [Section 7.7, “Working with Facts and Constraints,” on page 77](#).

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.

  TimeArgReq         Desc: Required Time arg test
                    Type: Time
                    Default: None! Value must be specified

  IntegerArg         Desc: Integer arg test
                    Type: Integer
                    Default: 1

  DateArg            Desc: Date arg test
                    Type: Date
                    Default: Wed Apr 05 09:00:00 EDT 2006

  RealArgReq         Desc: Required Real arg test
                    Type: Real
                    Default: None! Value must be specified

  IntegerArrayArg    Desc: Integer[] arg test
                    Type: Integer[]
                    Default: [100,200,300]

  RealArrayArg       Desc: Real[] arg test
                    Type: Real[]
                    Default: [1.23,3.456,7.0]

  DictArg            Desc: Dictionary arg test
                    Type: Dictionary
                    Default: {name=moe, dob=Fri Jan 02 00:00:00 EST 1970,
                               age=35}

  DateArrayArg       Desc: Date[] arg test
```

```

                                Type: Date[]
                                Default: [Wed Jul 08 22:00:00 EDT 2009,Thu Jan 02 00:01:00
                                           EST 2003]

StringArgReq      Desc: Required String arg test
                  Type: String
                  Default: None! Value must be specified

TimeArrayArg     Desc: Time[] arg test
                  Type: Time[]
                  Default: [79200000,60000]

StringArrayArg   Desc: String[] arg test
                  Type: String[]
                  Default: [abc,def,ghi jkl]

TimeArg          Desc: Time arg test
                  Type: Time
                  Default: 32400000

BooleanArgReq    Desc: Required Boolean arg test
                  Type: Boolean
                  Default: None! Value must be specified

BooleanArg       Desc: Boolean arg test
                  Type: Boolean
                  Default: true

IntegerArgReq    Desc: Required Integer arg test
                  Type: Integer
                  Default: None! Value must be specified

StringArg        Desc: String arg test
                  Type: String
                  Default: Hello World

BooleanArrayArg  Desc: Boolean[] arg test
                  Type: Boolean[]
                  Default: [true,false,true]

RealArg          Desc: Real arg test
                  Type: Real
                  Default: 3.1415

ListArg          Desc: List arg test
                  Type: List
                  Default: [abc, d, efghij]

DateArgReq       Desc: Required Date arg test
                  Type: Date
                  Default: None! Value must be specified

```

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 © 2008 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-INFRINGEMENT. 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,v 1.3 2008/02/27 20:50:31 john Exp $
13 # -----
-
14
15 """
16 Example job showing all available job argument types.
17
18 Example cmd line to run job:
19     zos run jobargs TimeArgReq="12:01:02" RealArgReq="3.14" IntegerArgR
eq="123" StringArgReq="foo" BooleanArgReq="true" ListArg="hi,mom"
20 """
21
22 import time
23
24 #
25 # Add to the 'examples' group on deployment
26 #
27 if __mode__ == "deploy":
28     try:
29         jobgroupname = "examples"
30         jobgroup = getMatrix().getGroup(TYPE_JOB, jobgroupname)
31         if jobgroup == None:
32             jobgroup = getMatrix().createGroup(TYPE_JOB, jobgroupname)
33         jobgroup.addMember(__jobname__)
34     except:
35         exc_type, exc_value, exc_traceback = sys.exc_info()
36         print "Error adding %s to %s group: %s %s" % (__jobname__, jobgr
oupname, exc_type, exc_value)
37
38
39 class jobargs(Job):
40
41     def job_started_event(self):
42
43         jobid = self.getFact("jobinstance.id")
44         print "*****Dumping %s JobInstance jobargs facts*****" % (jobi d)
45         keys = self.getFactNames()
46         keys.sort()

```

```

47         for s in keys:
48             if s.startswith("jobargs"):
49                 v = self.getFact(s)
50                 ty = type(v)
51
52                 if str(ty).endswith("Dictionary"):
53                     self.dump_dict(s,v)
54                 else:
55                     if s.endswith("TimeArg") or s.endswith("TimeArgReq "):
56                         v = time.ctime(v/1000)
57
58                     print "%s %s %s" % (s,type(v),str(v))
59         print "*****End %s dump*****" % (jobid)
60
61         #self.schedule(jobargsJoblet)
62
63     def dump_dict(self,name,dict):
64         print "Dict: %s" % (name)
65         keys = dict.keys()
66         for k in keys:
67             v = dict[k]
68             ty = type(v)
69             if k == "dob":
70                 v = time.ctime(v/1000)
71             print "    %s %s %s" % (k,ty,str(v))
72
73
74 class jobargsJoblet(Joblet):

```

jobargs.policy

```

1  <!--
2
=====
3  * Copyright © 2008 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 INFRINGEMENT. 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,v 1.2 2008/02/27 20:50:31 john Exp $
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"
33         description="Optional Real Arg"
34         type="Real"
35         value="3.14" />
36
37     <fact name="OptionalIntegerArg"
38         description="Optional Integer Arg"
39         type="Integer"
40         value="123" />
41
42     <fact name="OptionalStringArg"
43         description="Optional String Arg"
44         type="String"
45         value="foo" />
46
47     <fact name="OptionalString2ArgAsTag"
48         description="Optional String Arg as tag">
49         <string>bar</string>
50     </fact>
51
52     <fact name="OptionalString3ArgAsCDATA"
53         description="Optional String Arg as CDATA">
54         <string>
55             <![CDATA[this text is part of
56 a multi-line cdata section containing
57 xml <html>test</html>
58 <eq fact="foo.bar" value="qwerty" />
59 cool!
60 ]]>
61         </string>
62     </fact>
63
64     <fact name="OptionalBooleanArg"
65         description="Optional Boolean Arg"
66         type="Boolean"
67         value="true" />
68
69     <fact name="OptionalListArg">
70         <list>
71             <listelement value="hi" type="String" />
72             <listelement value="mom" />
73             <listelement value="42" type="Integer" />
74         </list>
75     </fact>
76
77     <fact name="OptionalDictArg">
78         <dictionary>
79             <dictelement key="name" type="String" value="joe" />
80             <dictelement key="date" type="Date" value="4/15/06" />
81             <dictelement key="time" type="Time" value="3:30 AM" />

```



```

82         <dictelement key="age" type="Integer" value="12" />
83     </dictionary>
84 </fact>
85
86 <fact name="OptionalDateArray">
87     <array>
88         <date>1/2/06 12:01 PM</date>
89         <date>1/3/06 12:02 PM</date>
90         <date>1/4/06</date>
91     </array>
92 </fact>
93 <fact name="OptionalTimeArray">
94     <array>
95         <time>12:01 PM</time>
96         <time>12:02 PM</time>
97     </array>
98 </fact>
99 <fact name="OptionalRealArray">
100     <array>
101         <real>1.1</real>
102         <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)

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:

- ♦ “zosadmin deploy” on page 195
- ♦ “job_started_event” on page 195
- ♦ “joblet_started_event” on page 195

zosadmin deploy

In `jobargs.jdl` (page 190), lines 10-229 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 ZENworks Orchestrator Console in the Explorer view at the location:

```
/ZOS/YOUR_GRID/Jobs/examples
```

For a general overview of how jobs are added to groups during deployment, see “[Walkthrough: Deploy a Sample Job](#)” in the *Novell ZENworks Orchestrator 1.3 Installation and Getting Started Guide*.

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 31 of `jobargs.jdl` (page 190). This list is sorted, filtered according to whether or not it’s a jobarg fact, and then enumerated (lines 32-42). Each jobarg fact is printed in a “name type value” format. When the complex Dictionary type is encountered (line 38), a separate method is used to print the values for all the key-value pairs (lines 49-57).

The list of optional and required arguments for this Jobargs example are available as facts within the `<jobargs>` section (see lines 3-139 in `jobargs.policy` (page 191)).

For more information about defining job arguments and their types, see [Chapter 5, “Developing Policies,”](#) on page 57 and [Section 4.4, “Policy Management,”](#) on page 53.

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 ZENworks Orchestrator installed and configured properly. No agents on separate resources are required. You also must be logged into your Orchestrator 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 \(page 191\)](#) 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

- ♦ Adding Jobs to Groups During Deployment (see how the JDL code can print the ID of group of jobs in [factJunction.job \(page 179\)](#)).
- ♦ Defining job arguments and their types
- ♦ Using ZENworks Orchestrator (“[How Do I Interact with ZENworks Orchestrator?](#)”)

Orchestrator Job Classes and JDL Syntax

A

- ♦ [Section A.1, “Job Class,” on page 197](#)
- ♦ [Section A.2, “Joblet Class,” on page 197](#)
- ♦ [Section A.3, “Utility Classes,” on page 197](#)
- ♦ [Section A.4, “Built-in JDL Functions and Variables,” on page 197](#)
- ♦ [Section A.5, “Job State Field Values,” on page 199](#)
- ♦ [Section A.6, “Repository Information String Values,” on page 200](#)
- ♦ [Section A.7, “Joblet State Values,” on page 200](#)
- ♦ [Section A.8, “Resource Information Values,” on page 201](#)
- ♦ [Section A.9, “JDL Class Definitions,” on page 201](#)

A.1 Job Class

To review the detailed JDL structure of the joblet class, see [Job \(page 222\)](#).

A.2 Joblet Class

To review the detailed JDL structure of the joblet class, see [Joblet \(page 224\)](#).

A.3 Utility Classes

The following are some of the main utility JDL classes you can use to customize your Orchestrator jobs:

- ♦ [DataGrid \(page 212\)](#)
- ♦ [Exec \(page 215\)](#)
- ♦ [MatrixInfo \(page 229\)](#)
- ♦ [ResourceInfo \(page 237\)](#)
- ♦ [RunJobSpec \(page 238\)](#)
- ♦ [ScheduleSpec \(page 239\)](#)

A.4 Built-in JDL Functions and Variables

The information in this section defines the built-in ZENworks Orchestrator JDL functions and variables.

- ♦ [Section A.4.1, “getMatrix\(\),” on page 198](#)
- ♦ [Section A.4.2, “system\(cmd\),” on page 198](#)
- ♦ [Section A.4.3, “Grid Object TYPE_* Variables,” on page 198](#)

- ♦ [Section A.4.4, “The `__agent__` Variable,” on page 198](#)
- ♦ [Section A.4.5, “The `__jobname__` Variable,” on page 198](#)
- ♦ [Section A.4.6, “The `__mode__` Variable,” on page 199](#)

A.4.1 `getMatrix()`

This function returns the matrix grid object. For more information, see [MatrixInfo \(page 229\)](#).

Purpose: The matrix object is used to retrieve other grid objects in the system.

A.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.

A.4.3 Grid Object `TYPE_*` Variables

The list of variables are constants for grid object type. For more information, see [MatrixInfo \(page 229\)](#).

Variable Names:

```
TYPE_USER
TYPE_JOB
TYPE_RESOURCE
TYPE_VMHOST
TYPE_REPOSITORY
TYPE_USERGROUP
TYPE_JOBGROUP
TYPE_RESOURCEGROUP
TYPE_REPOSITORYGROUP
```

Type: String.

Purpose: Use these in JDL functions for retrieving and creating grid objects.

A.4.4 The `__agent__` Variable

Variable Name: `__agent__`

Type: Boolean.

Purpose: Defines whether the JDL is executing on the agent.

A.4.5 The `__jobname__` Variable

Variable Name: `__jobname__`

Type: String.

Purpose: Defines the name of the deployed job.

A.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.

A.5 Job State Field Values

Here are the job state field values for the [Job \(page 222\)](#) class:

Constant	Value	Description
<code>int CANCELLED_STATE</code>	9	Cancelled end state.
<code>int CANCELLING_STATE</code>	6	Cancelling. Transitions to: Cancelled.
<code>int COMPLETED_STATE</code>	8	Completed end state.
<code>int COMPLETING_STATE</code>	5	Completing. Transitions to: Completing.
<code>int FAILED_STATE</code>	10	Failed end state.
<code>int FAILING_STATE</code>	7	Failing. Transitions to: Failed.
<code>int PAUSED_STATE</code>	4	Paused. Transitions to: Running/Completing/ Failing/Cancelling.
<code>int QUEUED_STATE</code>	1	Queued. Transitions to: Starting/Failing/ Cancelling.
<code>int RUNNING_STATE</code>	3	Running. Transitions to: Paused/Completing/ Failing/Cancelling.
<code>int STARTING_STATE</code>	2	Starting. Transitions to: Running/Failing/ Cancelling.
<code>int SUBMITTED_STATE</code>	0	Submitted. Transitions to: Queued/Failing.
<code>String TERMINATION_TYPE_ADMIN</code>	"Admin"	Indicates Job was cancelled by the admin and only applies if Job is in <code>CANCELLED_STATE</code> . Value is obtained from <code>jobinstance.terminationtype fact.</code>

Constant	Value	Description
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 <code>jobinstance.terminationtype</code> 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 <code>jobinstance.terminationtype</code> 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 <code>jobinstance.terminationtype</code> fact.

A.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.
TYPE_WAREHOUSE	warehouse	Specifies a warehouse repository.

A.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.

Constant	Value	Description
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 \(page 224\)](#).

A.8 Resource Information Values

Use the following values to specify the

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 \(page 237\)](#).

A.9 JDL Class Definitions

The following Orchestrator 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 203](#)
- ♦ [“BinaryConstraint” on page 204](#)
- ♦ [“BuildSpec” on page 205](#)
- ♦ [“CharRange” on page 206](#)
- ♦ [“ComputedFact” on page 207](#)
- ♦ [“ComputedFactContext” on page 208](#)
- ♦ [“Constraint” on page 209](#)
- ♦ [“ContainerConstraint” on page 210](#)

- ◆ “ContainsConstraint” on page 211
- ◆ “DataGrid” on page 212
- ◆ “DefinedConstraint” on page 213
- ◆ “EqConstraint” on page 214
- ◆ “Exec” on page 215
- ◆ “ExecError” on page 216
- ◆ “FileRange” on page 217
- ◆ “GeConstraint” on page 218
- ◆ “GridObjectInfo” on page 219
- ◆ “GroupInfo” on page 220
- ◆ “GtConstraint” on page 221
- ◆ “Job” on page 222
- ◆ “JobInfo” on page 223
- ◆ “Joblet” on page 224
- ◆ “JobletInfo” on page 225
- ◆ “JobletParameterSpace” on page 226
- ◆ “LeConstraint” on page 227
- ◆ “LtConstraint” on page 228
- ◆ “MatrixInfo” on page 229
- ◆ “NeConstraint” on page 230
- ◆ “NotConstraint” on page 231
- ◆ “OrConstraint” on page 232
- ◆ “ParameterSpace” on page 233
- ◆ “PolicyInfo” on page 234
- ◆ “ProvisionSpec” on page 235
- ◆ “RepositoryInfo” on page 236
- ◆ “ResourceInfo” on page 237
- ◆ “RunJobSpec” on page 238
- ◆ “ScheduleSpec” on page 239
- ◆ “Timer” on page 240
- ◆ “UndefinedConstraint” on page 241
- ◆ “UserInfo” on page 242
- ◆ “VMHostInfo” on page 243
- ◆ “VmSpec” on page 244

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 \(page 210\)](#)
- ◆ Javadoc: *AndConstraint* (http://www.novell.com/documentation/zen_orchestrator13/reference/jldoc/com/novell/zos/jdl/AndConstraint.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 \(page 211\)](#), [EqConstraint \(page 214\)](#), [GeConstraint \(page 218\)](#), [GtConstraint \(page 221\)](#), [LeConstraint \(page 227\)](#), [LtConstraint \(page 228\)](#), [NeConstraint \(page 230\)](#).
- ◆ Javadoc: [BinaryConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/com/novell/zos/jdl/BinaryConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/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/zen_orchestrator13/reference/jdl/doc/com/novell/zos/jdl/BuildSpec.html)

CharRange

Defines the attributes for creating a virtual machine. An instance of this class is passed to `resource.createInstance()`, `resource.createTemplate()`, `resource.clone()`.

See Also

- ♦ Javadoc: *CharRange* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/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 \(page 208\)](#)
- ♦ Javadoc: *ComputedFact* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/com/novell/zos/jdl/CharRange.html)

ComputedFactContext

Provides access to the evaluation context. See Example below.

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, VM host, and repository grid objects.

The VM host 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 \(page 207\)](#)
- ♦ Javadoc: [ComputedFactContext](http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/com/novell/zos/jdl/ComputedFactContext.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/com/novell/zos/jdl/ComputedFactContext.html)

Constraint

Defines the base class for all constraint classes.

See Also

- ♦ [BinaryConstraint \(page 204\)](#), [ContainerConstraint \(page 210\)](#), [DefinedConstraint \(page 213\)](#), [UndefinedConstraint \(page 241\)](#).
- ♦ Javadoc: *Constraint* (http://www.novell.com/documentation/zen_orchestrator13/reference/jldoc/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 203), [NotConstraint](#) (page 231), [OrConstraint](#) (page 232)
- ◆ Javadoc: [ContainerConstraint](http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/com/novell/zos/jdl/ContainerConstraint.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/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/zen_orchestrator13/reference/jldoc/com/novell/zos/jdl/ContainsConstraint.html)

DataGrid

General interface to the datagrid. See [Chapter 3, “The ZENworks Orchestrator Datagrid,”](#) on [page 43](#).

See Also

- ♦ [GridObjectInfo \(page 219\)](#)
- ♦ Javadoc: *DataGrid* (http://www.novell.com/documentation/zen_orchestrator13/reference/jldoc/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 \(page 209\)](#), [ContainerConstraint \(page 210\)](#), and [ContainsConstraint \(page 211\)](#)
- ♦ Javadoc: *DefinedConstraint* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/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 \(page 204\)](#).

See Also

- ♦ [BinaryConstraint \(page 204\)](#)
- ♦ Javadoc: *EqConstraint* (http://www.novell.com/documentation/zen_orchestrator13/reference/jldoc/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

- ♦ [BinaryConstraint \(page 204\)](#) and [ExecError \(page 216\)](#)
- ♦ Javadoc: *Exec* (http://www.novell.com/documentation/zen_orchestrator13/reference/jldoc/com/novell/zos/jdl/Exec.html)

ExecError

`ExecError` is raised for errors in executing a command line using the [Exec \(page 215\)](#) 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 \(page 215\)](#)
- ◆ Javadoc: *ExecError* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdldoc/com/novell/zos/jdl/ExecError.html)

FileRange

Define a range of values for a [ParameterSpace \(page 233\)](#) 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 \(page 212\)](#) file URL.

See Also

- ♦ [DataGrid \(page 212\)](#) and [ParameterSpace \(page 233\)](#)
- ♦ Javadoc: *FileRange* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/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 lexicographical 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 \(page 204\)](#).

See Also

- ♦ [Constraint \(page 209\)](#) and [BinaryConstraint \(page 204\)](#).
- ♦ Javadoc: *GeConstraint* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdldoc/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/zen_orchestrator13/reference/jldoc/com/novell/zos/jdl/GridObjectInfo.html)

GroupInfo

The 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 \(page 219\)](#).

See Also

- ◆ [GridObjectInfo \(page 219\)](#)
- ◆ Javadoc: *GroupInfo* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/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 lexicographical 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 \(page 209\)](#)
- ♦ Javadoc: *GtConstraint* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/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.

See Also

- ♦ [JobInfo \(page 223\)](#), [Joblet \(page 224\)](#), [JobletInfo \(page 225\)](#), [JobletParameterSpace \(page 226\)](#)
- ♦ Javadoc: *Job* (http://www.novell.com/documentation/zen_orchestrator13/reference/jldoc/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 \(page 222\)](#)
- ♦ Javadoc: *JobInfo* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/com/novell/zos/jdl/Job.html)

Joblet

Defines the attributes for creating a virtual machine. An instance of this class is passed to `resource.createInstance()`, `resource.createTemplate()`, `resource.clone()`.

See Also

- ◆ Job, JobInfo
- ◆ Javadoc: *Joblet* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdldoc/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/zen_orchestrator13/reference/jldoc/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/zen_orchestrator13/reference/jdldoc/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 lexicographical 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/zen_orchestrator13/reference/jldoc/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 lexicographical 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/zen_orchestrator13/reference/jldoc/com/novell/zos/jdl/LtConstraint.html)

MatrixInfo

The `MatrixInfo` class is a representation of the matrix grid object (see [GridObjectInfo \(page 219\)](#)). 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/zen_orchestrator13/reference/jdldoc/com/novell/zos/jdl/MatrixInfo.html)
- ♦ [Section A.4, “Built-in JDL Functions and Variables,” on page 197.](#)

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/zen_orchestrator13/reference/jldoc/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 \(page 209\)](#) and [ContainerConstraint \(page 210\)](#).
- ◆ Javadoc: *NotConstraint* (http://www.novell.com/documentation/zen_orchestrator13/reference/jldoc/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 \(page 209\)](#) and [ContainerConstraint \(page 210\)](#).
- ◆ Javadoc: *OrConstraint* (http://www.novell.com/documentation/zen_orchestrator13/reference/jldoc/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.

See Also

Javadoc: *ParameterSpace* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdldoc/com/novell/zos/jdl/ParameterSpace.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/zen_orchestrator13/reference/jdldoc/com/novell/zos/jdl/PolicyInfo.html)

ProvisionSpec

Defines the attributes for starting a provision. An instance of this class is passed to `self.provision()`. Defining a provision to reserve a provisioned resource "nightly" for a user is an instance of when this function might be used: `spec = ProvisionSpec()`
`spec.setReserveForUser('nightly')` `self.provision(spec)`

See Also

Javadoc: *ProvisionSpec* (http://www.novell.com/documentation/zen_orchestrator13/reference/jldoc/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 \(page 229\)](#) for how to script creation of Repository objects.

See Also

- ◆ See [GridObjectInfo \(page 219\)](#) and [MatrixInfo \(page 229\)](#).
- ◆ Javadoc: *RepositoryInfo* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdldoc/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 \(page 219\)](#) and adds the provisioning operations for provisionable resources such as VMs. See [MatrixInfo \(page 229\)](#) for how to script creation of Resource objects.

See Also

- ♦ [GridObjectInfo \(page 219\)](#) and [MatrixInfo \(page 229\)](#).
- ♦ Javadoc: *ResourceInfo* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/com/novell/zos/jdl/ResourceInfo.html)

RunJobSpec

Defines the attributes for starting a child job. An instance of this class is passed to `self.runJob()`.

See Also

Javadoc: *RunJobSpec* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdldoc/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()`. `schedule()` creates the joblets and schedules joblets to run on resources.

See Also

- ◆ [Joblet \(page 224\)](#)
- ◆ Javadoc: *ScheduleSpec* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/com/novell/zos/jdl/ScheduleSpec.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. Any Timers created in a job or joblet are shut down on job or joblet completion.

See Also

Javadoc: *Timer* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdldoc/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 \(page 209\)](#)
- ♦ Javadoc: *UndefinedConstraint* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/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/zen_orchestrator13/reference/jdldoc/com/novell/zos/jdl/UserInfo.html)

VMHostInfo

The `VMHostInfo` class is a representation of a virtual machine 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

- ♦ [VmSpec \(page 244\)](#)
- ♦ Javadoc: *VMHostInfo* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdl/doc/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()`.

See Also

- ♦ [VMHostInfo \(page 243\)](#)
- ♦ Javadoc: *VmSpec* (http://www.novell.com/documentation/zen_orchestrator13/reference/jdldoc/com/novell/zos/jdl/VmSpec.html)

Orchestrator Client SDK

B

This section provides the reference information for the internal Java* classes used by the Novell® ZENworks Orchestrator® Client:

- ◆ [Section B.1, “Constraint Package,” on page 245](#)
- ◆ [Section B.2, “Datagrid Package,” on page 249](#)
- ◆ [Section B.3, “Grid Package,” on page 250](#)
- ◆ [Section B.4, “Toolkit Package,” on page 258](#)

B.1 Constraint Package

The following Java files form the interfaces and exceptions for the internal Orchestrator constraint grid structure:

- ◆ [Section B.1.1, “AndConstraint,” on page 245](#)
- ◆ [Section B.1.2, “BetweenConstraint,” on page 246](#)
- ◆ [Section B.1.3, “BinaryConstraint,” on page 246](#)
- ◆ [Section B.1.4, “Constraint,” on page 246](#)
- ◆ [Section B.1.5, “ContainerConstraint,” on page 246](#)
- ◆ [Section B.1.6, “ContainsConstraint,” on page 246](#)
- ◆ [Section B.1.7, “ConstraintException,” on page 246](#)
- ◆ [Section B.1.8, “DefinedConstraint,” on page 247](#)
- ◆ [Section B.1.9, “EqConstraint,” on page 247](#)
- ◆ [Section B.1.10, “GeConstraint,” on page 247](#)
- ◆ [Section B.1.11, “GtConstraint,” on page 247](#)
- ◆ [Section B.1.12, “IfConstraint,” on page 247](#)
- ◆ [Section B.1.13, “LeConstraint,” on page 247](#)
- ◆ [Section B.1.14, “LtConstraint,” on page 247](#)
- ◆ [Section B.1.15, “NeConstraint,” on page 248](#)
- ◆ [Section B.1.16, “NotConstraint,” on page 248](#)
- ◆ [Section B.1.17, “OperatorConstraint,” on page 248](#)
- ◆ [Section B.1.18, “OrConstraint,” on page 248](#)
- ◆ [Section B.1.19, “TypedConstraint,” on page 248](#)
- ◆ [Section B.1.20, “UndefinedConstraint,” on page 248](#)

B.1.1 AndConstraint

Perform a logical and-ing of all child constraints.

For complete documentation of the class, see [AndConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/AndConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/AndConstraint.html).

B.1.2 BetweenConstraint

Binary Operator Constraints that have both a left and right side.

For complete documentation of the class, see [BetweenConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/BetweenConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/BetweenConstraint.html).

B.1.3 BinaryConstraint

Binary Operator Constraints that have both a left and right side.

For complete documentation of the class, see [BinaryConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/BinaryConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/BinaryConstraint.html).

B.1.4 Constraint

Basic Constraint interface which allows traversal and evaluation of a constraint tree.

For complete documentation of the class, see [Constraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/Constraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/Constraint.html).

B.1.5 ContainerConstraint

Container constraints that perform logical aggregation operations on contained constraints.

For complete documentation of the class, see [ContainerConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/ContainerConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/ContainerConstraint.html).

B.1.6 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 \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/ContainsConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/ContainsConstraint.html).

B.1.7 ConstraintException

For exceptions that occur in parsing or executing constraints.

For complete documentation of the class, see [ConstraintException \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/ConstraintException.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/ConstraintException.html).

B.1.8 DefinedConstraint

Evaluates to true only if the left side fact is defined in the match context.

For complete documentation of the class, see [DefinedConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/DefinedConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/DefinedConstraint.html).

B.1.9 EqConstraint

Performs a equality constraint operation.

For complete documentation of the class, see [EqConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/EqConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/EqConstraint.html).

B.1.10 GeConstraint

Performs a 'greater than or equal to' constraint operation.

For complete documentation of the class, see [GeConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/GeConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/GeConstraint.html).

B.1.11 GtConstraint

Performs a 'greater than' constraint operation.

For complete documentation of the class, see [GtConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/GtConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/GtConstraint.html).

B.1.12 IfConstraint

Perform a conditional if,then,else block.

For complete documentation of the class, see [IfConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/IfConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/IfConstraint.html).

B.1.13 LeConstraint

Performs a 'less than or equal to' constraint operation.

For complete documentation of the class, see [LeConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/LeConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/LeConstraint.html).

B.1.14 LtConstraint

Performs a 'less than' constraint operation.

For complete documentation of the class, see [LtConstraint \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/LtConstraint.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/LtConstraint.html).

B.1.15 NeConstraint

Performs a not equal constraint operation.

For complete documentation of the class, see [NeConstraint](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/NeConstraint.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/NeConstraint.html).

B.1.16 NotConstraint

Perform a logical not operation of all the child constraints.

For complete documentation of the class, see [NotConstraint](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/NotConstraint.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/NotConstraint.html).

B.1.17 OperatorConstraint

Operator constraints that perform comparison operation on facts.

For complete documentation of the class, see [OperatorConstraint](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/OperatorConstraint.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/OperatorConstraint.html).

B.1.18 OrConstraint

Perform a logical or-ing operation of all the child constraints.

For complete documentation of the class, see [OrConstraint](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/OrConstraint.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/OrConstraint.html).

B.1.19 TypedConstraint

Typed constraint must only be used as the outermost wrapper when it is necessary to override the default constraint type of 'resource'.

For complete documentation of the class, see [TypedConstraint](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/TypedConstraint.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/TypedConstraint.html).

B.1.20 UndefinedConstraint

Evaluates to true only if the left side fact is not defined in the match context.

For complete documentation of the class, see [UndefinedConstraint](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/UndefinedConstraint.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/constraint/UndefinedConstraint.html).

B.2 Datagrid Package

The following Java files form the classes, interfaces, and exceptions for the internal Orchestrator datagrid structure:

- ♦ [Section B.2.1, “DGLogger,” on page 249](#)
- ♦ [Section B.2.2, “GridFile,” on page 249](#)
- ♦ [Section B.2.3, “GridFileFilter,” on page 249](#)
- ♦ [Section B.2.4, “GridFileNameFilter,” on page 249](#)
- ♦ [Section B.2.5, “DataGridException,” on page 249](#)
- ♦ [Section B.2.6, “DataGridNotAvailableException,” on page 250](#)
- ♦ [Section B.2.7, “GridFile.CancelException,” on page 250](#)

B.2.1 DGLogger

Definitions of the DataGrid Logger options used for multicast.

For complete documentation of the class, see [DGLogger](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/DGLogger.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/DGLogger.html).

B.2.2 GridFile

Specifies the ZENworks Orchestrator datagrid interface for individual files and directories.

For complete documentation of the class, see [GridFile](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/GridFile.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/GridFile.html).

B.2.3 GridFileFilter

Filter for accepting/rejecting file names in a directory list.

For complete documentation of the class, see [GridFileFilter](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/GridFileFilter.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/GridFileFilter.html).

B.2.4 GridFileNameFilter

Filter for accepting/rejecting file names in a directory list.

For complete documentation of the class, see [GridFileNameFilter](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/GridFileNameFilter.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/GridFileNameFilter.html).

B.2.5 DataGridException

General exception class for datagrid errors.

For complete documentation of the class, see [DataGridException](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/DataGridException.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/DataGridException.html).

B.2.6 DataGridNotAvailableException

Exception thrown if the datagrid cannot be reached due to a network error.

For complete documentation of the class, see [DataGridNotAvailableException \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/DataGridNotAvailableException.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/DataGridNotAvailableException.html).

B.2.7 GridFile.CancelException

Exception thrown by cancelled requests.

For complete documentation of the class, see [GridFile.CancelException \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/GridFile.CancelException.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/dataGrid/GridFile.CancelException.html).

B.3 Grid Package

The Java classes included in the Grid package form the basis of the ZENworks Orchestrator infrastructure. For complete documentation of each class, click on the links to access the online documentation Javadoc.

- ◆ [Section B.3.1, “AgentListener,” on page 251](#)
- ◆ [Section B.3.2, “ClientAgent,” on page 251](#)
- ◆ [Section B.3.3, “Credential,” on page 251](#)
- ◆ [Section B.3.4, “Fact,” on page 252](#)
- ◆ [Section B.3.5, “FactSet,” on page 252](#)
- ◆ [Section B.3.6, “GridObjectInfo,” on page 252](#)
- ◆ [Section B.3.7, “ID,” on page 252](#)
- ◆ [Section B.3.8, “JobInfo,” on page 252](#)
- ◆ [Section B.3.9, “Message,” on page 252](#)
- ◆ [Section B.3.10, “Message.Ack,” on page 252](#)
- ◆ [Section B.3.11, “Message.AuthFailure,” on page 253](#)
- ◆ [Section B.3.12, “Message.ClientResponse,” on page 253](#)
- ◆ [Section B.3.13, “Message.Event,” on page 253](#)
- ◆ [Section B.3.14, “Message.GetGridObjects,” on page 253](#)
- ◆ [Section B.3.15, “Message.GridObjects,” on page 253](#)
- ◆ [Section B.3.16, “Message.JobAccepted,” on page 253](#)
- ◆ [Section B.3.17, “Message.JobError,” on page 253](#)
- ◆ [Section B.3.18, “Message.JobFinished,” on page 254](#)
- ◆ [Section B.3.19, “Message.JobIdEvent,” on page 254](#)
- ◆ [Section B.3.20, “Message.JobInfo,” on page 254](#)
- ◆ [Section B.3.21, “Message.Jobs,” on page 254](#)
- ◆ [Section B.3.22, “Message.JobStarted,” on page 254](#)

- ◆ Section B.3.23, “Message.JobStatus,” on page 254
- ◆ Section B.3.24, “Message.LoginFailed,” on page 254
- ◆ Section B.3.25, “Message.LoginSuccess,” on page 255
- ◆ Section B.3.26, “Message.LogoutAck,” on page 255
- ◆ Section B.3.27, “Message.NeedUpgrade,” on page 255
- ◆ Section B.3.28, “Message.RunningJobs,” on page 255
- ◆ Section B.3.29, “Message.ServerStatus,” on page 255
- ◆ Section B.3.30, “Message.SessionAck,” on page 255
- ◆ Section B.3.31, “Message.SessionChallenge,” on page 255
- ◆ Section B.3.32, “Message.SessionResponse,” on page 256
- ◆ Section B.3.33, “Message.SessionStatus,” on page 256
- ◆ Section B.3.34, “Node,” on page 256
- ◆ Section B.3.35, “Priority,” on page 256
- ◆ Section B.3.36, “WorkflowInfo,” on page 256
- ◆ Section B.3.37, “ClientOutOfDateException,” on page 256
- ◆ Section B.3.38, “FactException,” on page 256
- ◆ Section B.3.39, “GridAuthenticationException,” on page 257
- ◆ Section B.3.40, “GridAuthorizationException,” on page 257
- ◆ Section B.3.41, “GridConfigurationException,” on page 257
- ◆ Section B.3.42, “GridDeploymentException,” on page 257
- ◆ Section B.3.43, “GridException,” on page 257
- ◆ Section B.3.44, “GridObjectNotFoundException,” on page 257

B.3.1 AgentListener

Provides the interface necessary for processing messages sent from the Orchestrator Server.

For complete documentation, see [AgentListener](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/AgentListener.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/AgentListener.html).

B.3.2 ClientAgent

API for client communication with server for job and datagrid operations.

For complete documentation, see [ClientAgent](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/ClientAgent.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/ClientAgent.html).

B.3.3 Credential

A credential used for identity on the ZENworks Orchestrator system.

For complete documentation, see [Credential](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Credential.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Credential.html).

B.3.4 Fact

The Grid Fact object.

For complete documentation, see [Fact](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Fact.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Fact.html).

B.3.5 FactSet

Definition of a set of facts.

For complete documentation, see [FactSet](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/FactSet.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/FactSet.html).

B.3.6 GridObjectInfo

Client interface to any Grid object.

For complete documentation, see [GridObjectInfo](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridObjectInfo.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridObjectInfo.html).

B.3.7 ID

A unique identifier for an engine or a facility or Grid object.

For complete documentation, see [ID](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/class-use/ID.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/class-use/ID.html).

B.3.8 JobInfo

A client representation of a deployed job.

For complete documentation, see [JobInfo](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/JobInfo.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/JobInfo.html).

B.3.9 Message

A base interface for all the messages in the system.

For complete documentation, see [Message](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.html).

B.3.10 Message.Ack

A general acknowledgement of “action” message.

For complete documentation, see [Message.Ack](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.Ack.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.Ack.html).

B.3.11 Message.AuthFailure

Authentication failure messages indicates 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/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.AuthFailure.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.AuthFailure.html).

B.3.12 Message.ClientResponse

Message All messages that can optionally carry an error string back to the client extend this.

For complete documentation, see [Message.ClientResponse](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.ClientResponseMessage.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.ClientResponseMessage.html).

B.3.13 Message.Event

An Event is used to signal clients and workflows.

For complete documentation, see [Message.Event](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.Event.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.Event.html).

B.3.14 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/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.GetGridObjects.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.GetGridObjects.html).

B.3.15 Message.GridObjects

Server response to client request to retrieve grid a grid object set.

For complete documentation, see [Message.GridObjects](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.GetGridObjects.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.GetGridObjects.html).

B.3.16 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/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobAccepted.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobAccepted.html).

B.3.17 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/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobError.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobError.html).

B.3.18 Message.JobFinished

A JobFinished message is sent when processing of a job completes.

For complete documentation, see [Message.JobFinished](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobFinished.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobFinished.html).

B.3.19 Message.JobIdEvent

Base Event interface for retrieving JobID used for jobid messages.

For complete documentation, see [Message.JobIdEvent](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobIdEvent.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobIdEvent.html).

B.3.20 Message.JobInfo

A JobInfo message contains information describing a deployed job.

For complete documentation, see [Message.JobInfo](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobInfo.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobInfo.html).

B.3.21 Message.Jobs

A Jobs message contains a list of deployed job names.

For complete documentation, see [Message.Jobs](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.Jobs.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.Jobs.html).

B.3.22 Message.JobStarted

A JobStarted message is sent when a job is successfully started.

For complete documentation, see [Message.JobStarted](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobStarted.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobStarted.html).

B.3.23 Message.JobStatus

A JobStatus message contains the state of the specified job.

For complete documentation, see [Message.JobStatus](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobStatus.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.JobStatus.html).

B.3.24 Message.LoginFailed

Response message for an unsuccessful login

For complete documentation, see [Message.LoginFailed](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.LoginFailed.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.LoginFailed.html).

B.3.25 Message.LoginSuccess

Response message for a successful login.

For complete documentation, see [Message.LoginSuccess](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.LoginSuccess.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.LoginSuccess.html).

B.3.26 Message.LogoutAck

A LogoutAck indicates success or failure of logout operation.

For complete documentation, see [Message.LogoutAck](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.LogoutAck.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.LogoutAck.html).

B.3.27 Message.NeedUpgrade

to be documented

For complete documentation, see [Message.NeedUpgrade](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.NeedUpgrade.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.NeedUpgrade.html).

B.3.28 Message.RunningJobs

A RunningJobs message contains the list of running jobs.

For complete documentation, see [Message.RunningJobs](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.RunningJobs.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.RunningJobs.html).

B.3.29 Message.ServerStatus

A ServerStatus message.

For complete documentation, see [Message.ServerStatus](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.ServerStatus.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.ServerStatus.html).

B.3.30 Message.SessionAck

to be documented

For complete documentation, see [Message.SessionAck](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.SessionAck.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.SessionAck.html).

B.3.31 Message.SessionChallenge

to be documented

For complete documentation, see [Message.SessionChallenge](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.SessionChallenge.html) (http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.SessionChallenge.html).

B.3.32 Message.SessionResponse

to be documented

For complete documentation, see [Message.SessionResponse \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.SessionResponse.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.SessionResponse.html).

B.3.33 Message.SessionStatus

to be documented

For complete documentation, see [Message.SessionStatus \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.SessionStatus.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Message.SessionStatus.html).

B.3.34 Node

Internal interface for Node (Resource) Grid object.

For complete documentation, see [Node \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Node.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Node.html).

B.3.35 Priority

Priority information.

For complete documentation, see [Priority \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Priority.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/Priority.html).

B.3.36 WorkflowInfo

A WorkflowInfo can represent either a snapshot of a running instance or an historical record of an instance.

For complete documentation, see [WorkflowInfo \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/WorkflowInfo.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/WorkflowInfo.html).

B.3.37 ClientOutOfDateException

Grid exception indicating the client is not compatible with the server.

For complete documentation, see [ClientOutOfDateException \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/ClientOutOfDateException.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/ClientOutOfDateException.html).

B.3.38 FactException

For exceptions that occur in accessing or setting facts.

For complete documentation, see [FactException \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/FactException.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/FactException.html).

B.3.39 GridAuthenticationException

Thrown when authentication is denied by a ZENworks Orchestrator Server.

For complete documentation, see [GridAuthenticationException \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridAuthenticationException.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridAuthenticationException.html).

B.3.40 GridAuthorizationException

Thrown when credentials are insufficient for the desired grid operation.

For complete documentation, see [GridAuthorizationException \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridAuthorizationException.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridAuthorizationException.html).

B.3.41 GridConfigurationException

Grid exception thrown to indicate a Grid configuration error.

For complete documentation, see [GridConfigurationException \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridConfigurationException.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridConfigurationException.html).

B.3.42 GridDeploymentException

Thrown when credentials are insufficient for the desired grid operation.

For complete documentation, see [GridDeploymentException \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridDeploymentException.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridDeploymentException.html).

B.3.43 GridException

The base exception for all Grid exceptions.

For complete documentation, see [GridException \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridException.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridException.html).

B.3.44 GridObjectNotFoundException

Thrown when a grid object lookup does not find the requested object.

For complete documentation, see [GridObjectNotFoundException \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridObjectNotFoundException.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/grid/GridObjectNotFoundException.html).

B.4 Toolkit Package

The Client agent, Constraint, and Credentials factory patterns used internally by Orchestrator Server:

- ♦ [Section B.4.1, “ClientAgentFactory,” on page 258](#)
- ♦ [Section B.4.2, “ConstraintFactory,” on page 258](#)
- ♦ [Section B.4.3, “CredentialFactory,” on page 258](#)

B.4.1 ClientAgentFactory

Factory pattern used to create new clients for connection to a ZENworks Orchestrator Server.

For complete documentation of the class, see [ClientAgentFactory \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/toolkit/ClientAgentFactory.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/toolkit/ClientAgentFactory.html).

B.4.2 ConstraintFactory

Factory pattern used to create constraint objects which may be combined into larger constraint hierarchies for use in searches or other constraint based matching.

For complete documentation of the class, see [ConstraintFactory \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/toolkit/ConstraintFactory.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/toolkit/ConstraintFactory.html).

B.4.3 CredentialFactory

Factory pattern used to create a Credential used for connection to a ZENworks Orchestrator Server.

For complete documentation of the class, see [CredentialFactory \(http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/toolkit/CredentialFactory.html\)](http://www.novell.com/documentation/zen_orchestrator13/reference/javadoc/com/novell/zos/toolkit/CredentialFactory.html).

Documentation Updates

C

This section contains information on documentation content changes that have been made in the Developer Guide and Reference since the initial release of Novell® ZENworks® Orchestrator 1.3 (June 9, 2008). The information will help you to keep current on updates to the documentation.

All changes that are noted in this section were also made in the documentation. The documentation is provided on the Web in two formats: HTML and PDF. The HTML and PDF documentation are both kept up-to-date with the documentation changes listed in this section.

The documentation update information is grouped according to the date the changes were published. Within a dated section, the changes are alphabetically listed by the names of the main table of contents sections for ZENworks Orchestrator 1.3.

If you need to know whether a copy of the PDF documentation you are using is the most recent, the PDF document contains the date it was published on the front title page.

The documentation was updated on the following dates:

- ♦ [Section C.1, “Aug 3, 2008,” on page 259](#)

C.1 Aug 3, 2008

The following sections are affected by updates:

- ♦ [Section C.1.1, “Documentation Updates,” on page 259](#)

C.1.1 Documentation Updates

The following changes were made in this section:

Location	Change
Chapter 3, “The ZENworks Orchestrator Datagrid,” on page 43	Changed the name of this chapter to better reflect its contents.