



C o m m u n i t y   E x p e r i e n c e   D i s t i l l e d

# Zenoss Core 3.x Network and System Monitoring

A step-by-step guide to configuring, using, and adapting this free Open Source network monitoring system

**Michael Badger**

**[PACKT]** open source\*  
PUBLISHING community experience distilled

[www.it-ebooks.info](http://www.it-ebooks.info)

# Zenoss Core 3.x Network and System Monitoring

A step-by-step guide to configuring, using, and adapting this free Open Source network monitoring system

**Michael Badger**



BIRMINGHAM - MUMBAI

# Zenoss Core 3.x Network and System Monitoring

Copyright © 2011 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2011

Production Reference: 1120411

Published by Packt Publishing Ltd.  
32 Lincoln Road  
Olton  
Birmingham, B27 6PA, UK.

ISBN 978-1-849511-58-2

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Asher Wishkerman ([a.wishkerman@mpic.de](mailto:a.wishkerman@mpic.de) )

# Credits

**Author**

Michael Badger

**Editorial Team Leader**

Akshara Aware

**Reviewers**

Andrea Consadori

Jonny Gerold

Duncan M. McGreggor

Charlie Schluting

**Project Team Leader**

Priya Mukherji

**Project Coordinator**

Jovita Pinto

**Development Editor**

Neha Mallik

**Proofreader**

Stephen Silk

**Technical Editor**

Conrad Sardinha

**Graphics**

Nilesh Mohite

**Indexer**

Tejal Daruwale

**Production Coordinator**

Melwyn D'sa

**Cover Work**

Melwyn D'sa



# About the Author

Michael Badger is a freelance technical communicator with a knack for helping other people understand and use their computer software and technology. In addition to writing a previous book about Zenoss Core: *Zenoss Core Network and System Monitoring*, Badger authored *Scratch 1.4: Beginner's Guide*, a Scratch programming tutorial.

He lives in north central Pennsylvania (United States) on a small farm and has recently taken to raising pastured chickens, honeybees, and pigs. Michael is searching for a way to integrate Zenoss Core into the hen house so that he can receive an alert each time an egg is laid.

For more information, visit [www.badgerfiles.com/zenoss3](http://www.badgerfiles.com/zenoss3).

---

There are so many people to thank, starting with my family. They tolerate my late nights and weekend work.

My team at Packt deserves a nod for finally helping me get this revision done. Thanks for the help Rakesh Shejwal and Jovita Pinto.

Then there are the reviewers. It's not easy to provide substantive critique of another person's work because it takes time and thoughtful consideration for you to want to make my work better. You should know that even though I did not incorporate all your suggestions, I considered them carefully.

---

# About the Reviewers

**Andrea Consadori** is the lead technical support at Lais s.r.l. and has been working with Zenoss to monitor customer IT infrastructures for the past four years.

He has been implementing simple zenpacks to integrate all of the vendors' products he uses like Motorola Canopy/PTP, Alvarion, and so on.

Lais s.r.l. is a WISP and uses Zenoss to monitor its wireless infrastructure.

Earlier, he worked at Edslan s.p.a. (an Italian networking product distributor) where he studied lots of networking brands.

Andrea is passionate about solving hard networking issues and enjoys working with routing protocol and firewall rules.

---

I would like to thank Michela for her encouragement that makes my skill and knowledge grow every day.

---

**Jonny Gerold** loves Open Source, enjoys working with Linux/Unix/Solaris, and also enjoys dirt biking.

**Duncan M. McGregor** started his programming career at the ripe old age of 11 in the early 80s. From his adventures in rewriting games on Kaypro's luggable CP/M machine to the open source world, programming has been his passion. When Duncan wasn't hacking, he was an Army MI linguist; worked his way up to sous chef in a Massachusetts restaurant; studied quantum mechanics and mathematics as a physics major; learned meditation while living with Tibetan monks; and started his own software consulting company. His contract work included systems management solutions for the U.S. Federal Government as well as Zenoss, Inc. After consulting for several years, Duncan joined an engineering startup as the COO and eventually left that position for Canonical where he manages teams in the Product Strategy group, improving the Ubuntu Linux distribution.

**Charlie Schluting**, BS CS, MBA; is first a sysadmin, and second a technology strategy connoisseur, currently working as the IS Operations Manager at Canonical (the creators of Ubuntu Linux). Charlie also wrote *Network Ninja*, <http://stores.lulu.com/schluting>, a book designed to educate sysadmins and mid-level network engineers on the fundamentals of the protocols they work with. Charlie can frequently be found dabbling in various technology startups, attempting to change the world, when he's not touring off-road on his motorcycle (<http://charlierides.com>).

# www.PacktPub.com

## Support files, eBooks, discount offers and more

You might want to visit [www.PacktPub.com](http://www.PacktPub.com) for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## Why Subscribe?

- Fully searchable across every book published by Packt
- Copy & paste, print and bookmark content
- On demand and accessible via web browser

## Free Access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.



# Table of Contents

<b>Preface</b>	<b>1</b>
<b>Chapter 1: Network and System Monitoring with Zenoss Core</b>	<b>7</b>
Device management	9
Availability and performance monitors	10
Event management	11
Plugin architecture	12
System reports	13
Custom device reports	14
System architecture	14
User layer	15
Data layer	16
Collection layer	17
Device management daemons	18
Performance and availability daemons	18
Event daemons	19
Summary	20
<b>Chapter 2: Discovering Devices</b>	<b>21</b>
Zenoss Core installation	22
Preparing devices for monitoring	22
SNMP	23
SNMP versions	24
Configuring SNMP on Linux	25
Configuring SNMP and WMI on Windows	26
Zenoss Plugins	28
Installing Zenoss Plugins	29
Port scan	30
Opening monitoring-specific ports	30
Configuring Linux firewalls	31
Configuring Windows firewall	32



<b>Zenoss Core setup wizard</b>	<b>33</b>
Step 1: Setting up users	33
Step 2: Specify or discover devices to monitor	34
Adding devices	35
Manually find devices	35
Autodiscover devices	37
Our device inventory: A job well done	38
Reviewing device creation job log	40
<b>Adding a single device</b>	<b>42</b>
Entering device attributes	43
<b>Importing a list of devices with zenbatchload</b>	<b>46</b>
<b>Command line discovery with zendisc</b>	<b>48</b>
<b>Summary</b>	<b>49</b>
<b>Chapter 3: Device Setup and Administration</b>	<b>51</b>
<b>Organizing devices in Zenoss Core</b>	<b>52</b>
Locations	52
Systems and Groups	54
Organizer details	54
Editing organizers	56
Moving organizers	56
Classes	56
Viewing a list of device classes	57
Assigning devices to a class	58
<b>Modeling devices</b>	<b>59</b>
Modeler plugins gather device information	60
Assigning modeler plugins	62
<b>Troubleshooting data collection</b>	<b>62</b>
Troubleshooting SNMP problems	62
Running snmpwalk	63
Is the SNMP daemon running on Linux servers?	64
SNMP problems on Windows	64
Troubleshooting WMI problems	64
Zeneventlog—unable to connect to Windows	65
Zenoss Core does not collect WMI data	65
Troubleshooting Zenoss Plugins	66
A class of its own	66
<b>Device administration</b>	<b>67</b>
Locking or unlocking a device	67
Renaming a device	68
Resetting the IP address	69
Push changes	70
Deleting devices	70

---

<b>zProperties defined</b>	<b>71</b>
<b>Summary</b>	<b>74</b>
<b>Chapter 4: Monitor Status and Performance</b>	<b>75</b>
<b>Collectors collect</b>	<b>76</b>
Configuring the performance collector	77
<b>Monitoring components</b>	<b>79</b>
Interfaces	80
OS Processes	81
Add Process	81
Viewing or editing the process details	82
Configuration properties	82
Monitoring OS Processes	83
Services	85
Enable monitoring for a service	85
Configuration properties	87
Monitoring exceptions for services	87
Interactively monitor IP services	88
File Systems	89
Ignoring File Systems with zProperties	89
Network Routes	90
Networks	91
Add Components	92
Viewing and editing component details for a device	93
Performance Graphs	94
Interface template	95
<b>Performance Graphs</b>	<b>96</b>
Working with graphs	97
Monitoring performance thresholds	98
<b>Summary</b>	<b>98</b>
<b>Chapter 5: Custom Monitoring Templates</b>	<b>99</b>
<b>Monitoring Templates</b>	<b>99</b>
<b>Monitoring SNMP data sources</b>	<b>101</b>
Overriding templates	101
Editing the /Server/Linux template	103
Find OIDs for SNMP monitoring	105
<b>Monitoring with Nagios plugins</b>	<b>108</b>
Working with Nagios plugins	111
Nagios return codes	111
Nagios performance data	111
Adding the Nagios plugin to Monitoring Templates	112
Adding a Data Source	114
Adding a Data Point	116
RRDtool Data Point configurations	117

Defining monitoring thresholds	118
Graph definitions	120
RRDtool Graph Point configurations	123
Binding templates to the device class	125
Adding a device to monitor using the Bogo template	126
<b>Monitoring with Cacti plugins</b>	<b>127</b>
Data Source parser	128
<b>Summary</b>	<b>129</b>
<b>Chapter 6: Core Event Management</b>	<b>131</b>
<b>Event Console</b>	<b>132</b>
Event severities defined	133
Event statuses defined	134
Acknowledging an event	134
Viewing an event log	135
Events consoles are everywhere	137
Closing events	137
Displaying historical events	138
<b>Event Manager</b>	<b>138</b>
Event Fields	141
Event commands	143
Creating a command	143
<b>Working with events</b>	<b>145</b>
Simulating an event	145
Clearing the event	147
Event mapping	148
Event Classes	148
Event class zProperties	149
Mapping an event	150
Event mapping sequence	154
<b>Event de-duplication</b>	<b>154</b>
Turning off event de-duplication	155
<b>Summary</b>	<b>156</b>
<b>Chapter 7: Collecting Events</b>	<b>157</b>
<b>Routing syslog messages to Zenoss Core</b>	<b>157</b>
Collecting Cisco router syslogs	159
Testing syslog configuration with Logger	160
<b>Monitoring Windows event logs</b>	<b>161</b>
Windows event log severities	162
Testing the event log configuration with Eventcreate	163

<b>Incorporating event reporting into third-party scripts via zensendevent</b>	<b>163</b>
Simple backup script with zensendevent	165
<b>Creating events by e-mail</b>	<b>166</b>
Zenmail	167
Zenpop3	169
<b>Configuring alerting rules</b>	<b>170</b>
Alert filters	172
Alert escalations	173
Schedule	174
Alert messages	176
<b>Event transformations</b>	<b>177</b>
Some event transformation examples	178
<b>Programming in zendmd, an interactive shell</b>	<b>180</b>
<b>Summary</b>	<b>182</b>
<b>Chapter 8: Settings and Administration</b>	<b>183</b>
<b>Managing Zenoss Core users</b>	<b>183</b>
Administered Objects	185
Event Views	187
Groups	189
<b>Creating custom User Commands</b>	<b>189</b>
Adding a User Command	192
<b>System settings</b>	<b>193</b>
<b>Configuring Zenoss Core's Monitoring Dashboard</b>	<b>194</b>
Locations portlet with Google Maps	196
Device Issues portlet	197
Zenoss Issues portlet	198
Watch List portlet	198
Root Organizers portlet	199
Production States portlet	199
Portlet permissions	200
<b>Meet the Zenoss Daemons</b>	<b>200</b>
<b>Maintenance Windows</b>	<b>202</b>
<b>Adding MIBs</b>	<b>204</b>
<b>Backing up and restoring monitoring data</b>	<b>205</b>
Automating backups with zenbackup	206
Restoring backups with zenrestore	207
<b>Updating Zenoss Core</b>	<b>208</b>
<b>Summary</b>	<b>209</b>

---

<b>Chapter 9: Extending Zenoss Core with ZenPacks</b>	<b>211</b>
<b>Installing community ZenPacks</b>	<b>211</b>
Monitoring websites with HttpMonitor	212
Viewing a list of installed ZenPack objects	215
Configuring HttpMonitor	216
Configuring HttpMonitor settings	217
<b>Creating a ZenPack</b>	<b>218</b>
Adding files and objects to the ZenPack	221
Adding a new data source to the monitoring template	222
Adding objects to a ZenPack	223
Packaging the ZenPack	224
ZenPack development mode	225
<b>Developer resources</b>	<b>226</b>
<b>Summary</b>	<b>226</b>
<b>Chapter 10: Reviewing Built-in Reports</b>	<b>227</b>
<b>Report overview</b>	<b>227</b>
<b>Device Reports</b>	<b>228</b>
New Devices	229
Device Changes	229
Model Collection Age	229
Software Inventory	230
Manufacturers and Products	230
SNMP Status Issues	231
Ping Status Issues	232
All Devices	232
All Monitored Components	232
<b>Event Reports</b>	<b>233</b>
All Event Classes	233
All Event Mappings	234
All Heartbeats	234
<b>Graph Reports</b>	<b>234</b>
<b>Multi-Graph Reports</b>	<b>237</b>
Adding Collections	238
Adding Graph Definitions	240
Adding Graph Groups	241
<b>Performance Reports</b>	<b>243</b>
Aggregate Report	244
Availability	245
CPU Utilization	246
Filesystem Utilization	246
Interface Utilization	247

---

Memory Utilization	248
Threshold Summary	248
<b>User Reports</b>	<b>249</b>
Notification Schedules	249
<b>Summary</b>	<b>249</b>
<b>Chapter 11: Writing Custom Device Reports</b>	<b>251</b>
<b>Creating Custom Device Reports</b>	<b>251</b>
Custom Device Report fields	253
Building Custom Device Report queries	254
Using zendmd to test report queries	255
Exploring data in Zope	258
Using Python expressions in the columns	260
Convenience functions	261
convToUnits	261
<b>Scheduling reports for e-mail delivery</b>	<b>262</b>
Sending a CSV report	263
Scheduling a cron job	263
<b>Summary</b>	<b>263</b>
<b>Appendix A: Event Attributes</b>	<b>265</b>
<b>Appendix B: Device Attributes</b>	<b>269</b>
<b>Appendix C: Example snmpd.conf</b>	<b>273</b>
<b>Index</b>	<b>277</b>

---





# Preface

For system administrators, network engineers, and security analysts, it is essential to keep a track of network traffic.

Zenoss Core is an enterprise-level systems and network monitoring solution that can be as complex as you need it to be. While just about anyone can install it, turn it on, and monitor "something", Zenoss Core has a complicated interface packed with features. The interface has been drastically improved over version 2, but it's still not the type of software you can use intuitively – in other words, a bit of guidance is in order.

The role of this book is to serve as your Zenoss Core tour guide and save you hours, days, maybe weeks of time.

This book will show you how to work with Zenoss and effectively adapt Zenoss for System and Network monitoring. Starting with the Zenoss basics, it requires no existing knowledge of systems management, and whether or not you can recite MIB trees and OIDs from memory is irrelevant. Advanced users will be able to identify ways in which they can customize the system to do more, while less advanced users will appreciate the ease of use Zenoss provides. The book contains step-by-step examples to demonstrate Zenoss Core's capabilities. The best approach to using this book is to sit down with Zenoss and apply the examples found in these pages to your system.

The book covers the monitoring basics: adding devices, monitoring for availability and performance, processing events, and reviewing reports. It also dives into more advanced customizations, such as custom device reports, external event handling (for example, syslog server, zensendevent, and Windows Event Logs), custom monitoring templates using SNMP data sources, along with Nagios, and Cacti plugins. An example of a Nagios-style plugin is included and the book shows you where to get an example of a Cacti-compatible plugin for use as a command data source in monitoring templates.

In Zenoss Core, ZenPacks are modules that add monitoring functionality. Using the Nagios plugin example, you will learn how to create, package, and distribute a ZenPack. You also learn how to explore Zenoss Core's data model using zendmd so that you can more effectively write event transformations and custom device reports.

Implement Zenoss Core and fit it into your security management environment using this easy-to-understand tutorial guide.

## What this book covers

*Chapter 1, Network and System Monitoring with Zenoss Core*, provides an overview of Zenoss Core's monitoring capabilities and system architecture.

In *Chapter 2, Discovering Devices*, we prepare our monitoring environment by configuring SNMP, WMI, SSH, and firewall ports. We'll add devices to Zenoss Core via the setup wizard, zenbatchload, and zendisc.

*Chapter 3, Device Setup and Administration*, configures devices so that we ensure we collect the proper monitoring information by organizing, configuring, and troubleshooting the monitoring properties.

*Chapter 4, Monitor Status and Performance*, monitors and graphs the performance of device components such as routes, windows services, IP services, processes, file systems, and network interfaces.

*Chapter 5, Custom Monitoring Templates*, explores custom monitoring templates by configuring various data sources, including SNMP, Nagios plugins, and Cacti plugins.

*Chapter 6, Core Event Management*, introduces us to processing events via the Event Console. We create custom event commands, learn how to create test events, and perform event mapping.

*Chapter 7, Collecting Events*, allows Zenoss Core to receive and process events from third-party sources, such as syslog, Windows Event Log, e-mail, and home-grown system administration scripts.

*Chapter 8, Settings and Administration*, covers common Zenoss Core administration tasks, such as managing users, the monitoring dashboard, backups, and updates.

*Chapter 9, Extending Zenoss Core with ZenPacks*, installs, creates, and packages add-on modules. ZenPacks extend the functionality of Zenoss Core.

*Chapter 10, Reviewing Built-in Reports*, reviews each of Zenoss Core's included reports to help us troubleshoot, analyze, and view our monitoring performance over time. It also creates custom graph and multi-graph reports.

*Chapter 11, Writing Custom Device Reports*, provides an in-depth look at Zenoss Core's custom device report functionality, including the use of zendmd to explore the Zenoss data model.

*Appendix A, Event Attributes*, lists the available event attributes in Zenoss Core.

*Appendix B, Device Attribute*, lists the attributes that we may use when working with our devices.

*Appendix C, Example snmpd.conf*, lists a sample `snmpd.conf` file.

## What you need for this book

This book will work best if you have a working installation of Zenoss Core and some network servers, routers, switches, and other devices to monitor. Zenoss Core can be installed on Linux, Mac OS X, and Windows (via a virtual Zenoss Virtual Appliance and VMware).

## Who this book is for

This book is written primarily for network and systems administrators who are monitoring their IT assets with Zenoss Core or who plan to monitor them. In reality, this book will benefit anyone, regardless of job title, who recognizes the importance of proactively monitoring the servers, routers, computers, websites, and devices that connect companies to customers.

## Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Before you make any changes, back up the `snmpd.conf` file".


A block of code is set as follows:


```
syslocation Unknown (edit /etc/snmp/snmpd.local.conf)
syscontact Root <root@localhost> (configure
    /etc/snmp/snmpd.local.conf)
```

Any command-line input or output is written as follows:

```
python setup.py build
python setup.py install
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Select **Simple Network Management Protocol** and **WMI**".

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on [www.packtpub.com](http://www.packtpub.com) or e-mail [suggest@packtpub.com](mailto:suggest@packtpub.com).

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on [www.packtpub.com/authors](http://www.packtpub.com/authors).

## Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

## Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

## Questions

You can contact us at [questions@packtpub.com](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.





# 1

## Network and System Monitoring with Zenoss Core

Whether it's internal or public-facing technology, businesses of all sizes depend on the availability of their IT assets, which may include servers, routers, networks, switches, and websites. If you're picking up this book, then you already know the value of monitoring and more than likely have an installation of Zenoss Core running.

Zenoss Core is an open source network and system monitoring platform that is sponsored by Zenoss, Inc. Zenoss, Inc. develops two versions of Zenoss: Core and Enterprise. Core belongs to the community and is supported by the community.

Enterprise adds some value-added features on top of the Core version, such as an extended report library, synthetic web transactions, certified monitors (ZenPacks), and a global dashboard for multiple Zenoss installations. The additional features allow Zenoss Inc., to sell the enterprise version as a commercial software product with support. As open source consumers, we're familiar with this business model. Our focus in the book is on Zenoss Core, but the concepts will also apply to Zenoss Enterprise.

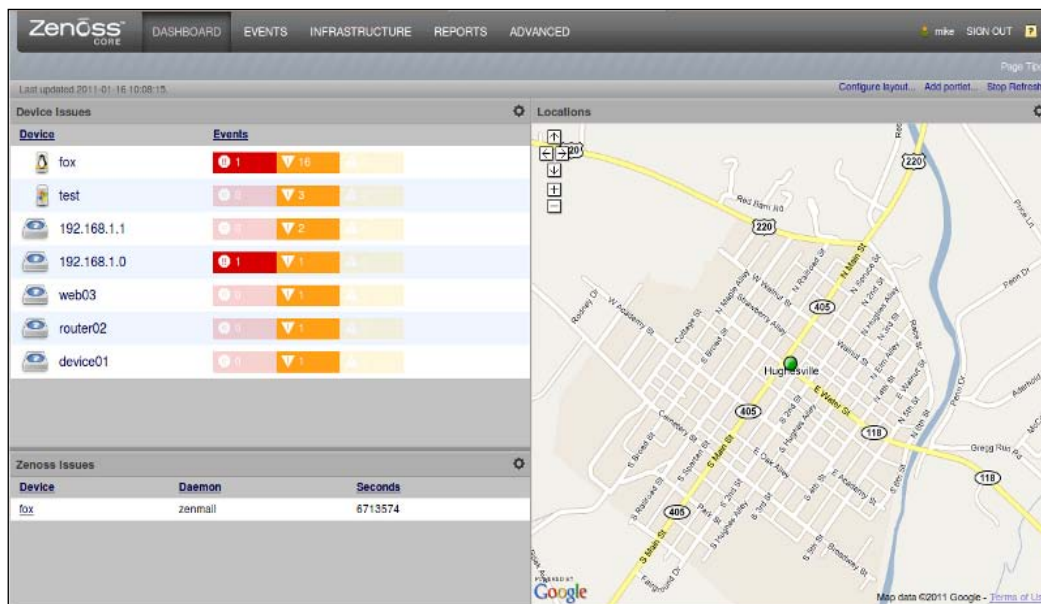
Zenoss Core is a monitoring solution that can be as complex as you need it to be. And while just about anyone can install it, turn it on, and monitor "something," Zenoss Core is packed with features in a complicated interface. The interface has been drastically improved over version 2, but it's not the type of software you can intuitively use—in other words, a bit of guidance is in order.

The role of this book is to serve as your Zenoss Core tour guide and save you hours, days, maybe weeks of time. It's designed to quickly acquaint you with the core features so you can customize Zenoss Core to your needs. It's loaded with screenshots and provides a handy reference guide. Zenoss Core provides a monitoring solution that incorporates the following:

- Device management
- Availability monitoring
- Performance graphs
- Event management
- User and alert management
- Plugin architecture
- Monitoring reports

To monitor your IT assets (servers, routers, switches, websites, and anything else attached to your network), you install Zenoss Core to a server. Even though Zenoss Core is intended to be installed on a Linux server, virtual appliances are available that allow Macintosh and Windows users to install a working version of Zenoss Core by using VMware.

After installation, you can manage your Zenoss Core installation and your monitoring setup from a web-based interface. The following screenshot shows a dashboard view:



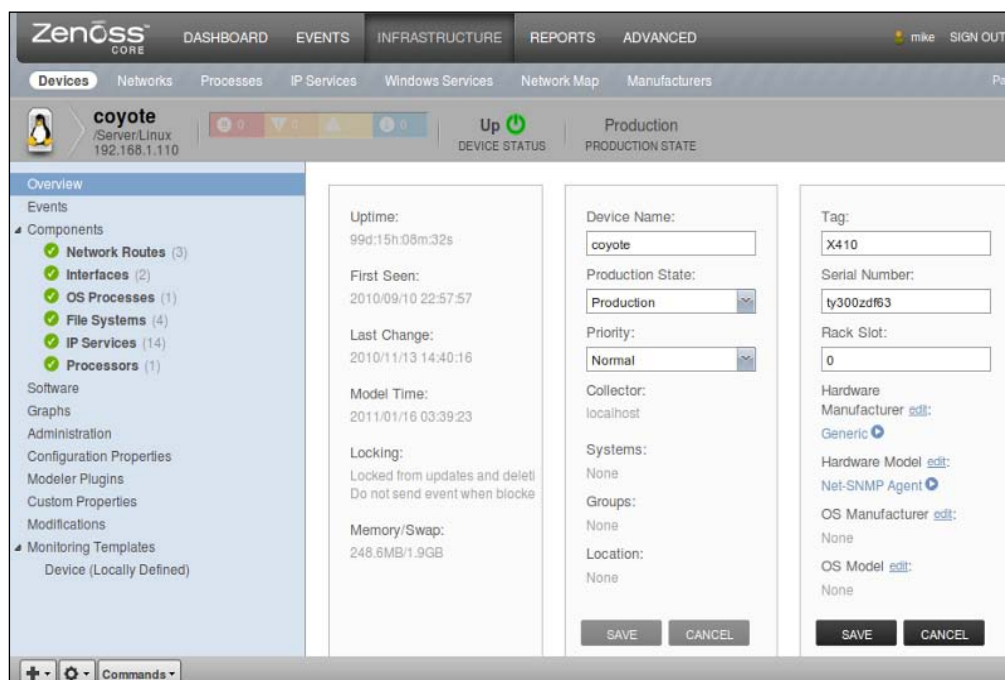
The web portal is the face of the Zenoss Core system and is the place where we spend most of our time. It provides a single access point to the monitoring system and requires no operating-system-specific knowledge to use. The web interface features drag-and-drop dashboard portlets that display a customized view of the network's health at any given time.

## Device management

At the heart of the device management capabilities, Zenoss Core uses a configuration management database (CMDB), which stores a model of the IT environment and its change history. Zenoss Core supports adding IT assets (I'll switch out of "executive-speak" and just refer to the "IT assets" as devices from this point on) to the CMDB one at a time or by auto-discovering active devices by walking the routing tables. Devices are then modeled via Simple Network Management Protocol (SNMP), SSH (or Telnet), or port scans.

Zenoss Core allows us to organize devices by user-defined locations, groups, and systems. One of Zenoss Core's most powerful organizational concepts is classes, which allow us to define monitoring characteristics based on a hierarchical classification of devices, which allows a device to inherit the monitoring properties of its parent class.

The following screenshot provides a look at a device status page:



## **Availability and performance monitors**

By using ICMP and SNMP monitoring, Zenoss Core reports on the availability of the following:

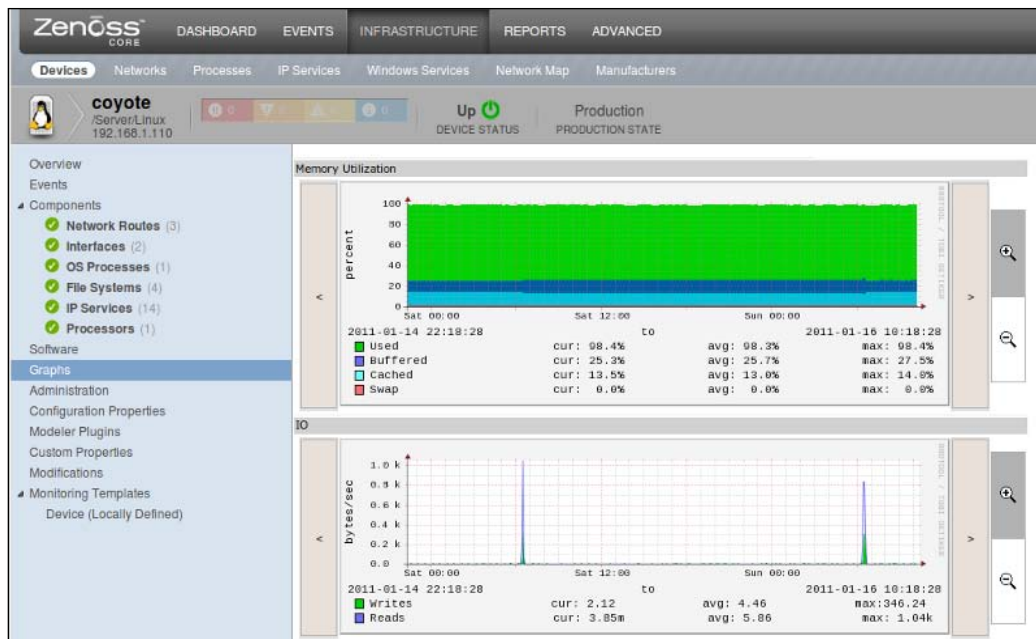
- Network devices
- TCP/IP services and ports
- URL availability
- Windows services and processes
- Linux/UNIX processes

Zenoss Core is Level-3 network topology aware, which reduces the amount of alert chatter by creating an event about the problem device only and not about the devices that depend on it.

Performance monitors collect time series data and provide us with a graphical analysis of the following components:

- File system statistics
- CPU and memory usage
- JMX monitoring for J2EE servers (available via a ZenPack)
- Nagios and Cacti plugin support

The following screenshot shows a graph based on Zenoss Core's monitoring activity:



Using the built-in event management system, we can configure Zenoss Core to generate an event if a monitored device crosses a defined threshold.

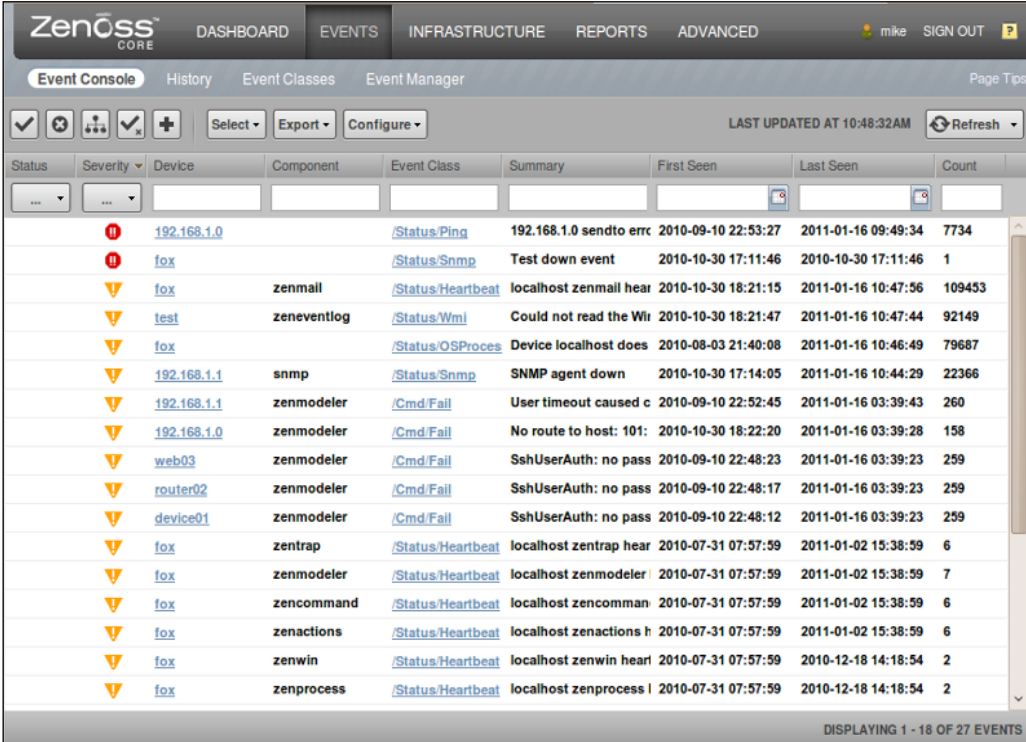
## Event management

Zenoss Core monitors a variety of sources for signs of trouble, including syslogs, availability and performance monitors, SNMP traps, Windows event logs, and custom scripts. Core features of the event management system include:

- Custom events
- Automatic event prioritization
- Event deduplication
- Up/down event correlation



The following screenshot shows the **Event Console**:



Status	Severity	Device	Component	Event Class	Summary	First Seen	Last Seen	Count
U		192.168.1.0		/Status/Ping	192.168.1.0 sendto err	2010-09-10 22:53:27	2011-01-16 09:49:34	7734
U		fox		/Status/Snmp	Test down event	2010-10-30 17:11:46	2010-10-30 17:11:46	1
V		fox	zenmail	/Status/Heartbeat	localhost zenmail hear	2010-10-30 18:21:15	2011-01-16 10:47:56	109453
V		test	zeneventlog	/Status/Wmi	Could not read the Wmi	2010-10-30 18:21:47	2011-01-16 10:47:44	92149
V		fox		/Status/OSProcess	Device localhost does	2010-08-03 21:40:08	2011-01-16 10:46:49	79687
V		192.168.1.1	snmp	/Status/Snmp	SNMP agent down	2010-10-30 17:14:05	2011-01-16 10:44:29	22366
V		192.168.1.1	zenmodeler	/Cmd/Fail	User timeout caused c	2010-09-10 22:52:45	2011-01-16 03:39:43	260
V		192.168.1.0	zenmodeler	/Cmd/Fail	No route to host: 101:	2010-10-30 18:22:20	2011-01-16 03:39:28	158
V		web03	zenmodeler	/Cmd/Fail	SshUserAuth: no pass	2010-09-10 22:48:23	2011-01-16 03:39:23	259
V		router02	zenmodeler	/Cmd/Fail	SshUserAuth: no pass	2010-09-10 22:48:17	2011-01-16 03:39:23	259
V		device01	zenmodeler	/Cmd/Fail	SshUserAuth: no pass	2010-09-10 22:48:12	2011-01-16 03:39:23	259
V		fox	zentrap	/Status/Heartbeat	localhost zentrap hear	2010-07-31 07:57:59	2011-01-02 15:38:59	6
V		fox	zenmodeler	/Status/Heartbeat	localhost zenmodeler	2010-07-31 07:57:59	2011-01-02 15:38:59	7
V		fox	zencommand	/Status/Heartbeat	localhost zencomman	2010-07-31 07:57:59	2011-01-02 15:38:59	6
V		fox	zenactions	/Status/Heartbeat	localhost zenactions h	2010-07-31 07:57:59	2011-01-02 15:38:59	6
V		fox	zenwin	/Status/Heartbeat	localhost zenwin heart	2010-07-31 07:57:59	2010-12-18 14:18:54	2
V		fox	zenprocess	/Status/Heartbeat	localhost zenprocess	2010-07-31 07:57:59	2010-12-18 14:18:54	2

The event system mitigates duplicate events and auto-clears events when the status of the event changes from down to up. Zenoss Core can also collect events from custom scripts and external applications.

In response to events, Zenoss Core can send e-mail or pager alerts, run a script, or do nothing. We configure how Zenoss Core responds to an event by defining alerting rules. Alerting rules are defined on a per user or user-group basis.

## Plugin architecture

Zenoss Core provides several ways for us to extend the base functionality:

- ZenPacks: Zenoss Core's add-on modules
- Nagios plugins
- Cacti plugins

We install and configure a Nagios plugin in *Chapter 5, Custom Monitoring Templates*.

The information presented there will be valuable for those of you who want to develop your own plugins.

The ZenPack architecture allows us to package plugins and configurations for distribution to other users and the community at-large. *Chapter 9, Extending Zenoss Core with ZenPacks* walks you through the steps of installing a community ZenPack, and how to create your own ZenPack as well.

## System reports

Zenoss Core packages a set of standard reports that allow us to view what is happening right now, as well as what has happened in the past. The reports integrate with the device management, performance monitors, events, and user functionalities.

The following screenshot shows the **All Monitored Components** report:

The screenshot displays the Zenoss Core web interface. The top navigation bar includes 'DASHBOARD', 'EVENTS', 'INFRASTRUCTURE', 'REPORTS', and 'ADVANCED'. The 'REPORTS' tab is active. The sidebar on the left lists 'REPORT CLASSES (28)' with categories like 'Device Reports (9)', 'Custom Device Reports (6)', 'Graph Reports (1)', 'Multi-Graph Reports (1)', and 'Performance Reports (7)'. The 'All Monitored Components' report is selected. The main content area shows a table of monitored components with columns: Device, Component, Type, Description, and Status. The table lists various components for devices 'covote', 'test', and 'fox'. Most components are 'Up', but the 'lo' interface for 'covote' is 'None'. The bottom of the table shows '1 of 11' items and a 'Page Size' of '40'.

Device	Component	Type	Description	Status
covote	/	FileSystem	/	Up
covote	/boot	FileSystem	/boot	Up
covote	/home	FileSystem	/home	Up
covote	/proc/bus/usb	FileSystem	/proc/bus/usb	Up
covote	0	CPU	0	Up
test	Eventlog	WinService	'Event Log' StartMode: StartName:	Up
fox	apache2	OSProcess	apache2	Up
covote	eth0	IpInterface	eth0	Up
covote	lo	IpInterface	lo	None
covote	http	IpService	tcp-80 ips:0.0.0.0	Up
covote	/usr/sbin/apache2 -k start	OSProcess	/usr/sbin/apache2 -k start	Up

Notice the number of additional reports listed in the sidebar.

## Custom device reports

The canned reports are nice, but sometimes we need to access and analyze data that the included reports do not cover. Zenoss Core enables users to write custom device reports from the web interface, as seen in the following screenshot:

The screenshot displays the Zenoss Core web interface for configuring a custom device report. The top navigation bar includes links for DASHBOARD, EVENTS, INFRASTRUCTURE, REPORTS, and ADVANCED, along with a user profile for 'mike' and a 'SIGN OUT' button. The left sidebar contains 'View Report' and 'Edit Report' options. The main content area is titled 'Reports > Custom Device Reports > Comments' and shows an 'RRD Graph state at time: 2011/01/16 11:02:45'. Below this, there are input fields for 'Name' (Comments), 'Title', 'Path' (/HTTP), 'Query' (here.comments != ''), 'Sort Column' (Name), 'Sort Sense' (asc), 'Columns' (getId, comments), and 'Column Names' (Name, Comment).

We step through the creation of custom device reports in *Chapter 11, Writing Custom Device Reports*.

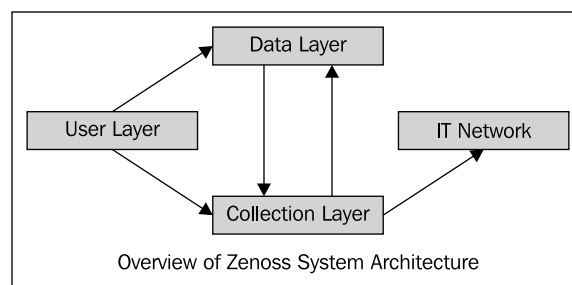
## System architecture

Reviewing the system architecture now provides us with an understanding that can help troubleshoot problems that may arise later. However, reading it to start using Zenoss Core is definitely not required. So feel free to come back later.

Zenoss Core builds upon several open-source software projects to create a robust network and systems management solution. The most notable open-source software components that integrate with Zenoss Core include Zope, Python, MySQL, RRDtool, and Twisted.

When we talk about the system architecture, it helps to conceptually segregate Zenoss Core into three layers:

- User
- Data
- Collection



## User layer

Zenoss Core is flexible enough to work from a command line, but most of our work will take place via a web interface, which is based on the Zope application server framework.

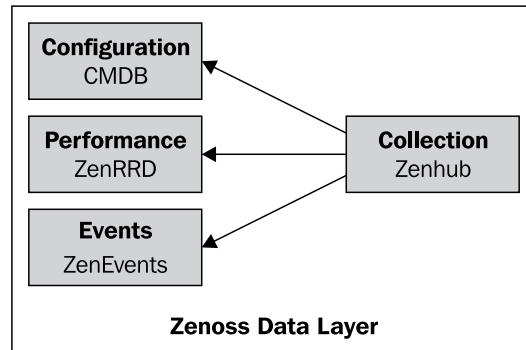
Zope is a popular, extensible application server written in Python. It features a built-in web server, transactional object database, and HTML templates. Python is the basis for Zope; it's also the basis for Zenoss Core.

Through the web interface, we provide input with both the data and collection layers to accomplish tasks related to the following areas:

- Navigation and organization
- Device management
- Availability and performance monitors
- System reports
- Event management
- Settings and administration

## Data layer

As we might expect, databases are the heart of the data layer, and Zenoss Core stores data in three types of databases. The Collection layer funnels device information to ZenHub, which in turn stores the data in the appropriate place, as seen in the following illustration.



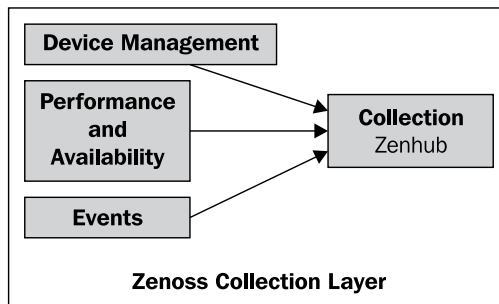
Events are stored in a MySQL database. Zenoss Core generates Events when an established threshold is crossed, such as a server outage or high memory usage. Events trigger actions, such as e-mail or pager alerts.

Time series performance data gets stored in a **Round Robin Database (RRD)**. A RRD differs from a linear database, such as MySQL, in that it's circular – which means the size does not increase over time. Data is stored in a first in, first out basis, which implies that monitoring data is consolidated and eventually lost over time. RRDtool provides Zenoss Core with the ability to log and graph performance data.

The third database deployed by Zenoss is a **Configuration Management Database (CMDB)**. The CMDB is an **Information Technology Infrastructure Library (ITIL)** standard for managing the configuration, relationship, and change history of the IT environment, which creates a detailed model of the network. Zenoss uses a **Zope Object Database (ZODB)** to house the CMDB.

## Collection layer

The collection layer includes several daemons that gather information about devices, performance, and Events. They feed information to ZenHub to distribute to the appropriate database. The Zenoss Core daemons are easy to identify – they all start with the prefix "zen".



As the following screenshot illustrates, the Daemons page provides us with the complete view of the Zenoss Daemons that includes the process ID (**PID**) and up/down **State**. Green is up; red is down. From the interface, we can also view the **Log File**, edit the **Configuration**, and start and **Stop** each daemon.

Zenoss CORE						
DASHBOARD   EVENTS   INFRASTRUCTURE   REPORTS   ADVANCED   SIGN OUT   ?						
Settings   Collectors   Monitoring Templates   MIBs   Page Tips						
Settings	Zenoss Daemons					
Commands	Zenoss Daemon	PID	Log File	Configuration	State	Actions
Users	zeoctl	3141	<a href="#">view log</a>	<a href="#">view config</a> <a href="#">edit config</a>	●	<a href="#">Restart</a> <a href="#">Stop</a>
ZenPacks	zopectl	3178	<a href="#">view log</a>	<a href="#">view config</a> <a href="#">edit config</a>	●	<a href="#">Restart</a> <a href="#">Stop</a>
Jobs	zenhub	3230	<a href="#">view log</a>	<a href="#">view config</a> <a href="#">edit config</a>	●	<a href="#">Restart</a> <a href="#">Stop</a>
Portlets	zenjobs	3296	<a href="#">view log</a>	<a href="#">view config</a> <a href="#">edit config</a>	●	<a href="#">Restart</a> <a href="#">Stop</a>
Daemons	zenping	3420	<a href="#">view log</a>	<a href="#">view config</a> <a href="#">edit config</a>	●	<a href="#">Restart</a> <a href="#">Stop</a>
Versions	zensyslog	3509	<a href="#">view log</a>	<a href="#">view config</a> <a href="#">edit config</a>	●	<a href="#">Restart</a> <a href="#">Stop</a>
Backups						

If we browse the file system, we will find each daemon in `$ZENHOME/bin`. `$ZENHOME` is an environment variable, which allows us to talk about the Zenoss installation directory without knowing exactly where it is. For example, I may install to `/usr/local/zenoss/zenoss` while you install to `/home/zenoss`.

Twisted is an integral network communication protocol for the daemons. The Twisted Core README file describes Twisted as an "event-based framework for Internet applications"

## Device management daemons

Finding the devices on our networks is a prerequisite to managing them, and Zenoss Core not only finds the devices, it models them. Device modeling builds a detailed overview of the network by recording the following types of information: system dependencies, available services, and change history.

The following table describes the daemons responsible for discovering and modeling devices:

Device daemon	Description
zenmodeler	Queries the devices via SSH/Telnet, SNMP, and port scans when we model the device. Each time zenmodeler runs on a device, it compares its findings with existing configuration and updates it as necessary.
zendisc	Runs each time a request is made to discover a network or device.

Zenoss uses SNMP as a primary collection protocol. However, it can also collect information via ICMP pings, port scans, and plugins.

## Performance and availability daemons

The Zenoss Core performance and availability daemons help us determine if the devices on our network are available and performing within our established guidelines. If our monitored systems perform in an unexpected way, Zenoss Core generates an event.

The following daemons play an important role in collecting performance and availability data:

Performance daemon	Description
zenperfsnmp	Stores the collected performance data in RRD files so that RRDtool can graph device performance over hourly, daily, weekly, monthly, or yearly durations.
zencommand	Provides a way to run custom scripts and third party plugins including Nagios and Cacti plugins from within Zenoss.
zenprocess	Monitors the processes on Linux, Unix, and Windows systems.
zenping	Pings a device and reports an up or down status to determine if a device is active or not.
zenstatus	Tests the TCP ports and reports an up or down service.

## Event daemons

When a device goes down or a service crosses a predetermined threshold, such as available disk space, Zenoss Core generates an event. Events can generate a notification alert or run a custom command (to automatically take remedial action to fix the event, for example).

Not only can Zenoss Core generate its own events, but it can collect events from external sources (for example, custom system administration scripts) and convert the information to a "Zenoss-style" event.

The following table outlines the Zenoss Core event daemons:

Event daemon	Description
zensyslog	Creates events from syslog messages.
zeneventlog	Creates events from Windows event logs.
zentrap	Creates events from SNMP traps. When a problem occurs on a monitored device, it generates an SNMP trap to alert Zenoss of the problem.



## Summary

In this chapter we provided a brief overview of Zenoss Core's monitoring capabilities and the underlying technology that makes it all work. It's our blueprint for what we'll discuss as we devote the rest of the book to configuring our monitoring environment. As we work through the rest of the book, we'll demonstrate the core concepts that will help you adapt Zenoss Core to your specific environment.

Feel free to treat each chapter as a stand-alone topic and skip around as needed. In *Chapter 2, Discovering Devices* we jump right in and discover devices.

# 2

## Discovering Devices

I assume that many of you already have a working Zenoss Core installation with devices in your inventory; however, adding devices is the easy part. Before we jump to monitoring, we need to think about how we will monitor. For example, we can use **Simple Network Management Protocol (SNMP)**, **Windows Management Instrumentation (WMI)**, or command-line plugins that run over SSH/Telnet. Each of your monitored devices may have different requirements. We'll review each of the monitoring protocols to help you decide how to monitor your devices with Zenoss Core.

Because our Zenoss Core server needs to communicate with our monitored devices, we will need to configure those devices to allow connections on various TCP and UDP ports.

The order in which you perform the items in this chapter is not important. If you are starting a new installation, then it makes sense to prepare your devices to be monitored before you add them to your inventory. If you've already added devices to your inventory, then you will still need to do these tasks in order to monitor them.

Here's what we'll do in this chapter:

- Prepare devices for monitoring via SNMP, WMI, or SSH
- Open ports for monitoring
- Run the Zenoss Core setup wizard
- Import devices with zenbatchload
- Discover devices with zendisc

Before we jump into SNMP, let's talk briefly about Zenoss Core installation options.

## Zenoss Core installation

Zenoss, Inc., makes several installation packages of Zenoss Core available to meet the needs of its community. For a list of current installation packages, visit [www.zenoss.com/download](http://www.zenoss.com/download). You'll find native packages, stack installers, and source files for several operating systems: Red Hat Enterprise Linux, Centos, Ubuntu, Fedora Core, OS X, OpenSuse, and Debian. The stack installers and native packages include all Zenoss Core dependencies.

If you need a no fuss installation to use as a sandbox or to follow along with the book, then I recommend loading one of the Zenoss Core VMware appliances for Windows, Linux, or OS X. To run the appliance, download the VMware Player or VMware Server for free from [www.vmware.com](http://www.vmware.com).

## Preparing devices for monitoring

One of the first questions we need to answer is, "How will I collect information from my devices and services?" Zenoss Core supports the following ways to model (collect data from) the device:

- Simple Network Management Protocol (SNMP)
- Windows Management Instrumentation (WMI)
- SSH/Telnet (for example, Zenoss Plugins)
- Port scan

SNMP gives us the most flexibility and the best device support. WMI lets us access information about services running on Windows servers and can co-exist with SNMP. If we want to monitor Windows event logs, then WMI is a must.

Even if your device supports SNMP, there may be times when you are unable to use it. For example, you're monitoring a remote device that's outside the local network. Zenoss, Inc., distributes a suite of command-line monitors that can be installed on a device via the Zenoss Plugins, and Zenoss Core runs the plugins over an SSH or Telnet session to collect the performance data.

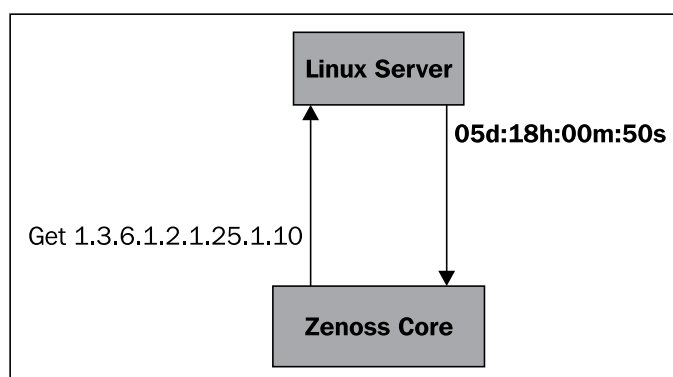
In the unfortunate event that your device doesn't support SNMP, WMI, SSH, or Telnet you will only be able to monitor availability via ping. You may be able to retrieve a list of services to monitor on the device via Zenoss Core's port scan plugin.

## SNMP

Zenoss Core uses SNMP to model the specific characteristics about a device, such as file system utilization, memory usage, or throughput on a network interface. The network devices report data to Zenoss Core via an SNMP agent that runs on each device. The agent on the device communicates with Zenoss Core—the SNMP network management station.

When Zenoss Core requests information from a device using SNMP, it asks the device to send information for a specific characteristic by including an **Object Identifier (OID)** in the request. For example, the OID 1.3.6.1.2.1.25.1.1.0 contains the system uptime value.

The following screenshot shows a sample transaction between Zenoss Core and a Linux server. Zenoss Core requests the value for an OID and the server responds with the value:



SNMP associates OIDs with a human-friendly variable name via **Management Information Bases (MIBs)**. For example, 1.3.6.1.2.25.1.1.0 is also known as sysUpTime.

OIDs are collected in modules, called **Management Information Bases (MIBs)**. Zenoss Core supplies many of the MIBs that we need to monitor a broad range of devices. However, we can download and install manufacturer-specific MIBs to extend our monitoring functionality. We'll cover adding MIBs in *Chapter 8, Settings and Administration* including how to know when you need to download a new MIB.

SNMP agents don't always wait to be asked for information. They have the ability to send traps to the network management station as a notification of a system problem. We'll investigate SNMP traps in *Chapter 7, Collecting Events*.

We'll encounter more SNMP as we move through the remainder of the book, but if you'd like to read more SNMP theory, consider the following resources:

- SNMP RFC: <http://www.ietf.org/rfc/rfc1157.txt>.
- Net-SNMP: <http://www.net-snmp.org>.

Let's take a quick look at the different SNMP versions before we configure our monitored devices.

## SNMP versions

SNMP comes in three versions: v1, v2c, and v3. SNMP v2c provides support for 64-bit counters, and it can embed multiple requests in one packet, whereas v1 needs to send each SNMP request in a separate packet.

SNMP v3 improves upon the "weak" security model of the v1 and v2c by implementing user authentication and roles. In the earlier versions of SNMP, authentication is done with community strings. It's the equivalent of protecting your childhood fort by asking, "What's the secret word," to everyone who knocks on the door.

Let's manufacture an example and configure a Linux server to respond to any SNMP request that contains the community string of "wildchicken". If the Zenoss Core server queries with a SNMP community string of "public", the Linux server won't respond and Zenoss Core won't collect data from the server.

Throughout the course of the book, we won't dwell on which version of SNMP you should use. Use the version that meets your requirements and your devices' capabilities. As we move through the device management chapter, we'll see how to adjust the SNMP collection options in Zenoss Core. Experiment with different configurations, as necessary.

Now, let's configure SNMP on a Linux host.

## Configuring SNMP on Linux

If we plan to collect device information from the network using SNMP, we need to install an SNMP agent on all devices that we plan to monitor with SNMP. The SNMP agent on the monitored devices listens for incoming SNMP requests from Zenoss Core and responds to the request appropriately. Although there are other SNMP options, Net-SNMP is widely used and recommended by the Zenoss team.

Net-SNMP also includes a set of utilities, such as *snmpwalk*, *snmpget*, and *snmptrap* that can help us manipulate SNMP values and troubleshoot problems. It won't hurt you to install both packages.

The Net-SNMP package names vary from one distribution to the next, so be sure to check with your distribution if you are unsure of which file you need. Here are a few examples:

- Red Hat users can install the agent and the utilities with the command:  
`yum -y install net-snmp net-snmp-utils`
- Ubuntu users can install the agent and utilities with the command:  
`apt-get install snmpd snmp`

Next, we must configure the SNMP agent to process requests and to publish the entire MIB tree. As root, we need to edit the `/etc/snmp/snmpd.conf` file.

1. Before you make any changes, back up the `snmpd.conf` file:  
`cp /etc/snmp/snmpd.conf /etc/snmp/snmpd.conf.bak`
2. In the section that begins "First, map the community name into a security name," add the following line and replace `public` with the value of your community string:  
`com2sec notConfigUser default public`
3. In the section that begins "Second, map the security names into group names," add:  
`group notConfigGroup v1 notConfigUser`  
`group notConfigGroup v2c notConfigUser`
4. In the section that begins "Third, create a view for us to let the groups have rights," add:  
`view systemview included .1`

5. In the section that begins, "Finally, grant the 2 groups access to the 1 view with different write permissions," add the following line:

```
access notConfigGroup "" any noauth exact systemview none none
```

6. Add the following lines to the System Contact Information section using your contact details:

```
syslocation Unknown (edit /etc/snmp/snmpd.local.conf)
syscontact Root <root@localhost> (configure /etc/snmp/snmpd.local.conf)
```

7. Add the following lines to the Further Information section to send SNMP traps to the Zenoss Core server. Replace the IP address with your Zenoss server IP and use the community string for your network:

```
trapsink 192.168.1.125 public 162
```



#### Downloading the example code

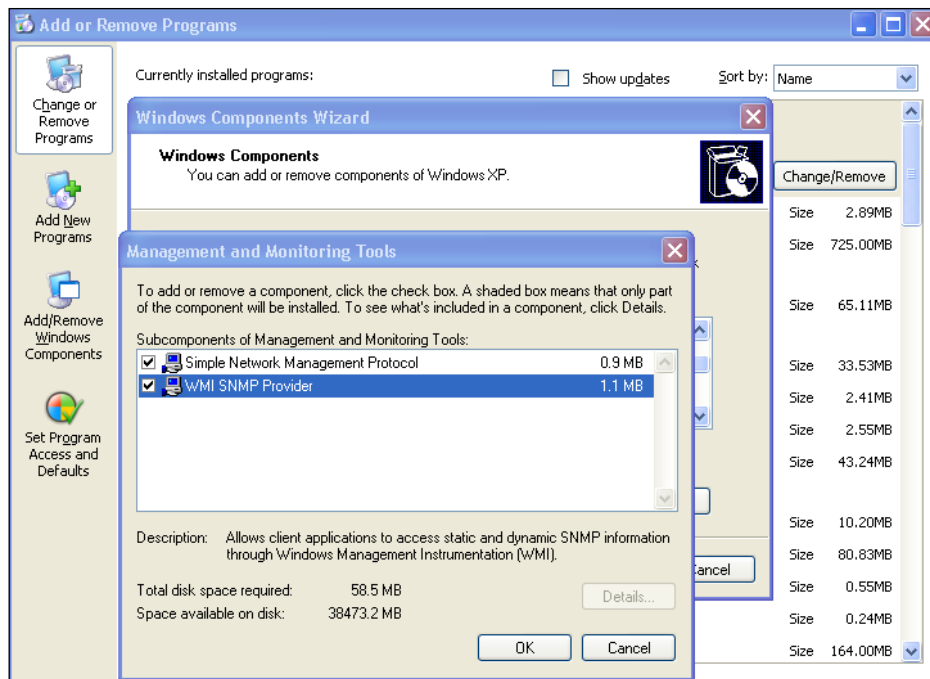
You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

## Configuring SNMP and WMI on Windows

Your Windows server most likely has WMI and SNMP already installed. However, in the event you need to install one or the other, you can do so through Add or Remove Programs.

WMI and SNMP are enabled from the Windows Management and Monitoring Tools packages. To install WMI and SNMP (refer to the following screenshot), follow these steps:

1. Open the Windows **Control Panel**.
2. Select **Add/Remove Windows Components**.
3. Click on **Management and Monitoring Tools** and select **Details**.
4. Select **Simple Network Management Protocol** and **WMI**.
5. Save the changes to install the **Windows Components**.

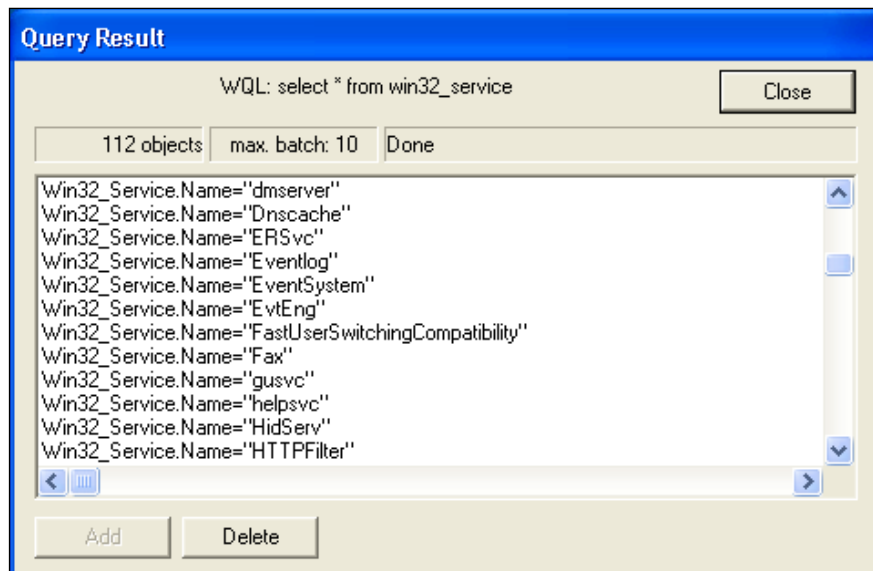


After WMI installs, we should confirm that WMI is properly configured:

1. From the Windows **Start** menu, select **Run**.
2. Enter the command `wbemtest`.
3. Select the **Connect** button.
4. Change the **Namespace** field to `\\HOST\root\cimv2`.
5. Enter an administrator username and password.
6. Click the **Query** button.



7. In the search box, type `select * from win32_service` to see a list of services as shown in the next screen capture.



The Windows SNMP agent does not return information about the server's CPU, memory, or file system. For these stats, the Zenoss team recommends the free SNMP (not open source) agent from SNMP Informant located at <http://www.snmp-informant.com>. No configuration is necessary for SNMP Informant.

## Zenoss Plugins

If the device we plan to monitor does not support SNMP, or if we need to monitor a device behind a firewall on a network far, far away, SSH provides an alternative to SNMP. Zenoss Core also supports Telnet, but we'll work with SSH in our examples.

In order to collect performance data from our device over SSH, the monitored device needs the Zenoss Plugins installed. And the monitored device needs to have an SSH server installed so that it can accept incoming requests from the Zenoss Core server. OpenSSH from [openssh.com](http://openssh.com) offers a good solution.



The Zenoss Plugins platform support is limited to Linux, Darwin, and FreeBSD.

The level of data provided by plugins varies between platforms. For this reason we may not achieve the same level of detail as we do with SNMP; however, SSH modeling provides more detail than a port scan.

## Installing Zenoss Plugins

As a prerequisite to installing the Zenoss Plugins, the monitored system needs a Python environment installed. This can be installed using your distribution's package manager. If you have `setuptools` installed, you can install the Zenoss Plugins package from the Cheese Shop (<http://pypi.python.org/pypi/>) with the following command as root:

```
easy_install Zenoss-Plug-ins
```

We can also build the plugins package from source:

1. Download the Zenoss Plugins package from <http://www.zenoss.com/download/>.
2. Extract the file.
3. From the extracted plugin source directory, run the following commands as root:

```
python setup.py build
python setup.py install
```

The `setuptools` procedure installs `zenplugin.py` to `/usr/bin`, which is important because we will need to configure the configuration properties (`zCommandPath`) of the device or device class to look for the plugins at the correct location. We'll review the configuration properties in *Chapter 3, Device Setup and Administration*.

To ensure that the plugin file is working correctly, run the following command on the monitored device:

```
zenplugin.py -list-plugins
```

This command outputs the detected platform and the supported plugins as shown in the following screenshot:

```
mike@coyote:~$ zenplugin.py --list-plugin
platform 'linux2' supports the following plugins:
  process
  mem
  disk
  cpu
  io
mike@coyote:~$
```

## Port scan

Sometimes, the only option we have to model our devices is a port scan. A port scan tries to guess which services are running on a device by connecting to various ports. Port scans provide the least detailed model and may raise security alerts on your network. Consult the security administrators before port scanning devices on the network.

We don't need to make any special provisions on our monitored device if we plan to check the availability of services with port scanning.

## Opening monitoring-specific ports

Both the Zenoss Core server and the monitored devices have port requirements. The Zenoss Core server needs to allow access to the following ports:

- TCP port 8080 for HTTP access
- TCP port 514 for syslog access, if Zenoss Core is acting as a syslog server
- TCP port 22 for remote SSH access
- UDP port 162 to process SNMP traps

To facilitate monitoring, the systems on the network need to allow access to the following ports:

- UDP 161 for SNMP requests
- TCP port 22 for remote collector plugins via SSH
- TCP port 23 for remote collector plugins via Telnet

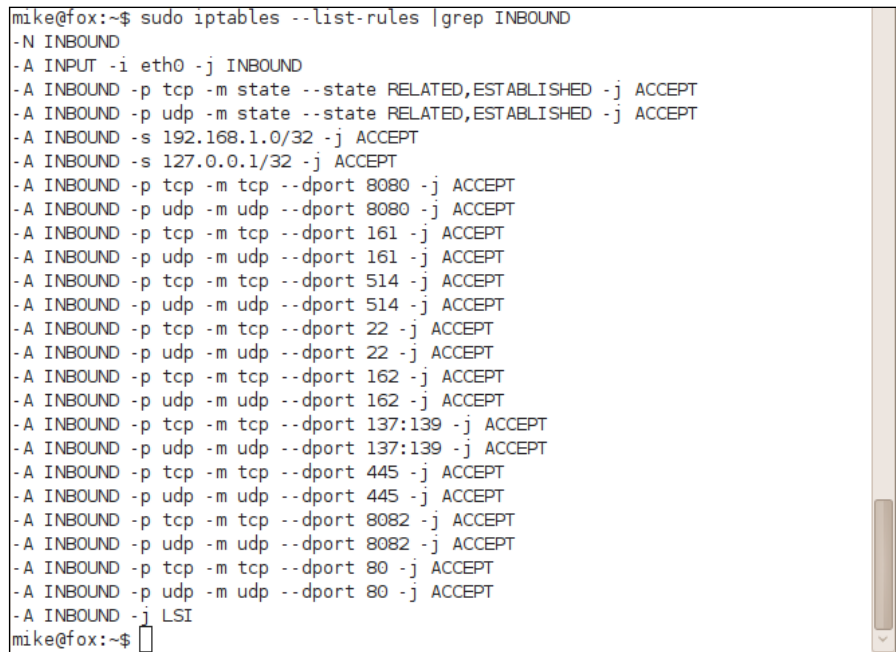
This is a common list of ports, but network and monitoring needs are unique from one site to the next. For example, if you do not plan to monitor any devices with the Zenoss Plugins over Telnet, then don't open port 23 on your monitored devices.

## Configuring Linux firewalls

Iptables is a popular tool for managing firewall access on Linux systems. Assuming, iptables is installed and configured, use the following command to view the current filtering rules:

```
iptables --list-rules |grep INBOUND
```

The following screenshot shows a sample list of iptables rules:

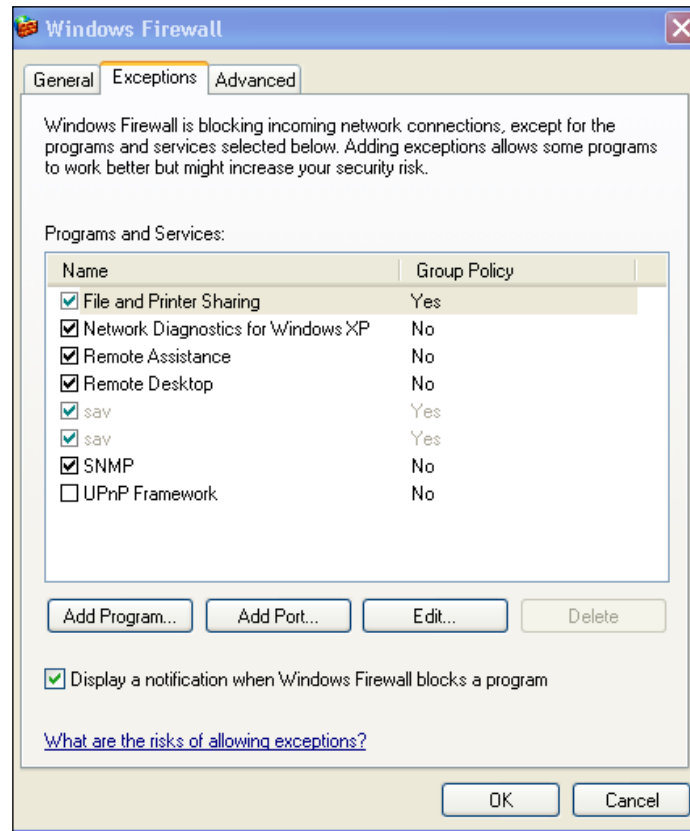


```
mike@fox:~$ sudo iptables --list-rules |grep INBOUND
-N INBOUND
-A INPUT -i eth0 -j INBOUND
-A INBOUND -p tcp -m state --state RELATED,ESTABLISHED -j ACCEPT
-A INBOUND -p udp -m state --state RELATED,ESTABLISHED -j ACCEPT
-A INBOUND -s 192.168.1.0/32 -j ACCEPT
-A INBOUND -s 127.0.0.1/32 -j ACCEPT
-A INBOUND -p tcp -m tcp --dport 8080 -j ACCEPT
-A INBOUND -p udp -m udp --dport 8080 -j ACCEPT
-A INBOUND -p tcp -m tcp --dport 161 -j ACCEPT
-A INBOUND -p udp -m udp --dport 161 -j ACCEPT
-A INBOUND -p tcp -m tcp --dport 514 -j ACCEPT
-A INBOUND -p udp -m udp --dport 514 -j ACCEPT
-A INBOUND -p tcp -m tcp --dport 22 -j ACCEPT
-A INBOUND -p udp -m udp --dport 22 -j ACCEPT
-A INBOUND -p tcp -m tcp --dport 162 -j ACCEPT
-A INBOUND -p udp -m udp --dport 162 -j ACCEPT
-A INBOUND -p tcp -m tcp --dport 137:139 -j ACCEPT
-A INBOUND -p udp -m udp --dport 137:139 -j ACCEPT
-A INBOUND -p tcp -m tcp --dport 445 -j ACCEPT
-A INBOUND -p udp -m udp --dport 445 -j ACCEPT
-A INBOUND -p tcp -m tcp --dport 8082 -j ACCEPT
-A INBOUND -p udp -m udp --dport 8082 -j ACCEPT
-A INBOUND -p tcp -m tcp --dport 80 -j ACCEPT
-A INBOUND -p udp -m udp --dport 80 -j ACCEPT
-A INBOUND -j LSI
mike@fox:~$
```

For more help with iptables, try the man page.

## Configuring Windows firewall

Windows has built-in firewall support via the Windows Firewall Control Panel (as shown in the following screenshot). If you are unsure about how to configure port access, consult your system documentation or friendly system administrator.

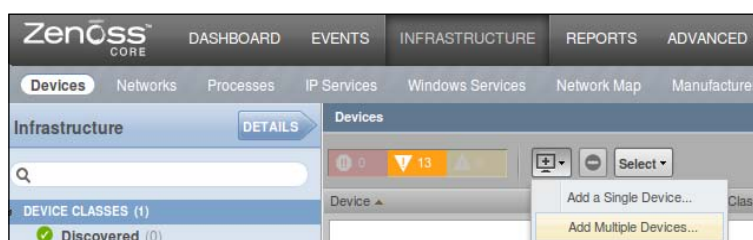


In order to allow Zenoss Core to retrieve data via WMI, you must add a remote administration exception. See this MSDN topic <http://msdn.microsoft.com/en-us/library/aa389286%28VS.85%29.aspx>.

## Zenoss Core setup wizard

New installations automatically run a setup wizard that creates a user account, defines an administrator password, and discovers devices attached to the network. Many of you have probably already breezed through this step, but we'll step through the process for the benefit of those of you who are working through an initial install.

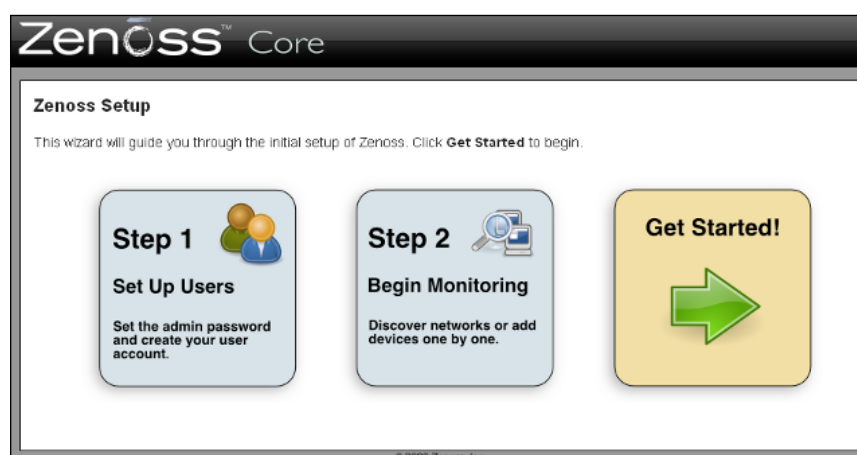
Even if you already ran the initial setup wizard, you can invoke the add device wizard by selecting the **Add Multiple Devices...** link on the **Infrastructure** | **Devices** page. See the following screenshot:



To access your Zenoss Core installation, open a browser and navigate to port 8080 of your Zenoss Core server. For example, go to `http://zenoss.example.com:8080` where `zenoss.example.com` is your web address.

## Step 1: Setting up users

If this is a new installation, you will be greeted with the setup wizard when you visit the Zenoss Core application. Otherwise, you will see the Zenoss Core Dashboard.



Click the big green arrow to advance the wizard to the **Set Up Initial Users** screen.

The user setup screen will ask you to provide two important pieces of information. The first is an administrative password for Zenoss Core's built-in admin user. You also set up a privileged user account by specifying a **User name**, **Password**, and **email** address.



The admin user in Zenoss Core has superuser privileges, similar to root access on a Linux machine.

It's generally considered bad form to work under a system's admin account, and the same philosophy holds with Zenoss Core. Zenoss Core logs each user's activities throughout the system, so you don't want multiple people making changes as the admin user.

The user account we setup should accommodate all our monitoring tasks.

As we'll see in the user management section of *Chapter 8, Settings and Administration* we can control user access to the system via roles.

Click the **Submit** button after you have entered all the information.

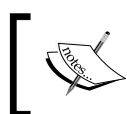
## Step 2: Specify or discover devices to monitor

Now we have some choices to make. Do we manually add devices, one at a time?

Or do we let Zenoss Core discover the devices attached to the network. Each choice will lead to a different set of choices about how the device is discovered and classified by Zenoss Core, and device classification is an important concept in our monitoring activities.

Classes in Zenoss Core establish a default set of monitoring characteristics for a type of device, specifically, the collector plugins. The collector plugins determine what information Zenoss Core collects from each type of device. As we'll find out in *Chapter 3, Device Setup and Administration* when we review device classes in-depth, we can always change the defaults and change the classification.

If we manually add devices, Zenoss Core will assign the device to the class based on the device type we select. For example, if we select Linux Server, the device class will be `/Server/Linux`. In contrast, if we autodiscover devices, Zenoss Core classifies all our devices in a generic `/Discovered` class, and we will manually have to classify the devices.



To enter devices in advanced mode, click the link titled **skip to the dashboard**. Then you can skip to the start of *Chapter 3, Device Setup and Administration*.

## Adding devices

When you add a device manually, you provide Zenoss Core the IP address or the hostname of the device. When you add devices automatically, you can specify an entire subnet or ranges of IP addresses.

Here are some reasons you may opt to manually add devices:

- You have a relatively small number of devices on your network
- Your network has a large number of devices with dynamic IP addresses (for example, Workstations)

Here are some reasons you may opt to autodiscover devices:

- You have a large number of devices on your network
- You need to probe multiple networks and IP ranges

Of course, what you choose to do is a judgment call. Let's look closer at each option.


## Manually find devices

If we choose to manually add devices, we then specify the device type. Available device types are Generic Switch/Router SNMP, Linux Server SNMP, or Window Server SNMP. Notice an SNMP trend?



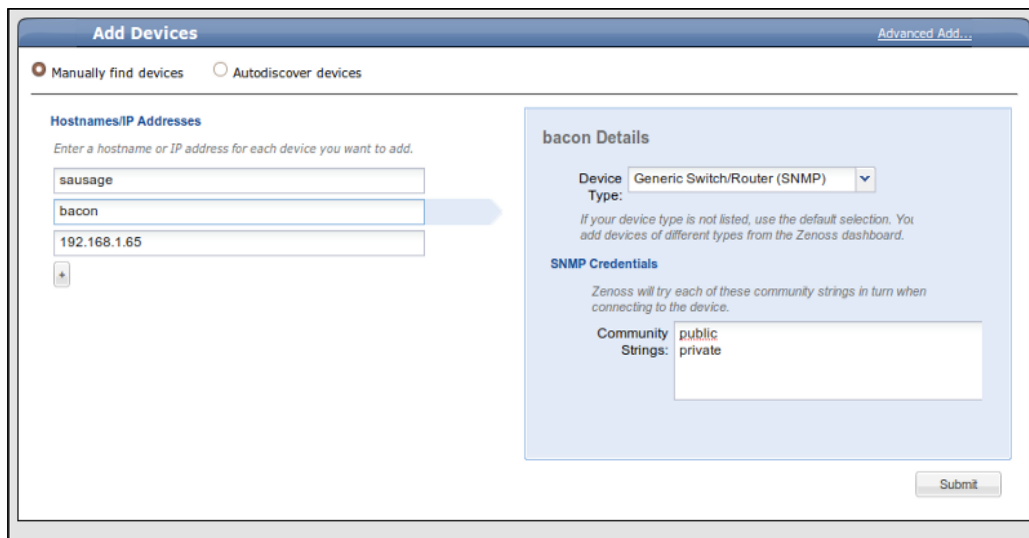
Let's step through the process:

1. Make sure the **Manually find devices** option is selected from the **Add Devices** screen.
2. In the first text field, enter the IP address or the hostname of the first device.

 Make sure the devices you want to add are turned on and connected to the network, else Zenoss Core won't be able to find them. Sounds obvious, right?

3. In the shaded **Details** box, select a device type from the drop-down box.
4. In the **SNMP Credentials** field, enter the community strings required to connect to the device. The defaults are public and private.
5. To add more devices, click the button with the plus (+) sign to display a new hostname/IP address entry field. Repeat for as many devices as necessary.

Before you click the **Submit** button, let's pause to talk about what is happening on this screen:



As you see in the screenshot, I have three devices specified: **sausage**, **bacon**, and **192.168.1.65**. The hostname **bacon** is highlighted and contains an arrow that points to the **bacon Details** section of the screen. As you click on the hostnames, you'll notice the details section changes too. This allows us to specify a unique set of details for each of the devices we're going to add.

This is a big interface improvement over the earlier versions of Zenoss Core. In the next chapter, we'll take a look at the old add device interface, which has some of its own advantages.

When you have all of your devices entered, click the **Submit** button. Zenoss Core will create a job that tries to connect to each device using the details you specified for it. You'll be looking at the Zenoss Core dashboard after you submit the job.

Take a coffee or tea break. When you come back, we'll review the results of our work.

## Autodiscover devices

If we choose to autodiscover devices, we will be prompted to specify SNMP information, SSH login, or Windows administrator credentials. What you enter depends on what types of devices you expect to monitor.

Let's autodiscover some devices. For this example, I'm going to scan my employer, Mojo Active's, network.

1. From the **Add Device** screen, select **autodiscover** devices.
2. In the text field for **Networks/Ranges**, specify a network or an IP range.
3. To add more than one network or IP range, click the button with the plus (+) sign on it.
4. In the **Authentication** section of the screen, specify the credentials that are required to connect to the devices on your network.

5. Click the **Discover** button to tell Zenoss Core to find your devices. You'll be transferred to the Zenoss Core dashboard.

**Step 2: Specify or Discover Devices to Monitor**  
I want Zenoss to:

☐ Manually find devices ☒ Autodiscover devices

**Networks/Ranges**  
*Enter one or more networks (such as 10.0.0.0/24) or IP ranges (such as 10.0.0.1-50).*

192.168.0.0/24

**Authentication**  
*Specify credentials to be used during the discovery process. Zenoss will apply these to each device it discovers.*

**Windows**  
*This user must be a member of the Local Administrators group.*

Username: administrator  
Password: .....

**SSH**

Username:   
Password:

**SNMP**  
*Zenoss will try each of these community strings in turn when connecting to the device.*

Community Strings: public  
private  
wildchicken

Discover

The authentication options correspond to the monitoring methods we talked about earlier in the chapter.

If Zenoss Core discovers a Windows server, it will try to log in with the supplied username and password to retrieve information via WMI. It will also test the server with the supplied SNMP community strings.

If Zenoss Core discovers a Unix based machine, it will query the device with the SNMP community strings. It will also try to log in via SSH to determine if the Zenoss Plugins are installed.

## Our device inventory: A job well done

We've just completed the easy part. Let's take a look at our results. From the main menu in the Zenoss Core interface, select **Infrastructure** to display a list of the devices you just added. Your device list will be unique, but it should resemble the following screenshot:

The screenshot shows the Zenoss Core Infrastructure page. On the left is a sidebar with 'DEVICE CLASSES (242)' including 'Class With Space (1)', 'Discovered (107)', 'Steak Julcer (2)', 'Network (6)', 'Ping (26)', 'Power (11)', 'Printer (22)', 'Robot (1)', 'Server (64)', 'Testing Device (1)', and 'Testsystem (1)'. The main area is titled 'Discovered' with a message: 'Discovered devices are not yet assigned to a device class.' Below this is a table with columns: Device, IP Address, Device Class, Production State, and Events. The table lists several devices, all with a 'Production' state and varying event counts. Some events are highlighted with orange warning icons.

Device	IP Address	Device Class	Production State	Events
cent5-java.zenoss.loc	10.175.211.93	/Discovered	Production	0
cent5bi-64.zenoss.loc	10.175.211.66	/Discovered	Production	0
cent5bi.zenoss.loc	10.175.211.99	/Discovered	Production	1
colibbons-dev.zenoss.loc	10.175.211.233	/Discovered	Production	0
demo-core.zenoss.loc	10.175.211.244	/Discovered	Production	0
dev.zenoss.loc	10.175.211.238	/Discovered	Production	0
entitlement.zenoss.loc	10.175.211.154	/Discovered	Production	0
esx1.zenoss.loc	10.175.211.61	/Discovered	Production	0

We have a table that lists the device name, the device IP, the class, the production state, and the number of events. That's a lot of information to process, and we're not going to be able to decode it all at once.

Right now, I'd like to draw your attention to the **Events** column.

If you're like me, you have several devices with warning level events, which are identified here by the color orange. These are errors that we need to address.

If you autodiscovered your devices, you're going to see more events per device than if you manually added each device. Let's take a look.

To view the details of an event, click on the event column for a device to display the **Event Console** for the selected device. See the following screenshot:

The screenshot shows the Zenoss Core Event Console for a device named 'Coyote' (IP: 192.168.1.110). The device status is 'Up' and 'Production'. The 'Event Console' is displayed, showing a table of events. The table has columns: Status, Severity, Component, Event Class, Summary, First Seen, Last Seen, and Count. Two events are listed: one informational event (blue 'i' icon) for 'Status:Snmp' and one alert event (red '!!' icon) for 'exim4'.

Status	Severity	Component	Event Class	Summary	First Seen	Last Seen	Count
i	Informational	192.168.1.110	/Status:Snmp	*Discovered de	2010-09-10 05:29	2010-09-10 05:29:14	1
!!	Alert	exim4	/Unknown	ALERT: exim p	2010-08-01 06:30	2010-09-09 06:29:53	14

Chapter 6, *Core Event Management* and Chapter 7, *Collecting Events* are devoted to events, but as you look at the details of the events you can begin to see some problems. You might see problems with WMI, SNMP, or refused connections. You may also see blue informational events to let you know the device was discovered.



Informational events will auto clear after four hours. You don't need to be concerned with them.

If you autodiscovered your network, you could see multiple events as Zenoss Core tried to connect to the device with WMI, SNMP, and SSH. Any failed attempts will generate an event that we have to deal with by properly configuring our devices to be monitored or by tuning how we monitor. We'll tune our monitoring in *Chapter 5, Custom Monitoring Templates*, *Chapter 6, Core Event Management*, and *Chapter 7, Collecting Events*.



To clear an event, highlight the event and click the **Close selected event** button at the top of the Event Console.


Manually adding the device will cause Zenoss Core to generate fewer events. However, you may see a warning event that indicates the SNMP agent is down, which indicates SNMP is not properly configured.

Before we can model our devices, we must resolve our SNMP, WMI, or Zenoss Plugin issues. Modeling our devices means we collect performance data, such as disk utilization, CPU utilization, and interface information.

Before we clean up our device inventory, let's take a look at the device discovery process in a bit more detail by examining the log file.

## Reviewing device creation job log

To review the log file from our device discovery job, click on the **Advanced** menu to display the **Settings** page. From the sidebar navigation, choose **Jobs**. Then select the **Jobs** tab to display a table of completed jobs. Actually, if the job is currently running, it will still show in this table with a status of running.

Jobs <span>zenjobs daemon is running.</span>						
Status	Job Type	Description	Started	Finished	Duration	Actions
✓ Succeeded	AutoDiscoveryJob	/usr/local/zenoss/zenoss/bin/Zendisc run --now --monitor localhost --deviceclass /Discovered --parallel 8 --job 89ca6250-07f6-4bbb-a139-f3dba396f3be --net 192.168.0.0/24	7 minutes ago	3 minutes ago	4 minutes	 

At a glance, we see some information about this job, most of which is self explanatory. The **Description** field provides the command that Zenoss Core ran. We can access the log file by clicking the icon that resembles a notepad under the **Actions** column

Click the notepad to display the log file:

```
<< Back to Job Manager

AutoDiscoveryJob "/usr/local/zenoss/zenoss/bin/zendisc run --now --monitor localhost
--deviceclass /Discovered --parallel 8 --job 89ca6250-07f6-4bbb-a139-f3dba396f3be --net
192.168.0.0/24"

If no output is being displayed, make sure the zenjobs daemon is running.

2009-12-28 11:48:57,818 INFO zen.ZenDisc: Connecting to localhost:8789
2009-12-28 11:48:57,823 INFO zen.ZenDisc: Connected to ZenHub
2009-12-28 11:48:57,833 INFO zen.ZenDisc: Discover network '192.168.0.0/24'
2009-12-28 11:49:46,345 INFO zen.ZenDisc: Discovered 15 active ips
2009-12-28 11:50:38,926 INFO zen.ZenDisc: Result: Discovered 15 devices
2009-12-28 11:50:39,140 INFO zen.ZenDisc: WMI collector method for device NPIEC86B4
2009-12-28 11:50:39,141 INFO zen.ZenDisc: plugins: zenoss.wmi.WinServiceMap
2009-12-28 11:50:39,165 INFO zen.ZenDisc: No Python plugins found for NPIEC86B4
2009-12-28 11:50:39,168 INFO zen.ZenDisc: Using SSH collection method for device NPIEC86B4
2009-12-28 11:50:39,168 INFO zen.ZenDisc: plugins: zenoss.cmd.linux.netstat_rn, zenoss.cmd
2009-12-28 11:50:39,171 INFO zen.ZenDisc: SNMP collection device NPIEC86B4
2009-12-28 11:50:39,171 INFO zen.ZenDisc: plugins: zenoss.snmp.NewDeviceMap, zenoss.snmp.D
2009-12-28 11:50:39,176 INFO zen.ZenDisc: No portscan plugins found for NPIEC86B4
2009-12-28 11:50:39,176 INFO zen.ZenDisc: Running 3 clients
```

At the top of the log file, we can see that our device creation job ran the zendisc command:

```
$ZENHOME/bin/zendisc run --now --monitor localhost \
--deviceclass /Discovered --job 015be396-c020-424f-bbd1-ad37cc28e6ec \
--net 192.168.0.0/24
```

Zendisc is a device management command that operates in the collection layer of Zenoss Core's architecture.

The job of zendisc is to scan the supplied network or IP address for a device, and for each device it finds, create a device in the inventory. The discovered devices are added to the Zope database ZODB.

The command accepts several options. The --deviceclass option sets the default class for each new device. The --net option scans the supplied network. These command line options correspond to the values we entered in the add device wizard.



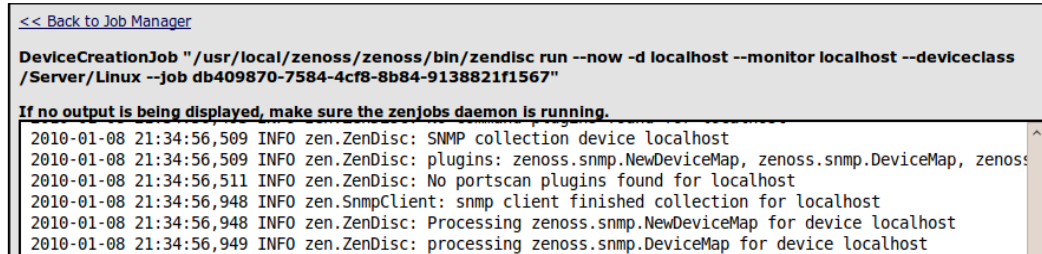
To see the available options for zendisc, run the following command as the zenoss user: zendisc help.

When we work with the Zenoss Core commands from the command line, we need to work as the zenoss user, which was set up during installation. To become the zenoss user, run the following command:

```
su - zenoss
```

```
<enter zenoss user's password>
```

Let's take a look at the log file for a manually added device and note the differences:



```
<< Back to Job Manager

DeviceCreationJob "/usr/local/zenoss/zenoss/bin/zendisc run --now -d localhost --monitor localhost --deviceclass
/Server/Linux --job db409870-7584-4cf8-8b84-9138821f1567"

If no output is being displayed, make sure the zenjobs daemon is running.
2010-01-08 21:34:56,509 INFO zen.ZenDisc: SNMP collection device localhost
2010-01-08 21:34:56,509 INFO zen.ZenDisc: plugins: zenoss.snmp.NewDeviceMap, zenoss.snmp.DeviceMap, zenoss
2010-01-08 21:34:56,511 INFO zen.ZenDisc: No portscan plugins found for localhost
2010-01-08 21:34:56,948 INFO zen.SnmpClient: snmp client finished collection for localhost
2010-01-08 21:34:56,948 INFO zen.ZenDisc: Processing zenoss.snmp.NewDeviceMap for device localhost
2010-01-08 21:34:56,949 INFO zen.ZenDisc: processing zenoss.snmp.DeviceMap for device localhost
```

As the screenshot indicates, our command is:

```
$ZENHOME/bin/zendisc run --now -d coyote --monitor localhost \
--deviceclass /Server/Linux --job 015be396-c020-424f-bbd1-ad37cc28e6ec
```

zendisc used the `--deviceclass` option to set the device class to `/Server/Linux` and the `-d` option specifies a hostname, instead of the `--net` option we used in the autodiscovery example.

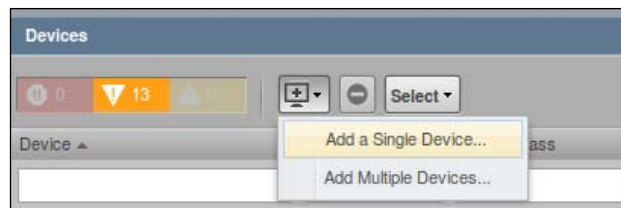
As we step through each of our log files, we can see what transpires between Zenoss Core and the device. The zendisc daemon scans each device looking for SNMP, WMI, and the Zenoss Plugins.

In the screenshot of the manually added device, there are some lines that follow the format of **Processing zenoss.snmp.DeviceMap for device localhost**. This indicates that zendisc successfully connected via SNMP and is running each of the collector plugins for the `/Server/Linux` device class, of which `zenoss.snmp.DeviceMap` is one.

## Adding a single device

We've seen how you can autodiscover the devices on your network, but you may not want to monitor all of the discovered networks. For example, you may have workstations, test servers, and other systems not worth monitoring. By adding a single device at a time, you can also specify device attributes, including classes, SNMP properties, serial numbers, notes, and more.

The **Add Single Device** option is available by clicking on **Infrastructure | Devices**. See the following screenshot:



The **Add a Single Device** window displays as a pop up with the option to add several device attributes:

 A screenshot of the 'Add a Single Device' window. The window has a dark background. It contains several input fields and dropdown menus. The fields are: 'Name or IP:' with a text input field; 'Device Class:' with a dropdown menu showing a path; 'Collector:' with a dropdown menu showing 'localhost'; 'Model Device:' with a checked checkbox; 'Title:' with a text input field; 'Production State:' with a dropdown menu showing 'Production'; and 'Priority:' with a dropdown menu showing 'Normal'. There are also 'ADD' and 'CANCEL' buttons at the bottom.

The advantage of adding devices from this view is that we can add a device with a specific set of properties. The wizard didn't give us this kind of flexibility.

## Entering device attributes

At a minimum, we should enter a **Device Name** and **Device Class**. The **Device Name** identifies the IP address or resolvable hostname, while the Device Class Path sets the monitoring properties we want our device to inherit by default. Device classes are a core concept in Zenoss Core and we will discuss them at length in *Chapter 3, Device Setup and Administration*.

As you look at the **Add Single Device** window, you will notice there is a **More** link that displays additional attributes.

The attributes and relations that we specify on the Add Device page have several benefits. We can describe the devices in our inventory and we can actually use Zenoss Core as a way to manage the information about each device. Using the built-in reporting features of Zenoss Core, we can retrieve that information for later use, analysis, or audit.



Some of the fields, such as groups, class, systems, and location, are often referred to as "organizers" because they create hierarchies that help categorize the device inventory. The organizers help us define monitoring properties, categorization, and alerting rules. Alerting rules dictate how administrators are notified when events occur.

The Add Device Options table lists the available attributes we can set when manually adding a device:

Add Device Options		
Field name	Description	More info
Name or IP	Specify the IP address or resolvable hostname of the device to be monitored.	You can rename the device later to anything you want.
Device Class	Select a class from the drop-down list that describes the device.	The classes form a hierarchy of monitoring properties where devices inherit the properties of the parent class.
Model Device	Determines whether or not Zenoss will try to model the device using SNMP, WMI, and SSH.	Leave unchecked if you know the device doesn't support any of the monitoring protocols.
Collector	The default collector is localhost.	The collector defines how Zenoss collects information, including polling intervals.
SNMP Community	Specify the SNMP community string to try when zendisc discovers the device.	Leave this field blank if the device doesn't support SNMP.
SNMP Port	The default SNMP port is 161.	Zenoss Core will send SNMP get requests to the port specified for the device.
Tag Number	Text field to record service tag numbers or model numbers for the device.	This field helps you manage your devices by providing a place to store service-related information.
Serial Number	Text field to record serial numbers or other identifying information.	Like the Tag Number, this field enables you to enter information that describes the device.

Add Device Options		
Field name	Description	More info
Production State	<p>Select the appropriate state of this device:</p> <ul style="list-style-type: none"> <li>• Production (default)</li> <li>• Pre-Production</li> <li>• Test</li> <li>• Maintenance</li> <li>• Decommissioned</li> </ul>	<p>Production states provide rules for monitoring, alerting, and display:</p> <ul style="list-style-type: none"> <li>• Production: monitor, alert, and display on dashboard</li> <li>• Pre-Production: Excludes alerting and dashboard display</li> <li>• Test: Excludes dashboard display</li> <li>• Maintenance: Excludes alerting</li> <li>• Decommissioned: Do not monitor</li> </ul>
Priority	<p>Select the device's priority:</p> <ul style="list-style-type: none"> <li>• Highest</li> <li>• High</li> <li>• Normal (default)</li> <li>• Low</li> <li>• Lowest</li> <li>• Trivial</li> </ul>	<p>How important are your devices? Use the priority field to rank how important one device is in relation to the others.</p>
Rack Slot	Enter the physical location.	Never let a device hide from you again.
Title	Enter an alternate name.	The title will be used throughout the interface to refer to the device.
Comments	Text field to file notes about the device.	Enter troubleshooting steps, configuration information, or share a haiku.
HW Manufacturer	Select the name of the device manufacturer.	Describe the device.

---

<b>Add Device Options</b>		
<b>Field name</b>	<b>Description</b>	<b>More info</b>
HW Product	Select the product name from the drop-down list. The available products are dependent on the selected HW Manufacturer.	Describe the device.
OS Manufacturer	Select the operating system information from the drop-down list.	Describe the device in terms of the operating system.
OS Product	Select the appropriate operating system version. The available products depend on the selected OS Manufacturer.	Describe the device in terms of the operating system.
Location Path	Select a location from the drop-down list.	The location can describe the device's location in terms of office, city, room, country, or whatever you deem appropriate.
Systems	Select a Systems organizer from the drop-down list. You can select multiple systems.	The Systems organizer is a free form classification that helps you describe the device in the context of your organization.
Groups	Select a Groups organizer from the drop-down list. You can select multiple groups.	The Groups organizer is a free form classification that helps you describe the device in the context of your organization.

---

After we enter the configuration information for the device, click the **Add** button to schedule the job to run via Zenoss Core's zenjobs daemon. You may check on the status of the job by going to **Advanced** | **Settings** | **Jobs**. Refresh the device list to see the newly-added device.

## Importing a list of devices with zenbatchload

So far, we've seen ways to build our device inventory through the web interface. In one case, we auto-discover everything and in the other case, we add the devices one at a time.

What if you already have a list of the SNMP capable devices by hostname or IP address? Wouldn't it be nice to import those devices. Zenoss Core can import a list of devices and attributes via the zenbatchload command.

In its simplest form, zenbatchload will process a text file that lists one device per line. Here's a sample list of devices that I will call `deviceList.txt`:

```
device01
router02
web03
```

Since zenbatchload is a Zenoss Core daemon, we need to run it as the zenoss user. Here are the commands:

```
su - zenoss
<enter zenoss user's password>
zenbatchload deviceList.txt
```

The zenbatchload command will attempt to run SNMP on each device. If the device doesn't support SNMP, it won't be added to the inventory. The zenbatchload command accepts device classes, options, and zProperties in the text file. We'll talk about zProperties in the *Chapter 3, Device Setup and Administration*.

Let's modify `deviceList.txt` with some options:

```
/Network/Router
device01 comments ='Sample Device'
/Server/Linux
router2 zSnmpMonitorIgnore='true'
web03 serialNumber='013478783'
```

The first thing we added was device class definitions. That's the `/Network/Router` and `/Server/Linux` lines. The second thing we added was a list of comma-separated options for each device. The option's value is enclosed in quotes.

When you specify the device class, zenbatchload will identify the class name and assign all subsequent devices to that class name until it detects a new class name. So, in our example, device01 is automatically assigned to `/Network/Router`, router2 and web3 are assigned to `/Server/Linux`. Classes are coming up in the next chapter.

If you want to see which options you can use within the zenbatchload text file, run the following command:

```
zenbatchload --show_options
```



The zenbatchload import utility will not update devices that are already in the device inventory.

The `zenbatchload` file is a harness file for the `$ZENHOME/Products/ZenModel/BatchDeviceLoader.py` file. It's well documented and gives more usage examples.

## Command line discovery with zendisc

The `zendisc` daemon gives us an opportunity to discover devices from the command line. It can be a helpful troubleshooting tool, or it can be a way for you to shed the Web interface and satisfy your curiosity about how Zenoss Core works. Let's take a look at a few example commands.

To see a list of available options, run the following command as the `zenoss` user:

```
zendisc help
```

Let's work through an example where we remodel a device on our network. Modeling the device gives Zenoss Core the characteristics about the device that we monitor. Sometimes Zenoss Core may not be collecting the information we think it should, so we need to figure out why.

Run the following command as the `zenoss` user:

```
zendisc run --remodel -d coyote --logseverity=10
```

Because we used the `--logseverity=10` option, the output is verbose and we can step through the remodeling process, one line at a time. A value of 10 provides the highest level of verbosity.

The `-d coyote` option ran the `--remodel` on the device named `coyote`. Now, if you ran the command exactly as I wrote it, `zendisc` should have reported that it couldn't find an IP address for the device. Feel free to run the command again and substitute the IP address or hostname of a device on your network.

The `run` option tells `zendisc` to immediately run in the foreground, specifically for debugging.

Let's try another command to scan a network for available devices:

```
zendisc run --net=192.168.0.0/24 --parallel=8
```

The `--net=192.168.0.0/24` option scans for IP addresses on the 192.168.0.0 network. We could also specify a range of IP addresses using the `--range` option. The `--parallel=8` option tells `zendisc` to collect from eight devices at a time, which helps discover large networks more efficiently.

One more example: This time let's find all the routers on a network and automatically assign them a device class:

```
zendisc run --net=192.168.0.0/24 --routers \ --deviceclass=/Server/  
Network/Routers
```

While zendisc is helpful for debugging, you could use it to work directly with your device inventory. I know many of you will prefer the command line to the web environment, so enjoy.

## Summary

In order to monitor a device, we need to tell Zenoss Core how to monitor the device. Our options include SNMP, WMI, or Zenoss Plugins over SSH. Then we have to make sure the monitored devices are configured to respond to the monitoring protocol Zenoss Core uses for each device.

As you look at your Zenoss Core device list, you should see a list of devices. However, you may see several events that indicate Zenoss Core is not able to model devices correctly. Events could also signify there's a problem with the monitored device, but we're just starting out. It's more likely we need to tweak the monitoring properties of each device.

A common point of failure in establishing the relationship between the monitoring server and monitored device is credentials. If you are having problems, make sure Zenoss Core can authenticate using the unique credentials for each device.

Our monitoring protocols also require access to specific ports. In this chapter, we outlined the common ports that Zenoss Core will need open in order to monitor via SNMP and SSH.

In the next chapter, we'll refine the monitoring properties for each device so that we can ensure we're collecting the correct information.



# 3

## Device Setup and Administration

Based on our work in *Chapter 2, Discover Devices* Zenoss Core is now monitoring all the devices we added to our inventory. However, Zenoss Core is probably not monitoring the way we need it to, and if we look closely, we may notice that some devices have events associated with them. In this chapter, we're going to configure our devices so that we ensure we collect the proper information.

As we work through the chapter, we will:

1. Organize devices by class, location, system, and group
2. Model devices via SNMP, WMI, and Zenoss Plugins
3. Troubleshoot our Zenoss Core data collection
4. Administer device properties

We'll spend the bulk of this chapter fine-tuning our device inventory and getting it into monitoring shape. One of Zenoss Core's critical concepts is inheritance, which means that the devices inherit monitoring properties from their parent device class.

We can change the monitoring rules at the device level by tweaking individual device properties, called *zProperties*.

Let's start by taking a look at some of Zenoss Core's organizers.



## Organizing devices in Zenoss Core

Zenoss Core provides multiple ways to organize our devices. We're going to take a closer look at Locations, Systems, Groups, Networks, and Classes. Collectively, these organizers allow us to describe devices in a way that's meaningful to your organization.

We can define the organizers to be as specific as we need them to be, and not all organizers are required. In other words, no one will care if you do not define any groups. The information that we do specify can be used to establish monitoring rules, event handling, and alerting rules.

Let's start by adding a location.

### Locations

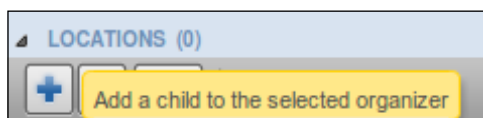
Zenoss Core does not include any location organizers by default. To add a location, click on the **Infrastructure** menu. A list of organizers displays in the left sidebar of the page, as seen in the following screenshot:



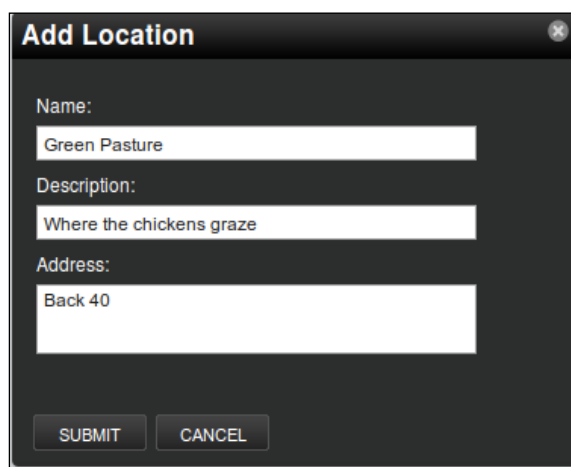
Potential location names are floors, buildings, wings, room numbers, or office locations. Use something that's meaningful to you.

To enter a new location:

1. Select **Locations** from the sidebar of organizers.
2. Click the plus sign (+) at the bottom of the sidebar to **Add a child to the selected organizer** to display the **Add Location** dialog:



3. Enter the **Name** of the location, a **Description**, and an **Address**.
4. Click on **SUBMIT** to add the location:

A screenshot of a 'Add Location' dialog box. It has a title bar with a close button. Inside, there are three text input fields: 'Name:' with 'Green Pasture', 'Description:' with 'Where the chickens graze', and 'Address:' with 'Back 40'. At the bottom are two buttons: 'SUBMIT' and 'CANCEL'.

Enter anything you think may be important about the location of that device. For example, phone extension, driving directions, or site contacts may be practical items to include in the description.

The real-world address may be something you want to record about a location, especially if you have multiple locations. Zenoss Core provides a separate **Address** field for each location name that ties into the Google Maps portlet.

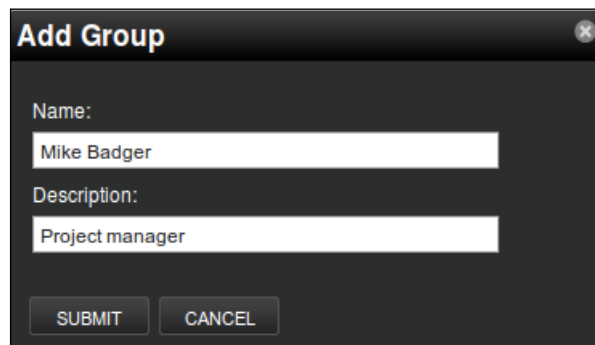
The Google Maps portlet displays your locations on a map and connects them for some nice dashboard eye candy. If a marketer happens to come by, be sure to show them the Google Maps portlet. You may make a friend. We'll cover the maps portlet in *Chapter 9, Extending Zenoss Core with ZenPacks*.

We can add as many locations as we need, or we can define a hierarchy of locations by adding a sub-location. To add a sub-location, click on the name of the location from the list and click on the **Add a child to the selected organizer** button.

## Systems and Groups

Groups are closely related to Systems and how you use them is an exercise of your imagination or operational needs. As an example, you might use groups to define nodes on the org chart while the Systems describe the devices in terms of function. In my monitoring environment, I group websites by project manager, so my groups are names of people. I don't define any "systems" organizers, but you can. There's no wrong answer.

Systems and Groups are displayed in the same spot as the locations, by clicking on the **Infrastructure** menu. Adding either a system or a group works the same way, too. Click on either **System** or **Group**, and then click the **Add a child to the selected organizer** button. Enter the **Name** and **Description**, as shown in the following screenshot:

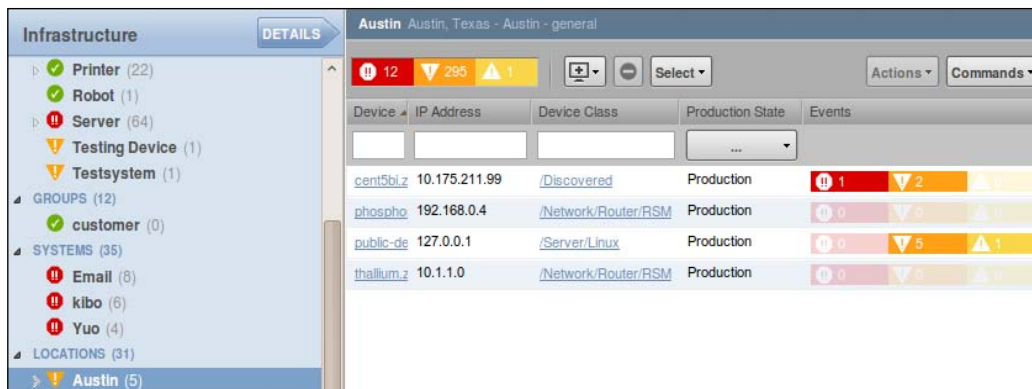


Adding a system is the same as adding a group, except that you select the **Systems** organizer in the sidebar first.

## Organizer details

Before we look at classes, let's take a quick look at how we can work with Locations, Groups, and Systems in the Zenoss Core interface.

You will notice that as you click on an organizer, the device list filters to include all the devices assigned to an organizer. We haven't assigned any devices to an organizer yet, so the device list is empty. However the following screenshot shows an established Zenoss Core installation:



If you click on the **Details** button, the sidebar changes to display the following items: **Devices**, **Events**, **Administration**, **Map**, and **Modifications**.



As we move through the book, we will cover each of these items in more detail, so I'll only provide a brief overview now. If you click on the **Events** link, you see all the events associated with that organizer. We cover events in *Chapter 6, Core Event Management*.

Click on the **Administration** link to display a list of commands, maintenance windows, and device administrators. We cover administration in *Chapter 8, Settings and Administration*.

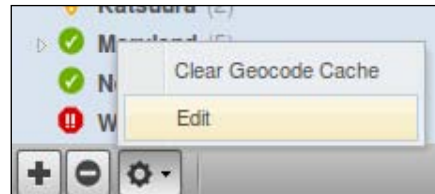
The **Map** page displays a map of the location based on the address information you entered for the organizer. The map requires a Google Maps key and we'll set that up in *Chapter 8, Settings and Administration*.

To see a list of changes for the device, click on the **Modifications** link.

To exit the **Details** page, click on the **See All** button.

## Editing organizers

You can change the name or description of an organizer by selecting the organizer and then clicking **Edit**. You can find the Edit option by clicking on the Actions menu button (which looks like a sprocket) at the bottom of the **Infrastructure** sidebar. See the following screenshot:



## Moving organizers

You can nest organizers by dragging and dropping one organizer into another one to create a hierarchy. You can only nest similar organizers, which means that you cannot move a system into a group or a group into a location.

## Classes

Classes provide one of the most important organizers in Zenoss Core because we can use the classes to establish monitoring properties for groups of devices. Each device inherits the properties of its parent class. Using classes, we can define specific monitoring rules and settings via zProperties. All devices in a class share the same zProperties.

Classes also define what types of information Zenoss Core collects about a device by assigning a set of collector plugins. Like the zProperties, each class contains a set of collector plugins that can change from one class to another.



The Zenoss Plugins that we install on remote Unix servers are not the same as the collector plugins we configure for our devices.

Several classes exist to organize devices, events, services, and processes based on common groupings, but we'll stick to talking about devices right now.

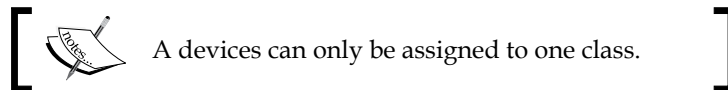
## Viewing a list of device classes

You have probably already noticed that the list of device classes is available when you click on the **Infrastructure** menu. Zenoss Core includes a default hierarchy of classes; for now, we'll work within those default classes.

Next to each class is a number within parenthesis that indicates the number of devices that are associated with the class, including the sub-classes. The icon to the left shows the highest severity event for each class.



Click on the class name to display all the devices assigned to that class.



A hierarchical list of the default device classes in Zenoss Core follows. However you can add as many classes as you want.

### List of device classes

- Discovered
- KVM
- Network
  - Router
  - Cisco
  - Firewall
  - RSM
  - Terminal Server
  - Switch
- Ping

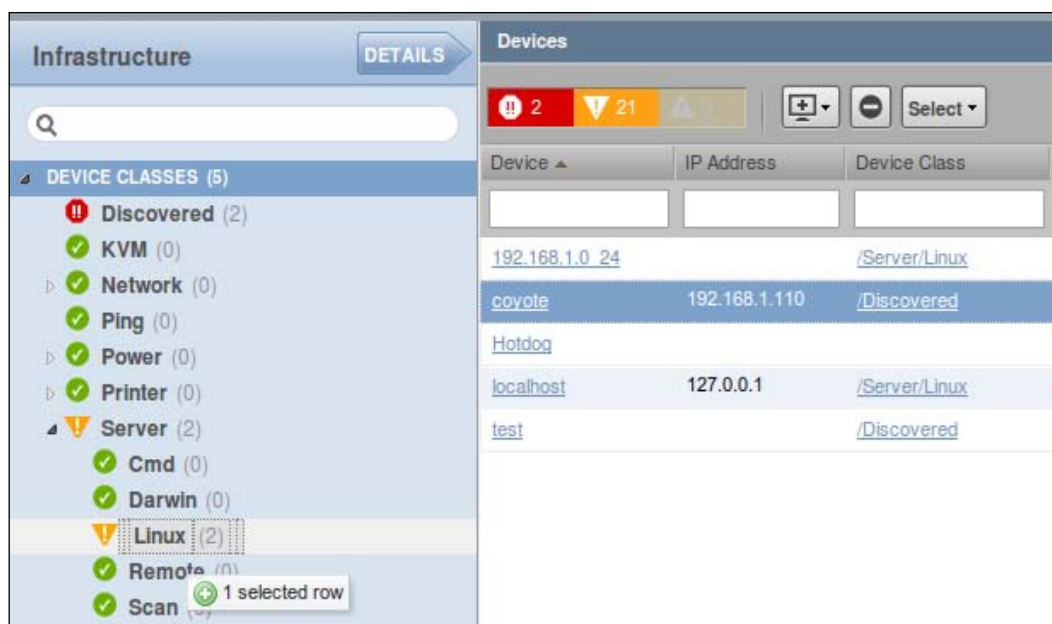
- Power
  - UPS
  - APC
- Printer
  - InkJet
  - Laser
- Server
  - Cmd
  - Darwin
  - Linux
  - Remote
  - Scan
  - Solaris
  - Windows

## Assigning devices to a class

Depending on how you added devices, you may need to reclassify them. For example, if you auto discovered your network, all the devices in your inventory will be classified as **/Discovered**. There's no long term value in that.

To assign devices to a class:

1. Click on the **Infrastructure** menu to display a list of all devices.
2. Select one or more devices.
3. Drag and drop the selected devices onto the appropriate device class.
4. Click **OK** when prompted to confirm the move.



The screenshot shows the device named **coyote** being moved to the **/Server/Linux** class. Notice that the interface is giving you some feedback. You can only assign a device to one class, so the selected class is highlighted.

The tool tip box in the screenshot is showing **1 selected row**, which corresponds to the number of devices you're adding to the class. In the screenshot, only one device is being moved to **/Server/Linux**.

After you classify the devices, it's time to model them.

## Modeling devices

Let's get our devices in shape to be modeled.

When we talk about Zenoss Core, two related terms often come up, monitoring and modeling. Monitoring refers to the availability of the device and answers the question, "Is the device accessible?". Modeling defines relationships between devices and identifies the components available on a device, such as services, interfaces, and file systems. It then collects varying amounts of data about each component.

Zenoss models devices via WMI, SNMP, SSH/Telnet, and port scan. Each class has a default set of collector plugins that tells Zenoss how to model the devices assigned to the class.



Remember in *Chapter 2, Discovering Devices* when we set up our servers to be monitored by setting up WMI, SNMP, and the Zenoss plugins? Now is the time to put that work to use. Feel free to go back and review that section now.

As we model our servers, routers, and other network devices, we'll be interested in setting the collector plugins and zProperties for each device.

## Modeler plugins gather device information

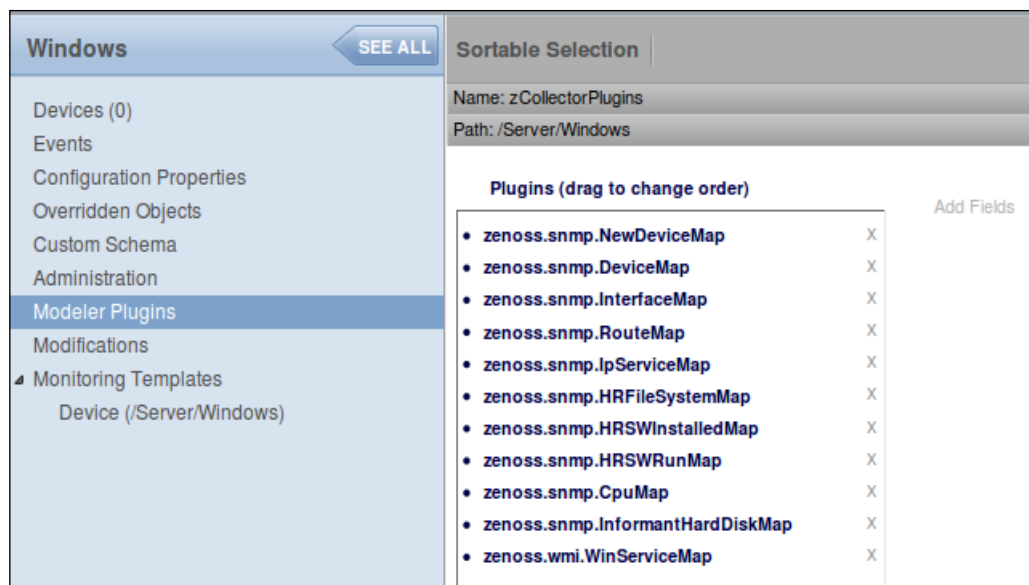
Each modeler plugin queries the device for a specific set of information that it will store in Zenoss Core's device database. What we collect depends on multiple factors, such as the capabilities of the device and the modeler plugins assigned to the device. In general, we can expect to collect information about the operating system, file systems, interfaces, routes, processes, IP services, CPU, and memory.



Modeler plugins can be assigned at the class level. However, we can configure exceptions from the class properties at the device level.

Let's examine these modeler plugins I've been talking about.

1. From the **Infrastructure** menu, select **Devices**.
2. Navigate to the **/Servers/Windows** class.
3. Click on the **Details** button to display a list of settings.
4. Find and click the **Modeler Plugins** link to display a sortable list of plugins.



A page showing the assigned collector plugins displays in the left column of the page with an **Add Fields** link on the right.

To see a list of unassigned plugins click on the **Add Fields** link.

The plugin names are intuitive in that the name suggests the type of information we expect to collect. For example, `zenoss.snmp.IpServiceMap` uses SNMP to return a list of active IP services on the device, such as HTTP. The Dell-specific plugins retrieve more detailed information from Dell devices using OpenManage, and the HP plugins provide more information about devices using Insight Management agents.

The plugins follow a three part naming convention. The first part identifies the author (for example, `zenoss`). The second part lists the collection method (for example, `SNMP`, `WMI`, `cmd`). The final part identifies the specific information collected by the plugin (for example, `IpServiceMap`, `CpuMap`, `uname`).

Actually, the command plugins, identified by the "cmd" in the name, have an optional field to specify system architecture. For example, `cmd.linux` applies to Linux servers while `cmd.darwin` plugins apply to OSX.

To remove a plugin from the assigned plugin list, click on the **x** next to the plugin name. To assign a plugin, drag the plugin name from the available list to the assigned list.



Any change we make to the collector plugins at the class level will be applied to all devices in the class. If we have one-off changes, we should make the changes at the device level.

## Assigning modeler plugins

Take the time now to review the devices in your inventory and set the appropriate collector plugins. If you're not sure where to start, consider the following questions as a starting point:

- Will you monitor with WMI? If not, remove the `zenoss.wmi.WinServiceMap` plugin.
- Does the device support SNMP? If not, remove the `snmp` plugins.
- Is the server a Dell or HP? If so, add the `PowerwareDeviceMap` or `SysedgeDiskMap` and `SysedgeMap` plugins.
- Are you monitoring a remote OSX server with the Zenoss Plugins? If so, add the `zenoss.cmd.darwin` plugin.
- Plan to rely on port scan? Then use the port scan plugin.

Obviously, we can't cover every possible scenario; however that should get you started.

## Troubleshooting data collection

Wouldn't it be nice if everything just worked? Unfortunately, we can expect to make a mistake or two.

## Troubleshooting SNMP problems

If you see an event in Zenoss that reports SNMP agent down, we first need to find out whether the issue is with Zenoss Core or with the device.



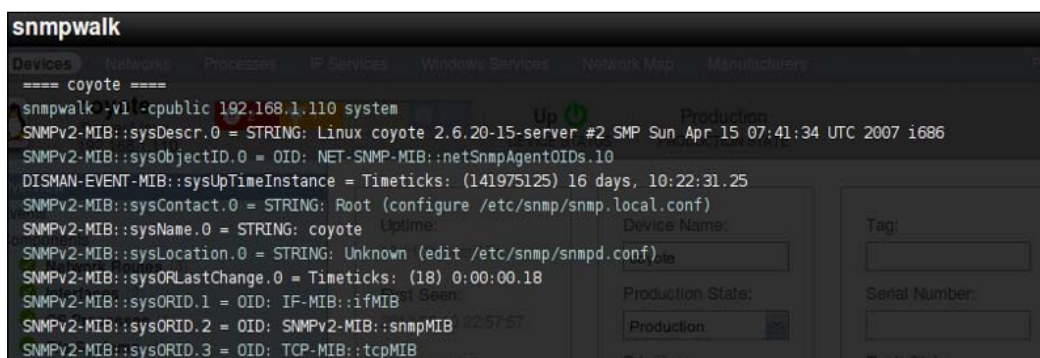
Not all network devices support SNMP. So, before you beat yourself up troubleshooting, make sure your device supports SNMP.

## Running snmpwalk

Zenoss Core includes a command that will run the `snmpwalk` command on the selected device. The `snmpwalk` command will query the device using the SNMP community string we assigned to the device and will try to get a list of the OIDs. Let's try it.

Navigate to **Infrastructure** and select **Devices**. Select a device (by clicking on the device name) that should respond to SNMP requests. The device's **Overview** page displays.

1. At the bottom of the sidebar, expand the **Commands** menu and select **snmpwalk**.
2. The command runs against the device and returns the result, as shown in the following screenshot:



The previous screenshot indicates that Zenoss Core was able to retrieve data from the device successfully. If you have problems, the `snmpwalk` command will either timeout or be refused.

If you're having problems connecting, check the SNMP Community attribute (`zSnmpCommunity`) from the Configuration Settings of the device to make sure it's correct.

If problems continue, let's move to the device itself and run the `snmpwalk` command, assuming we're working with a Linux server. The `snmpwalk` command is included in the SNMP utilities we installed in *Chapter 2, Discovering Devices*.

Run this command on the problem device:

```
snmpwalk -c public -v 2c localhost
```

If the command is successful, pages of results will scroll by the terminal. You should check to make sure the device is allowing incoming connections on port 161/UDP and that both the device and Zenoss Core are using the correct community strings.

## Is the SNMP daemon running on Linux servers?

You should also check to make sure the `snmpd` daemon is running. Run the following command to check:

```
ps aux | grep snmpd
```

We expect to see `/usr/bin/snmpd` running in the process list. If you don't see `snmpd`, then install it as per the *Prepare devices for monitoring* section in *Chapter 2, Discovering Devices*.

If you are still having problems, review the SNMP zProperties for the device. You can try different SNMP versions and make sure SNMP monitoring is enabled. To find the zProperties for a device, navigate to the device's status page. Then click on the **Configuration Properties** link.

A description of the zProperties of all devices is included at the end of this chapter.

## SNMP problems on Windows

Assuming the `snmpwalk` fails when run from Zenoss Core, make sure SNMP is installed on the server (see *Chapter 2, Discovering Devices*).

The built-in Windows SNMP client doesn't provide as much information as Net-SNMP does for our Unix-based systems. You may be wondering why Zenoss Core isn't collecting file system information or CPU statistics. You need a third-party SNMP agent for Windows. Try the free SNMP agent from <http://www.snmp-informant.com>.

## Troubleshooting WMI problems

The first thing you should do is to make sure you have WMI installed as per the instructions in *Chapter 2, Discovering Devices*, and have verified the setup.

## Zeneventlog—unable to connect to Windows

If you see an event that indicates zeneventlog is unable to connect to Windows, that's an indication Zenoss Core is not able to authenticate to the Windows server. Check the Configuration Properties (zProperties) of the device to ensure you have the correct username and password set.

The following screenshot shows the WMI specific zProperties:

zWinEventlog	False ▾	boolean /
zWinEventlogMinSeverity	2	int /
zWinPassword	••••••••••	password /
zWinUser	administrator	string /
zWmiMonitorIgnore	True ▾	boolean /
zXmlRpcMonitorIgnore	True ▾	boolean /

Remember, the zWinUser and zWinPassword properties must be an administrator on the Windows server you're monitoring. It's possible that each of your Windows servers could require a different username and password.

The zWinUser property can be either a local or a domain account. If you use a domain account, specify the domain for the zWinuser value (for example, mojoactive\administrator).

## Zenoss Core does not collect WMI data

It's possible that Zenoss Core isn't collecting any data from the Windows Servers via WMI, but there are no events reported. You can check the following zProperties at the device or class level:

- Make sure the device is using the `zenoss.wmi.WinServiceMap` collector plugin
- Set the **zWmiMonitorIgnore** property to **False** in the **Configuration Properties** for the device or device class
- If you want to collect event logs, set the **zWinEventlog** property to **True** in the **Configuration Properties** for the device or device class

## Troubleshooting Zenoss Plugins

In *Chapter 2, Discovering Devices* we installed the Zenoss plugins on Unix hosts that we may not be able to monitor with SNMP, such as remote Linux or OSX servers. If you're encountering problems, there are several `zProperties` that affect how successfully you can monitor remote machines. See the following screenshot:

zCommandCommandTimeout	15.0	float	/
zCommandCycleTime	60	int	/
zCommandExistenceTest	test -f %s	string	/
zCommandLoginTimeout	10.0	float	/
zCommandLoginTries	1	int	/
zCommandPassword		password	/
zCommandPath	/usr/local/zenoss/libexec	string	/
zCommandPort	22	int	/
zCommandProtocol	ssh	string	/
zCommandSearchPath		lines	/
zCommandUsername		string	/

There are several points of failure, but the most probable error is that the **zCommandPath** value is not valid if you installed the Zenoss Plugins as I described. The `zenplugin.py` file is installed to `/usr/bin`. And as you can see, the **zCommandPath** is pointing to `/usr/local/zenoss/libexec`. This means that each time Zenoss Core tries to collect data from the device using the remote command, it's trying to run `/usr/local/zenoss/libexec/zenplugin.py` on the remote server.

Change the **zCommandPath** property to reflect the actual path to `zenplugin.py` on the remote server, such as `/usr/bin`.

In order for Zenoss Core to connect to the remote server, it must supply valid user credentials. Check the **zCommandUsername** and the **zCommandPassword** properties to ensure they contain a valid username and password for the remote server.

## A class of its own

Zenoss Core provides the `/Server/Cmd` class specifically for monitoring remote machines. The collector plugins are specifically designed to interact with `zenplugin.py` and retrieve data about the remote server. The following screenshot shows the collector plugins assigned to the `/Server/Cmd` class.

**Sortable Selection**

Name: zCollectorPlugins

Path: /Server/Cmd

**Plugins (drag to change order)**

- zenoss.cmd.uname X
- zenoss.cmd.df X
- zenoss.cmd.linux.ifconfig X
- zenoss.cmd.linux.memory X
- zenoss.cmd.linux.netstat\_an X
- zenoss.cmd.linux.netstat\_rn X
- zenoss.cmd.linux.process X
- zenoss.cmd.darwin.cpu X
- zenoss.cmd.darwin.ifconfig X
- zenoss.cmd.darwin.memory X
- zenoss.cmd.darwin.netstat\_an X
- zenoss.cmd.darwin.process X
- zenoss.cmd.darwin.swap X

Save Delete

You won't find any SNMP collectors here.

## Device administration

In this section, we'll take a look at some basic device administration tasks, including: rename a device, delete a device, reset the IP address, and lock the device's configuration.

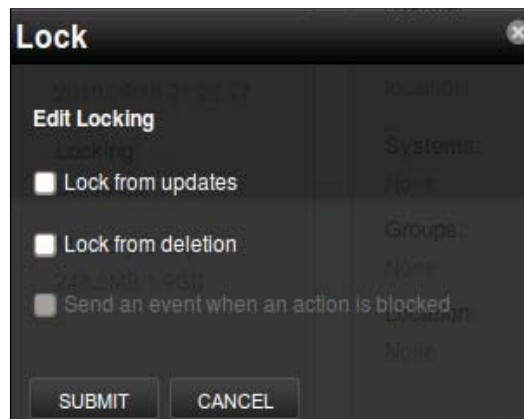
### Locking or unlocking a device

Zenoss Core automatically polls the devices in our inventory and remodels the devices when it finds changes. To prevent changes to the device properties, we can lock the configuration, and we can also lock the device from being deleted from the inventory.



To change the lock status of a device:

1. From the **Device Overview** page, select **Lock** from the **Actions** menu.
2. Select from these choices, as shown in the following screenshot:
  - **Lock from updates**
  - **Lock from deletion**
  - **Send event when actions are blocked**



3. The device status page displays after we choose a locking option.

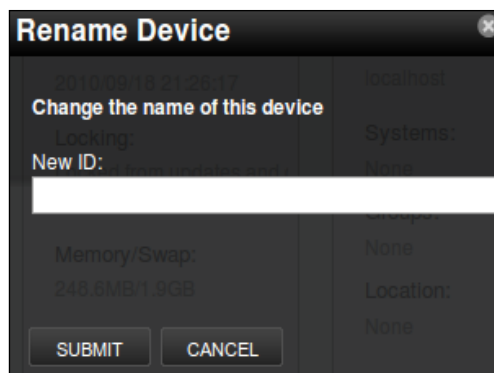
The options should be self explanatory. If you choose to send an event when actions are blocked by a lock, the event will show up in the event console for the device.

## Renaming a device

Zenoss automatically detects and populates the device name, but we can name the device anything we want. On my test network, I prefer to use the names of furbearers.

1. From the **Device Overview** page, select **Rename Device** from the **Actions** menu.
2. Enter the new name (for example, Coyote) in the **ID** field of the **Rename Device** dialog.
3. Click on **OK** to save the change.

On the **Device Status** page, the device information updates to reflect the new name. Even the breadcrumb navigation changes to reflect the new name.



Now, you can manage your devices using whatever slang you wish.

## Resetting the IP address

Sometimes we need to move machines around and allocate new IP addresses. At other times, we may try to monitor a server only to discover that it has a dynamic IP address. Since Zenoss Core requires a static IP address on the monitored device, we need to assign an IP address to the server, and therefore, IP information will need to be updated in Zenoss Core.

To change the IP address:

1. From the **Device Overview** page, select **Reset/Change IP** from the **Actions** menu.
2. Enter the new resolvable hostname or IP address in the **IP Address** field of the **Reset/Change IP** dialog box (shown in the following screenshot) or leave it blank to allow Zenoss to lookup the IP based on the device name.
3. Click **OK** to save the change.



## Push changes

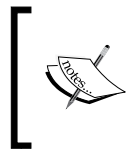
After we make changes to the device, we can "push" the changes to the collectors right away instead of waiting for Zenoss Core to remodel the device. From the **Device Overview** page, select **Push Changes** from the **Actions** menu.

Zenoss Core confirms the action with a status message in the bottom-right corner of the page.

The collectors in Zenoss Core define settings that determine how each device assigned to a collector is monitored. For example, we can configure the default cycle times for the modeler protocol (SNMP, WMI, Ping). On Zenoss Core, the default collector is localhost. Zenoss Core supports only one collector on the same host.

## Deleting devices

Given enough time, we will find devices in our inventory that we want to delete. For example, maybe we accidentally added a bunch of workstations with dynamic IP addresses that go up and down. There are many other reasons to delete a device from Zenoss Core, but I trust you have a firm grasp on when deleting a device from the inventory is necessary.



If you think you might want to monitor the device in the future, you should consider setting the production state to decommissioned. This keeps all the historical data and retains the device in the inventory; however, it's no longer monitored.

To remove a device from inventory:

1. Click on the **Infrastructure** menu and select the devices you want to remove.
2. From the **Device Overview** page, select **Delete** from the **Actions** menu.
3. Select the **Remove Devices** button (it looks like a minus sign) located at the top of the device list.
4. The **Remove Devices** dialog displays and prompts you to select from the following options:
  - Delete current events for these devices
  - Delete performance data for these devices
5. Make the appropriate selections and click on **OK** to confirm the delete.
6. The device will no longer show in inventory and Zenoss Core will no longer monitor it.

If you choose to retain the performance data by not deleting it, the data will be available to the device should we decide to add the device back to Zenoss Core at a later time. We'll review performance data in more detail in *Chapter 5, Custom Monitoring Templates* and *Chapter 6, Core Event Management*.

## zProperties defined

Throughout this chapter, we've seen several specific examples of zProperties, but there are many properties that impact how we monitor. The following table lists the device zProperties along with a brief description of the property. The zProperties are accessible via the Configuration Properties link at the device or class level.

zProperty	Description
zCollectorClientTimeout	Set the collector timeout in seconds. The default value is 180 seconds.
zCollectorDecoding	Specify the character encoding. The default is latin-1.
zCollectorLogChanges	Set to true to log changes and false to not log changes to the collector.
zCollectorPlugins	Click the Edit link to open the collector plugin selection page.
zCommandTimeout	Time in seconds to wait for a command to finish. The default is 15.
zCommandCycleTime	Specifies a time in seconds to cycle through zCommands. The default is 60.
zCommandExistenceTest	Test to see if a command exists on the monitored device. The default is 'test -f %s'.
zCommandLoginTimeout	Wait the specified number of seconds for a login prompt to display. The default is 10.
zCommandLoginTries	Attempt to log in to the remote sever the specified number of times. The default is 1.
zCommandPassword	Enter the password for the user's shell account on the monitored device.
zCommandPath	When you specify a command, Zenoss Core searches the specified path for the command. You can enter fully-qualified commands here.
zCommandPort	The port the monitored system uses for remote connections. The default is 22 for SSH. Specify 23 for telnet.
zCommandProtocol	Specify the protocol to use (Telnet, SSH). The default is SSH.
zCommandSearchPath	Not currently used.

<b>zProperty</b>	<b>Description</b>
zCommandUsername	Enter the user name for the remote device.
zDeviceTemplates	Enter the templates by name to use in order to display information. The default is device.
zFileSystemMapIgnoreNames	Enter the names of the file systems to ignore. For example: /boot.
zFileSystemMapIgnoreTypes	Specify the file systems you want Zenoss Core to ignore.
zFileSystemSizeOffset	Adjust the file system capacity values reported by Net-SNMP. Useful when the Linux server and Net-SNMP report different capacities.
zHardDiskMapMatch	A regular expression to match the names of hard disks.
zIcon	Each device class has a default icon that can be changed as necessary.
zIfDescription	Displays the interface description on the Interfaces table of the OS tab. Select either true or false. The default is false.
zInterfaceMapIgnoreNames	Enter the names of the interfaces to ignore. For example: lo.
zInterfaceMapIgnoreTypes	Enter the type of interfaces to ignore. For example: local loopback.
zIpServiceMapMaxPort	Specify the maximum port number to port scan. The default is 1024.
zKeyPath	Specify the path to the user's public key file for use with public-key authentication.
zLinks	Enter HTML or TALEX expressions to display content on the device's status page. For example, you can create a link to a router's administration console that will display on the Device Status page.
zLocalInterfaceNames	A regular expression match to identify local interface names. The default expression looks for lo (loopback) and vmnet (VMware).
zLocalIpAddresses	A regular expression match to identify local IP address.
zMaxOIDPerRequest	Specify the number of OIDs Zenoss collects with a single query. The default is 40.
zNmapPortscanOptions	Specify the options to use for the zenoss . portscan . IpServiceMap modeler plugin.
zPingMonitorIgnore	Select true to not ping the device or false to ping the device.

<b>zProperty</b>	<b>Description</b>
<code>zProdStateThreshold</code>	Monitor a service that is higher than the production state listed. Possible values include 1000 (Production), 500 (Pre-Production), 400 (Test), 300 (Maintenance), and -1 (Decommissioned).
<code>zPythonClass</code>	Not currently used.
<code>zRouteMapCollectOnlyIndirect</code>	Set to true to collect only the indirect routes. Default is false.
<code>zRouteMapCollectOnlyLocal</code>	Set to true to collect only the local routes. Default is false.
<code>zRouteMapMaxRoutes</code>	Specify the maximum number of routes to map. The default is 500.
<code>zSnmpAuthPassword</code>	Specify SNMP password, if applicable.
<code>zSnmpAuthType</code>	If using <code>zSnmpAuthPassword</code> , select either MD5 or SHA authentication protocol.
<code>zSnmpCommunities</code>	List of communities Zenoss tries to collect information for. The defaults are public and private. Enter more as needed.
<code>zSnmpCommunity</code>	The default community name on the monitored device.
<code>zSnmpMonitorIgnore</code>	Set whether or not Zenoss should monitor the device with SNMP. Defaults to false.
<code>zSnmpPort</code>	The SNMP communication port. It defaults to port 161.
<code>zSnmpPrivPassword</code>	Enter the security user's password.
<code>zSnmpPrivType</code>	Select either AES or DES encryption.
<code>zSnmpSecurityName</code>	Enter the security user's name.
<code>zSnmpTimeout</code>	Length of time in seconds that Zenoss waits for a response from the remote SNMP agent. Defaults to 2.5.
<code>zSnmpTries</code>	Number of times Zenoss tries to connect via SNMP before reporting a failure.
<code>zSnmpVer</code>	The version of SNMP. Available options are 1, 2c, and 3. Defaults to 1.
<code>zSshConcurrentSessions</code>	Set the number of concurrent SSH sessions that Zenoss Core will make. The default is 10.
<code>zStatusConnectTimeout</code>	Specifies the time in seconds for an IP service to respond before the service is marked as down. The default is 15.
<code>zSysedgeDiskMapIgnoreNames</code>	Not currently used.

<b>zProperty</b>	<b>Description</b>
zTelnetEnable	On Cisco routers, send the enable command to enable command collection. Default is false.
zTelnetEnableRegex	Match the enable prompt with the specified regular expression.
zTelnetLoginRegex	Match the login prompt with the specified regular expression.
zTelnetPasswordRegex	Match the password prompt with the specified regular expression.
zTelnetPromptTimeout	Specify the time in seconds to wait for the login prompt to display.
zTelnetSuccessRegexList	Match the command prompt with the specified regular expression.
zTelnetTermLength	Select true to enable Telnet terminal length.
zWinEventLog	Specifies whether or not Zenoss collects the Windows event log. Default is false.
zWinEventLogMinSeverity	Collect all Windows event logs that match the specified severity. Enter a value between 1 and 5, where 1 is the most severe. The default is 2.
zWinPassword	Enter the Windows user's password.
zWinUser	Enter the username of an account on the monitored Windows system.
zWmiMonitorIgnore	Set to true to ignore WMI monitoring and set to false to monitor WMI services.
zXmlRpcMonitorIgnore	Set to true to enable XML/RPC monitoring.

## Summary

At this point, the devices in your inventory should be organized and configured for monitoring. We defined the monitoring settings by assigning devices to classes, adding/removing collector plugins, and fine tuning the zProperties for the class or the individual device.

As we work with our devices and develop our Zenoss Core monitoring environment, we may realize that we want to change some of the work we did in this chapter. And that's no problem.

In the next chapter, we'll dig into the availability and performance data that Zenoss Core collects for each component.

# 4

## Monitor Status and Performance

Is the device available? How has the device performed over time? We answer these questions and more in our discussion about status and performance monitors. Status monitoring lets us know if the device or service is up or down while performance monitoring shows us the device's performance over time in the form of a graph.

In the previous chapters, we have built an inventory of the devices we wish to monitor and Zenoss Core is happily monitoring them according to the classes, `zProperties`, and modeler plugins we defined. In this chapter, we turn our attention to the components we want to monitor for each device. In Zenoss Core terminology, a component refers to some detail about the device, such as routes, services, processes, file systems, and network interfaces.

In this chapter, we will:

- Configure collectors to gather data from each device
- Monitor routes, window services, IP services, processes, file systems, and interfaces
- Graph the performance of individual components
- Customize the event threshold of an existing performance template

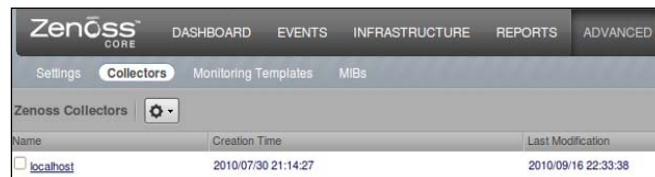
Let's jump right in.



## Collectors collect

The collector for a device sets the frequency that Zenoss Core monitors the device. For example, we can configure how often Zenoss Core checks for device configuration changes, how often it polls Windows events log, polls for SNMP data, and more. We have already encountered collectors while editing device settings. Zenoss Core has one default collector called localhost.

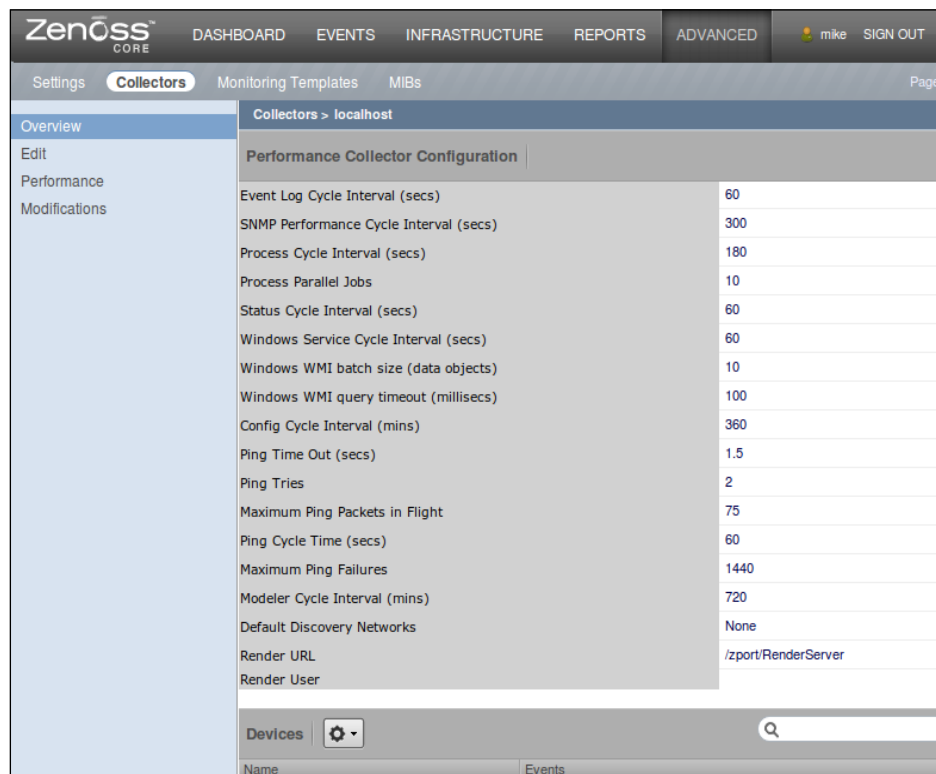
To view the collectors, click on the **Advanced | Collectors** menu. A simple table containing a list of the available **Zenoss Collectors** will be displayed:



This screenshot shows the 'Collectors' page in the Zenoss Core interface. It features a table with three columns: Name, Creation Time, and Last Modification. A single entry, 'localhost', is listed with a creation time of 2010/07/30 21:14:27 and a last modification time of 2010/09/16 22:33:38. The interface includes a top navigation bar with tabs for Dashboard, Events, Infrastructure, Reports, and Advanced, and a sub-navigation bar with links for Settings, Collectors, Monitoring Templates, and MIBs.

Name	Creation Time	Last Modification
localhost	2010/07/30 21:14:27	2010/09/16 22:33:38

Click on the **localhost** monitor to display an overview of the settings as per the following screenshot:



This screenshot displays the configuration page for the 'localhost' collector in Zenoss Core. The page is divided into a left sidebar with navigation links (Overview, Edit, Performance, Modifications) and a main content area titled 'Performance Collector Configuration'. The configuration area contains a list of settings, each with a label and a value. At the bottom, there is a 'Devices' section with a search bar and a table header for Name and Events.

Setting	Value
Event Log Cycle Interval (secs)	60
SNMP Performance Cycle Interval (secs)	300
Process Cycle Interval (secs)	180
Process Parallel Jobs	10
Status Cycle Interval (secs)	60
Windows Service Cycle Interval (secs)	60
Windows WMI batch size (data objects)	10
Windows WMI query timeout (milliseconds)	100
Config Cycle Interval (mins)	360
Ping Time Out (secs)	1.5
Ping Tries	2
Maximum Ping Packets in Flight	75
Ping Cycle Time (secs)	60
Maximum Ping Failures	1440
Modeler Cycle Interval (mins)	720
Default Discovery Networks	None
Render URL	/zport/RenderServer
Render User	

In the overview we see a list of properties, which we will describe in a moment. The bottom table shows all the devices attached to the localhost collector.

From the **Devices** menu, we can assign the device to a new monitor by selecting the **Set Perf Monitor** option. Since we only have one monitor at the moment, we won't be able to reassign the device to a new collector.



The collectors primarily set the polling cycles for Zenoss Core's collection processes, such as WMI, SNMP, and ping.

The previous screenshot shows the properties with default values filled in. We'll explain each property in the next section, but generally speaking, these properties tell Zenoss Core how often it should do each of the items listed.

For example, the zenping daemon pings the device every 60 seconds to check its availability. The zenping daemon expects a ping response within 1.5 seconds. If no response is received, a second ping request is sent. The default number of ping tries is two. If zenping fails to receive a response after two ping tries (three seconds), it generates an event.

The first question, that comes to mind is, "do all these servers deserve to be monitored every minute for availability?" Maybe we want to check some servers only every five minutes. In that case, we can increase the **Ping Cycle Time** to 300 seconds.

Let's take a look at the collector properties.

## Configuring the performance collector

The following table lists each property, data type, and description. The description indicates which underlying Zenoss Core daemon requests the data. Using the following settings, we can create collectors with highly customized monitoring properties that we can assign to one or more device classes.

Property	Data type	Description
Event Log Cycle Interval	int	The time in seconds that the zenwin daemon collects Windows event logs. The default is <b>60</b> .
SNMP Performance Cycle Interval	int	The time in seconds that the zenperfsnmp daemon collects SNMP performance data. The default is <b>300</b> .

Property	Data type	Description
Process Cycle Interval	int	The time in seconds that the zenprocess daemon collects performance data about the OS process. The default is <b>180</b> .
Process Parallel Jobs	int	The number of jobs to process at one time. The default is <b>10</b> .
Status Cycle Interval	int	The time in seconds that the zenstatus daemon collects data about IP services. The default is <b>60</b> .
Windows Service Cycle Interval	int	The time in seconds that the zenwin daemon collects performance data about Windows services. The default is <b>60</b> .
Windows WMI Batch Size	int	The number of items zenwin asks for at one time. The default is <b>10</b> .
Windows WMI query timeout	int	The time in milliseconds that zenwin will attempt to connect to the Windows server. The default is <b>100</b> .
Config Cycle Interval	int	The time in minutes that Zenoss reloads the monitor configuration. The default is <b>360</b> .
Ping Timeout	float	The time in seconds that zenping waits for a reply to the ping command. Default is <b>1.5</b> .
Ping Tries	int	The maximum number of ping attempts per cycle Interval. The default is <b>2</b> .
Cycle Interval	int	The time in seconds that Zenoss collects availability data. The default is <b>60</b> .
Maximum Ping Failures	int	If the device fails to respond to a ping for the specified number of consecutive tries, remove it. The default is <b>1440</b> (36 hours).
Chunk Size	int	Specifies a default chunk size for a ping, in bytes The default is <b>75</b> .
Configuration Reload Interval	int	The time in minutes that Zenoss reloads the configuration. The default is <b>20</b> .
Default Discovery Networks	line	Specify the networks to auto-discover in CIDR format. Enter one IP number per line. For example: 192.168.0.0/24.
Render URL	string	Used for inter-daemon communication (XML/RPC) for graphing information. The default is <b>/zport/RenderServer</b> .

Property	Data type	Description
Render User	string	The username required to connect to the Render URL. Default is blank.
Render Password	string	The password associated with the Render User.
Default RRD Create Command	line	A set of default commands that Zenoss Core uses to create performance graphs from the collected data.

## Monitoring components

Now that we know a bit about how Zenoss collects status and performance information, we turn our attention to monitoring the status of the components from the device's overview page. In Zenoss Core, our components are categorized by Network Routes, Interfaces, OS Processes, File Systems, Win Services, IP Services, and Processors.

To view the components for a device, click on the device name from the list of devices in the Infrastructure menu. The list of components for each device varies based upon several variables, such as the modeling protocol and the modeler plugins used for the device. Therefore, not all the devices will have the same components listed.

The screenshot displays the Zenoss Core web interface. The top navigation bar includes 'DASHBOARD', 'EVENTS', 'INFRASTRUCTURE' (selected), 'REPORTS', and 'ADVANCED'. Below this, a sub-menu shows 'Devices' (selected), 'Networks', 'Processes', 'IP Services', 'Windows Services', 'Network Map', and 'Manufacturers'. The main content area is for the device 'coyote' (Server/Linux, 192.168.1.110). It shows a status bar with 'Up' and 'Production' state. A left sidebar lists components: Overview, Events, and Components (expanded). The Components list includes: Network Routes (3), Interfaces (2), OS Processes (1), File Systems (4), IP Services (15), and Processors (1). The main panel displays device details: Uptime (02d:17h:11m:43s), First Seen (2010/09/10 22:57:57), Last Change (2010/09/18 09:26:24), Device Name (coyote), Production State (Production), and Priority (Normal).

We'll cover each of the following components in turn: Interfaces, OS Process, IP Services, Win Services, File Systems, Processors, and Network Routes.

## Interfaces

As part of the modeling process, Zenoss Core discovers the interfaces running on the device and automatically begins monitoring them for throughput and errors. However, Zenoss Core does not monitor the interface for up/down status by default. There are community workarounds involving event transformations and custom data sources in the interface template.

The following screenshot shows the discovered interfaces for a server.

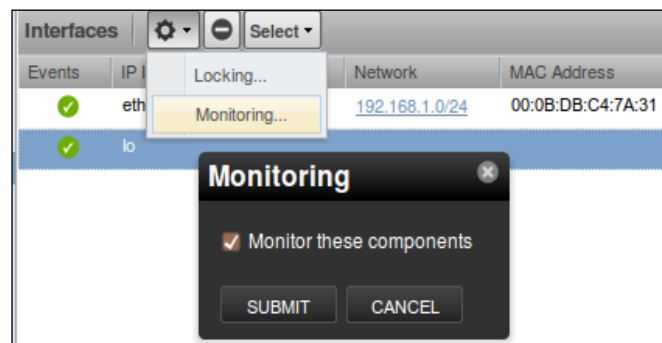


Events	IP Interface	IP Address	Network	MAC Address	Status	Monitored	Locking
✓	eth0	192.168.1.110	192.168.1.0/24	00:0B:DB:C4:7A:31	Up	true	
✓	lo				Unknown	false	

As you review the screenshot, you see that the **Interfaces** table lists some descriptive information about the device such as highest event associated with the interface, the interface name, IP address, network, MAC address, availability status, monitoring status, and locks. The action menu allows us to lock the device or change the monitoring status of the device. Just because Zenoss discovers the interface doesn't mean we need to monitor it. For example, we may decide we don't really need to monitor the loopback adapter.

The loopback adapter (lo) provides an interface for network traffic that only takes place on the local machine, and it always has an IP address of 127.0.0.1. To disable monitoring of the loopback interface:

- Select the **lo** interface from the list
- From the **Actions** menu, select **Monitoring** to display the **Monitoring** dialog box
- Uncheck **Monitor these components** and click **SUBMIT**



The **Monitored** status for the **lo** interface is now **false**, indicating that the device is not monitored. To make sure we stop monitoring the loopback adapter, we can edit the Configuration Properties for the device and set `zInterfacemapIgnoreName` to `^lo`.

## OS Processes

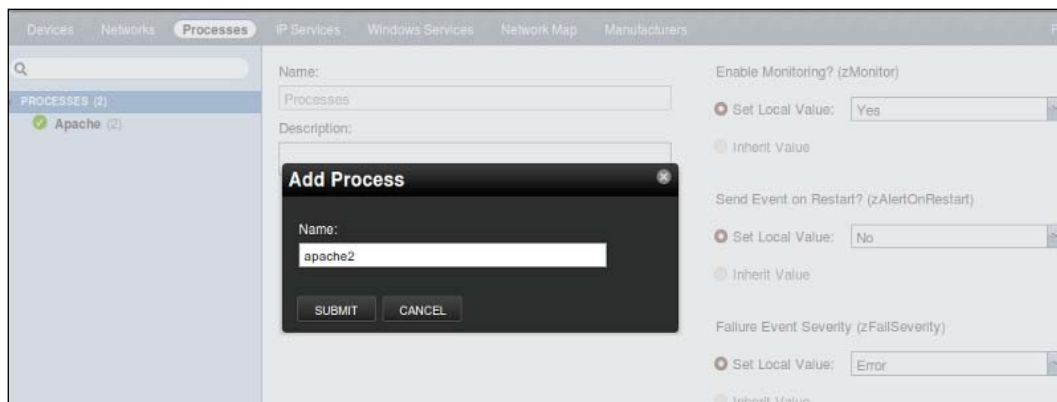
Zenoss Core can keep tabs on almost any Unix process we want to monitor; however, it won't be able to gather reliable statistics for short-lived processes which are up at irregular intervals.

Before we can monitor an OS process, we need to add it to Zenoss Core's list of processes. By default, Zenoss Core does not monitor any processes.

## Add Process

Let's add the `apache2` process, so we can monitor it:

- Click on the **Infrastructure** menu and then select **Processes**. The Processes page displays.
- From the Add menu (at the bottom of the Processes sidebar), select **Add Process**.
- Type `apache2` in the name field of the **Add Process** dialog box and click **SUBMIT**. When we enter a "name" in this dialog box, we're really specifying the pattern we want Zenoss Core to search for when it monitors the process.



The new process is added to the list.



You can create organizers, a.k.a subfolders, to help categorize your list. The add menu at the bottom of the **Processes** sidebar also has an **Add Process Organizer** option. After you have a process organizer defined, you can drag and drop processes.

## Viewing or editing the process details

To view or edit the monitoring details of the process, click on the name of the process. See the following screenshot:

Most of the settings on this screen should be self explanatory, but it pays to review a few of the more important properties.

The **Pattern** field allows you to enter a new regular expression pattern.

By default, Zenoss Core will monitor the process. You can override the default setting by selecting **No** under the **Enable Monitoring** heading. This setting is also configurable via the zMonitor zProperty, which is covered in the next section.

If you want a down process to generate an event with a higher priority, you can set that using the **Failure Event Severity**, which is tied to the zFailSeverity zProperty.

## Configuration properties

We can specify monitoring properties for an individual process or the process organizer. Processes will inherit the properties of their parent organizer. The configuration settings are handled via the zProperties.

To view the zProperties, select the process or process organizer from the list under **Infrastructure | Processes**. Then find the **Display** drop-down menu and choose **Configuration Properties**.

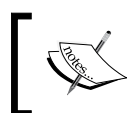
The following table shows the zProperties:

Property	Data type	Descriptions
zAlertOnRestart	Boolean	Determines if Zenoss generates an event when the process restarts. The default value is False.
zCountProcs	Boolean	Set this value to true if you want to find all the instances of the process that may be running.
zfailSeverity	Int	Specify the default severity for events: The default is Error.
zMonitor	Boolean	Specify whether or not you want to monitor this process.

Don't forget to save any changes you make.

## Monitoring OS Processes

The next time Zenoss Core models the devices, it will automatically scan each device for the process and automatically add discovered instances to the device's OS Processes list. Remember our discussion on collectors earlier in the chapter? The default collector models devices every 12 hours.



To force remodeling of individual devices, select the **Remodel Device** option from the **Actions** menu, which can be found on the devices overview page.

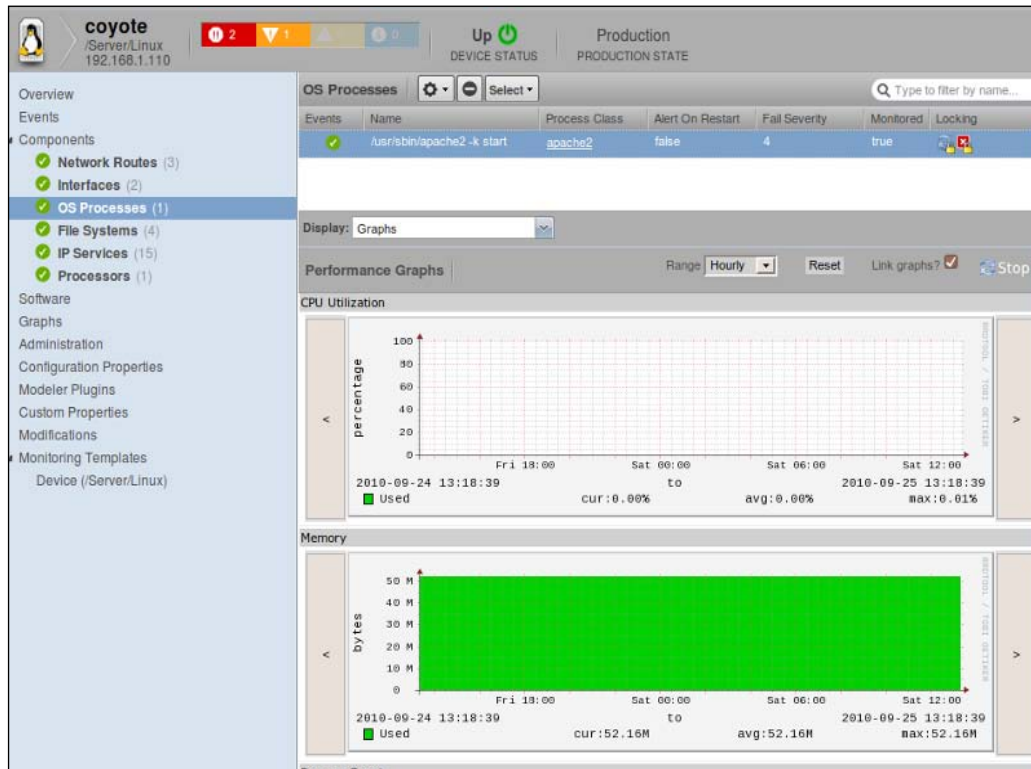
Once Zenoss Core remodels the devices, we can view a list of all the monitored process instances from the **Processes** page. Let's look at our Apache example navigating to the apache2 process. The page should display the Process Instances by default. The following screenshot shows all the devices with the apache2 process running:

Display: Process Instances			
Device	Name	Monitored	Status
<a href="#">localhost</a>	apache2	true	Up
<a href="#">covote</a>	/usr/sbin/apache2 -k start	true	Up



In the **Process Instances** list, each device lists the process name, monitoring status, and process availability.

But what does the monitored process look like from the Device's point of view. Let's find out. The following screenshot shows the **OS Processes** for a device with **apache2** selected.



We see performance graphs for CPU Utilization, Memory, and Processes Found (not pictured).

## Services

Unlike processes, Zenoss Core provides a list of services that we can monitor across Windows and Unix servers. Zenoss Core monitors two classes of services: IP Services and Windows Services. To view the list of services, select **Windows Services** or **IP Services** from the **Infrastructure** menu. The IP Services are organized by **Privileged** and **Registered**.

The rules for how we make Zenoss Core monitor a service is the same whether we want to monitor IP services (for example, Telnet, SMTP, HTTP) or Windows Services (for example, Event Logs).

To monitor a service:

- Enable monitoring for the service at the class level
- Configure the monitoring properties
- Make monitoring exceptions at the device level

Let's walk through the process of monitoring an IP Service.

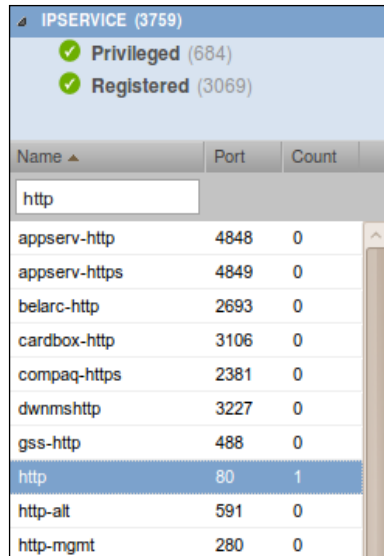


Monitoring is not set for individual services by default in Zenoss Core. If you want to monitor a service, you need to explicitly enable it.

## Enable monitoring for a service

To show how the services work, let's enable a service. Pick any service you want, but this exercise will work better if you actually have a device running that service. In my example, I will monitor http.

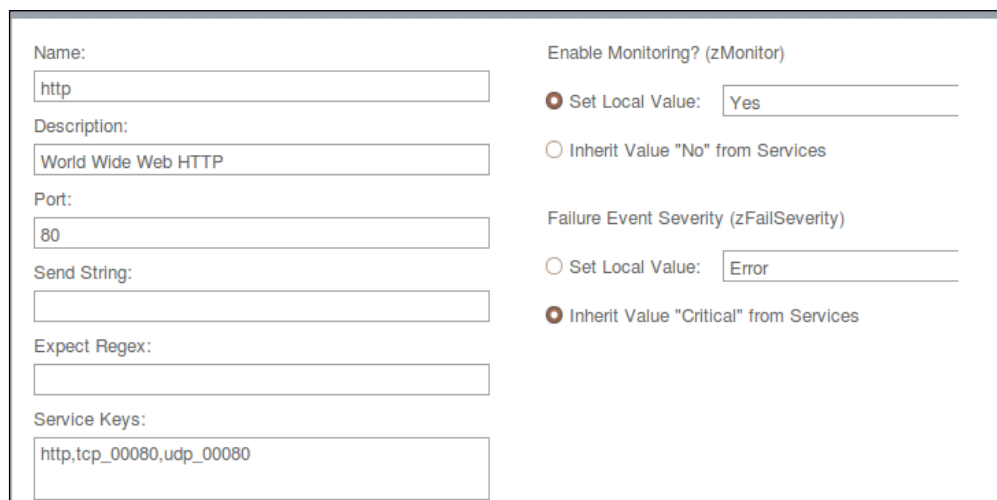
Navigate to **Infrastructure | IP Services** to display a list of services. The sidebar should display a long list of services. Zenoss Core provides a filter box, located just above the list of services. You can narrow down the list and save some scrolling by typing http into the filter. See the following screenshot:



The screenshot shows the 'IPSERVICE (3759)' interface. At the top, there are two status indicators: 'Privileged (684)' and 'Registered (3069)', both with green checkmarks. Below these is a table with columns 'Name', 'Port', and 'Count'. A search filter 'http' is applied to the 'Name' column. The table lists several services, with 'http' (port 80, count 1) highlighted in blue.

Name	Port	Count
appserv-http	4848	0
appserv-https	4849	0
belarc-http	2693	0
cardbox-http	3106	0
compaq-https	2381	0
dwnmshhttp	3227	0
gss-http	488	0
http	80	1
http-alt	591	0
http-mgmt	280	0

Each service class has a set of properties that define how we monitor the service, including whether or not monitoring is enabled. Since you have the service selected, you see a list of monitoring properties, as seen in the following screenshot:



The screenshot shows the configuration form for the 'http' service. It includes fields for Name, Description, Port, Send String, Expect Regex, and Service Keys. On the right, there are sections for 'Enable Monitoring? (zMonitor)' and 'Failure Event Severity (zFailSeverity)', each with radio buttons for 'Set Local Value' and 'Inherit Value from Services'.

Name:	http	Enable Monitoring? (zMonitor)	<input checked="" type="radio"/> Set Local Value: Yes
Description:	World Wide Web HTTP		<input type="radio"/> Inherit Value "No" from Services
Port:	80	Failure Event Severity (zFailSeverity)	<input type="radio"/> Set Local Value: Error
Send String:			<input checked="" type="radio"/> Inherit Value "Critical" from Services
Expect Regex:			
Service Keys:	http,tcp_00080,udp_00080		

To monitor this service, select **Yes** from the **Enable Monitoring** heading. As you can read on the screen, this is also a zProperty. Let's look at the zProperties now.

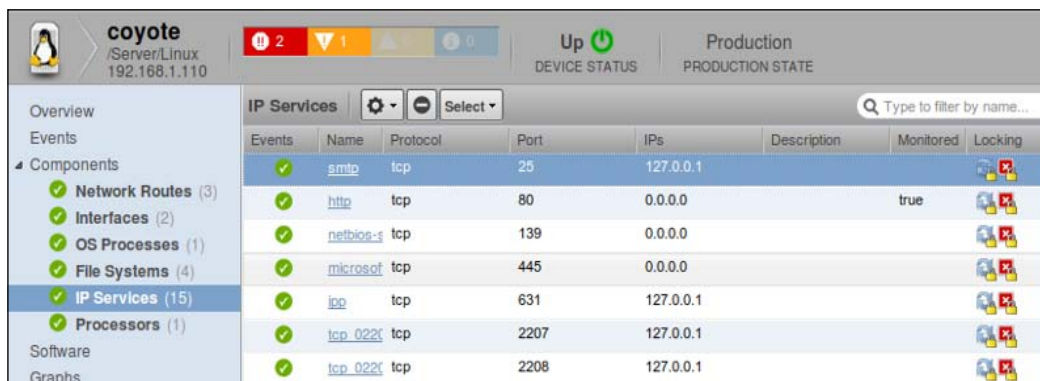
## Configuration properties

With the service selected, choose **Configuration Properties** from the **Display** menu. The following table explains the zProperties you will find:

Zproperty	Data type	Description
zFailSeverity	Int	Zenoss will generate an event with the specified severity when the service becomes unavailable.
zHideFieldFromList	Lines	Hide the list of columns from the services lists. Available options are <b>port</b> , <b>description</b> , <b>monitor</b> , and <b>count</b> . Enter each value on a new line.
zMonitor	Boolean	To monitor the service, select <b>True</b> .

## Monitoring exceptions for services

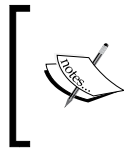
At this point we just told Zenoss Core to monitor all instances of http, but we may not want to monitor every instance.. We can selectively disable monitoring for a service on a per device basis. The following screenshot shows the **IP Services** list for a device:



The screenshot shows the Zenoss Core interface for a device named 'coyote' (Server/Linux, 192.168.1.110). The 'IP Services' tab is selected, displaying a list of services. The 'http' service is highlighted, and its 'Monitored' status is set to 'true'. Other services listed include 'smtp', 'netbios-s', 'microsoft', 'ipp', 'tcp\_0220', and 'tcp\_0220'.

Events	Name	Protocol	Port	IPs	Description	Monitored	Locking
✓	smtp	tcp	25	127.0.0.1			
✓	http	tcp	80	0.0.0.0		true	
✓	netbios-s	tcp	139	0.0.0.0			
✓	microsoft	tcp	445	0.0.0.0			
✓	ipp	tcp	631	127.0.0.1			
✓	tcp_0220	tcp	2207	127.0.0.1			
✓	tcp_0220	tcp	2208	127.0.0.1			

Notice in the screenshot that I have several services listed, but only one of them is being monitored. The **http** service has **true** in the **Monitored** column. That tells us Zenoss Core has discovered several services for this device but monitoring is disabled for most of them.



Remember, our example is using IP Services, but the same logic applies to Windows Services. If you enabled a Windows Service and viewed your Windows Server, the Components list would include a page for Windows Services.

We can change the status of monitored services:

- Select a service (for example, http) from the list
- Select Monitoring from the Actions menu
- When the **Monitoring** dialog box displays, uncheck **Monitor these components** and click on **Submit**
- Zenoss Core will change the monitored status of the http service to false for the current device

If the Monitored column is blank, you will not be able to set the monitoring status at the device level. You will first have to enable monitoring at the service level.

## Interactively monitor IP services

We can configure Zenoss Core to **Send String** when it checks the status of an IP service. Then we can define the expected result in the **Expect Regex** field. See the following screenshot, which shows the properties of the http service class:

The screenshot shows the Zenoss Core interface for configuring IP services. The 'IP Services' tab is selected, and the 'http' service is chosen. The left pane shows a list of services with columns for Name, Port, and Count. The right pane shows the configuration for the selected service.

Name	Port	Count
http	80	1
appserv-http	4848	0
appserv-https	4849	0
belarc-http	2693	0
cardbox-http	3106	0
compaq-https	2381	0
dwnmshhttp	3227	0
gss-http	488	0

Configuration details for the 'http' service:

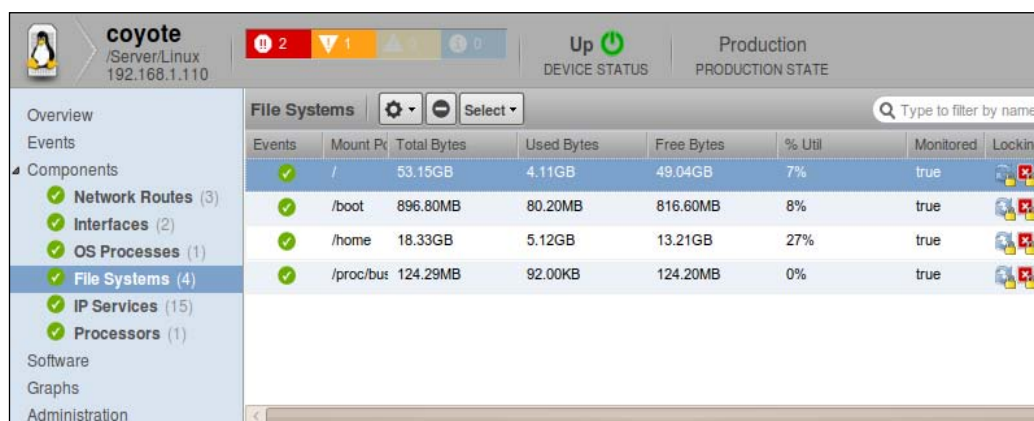
- Name:** http
- Description:** World Wide Web HTTP
- Port:** 80
- Send String:** (empty field)
- Expect Regex:** (empty field)
- Service Keys:** http.tcp\_00080,udp\_00080
- Enable Monitoring? (zMonitor):**
  - ☒ Set Local Value: Yes
  - ☐ Inherit Value "No" from Services
- Failure Event Severity (zFailSeverity):**
  - ☐ Set Local Value: Error
  - ☒ Inherit Value "Critical" from Services

Any change we make at the service class level will apply to all instances of the service, and the next time Zenoss models the devices, it will apply the new monitoring rules. It will send the string and expect the result you specify.

## File Systems

Zenoss Core models the file system hierarchy and reports the volume name (**Mount**), capacity in **Total Bytes**, **Used Bytes**, **Free Bytes**, **% Utilization**, and whether or not the file system configuration is locked. If we add a new **Mount** point to the device, such as an extra drive, Zenoss Core automatically detects and adds the new file system when it models the device.

The File Systems component is only visible at the device level, which is different than the processes and services we have been working with. See the following screenshot:



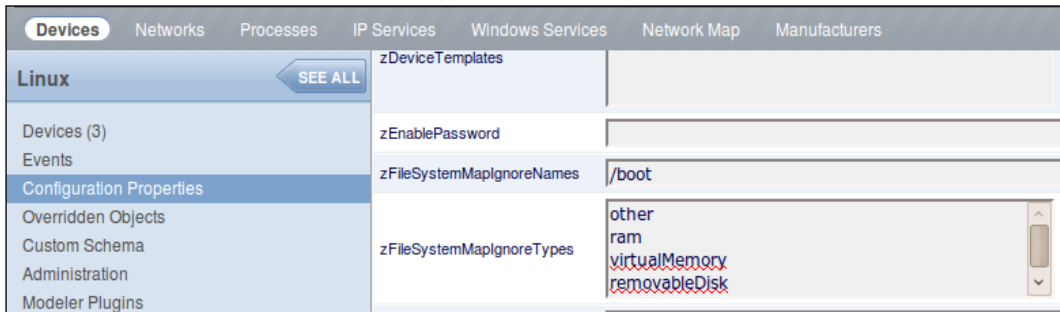
The screenshot shows the Zenoss Core interface for a device named 'coyote' (Server/Linux, 192.168.1.110). The 'File Systems' component is selected in the left sidebar. The main panel displays a table of file systems with columns: Events, Mount Pt, Total Bytes, Used Bytes, Free Bytes, % Util, Monitored, and Locking. The table lists four file systems: '/', '/boot', '/home', and '/proc/bus'. Each row has a green checkmark in the 'Events' column and a 'true' value in the 'Monitored' column. The 'Locking' column contains icons for each file system.

Events	Mount Pt	Total Bytes	Used Bytes	Free Bytes	% Util	Monitored	Locking
✓	/	53.15GB	4.11GB	49.04GB	7%	true	[Icons]
✓	/boot	896.80MB	80.20MB	816.60MB	8%	true	[Icons]
✓	/home	18.33GB	5.12GB	13.21GB	27%	true	[Icons]
✓	/proc/bus	124.29MB	92.00KB	124.20MB	0%	true	[Icons]

## Ignoring File Systems with zProperties

Zenoss Core provides a couple of device zProperties that help us configure what file systems we monitor by default. They are **zFileSystemMapIgnoreNames** and **zFileSystemMapIgnoreTypes**.

Let's say we want to ignore the `/boot` partition for all Linux servers. Navigate to the **Configuration Properties** of the **/Server/Linux** device class and enter `/boot` into the **zFileSystemMapIgnoreNames** field.



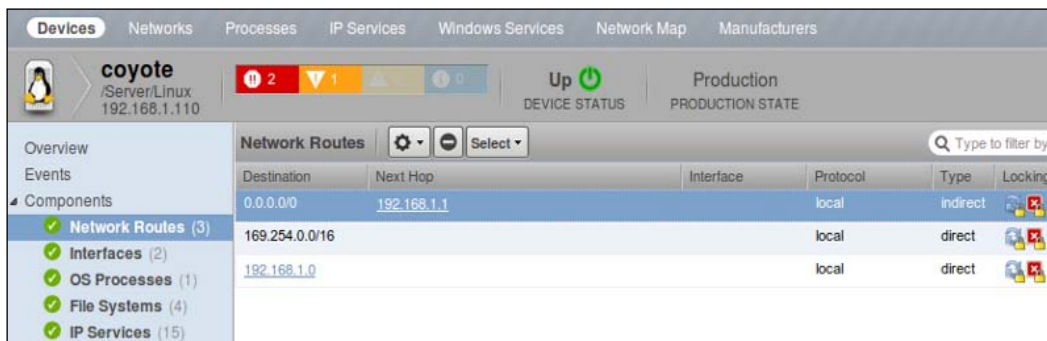
The **zFileSystemMapIgnoreNames** field accepts a regular expression.

The **zFileSystemMapIgnoreTypes** property is a multi-line entry that lists the types of memory that are ignored by Zenoss Core. If for some reason, you wanted to monitor the virtual memory of a server or a class of servers, remove the **virtualMemory** option.

Changes to these zProperties will be reflected the next time Zenoss Core models the device.

## Network Routes

For each device, Zenoss Core discovers the routing table and displays the following information in the **Network Routes** table, which is listed under the **Components** for each device: **Destination**, **NextHop**, **Interface**, **Protocol**, **Type**, and configuration **Locks** (as shown in the following screenshot).



If the **Destination** or **NextHop** values correspond to a discovered network, then we can click on the IP Address to display the network properties. This will open the **Networks** page in Zenoss Core, which is also a main menu under **Infrastructure**.

## Networks

Zenoss Core allows you to enter multiple networks and configure the default monitoring properties for each network. Some of the interesting things we can decide on a network-by-network basis are whether or not we want to auto-discover the network or whether Zenoss Core should only add devices that are available via SNMP.

We don't need to review each of these settings, but I'll draw your attention to some of them. First, in the left middle of the screen, Zenoss Core reports the number of used and free IP addresses.

Then as we work down the right side of the settings, you can see options to disable **Auto-discovery** on a network. Setting auto-discovery to false means that Zenoss Core will not automatically add new devices you may add to the network. You will have to add them manually.

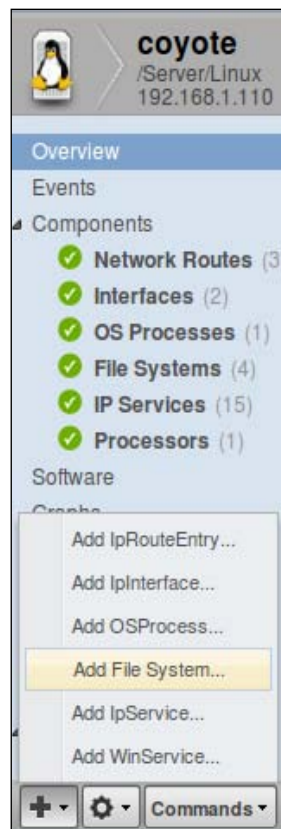
If, for some reason, you want Zenoss Core to name the device according to the SNMP name and not DNS, you enable that here.



If Zenoss Core discovers a device that doesn't respond to SNMP, it will still add the device to the inventory. You can set the **Only create devices if SNMP Succeeds** to yes to drop those devices.

## Add Components

We've just run through the configuration of the components that Zenoss Core could monitor for a device. On a device-by-device level, we can add components to the device by clicking the **Add Components** button (which looks like a + sign).



As we talked about the individual components, we mentioned many times that Zenoss Core would automatically detect the component. So why add it to the device manually?

There are a few reasons you might want or need to add a component manually. Maybe you previously removed it from the device. Maybe Zenoss Core did not discover it, or maybe you don't want to wait 12 hours for Zenoss Core to find the component.

For whatever reason, when you select one of the add menus, it will display a component-specific dialog box. Try it and see. The **Add OSProcess** menu is a unique menu in that it only displays the items defined in the Processes class. Remember, we added `apache2` as our example.

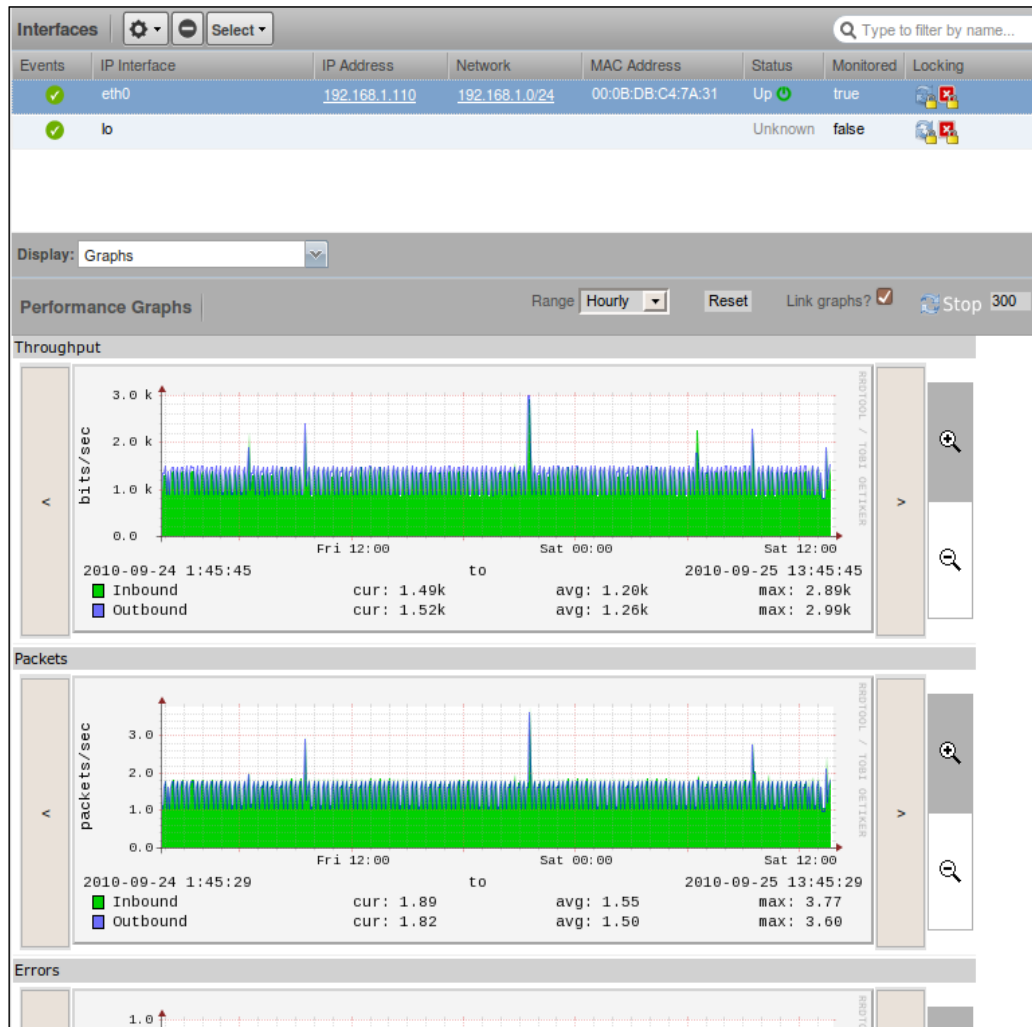
## Viewing and editing component details for a device

If Zenoss Core is monitoring a component for a device, it will display in the Components list on the device overview. To see information, such as events, configuration properties, or graphs related to a component, click on the component while viewing the device details.

We're going to walk through an example using interfaces, but the same concepts apply to all the components. Navigate to a device and click on **Interfaces** from the **Components** list.

## Performance Graphs

To view the graph, you must select a specific component name. In the screenshot that follows, we're looking at the **eth0** interface. Note that a performance graph for an interface has three graphs: **Throughput**, **Packets**, and **Errors** (not pictured).



If you do not see the list of graphs when you select an interface, select **Graphs** from the **Display** drop-down list (see previous screenshot). You can also display events, details, templates, and modifications.

The data on the graph is collected and displayed according to the template associated with the interface.

## Interface template

A template tells Zenoss Core what data to collect, how to collect it, how to graph it, and when to generate events based on the collected data. Templates are bound to devices, classes, and components.

Zenoss Core binds templates automatically based on the monitoring rules we've set up, but we can change template configurations, as needed. For example, Zenoss Core's default interface template will generate an event when the interface is at 75 % utilization or higher.

To view the template for an interface, select **Template** from the **Display** drop-down list, and then click on the template name to display its properties.

As you can see in the following screenshot, the template consists of **Data Sources**, **Thresholds**, and **Graph Definitions**:

Data Sources		Thresholds															
<div> <div>+</div> <div>-</div> <div>⚙</div> </div> <div>Data Points by Data Source ▲</div> <table> <tr> <th></th> <th>Source</th> </tr> <tr> <td>▶ ifInErrors</td> <td>1.3.6.1.2.1.2.2.1.14</td> </tr> <tr> <td>▶ ifInOctets</td> <td>1.3.6.1.2.1.2.2.1.10</td> </tr> <tr> <td>▶ ifInUcastPackets</td> <td>1.3.6.1.2.1.2.2.1.11</td> </tr> <tr> <td>▶ ifOperStatus</td> <td>1.3.6.1.2.1.2.2.1.8</td> </tr> <tr> <td>▶ ifOutErrors</td> <td>1.3.6.1.2.1.2.2.1.20</td> </tr> <tr> <td>▶ ifOutOctets</td> <td>1.3.6.1.2.1.2.2.1.16</td> </tr> <tr> <td>▶ ifOutUcastPackets</td> <td>1.3.6.1.2.1.2.2.1.17</td> </tr> </table>		Source	▶ ifInErrors	1.3.6.1.2.1.2.2.1.14	▶ ifInOctets	1.3.6.1.2.1.2.2.1.10	▶ ifInUcastPackets	1.3.6.1.2.1.2.2.1.11	▶ ifOperStatus	1.3.6.1.2.1.2.2.1.8	▶ ifOutErrors	1.3.6.1.2.1.2.2.1.20	▶ ifOutOctets	1.3.6.1.2.1.2.2.1.16	▶ ifOutUcastPackets	1.3.6.1.2.1.2.2.1.17	<div> <div>+</div> <div>-</div> <div>⚙</div> </div> <div>Name</div> <div>high utilization</div>
	Source																
▶ ifInErrors	1.3.6.1.2.1.2.2.1.14																
▶ ifInOctets	1.3.6.1.2.1.2.2.1.10																
▶ ifInUcastPackets	1.3.6.1.2.1.2.2.1.11																
▶ ifOperStatus	1.3.6.1.2.1.2.2.1.8																
▶ ifOutErrors	1.3.6.1.2.1.2.2.1.20																
▶ ifOutOctets	1.3.6.1.2.1.2.2.1.16																
▶ ifOutUcastPackets	1.3.6.1.2.1.2.2.1.17																

Graph Definitions	
<div> <div>+</div> <div>-</div> <div>⚙</div> </div> <div>Name</div> <div>Throughput</div> <div>Packets</div> <div>Errors</div>	

The reason I have brought you to this awkward looking screen is to show you how the data is being tied together. In the template shown in the screenshot, the **Data Sources** correspond to OID values that Zenoss Core monitors via SNMP.

The **Thresholds** contain rules that generate events. Not all templates have thresholds established by default.

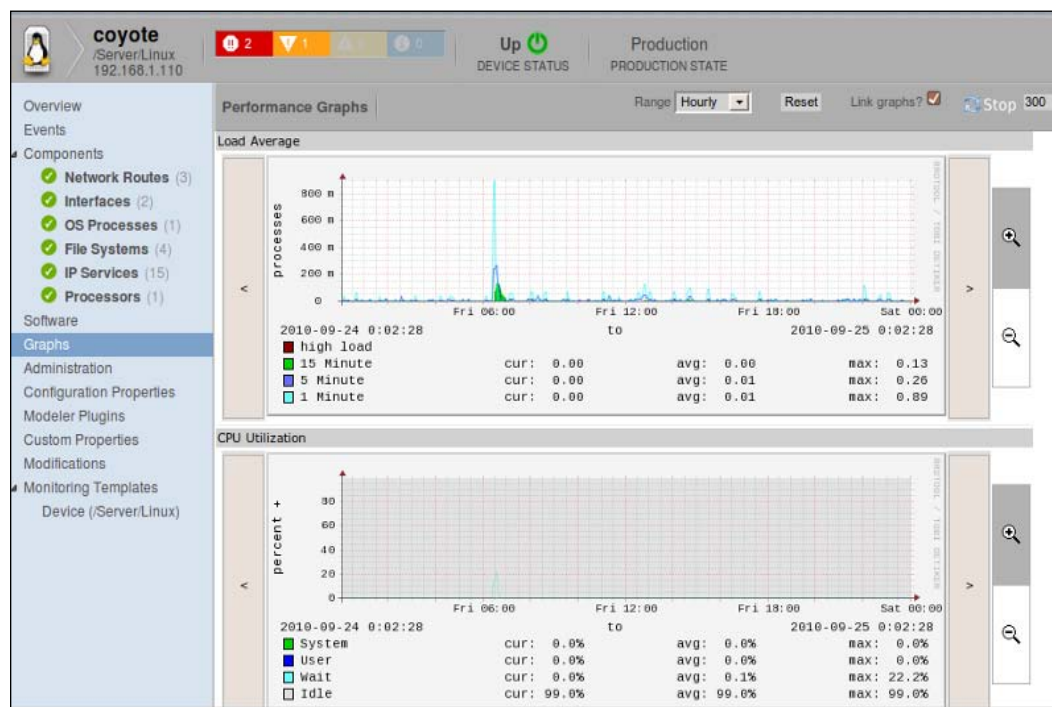
The **Graph Definitions** take the information from **Data Sources** and display it in a graph. As the screenshot shows, there are **graph definitions** for **Throughput**, **Packets**, and **Errors**. If that doesn't look familiar, you skipped the last couple of pages.

In the next chapter, we explore templates in more depth.

## Performance Graphs

Zenoss Core creates time series graphs using the RRDTool for the device and its components. "Time series" implies that we continuously measure data at regular intervals.

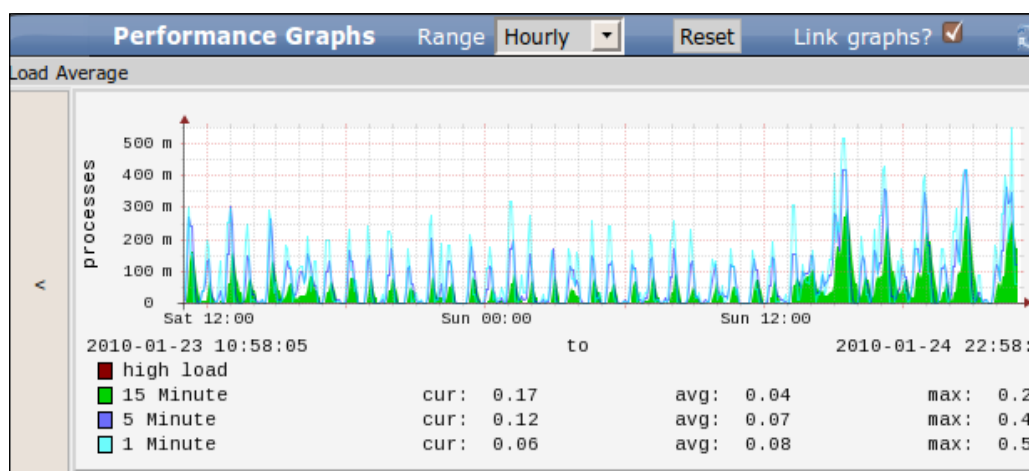
Devices may have graphs to display CPU utilization, load average, memory utilization, and IO. To display these graphs, navigate to a device and click on the **Graphs** link. See the following screenshot:



The other place we find performance graphs is at the Component level, which we've been discussing.

## Working with graphs

All performance graphs function in the same way. They can be viewed on an hourly, daily, weekly, monthly, or yearly range. The default **Range** is **Hourly** (see the following screenshot):



Each group of graphs has a **Reset** button that restores all the graphs on a page to the default view. The **Link graphs** checkbox allows all the graphs on the page to stay synchronized as we navigate through the time line of an individual graph.

Each graph also has its own set of controls. On either side of the graph, we have time line navigation controls. The < navigates backward through the date range while the > navigates forward. The magnifying glasses allow us to zoom in and out on the graph. To zoom in, click the +, and then move the cursor over the graph. When the cursor turns to a cross hair, click the mouse button to zoom in. The same process applies to zoom out, except that we select the - magnifying glass.

The layout of each graph also follows a common format. The time measurement plots on the x axis and the data point being measured plots on the y-axis. Beneath the x-axis, the graph identifies the visible time range.

At the bottom of the graph, we see the color-coded data points represented on the graph. Each data point displays current, average, and maximum measurements for the visible time range.

Defining which data points a graph displays is controlled in the performance template that is bound to the device. Collecting data via data points will be a central topic in *Chapter 5, Custom Monitoring Templates*.

## Monitoring performance thresholds

Monitoring and graphing performance is related to monitoring a component for a specific threshold. For example, if a file system exceeds certain utilization, Zenoss Core can create an event to alert us of the state of the system.

By default, Zenoss Core has several default thresholds established via the performance templates. Some thresholds are as follows:

- File system: 90 % utilization on /Device class
- Interface: 75 % utilization on /Device class
- CPU percentage: Less than 2 % on /Device class
- CPU high load: Greater than 1200 on /Device/Linux class

In order to change or add to our graphs, including thresholds, we modify the device's performance template, which we will do in the next chapter.

## Summary

We now have everything we need to configure our monitoring environment. This chapter showed us how to monitor interfaces, processes, services, file systems, and network routes.

In *Chapter 5, Custom Monitoring Templates*, we will fine-tune our data collection by editing and creating monitoring templates. We'll use the data we collect to build graphs and thresholds for event generation.

# 5

## Custom Monitoring Templates

In the preceding chapters, we've seen how to set up our devices to be monitored by defining classes, modeler plugins, and components. We now turn our attention to one of the most flexible features within Zenoss Core: monitoring templates.

Templates are a graphical frontend to RRDtool. They collect performance data from our devices and add that data into Zenoss Core's data model so we can monitor and graphically display it.

In this chapter we will:

- Monitor using SNMP data sources, including MIBs and OIDs
- Monitor using a command data source (Nagios and Cacti plugins)
- Bind monitoring templates to device classes

This chapter roughly divides into two halves. In the first half, we tweak an existing template and add a new SNMP data source and OID.

In the second half, we create a template that uses a Nagios style plugin that randomly generates monitoring values; therefore it provides a great way to demonstrate or experiment with events, data collection, and the RRDtool. And the beauty of the plugin is that it does not require a real device to work.

### Monitoring Templates

Templates allow us to customize our data collection when we decide the Zenoss Core defaults no longer meet our needs. We could for example, collect data for a specific OID, implement a Nagios plugin, or create our own plugin.



Monitoring Templates tell Zenoss what data sources to collect from and how to graph the data. They can also establish monitoring thresholds on the collected data, which we can use to generate events. We apply templates to the device class or to the individual device. In Zenoss terminology, the process of applying a template to a device or a class is called **binding**.

To view all the monitoring templates in Zenoss Core, click on the **Advanced** menu and then select **Monitoring Templates**. In the sidebar you will see an expandable list of templates. Let's look at an example.

Expand the **Device** heading and click on **/Server/Linux**. The details of the template will display, as shown in the following screenshot:

The screenshot shows the Zenoss Core interface for the 'Monitoring Templates' page. The left sidebar has a search bar and a tree view under 'Device'. The tree view includes categories like '/Devices', '/Network/Router/Cisco', '/Ping', '/Power/UPS/APC', '/Server', and '/Server/Linux' (which is selected). Below these are specific templates like 'ethernetCsmacd', 'ethernetCsmacd\_64', 'FileSystem', 'HardDisk', 'IpService', 'OSProcess', and 'WinService'. The main content area is divided into two sections: 'Data Sources' and 'Thresholds'. The 'Data Sources' section contains a table with columns for 'Data Points by Data Source', 'Source', 'Enabled', and 'Type'. The 'Thresholds' section contains a table with columns for 'Name' and 'Value'. The 'Graph Definitions' section contains a table with columns for 'Name' and 'Value'.

Data Points by Data Source	Source	Enabled	Type
laLoadInt1	1.3.6.1.4.1.2021.10.1.5.1	true	SNMP
laLoadInt15	1.3.6.1.4.1.2021.10.1.5.3	true	SNMP
laLoadInt5	1.3.6.1.4.1.2021.10.1.5.2	true	SNMP
memAvailReal	1.3.6.1.4.1.2021.4.6.0	true	SNMP
memAvailSwap	1.3.6.1.4.1.2021.4.4.0	true	SNMP
memBuffer	1.3.6.1.4.1.2021.4.14.0	true	SNMP
memCached	1.3.6.1.4.1.2021.4.15.0	true	SNMP
ssCpuIdle	1.3.6.1.4.1.2021.11.11.0	true	SNMP
ssCpuRawWait	1.3.6.1.4.1.2021.11.54.0	true	SNMP
ssCpuSystem	1.3.6.1.4.1.2021.11.10.0	true	SNMP
ssCpuUser	1.3.6.1.4.1.2021.11.9.0	true	SNMP
ssIORawReceived	1.3.6.1.4.1.2021.11.58.0	true	SNMP
ssIORawSent	1.3.6.1.4.1.2021.11.57.0	true	SNMP
sysUpTime	1.3.6.1.2.1.25.1.1.0	true	SNMP

Name	Value
high load	
low CPU idle	

Name	Value
Load Average	
CPU Utilization	
Memory Utilization	
IO	

As you look over the list of monitoring templates, you should make some connections to *Chapter 4, Monitor Status and Performance*, where we reviewed device components. The **FileSystem**, **IpService**, **OSProcess**, **WinService**, **HardDisk**, **ethernetCsmacd\_64**, and **ethernetCsmacd** monitoring templates match the components we saw on the OS tab in the previous chapter.

The **Data Sources** tell Zenoss Core what information to collect and define the type of data it is. Zenoss Core supports SNMP, command, and built-in data types. It supports the following data types:

- **SNMP:** Uses an OID value.
- **Command:** Uses the output from a shell command in a Nagios or Cacti compatible plugin. The Zenoss Plugins we talked about in *Chapter 2, Discover Devices* are a command type data source.
- **Thresholds:** If defined, this will generate an event when the data source collects information that crosses the threshold.
- **Graph Definitions:** This provides rules to display a graphical representation of the information collected from the data source and the thresholds.



Changes we make at the Monitoring templates level will be inherited by all devices bound to the template. Zenoss Core allows us to override each template at the device level.

## Monitoring SNMP data sources

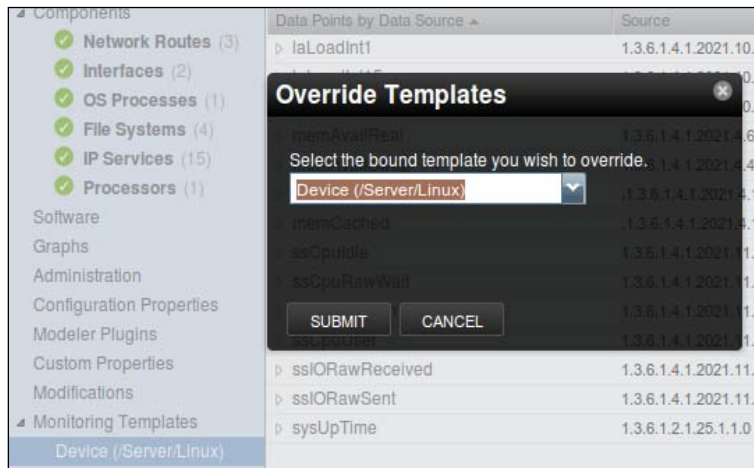
We're going to edit an existing monitoring template that's already set up for SNMP. This will allow us to become familiar with the monitoring templates and discuss how Zenoss Core incorporates OIDs into the template's data source.

## Overriding templates

Rather than make changes at the class level, let's modify the device template for an individual device. In my example, I'll continue to work with my test server which is a member of the `/Server/Linux` class. If you don't have a Linux server in the `/Server/Linux` class, you can work through the example by editing the `/Server/Linux` template.

1. Navigate to the device and select **Details**, and from the **Monitoring Templates** heading, select **Devices (/Server/Linux)**.
2. From the **Actions** menu, select **Override Template Here**. In the **Override Templates** dialog box, select **Device (/Server/Linux)**, and then click on **SUBMIT**.

3. The monitoring template name changes to **Device (Locally defined)**.  
Creating a copy of the template provides us with some protection from ourselves and is equivalent to making a backup of a file. We preserve the original template settings, so that if we don't like the changes we make to the local copy of the template, we can delete the local copy and restore the original settings.
4. Let's Edit the **Device (Locally Defined)** template. Click on the template name to display the **Data Sources**, **Thresholds**, and **Graph Definitions**.



The **Data Sources** table lists the **Name**, **Source**, **Source Type**, and **Enabled** status for each data source. The **Source Type** is either SNMP or command by default. Additional source types can be added via plugins, ZenPacks, and add-on modules in order to extend Zenoss Core's functionality.

Data Sources			
<div> <div></div> <div></div> <div></div> </div>			
Data Points by Data Source	Source	Enabled	Type
cpu	\$(zCommandPath)/zenplugin.py cpu over SSH	true	COMMAND
mem	\$(zCommandPath)/zenplugin.py mem over SSH	true	COMMAND
uptime	\$(zCommandPath)/zenplugin.py uptime over SSH	true	COMMAND



The data sources vary depending on the device class. Here are a few examples:

The `/Server/Linux` class monitors multiple OIDs, which we've already seen.

The `/Server/Cmd` class monitors with the Zenoss Plugins over SSH or Telnet, so its **Source** is the `zenplugin.py` file with a **Source Type** of command.

The `/Server/Windows` class relies on the SNMP Informant to collect data on a couple of OIDs; the `/Server/Linux` class by comparison, uses Net-SNMP and collects data for three times as many OIDs.

Not all device classes collect or graph performance data with their default templates. The `/Server/Scan` and `/Server/SSH` device classes have blank performance templates because they primarily monitor services in a way that may not report performance data, such as port scan in `/Server/Scan`.

Data sources and the related data points are the key components of a template because they define what data we have available to build thresholds and graphs. We'll explore the relationship between data points, thresholds, and graphs later in this chapter.

Let's make a few changes to the data sources for `/Server/Linux` to demonstrate how we can work with SNMP data sources. In the second part of this chapter, we'll work with command data sources when we create a template from scratch.

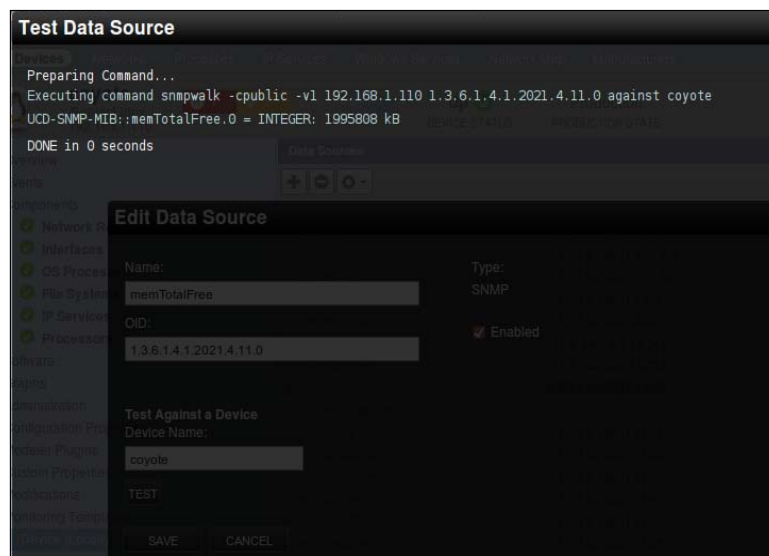
## Editing the `/Server/Linux` template

Currently, the `/Server/Linux` device template monitors the available physical memory (**memAvailReal**) and the available swap memory (**memAvailSwap**). The template defines what we graph and creates a minimum threshold for available memory.

That's great stuff, if you need it. Maybe you need to monitor physical memory and swap memory separately. Maybe you don't want to monitor any memory characteristics. No problem. There is nothing magical about the Zenoss Core default values. Zenoss Core can be easily customized to your needs, and we'll work through some template customization now.

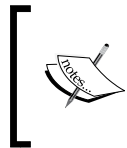
Let's work through an example where we monitor memTotalFree for our Linux server:

1. Navigate to the Linux server from the device list. Then select **Device (Locally Defined)** from **Monitoring Templates**.
2. Let's add the memTotalFree data source. From the **Data Sources** menu, choose **Add Data Source**.
3. In the **Add Data Source** dialog box, type **memTotalFree** for the **Name**. Select **SNMP** for the **Type**.
4. Click **Add**. The **memTotalFree Data Source** is added, but it's not configured.
5. To configure the data source, select **memTotalFree** from the list. Then select **View and edit details** from the **Data source edit options** menu.
6. Set the following properties in the **Edit Data Source** dialog box.
7. Set **Enabled** to **True**.
8. Set **OID** to **1.3.6.1.4.1.2021.4.11.0**.
9. Set **Type** to **Gauge**.
10. The **Edit Data Source** allows us to test the data source against a device. To run the test, type the name of the device name (my example uses **coyote**) and click the **Test** button.
11. When you're sure that the data source is configured properly, click **Save**.
12. The following screenshot shows the output of the data source **Test** command against a device:



The **Test Against Device** field just ran the `snmpwalk` command against the device we specified using the community string and SNMP version from the `zProperties`. And it asked the device to report the value of the `memTotalFree` OID.

The value of `memTotalFree` or OID `1.3.6.1.4.1.2021.4.11.0` is reported in the screenshot with the line **UCD-SNMP-MIB::memTotalFree.0 = INTEGER: 1995808 kB**.



Sometimes it's just faster to work from the command line. You can use the `snmpwalk` command to test a device for a specific OID by using a command in the following format:

```
snmpwalk -c <community> -v <SNMP version> <device> <oid>
```

If the `snmpwalk` command fails, then you have some troubleshooting to do. A good place to start is to verify your SNMP setup, which we covered in *Chapter 2, Discover Devices*.

The other obstacle here is knowing which OIDs you have available to monitor. MIBs are collection of OIDs and Zenoss Core includes several MIBs that we can use to find OIDs.

## Find OIDs for SNMP monitoring

Zenoss Core provides a MIB browser in the Zenoss Core web interface that we can display by clicking on the MIBs menu from the navigation sidebar. The problem is that the list is blank by default; however Zenoss Core stores lists of MIBs in `$ZENHOME/common/share/snmp/mibs` and `$ZENHOME/common/share/mibs`.

We can use the `zenmib` command to add the MIBs into the graphical MIB browser so we can find what we're looking for.

Since we used the `1.3.6.1.4.1.2021.4.11.0` OID in our example, let's pull its parent MIB into Zenoss Core so we can browse it. If we look back at the output of the `snmpwalk` test command we ran in the previous section, we see that the response line provides us with the MIB we care about, UCD-SNMP-MIB, which happens to be located in `$ZENHOME/common/share/snmp/mibs`.

To add all the OIDs in the UCD-SNMP-MIB to the MIB browser, let's open a command line and run the following commands as the zenoss user:

```
cd $ZENHOME/common/share/snmp/mibs
zenmib run UCD-SNMP-MIB.txt
```

When the `zenmib` command completes, it will provide some status information, such as the number of nodes that the MIB contained, which is 156 in our example.



To add all the MIBs in a directory, use the wild card, `*`, in place of the MIB name. For example: `zenmib run *`.

Now, let's view the MIBs from Zenoss Core, which you'll find in the **Advanced** menu. We see **UCD-SNMP-MIB** listed with a **Description**, the number of **Nodes**, and the number of **Notifications**:

Name	Description	Nodes	Notifications
<input type="checkbox"/> <a href="#">UCD-SNMP-MIB</a>	Clarify behaviour of objects in the memory & systemStats groups	156	2

As the screenshot indicates, you can create sub-folders to organize your Mibs. We're not going to worry about that now. Let's browse the MIB instead by clicking on **UCD-SNMP-MIB** to display the properties of the MIB:

**MIBs > UCD-SNMP-MIB**

State at time: 2010/10/05 20:20:16

Name: UCD-SNMP-MIB

Language: SMiv2

Contact:

This mib is no longer being maintained by the University of California and is now in life-support-mode and being maintained by the net-snmp project. The best place to write for public questions about the net-snmp-coders mailing list at net-snmp-coders@lists.sourceforge.net, postal: Wes Hardaker P.O. Box 382 Davis CA 95617 email: net-snmp-coders@lists.sourceforge.net

Description:

Clarify behaviour of objects in the memory & systemStats groups (including updated versions of malnamed mem\*Text objects). Define suitable TCs to describe error reporting/fix behaviour.

OID Mappings:

Name	OID	Type
<input type="checkbox"/> <a href="#">bsdi</a>	1.3.6.1.4.1.2021.250.11	node
<input type="checkbox"/> <a href="#">dskAvail</a>	1.3.6.1.4.1.2021.9.1.7	column
<input type="checkbox"/> <a href="#">dskDevice</a>	1.3.6.1.4.1.2021.9.1.3	column
<input type="checkbox"/> <a href="#">dskEntry</a>	1.3.6.1.4.1.2021.9.1	row


We learn some details about our MIB including contact and description information. In our example, we learn that UCD-SNMP-MIB addresses memory and system status, and it's maintained by the NET-SNMP project.

The list of OID Mappings is paginated, and we find memTotalFree on the second page. Click on it to display the properties for the OID:

Mibs > UCD-SNMP-MIB > memTotalFree	
Mib Node	
Name	UCD-SNMP-MIB::memTotalFree
OID	1.3.6.1.4.1.2021.4.11
Access	readonly
Node Type	scalar
Status	current
Description	The total amount of memory free or available for use on this host. This value typically covers both real memory and swap space or virtual memory.

On the properties page for **memTotalFree**, we learn some additional details, such as the **Access** type, **Node Type**, **Status**, and **Description**. So now we know, in detail, what we are collecting. The chances are good, however, that you know most of this information before you import the MIB.

If we go back one page to the **UCD-SNMP-MIB** page, we see a **Traps** table at the bottom of the page:

Traps 		
Name	OID	Type
<input type="checkbox"/> <a href="#">ucdShutdown</a>	1.3.6.1.4.1.2021.251.2	notification
<input type="checkbox"/> <a href="#">ucdStart</a>	1.3.6.1.4.1.2021.251.1	notification

We'll cover traps in more detail in *Chapter 7, Collect Events*, but traps allow the monitored devices to report events back to Zenoss Core. Then Zenoss Core can turn the received trap into an event.



We've walked through how we import a MIB, and as we've seen the MIB contains OIDs that we can use as our SNMP data sources in our monitoring templates. Even though Zenoss Core includes many MIBs, hardware manufacturers release MIBs that retrieve device-specific information. If you start seeing events with OIDs instead of a friendlier name, then you know you need a MIB.



You can find MIBs from the manufacturers, directory sites, such as [www.mibdepot.com](http://www.mibdepot.com), or by performing an Internet search.

Have you had enough MIB browsing for now? Let's shift gears and set up a monitoring template using a Nagios plugin. In the next section we will explore command data sources, thresholds, and graphs.

## Monitoring with Nagios plugins

With our exploration of SNMP-based templates behind us, let's add a demonstration plugin to Zenoss Core that randomly generates a range of performance data. We'll create a new device class called Demo with its own device template to collect, generate events, and graph data from the `bogo_check.py` command.



First, let's give some credit. The `bogo_check` plugin we'll be using was submitted to the Zenoss bug tracking database as bug number 2031 by Kells Kearney. I found out about Kells' plugin while he reviewed the first version of this book, but it was too late to incorporate it. The ticket is still open and there's some discussion on the ticket about making this into a ZenPack. So, we're going to bring this plugin out into the light of the day. See <http://dev.zenoss.com/trac/ticket/2031> for more information.

When we're finished with this chapter, we will be monitoring a fictitious device with a plugin that generates random data. You might be inclined to ask why we'd want to do such a thing, and that's a fair question. Generally speaking, we're not likely to match monitoring environments, so this gives us a rare chance to have a similar environment, and we'll turn this plugin into a ZenPack in *Chapter 9, Extend Zenoss Core with ZenPacks*. The demo environment we're about to create can also become a playground to explore templates, command data sources, and performance graphs via RRDtool.

Step one is to obtain the `bogo_check` plugin file. I'd recommend you download the code files for this book to obtain the plugin file. To do so, visit <http://www.packtpub.com/support>, select this book from the **Title** drop-down list, and click on **Go**. However, you can also extract the code from the attachment at <http://dev.zenoss.com/trac/ticket/2031>.

**Zenoss™** | Open Source Enterprise Monitoring

[Login](#) | [Help/Guide](#) | [About Trac](#) | [Preferences](#)

[Wiki](#) | [Timeline](#) | [Roadmap](#) | [Browse Source](#) | [View Tickets](#) | [New Testcase](#) | [Search](#)

[Back to Ticket #2031](#)

### [Ticket #2031: bogo\\_check.py](#)

**File `bogo_check.py`, 3.4 kB (added by kpg123, 2 years ago)**

Nagios-style plugin capable of data generation for demonstration purposes

Line	
1	<code>#!/usr/bin/python</code>
2	<code>"""This check is a test for creating sine-wave or random performance</code>
3	<code>data (percentages) suitable for testing out graphing etc.</code>
4	<code>"""</code>
5	<code>The intent is that it be run at 'demo-speed' (ie once per minute)</code>
6	<code>"""</code>
7	
8	<code>from optparse import OptionParser</code>
9	<code>import os, sys, random</code>
10	<code>import time</code>
11	<code>from math import sin, radians</code>
12	<code>import gettext</code>
13	
14	<code>#</code>
15	<code># Nagios return codes</code>
16	<code>#</code>
17	<code>nagios_ok= 0</code>

Let's install `bogo_check.py`. If you work as the `zenoss` user, you'll save yourself some permission problems:

1. If you downloaded the code from Trac, copy all 130 lines of the script and paste it into a text file named `bogo_check.py`.
2. Copy the file to `$ZENHOME/zenoss/Extensions`. Then navigate to the `Extensions` directory.



`$ZENHOME` represents the installation path of Zenoss Core. It's `/usr/local/zenoss` by default.

3. If you're not working as the zenoss user, you'll need to copy the file as a superuser and change file ownership to the zenoss user with the command:  

```
chown zenoss bogo_check.py
```
4. Make the script executable with the following command:  

```
chmod +x bogocheck.py
```
5. If you downloaded the `bogo_check.py` file from the Trac ticket, you'll need to change the first line of the script from `#!/usr/bin/python` to `#!/usr/bin/env python` because Zenoss Core installs its own version of Python. The zenoss user's environment is configured to use the version of python that is packaged with Zenoss Core. Zenoss Core packages Python 2.6 in `$ZENHOME/python`.

We're now ready to test our command, which we'll do from our terminal window. As the zenoss user, run the following command several times from the `$ZENHOME/Extensions` directory:

```
./bogo_check.py
```

Each time you execute the command, it returns a different result that resembles the following output:

```
bogo_check OK datapoint 58 | bogopoint=58%;70;90;0;100
```

The OK in the results will randomly switch between OK, WARN, and CRITICAL based on the datapoint value, which in my sample output is 58. The values 70 and 90 represent the default thresholds defined in the `bogo_check` plugin. So, if your datapoint returns a 70 or higher, the plugin will trigger an event with a severity of warning. If the datapoint is 90 or higher, the plugin generates an event with a severity of critical.

The `bogo_check` plugin supports runtime options. For example, we can change the warning event threshold at runtime with the `--warn` option. See the following screenshot:

```
zenoss@fox:/usr/local/zenoss/zenoss/Extensions$ ./bogo_check.py --warn=35
bogo_check WARN datapoint 43 | bogopoint=43%;35;90;0;100
```

We need to specify the new warning threshold value with an integer, which is the `--warn=35` part the command in the previous screenshot. To find all the available options for the `bogo_check` plugin, run it with the `--help` option.

The `bogo_check` plugin returns data in a Nagios plugin compliant way, which Zenoss Core inherently understands.

## Working with Nagios plugins

Because Zenoss Core can understand the data returned by Nagios plugins, we have a wide variety of command data sources available for us to plug into our device templates. Likewise, if you need to write a custom plugin to pull data, you can write it according to Nagios standards.



You can find Nagios plugins in a variety of places including Nagios Exchange at <http://exchange.nagios.org/> and the official Nagios Plugins at <http://nagiosplugins.org/>.

Under the hood, Zenoss Core uses a command called `zencommand` to run the plugin against a device. The important piece for now is that `zencommand` expects the output of the plugin to conform to Nagios conventions in two important ways: return code and performance data.

## Nagios return codes

The following table shows the Nagios return codes:

Return Code	Description
0	Status is OK, which means the device is available and does not violate defined thresholds.
1	Status is WARN, which means the device is available but is crossing a predetermined warning threshold.
2	Status is CRITICAL, which means the device may not be available. Or the device is available but is performing at a threshold deemed critical.
3	Status is UNKNOWN, which implies the plugin failed in some way or was unable to connect to the device.

The `bogo_check` plugin we're using does exit with one of these four return codes, based on the random datapoint it generates.

## Nagios performance data

Returning the status of the monitored component is a good start, but we often want to collect the actual performance data so we can graph it and manipulate it inside Zenoss Core.

The performance data can be found after the | in the plugin output and is represented in the following format:

```
label=value;[warn];[crit];[min];[max]
```

Does that look familiar? Let's look at the relevant part from our `bogo_check` plugin output:

```
bogopoint=58%;70;90;0;100
```

The label is `bogopoint` and it reports a value (58) that uses a percentage as the unit of measure. The unit of measure could also be none, seconds, bytes, and a counter.

We've identified the threshold values earlier (70 and 90), but now we know that the 0 and the 100 represent the minimum and maximum values returned by the plugin.

For more information about developing Nagios compliant plugins that you can use with Zenoss Core, refer to the Nagios Developers Guidelines at <http://nagiosplug.sourceforge.net/developer-guidelines>. Remember one fine point, however. Zenoss Core is not Nagios and is not built upon Nagios.

## Adding the Nagios plugin to Monitoring Templates

OK, we're ready to monitor with `bogo_check.py`. Since this is really a demo plugin, we want to make sure we keep it separate. Therefore, we'll add a special device class, add a new monitoring template to bind to the class, and then add a device to monitor. Even though we're working with a fictitious device, this is a common and powerful capability that will provide a near endless amount of Zenoss Core customizations.

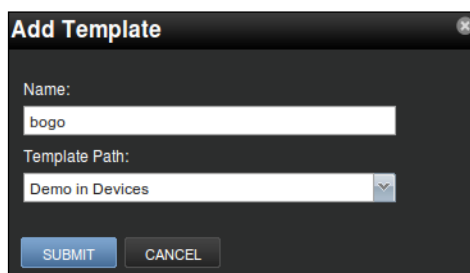
Let's get back to the Zenoss Core web interface to add the device class:

1. From the **Infrastructure** menu in Zenoss Core, click on **Devices** to display the **Device Classes**.
2. To add the Demo class, select the top level organizer, **Device Classes**. Then click on the **Add a child to the selected organizer** button. The **Add Device Class** dialog box displays.
3. In the **Name** field, enter **Demo**. The **Description** is optional.
4. In the **Add Organizer** dialog box, enter **Demo**.
5. Click on **SUBMIT** to add the device class.

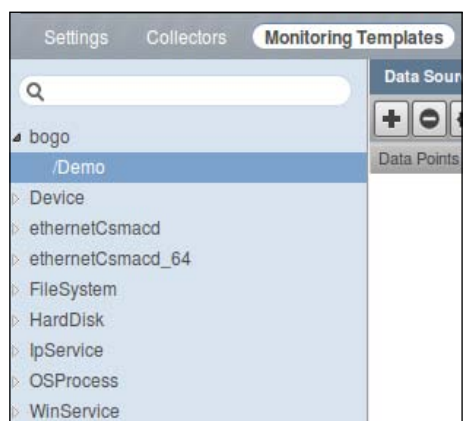


Let's add a new monitoring template and data source:

1. From the **Advanced** menu, select **Monitoring Templates** to display a list of templates. Click the **Add monitoring template** button.
2. Enter a descriptive **Name** in the **Add Template** dialog box. I'll use **bogo** in my example.
3. For the **Template Path**, select **Demo in Devices**. The **Demo** corresponds to the Demo device class we created.



4. Click on **SUBMIT** to add the **bogo** template to the list.

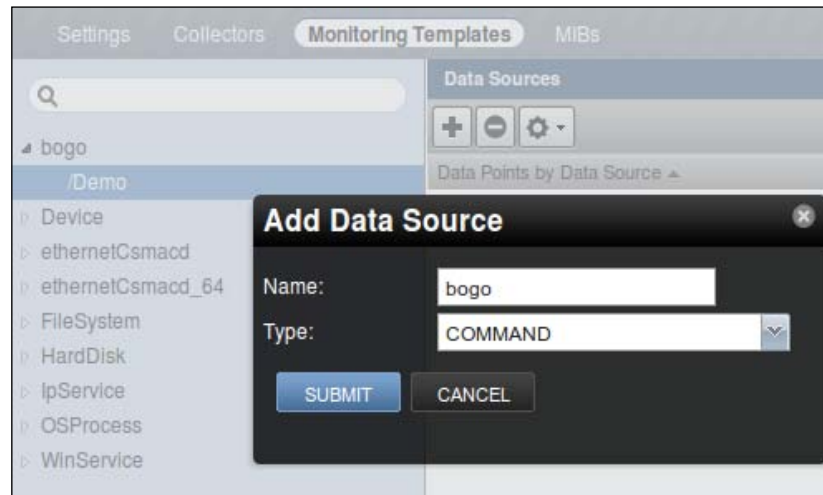


5. As you can see in the previous screenshot, there is a new template organizer named **bogo** that contains a template named **Demo**.

## Adding a Data Source

So far, so good. We have our class and associated template created. Let's configure the data sources for our template:

1. From the **Monitoring Templates** page, select the **Demo** template. Then, from the **Data Sources** menu, choose **Add DataSource**.
2. In the **Add a New DataSource** dialog box enter a descriptive **Name (bogo)**, and select **COMMAND** as the **Type**.



3. Click on **SUBMIT** to add the data source.
4. Now we need to configure the new data source. Select **bogo** from the Data Sources table. To display the **Edit Data Source** screen, select **View and Edit Details** from the **Data source edit options** menu.
5. In the **Command Template** field, enter the full path to the bogo plugin: `/usr/local/zenoss/zenoss/Extensions/bogo_check.py`.
6. Click on **SAVE** when finished.

**Edit Data Source**

Name:  Type:  Enabled: ☒ true

☒ Enabled Severity:

Event Class:  Cycle Time (seconds):

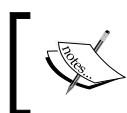
Parser:  ☐ Use SSH

Component:  Event Key:

Command Template:

**Test Against a Device**

Device Name:



We must specify the full command path for our plugin, because Zenoss Core will prefix a relative command with the path value from the `zCommandPath` configuration property.

At some point in the future, you may want to change the properties for a command data source. The following table describes each Data Source property:

Property	Description
Name	The friendly name of the data source.
Source Type	Identifies the data source type. You can't change this value.
Enabled	Select true to monitor the data source. False to not monitor it.
Use SSH	If this value is set to True, then Zenoss Core will try to run the command on the host over SSH.
Component	Specify a component name for this data.



Property	Description
Event Class	Determines how Zenoss Core classifies events generated against this command by selecting a predefined device class.
Severity	Sets the default severity for events generated with the device. Can be overridden in other places.
Cycle Time	The default is 300 seconds (5 minutes). Sets the interval that Zenoss Core collects data for the data source.
Parser	Tells Zenoss Core how to parse the data that is returned from the plugin. The default is Auto, but Cacti and Nagios are options.

Based on our review of the available options and considering this plugin is used for demonstration, I will change my **Cycle Time** to **60**, so we poll data every 60 seconds. Feel free to do the same.

## Adding a Data Point

Next we need to enter a data point to the bogo data source. The terminology can be a challenge in Zenoss Core. The data point is the output from the data source, and the data source is the command.

Let's add the data point:

1. From the **Data Sources** menu, select the **bogo data source**.
2. From the **Data source edit options** menu, select **Add Data Point**.
3. In the **Add Data Point** dialog box enter the name of our plugin data point.



The data point name must correspond to the label listed in the plugin's output. For the `bogo_check.py`, the appropriate data point name is **bogopoint**.

4. Click the **Add** button to save the changes. Now, bogopoint is listed as a sub-item of the bogo data source. See the following screenshot:

Data Sources			
<div> <div>+</div> <div>−</div> <div>⚙</div> </div>			
Data Points by Data Source ▲	Source	Enabled	Type
▲ bogo	/usr/local/zenoss/zenoss/Extensions/bogo_check.py	true	COMMAND
bogopoint			GAUGE

5. To configure the data point, select **bogopoint**. Then select **View and Edit Details** from the **Edit data source options** menu.
6. In the **Edit Data Point** screen, set the following properties:
  - **Type**: Gauge.
  - **RRD Minimum**: 0.
  - **RRD Maximum**: 100.
7. Click on **SAVE** and see the following screenshot:

**Edit Data Point**

Name: bogopoint

RRD Type: GAUGE

Create Command:

RRD Minimum: 0

RRD Maximum: 100

☒ Read Only

Alias:

ID / FORMULA

DELETE

SAVE CANCEL

Before we proceed, let's take a small diversion and discuss the RRDtool values we have available to us.

## RRDtool Data Point configurations

In the Data Point properties, we can specify the **Type** of data we're measuring, and the RRD minimum and maximum values. The **RRD Minimum** field specifies the minimum value we expect to get back from our plugin, while the **RRD Maximum** field specifies the maximum value we expect to get back from our plugin.

Obviously, you'll need to understand the device or service you're monitoring to know what data range you should specify. Because our `bogo_check` plugin uses a percentage, a minimum value of 0 and a maximum value of 100 makes sense.

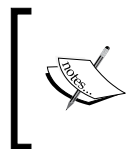
As for the **Type**, the following table shows a list of the RRDtool types and when we should use each type.

Type	Description
Gauge	Displays the value at the time the plugin collects the data. This is analogous to a gas gauge.
Counter	Measures the rate of change for the data point by subtracting the previous value from the current value. Assumes a continual increase in value over each polling cycle.
Derived	Same as counter; however it will also show the rate of change for a negative value, such as the available disk space.
Absolute	Measures the rate of change divided by the cycle time. The previous measurement is assumed to be 0.

You can learn more about RRDtool at <http://oss.oetiker.ch/rrdtool/>.

## Defining monitoring thresholds

Now that we have our data source and associated data point defined, we can set up our thresholds. The thresholds will generate events based on the values returned by the plugin.



We can only set a threshold for a defined data source. However, not all data sources need to have a threshold. Whether you add a threshold is really a question of whether or not you want to be notified when the monitored value exceeds a particular value.

With the Bogo (/Demo) monitoring template selected, let's add two thresholds that correspond to the data the `bogo_check` plugin already reports:

1. From the **Thresholds** table, select **Add Threshold**.
2. In the **Add Threshold** dialog box, enter a descriptive name in the **Name** field, such as **Warning**. For **Type** select **MinMaxThreshold**.
3. Click **OK** to add the threshold.
4. Now we need to configure the threshold. Select **Warning** from the thresholds list, and then click the **View and edit threshold details** button. The **Edit Thresholds** window displays.

5. For this threshold, we want to be notified if the data point exceeds 70%, so enter **70** in the **Max Value** field.
6. Change the **Event Class** to **/Cmd/Fail** from the default **/Perf/SNMP**. We'll talk more about event classes in *Chapter 6, Core Event Management*. However, this is a command data type; it makes sense to change the default SNMP class.
7. All other values can remain at default. Click on **SAVE**.

**Edit Threshold**

Name: Warning

Type: MinMaxThreshold

DataPoints: Available Selected

Severity: Warning

☒ Enabled

Minimum Value:

Maximum Value: 70

Event Class: /Cmd/Fail

Escalate Count: 0

There's one thing I glossed over as we configured the threshold. The Edit Threshold page provides a selection box of available and selected data points. As you can see in our example, we can only select data points that are configured for the template we're working in. In our example, we only have one data point.



The Escalate Count field, shown in the previous screenshot, specifies the number of times the threshold can be crossed before the event severity is increased to the next level.

Use the same procedure to add a critical threshold with a Max value of 90 and a severity of Critical. Our list of thresholds for the bogo template should look like the following screenshot:



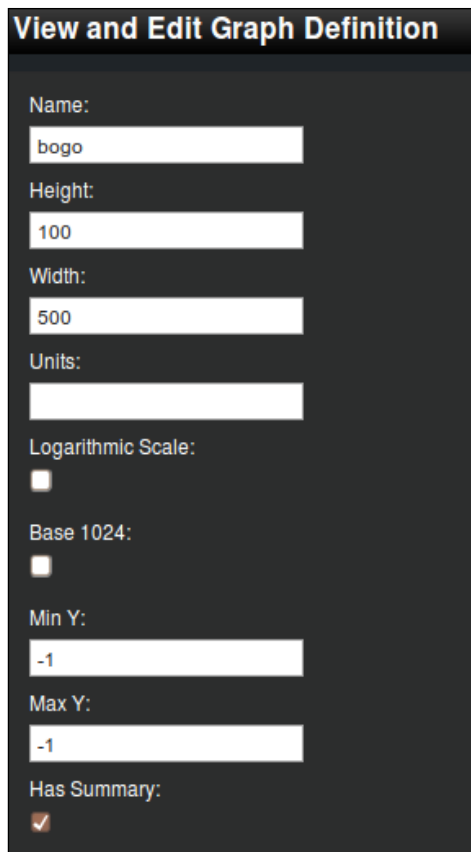
By defining the data source and the threshold, we have the necessary information to collect information about a device and be notified of problems. For some of us, that's enough. However, Zenoss Core provides us the ability to visualize our data over time in the form of graphs.

So, let's complete our template and set up some graphs.

## Graph definitions

We're almost finished. Our final step is to represent our data with a graph. Let's get right to it:

1. From the **Graphs** table, select **Add Graph**.
2. Enter a descriptive name in the **Name** field of the **Graph**, such as bogo. Click on **SUBMIT** to create an item in the **Graph Definitions** list.
3. To configure the graph, select bogo and then select **View and edit details** from the **Manage graph definitions** button.
4. From the **View and Edit Graph Definition** window, specify the % as the **Units**. The other values should be fine as defaults. Click on **SUBMIT** to save it.

A screenshot of a 'View and Edit Graph Definition' dialog box. The dialog has a dark background with white text and input fields. It contains the following fields: 'Name:' with the value 'bogo', 'Height:' with the value '100', 'Width:' with the value '500', 'Units:' (empty), 'Logarithmic Scale:' with an unchecked checkbox, 'Base 1024:' with an unchecked checkbox, 'Min Y:' with the value '-1', 'Max Y:' with the value '-1', and 'Has Summary:' with a checked checkbox.

**View and Edit Graph Definition**

Name:  
bogo

Height:  
100

Width:  
500

Units:

Logarithmic Scale:  
☐

Base 1024:  
☐

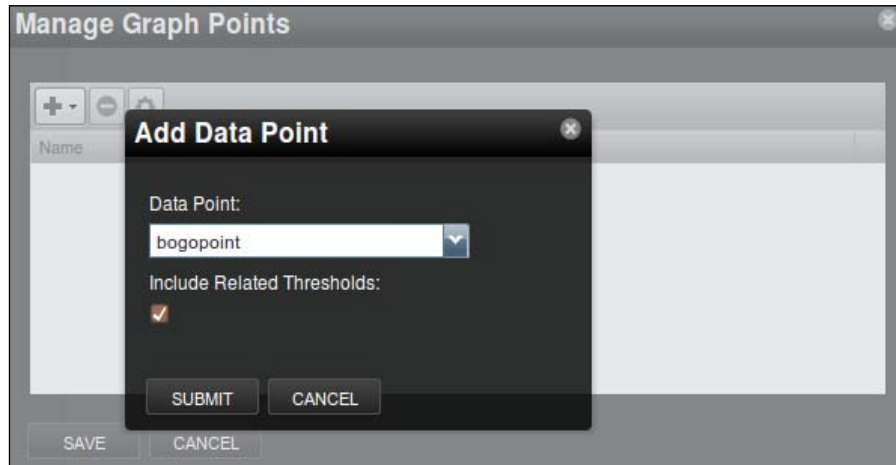
Min Y:  
-1

Max Y:  
-1

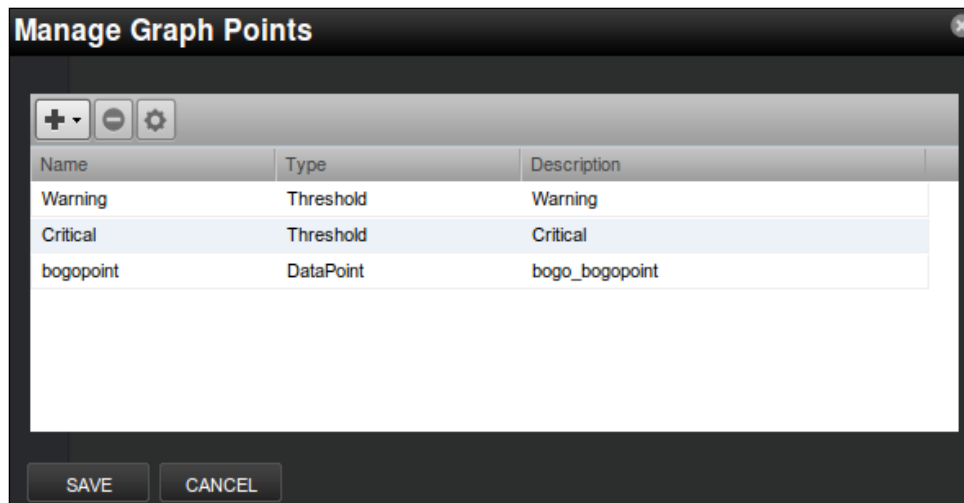
Has Summary:  
☒

5. Next we need to specify the points on the graph. With the bogo graph definition selected, click **Manage graph points** from the **Manage graph definitions** button. The Manage Graph Points dialog box appears.
6. From the **Add graph point** menu, click **Data Point** to display the **Add Data Point** dialog box.

7. Select **bogopoint** from the **Data Point** drop-down list:



8. Check the box for **Include Related Thresholds**.
9. Click on **SUBMIT** to add the data point and both thresholds. See the following screenshot:



Save all changes. Our next step in this process is to add a device that we can monitor with our new bogo template. Before we do that, let's pause for a moment and review some of the advanced RRDtool operations we can do to each data point on our graph.

## RRDtool Graph Point configurations

To configure a graph, click on its name. If our plugin graphs data exponentially, set the **Logarithmic Scale** value to true. To graph data that is measured in multiples of 1024, set **Base 1024** to true. We can change the way the values get displayed on the y-axis by specifying a **Min** or **Max Y** value.



For more information on RRD graph commands, see [http://oss.oetiker.ch/rrdtool/doc/rrdgraph\\_data.en.html](http://oss.oetiker.ch/rrdtool/doc/rrdgraph_data.en.html).

Obviously, we have a lot of flexibility with how we can display the data we're collecting and the more RRD tool knowledge you have, the more you'll be able to manipulate your data.

You can edit the properties of the graph from the **Manage Graph Points** dialog box:

1. To get to the **Manage Graph Points** dialog box, select the graph.
2. Select **Manage Graph Points** from the **Manage Graph Definition** menu.
3. Next, select the bogopoint data point and then click on **View and edit graph point details**.



See the following screenshot:

**Edit Graph Point**

Name: bogopoint

Type: DataPoint

DataPoint: bogo\_bogopoint

Line Type: Line

Line Width: 1

☐ Stacked

Format: %5.2lf%s

RPN:

Limit: -1

Consolidation: AVERAGE

Color (Hex value RRGGBB):

Legend: \${graphPoint/id}

Available RRD Variables: None

SAVE CANCEL

The following table shows the options available in the **Edit Graph Point** screen:

Property	Description
<b>Name</b>	Name of the graph point.
<b>Type</b>	This will be the data point and is not changeable.
<b>DataPoint</b>	A read only view of the data point name.
<b>Consolidation</b>	Set to average by default. Other options include minimum, maximum, total, and last. This setting determines how RRDtool consolidates data over a period of time so that it doesn't have to save minute-by-minute records. This helps save disk space.

Property	Description
<b>RPN</b>	This field accepts commands in Reverse Polish Notation (RPN) to evaluate the collected data. For example, you could convert bits to bytes, seconds to milliseconds, calculate utilizations, and a whole lot more. Accepts TALEs expressions.
<b>Limit</b>	The default is -1. Any data that exceeds the specified limit will not be used.
<b>Line Type</b>	Select how the graph should be drawn. The available options are Line, Area, and Not Drawn. Line will do as it says and use a line to graph the data. Area will fill the area from the horizontal axis to the graph line using the specified color, which is a separate configuration. The Not Drawn value will not create a graph line.
<b>Line Width</b>	Specify the width of the graph line in pixels. The default is 1.
<b>Stacked</b>	Select true to stack the data point on top of another data point on the graph.
<b>Color</b>	Displays the data point in the specified color when the graph is displayed. Specify the hex value in RRGGBB.
<b>Format</b>	Determines how the data on the graph is displayed using the print option to the <code>rrdgraph_graph</code> command of RRDtool.
<b>Legend</b>	Creates a legend on the graph for the data point. The default value is a TALEs expression that evaluates to the name of the data point: <code>\${graphPoint/id}</code> .
<b>Available RRD Variables</b>	An automatically generated list of variables that can be used in the RPN field.



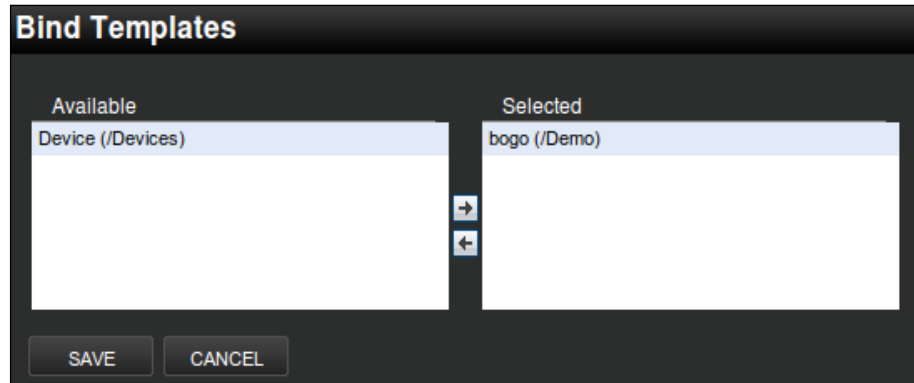
For more information regarding RRDtool, check out the project's home page at <http://oss.oetiker.ch/rrdtool/>.

## Binding templates to the device class

We want our Demo device class to use the bogo template we just created. To ensure that happens, we bind the template to the device class:

1. From the **Devices** page, select the **Demo** device class.
2. From the **Actions** menu, select **Bind Templates**.

3. From the **Bind Templates** dialog box, move **bogo** from **Available** to **Selected** and move **Device** from **Selected** to **Available**:



4. Click on **SAVE** to preserve the binding and return to the list of available templates.

Now, any device we add to the Demo class, will inherit the monitoring properties we defined in the bogo template. We unselected the Device template because that template only monitors system uptime by default, which will not make sense on our fictitious device.

Let's add a device now.

## Adding a device to monitor using the Bogo template

We're left with the easy part. We need to add a device to Zenoss Core and add it to the /Demo class. Click on the **Add Single Device** menu item from the Devices page and specify the following properties:

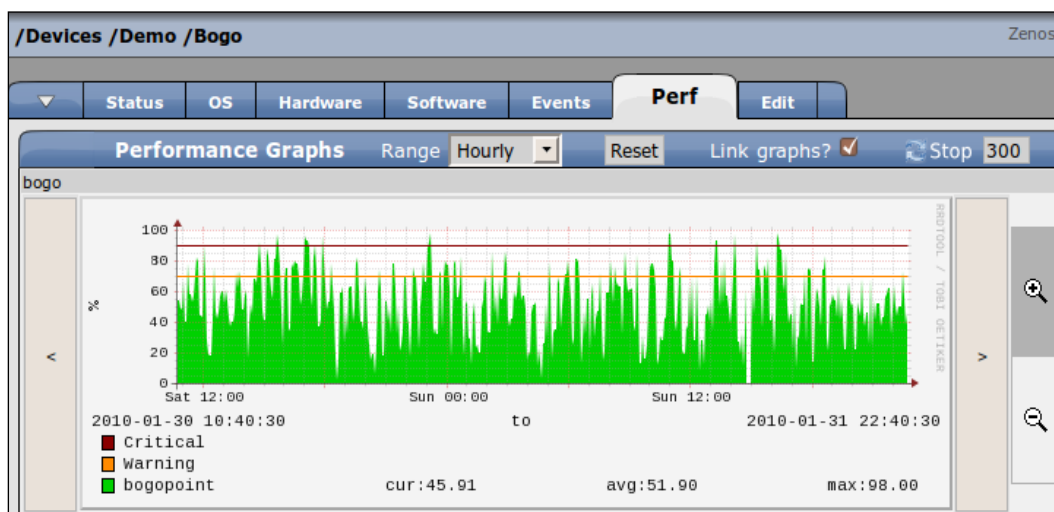
- **Device Name:** Make up a name; I'm using Bogo, of course.
- **Device Class Path:** /Demo.

Set any of the other properties as you see fit and then click on **Add Device**.

Unless you really have a device on your network named Bogo, we just added a device with no IP address. And Zenoss Core is going to monitor it with our new command template that generates random data points.

Cool, huh?

It will take a couple of monitoring intervals to actually get the data to display on the graph for the Bogo device. However, once it does, this is what you'll see:



The screenshot shows an hourly view of the data and the horizontal lines on the graph represent the thresholds at 90 % and 70 %. Each time the graph exceeded the threshold lines, Zenoss Core generated an event.

## Monitoring with Cacti plugins

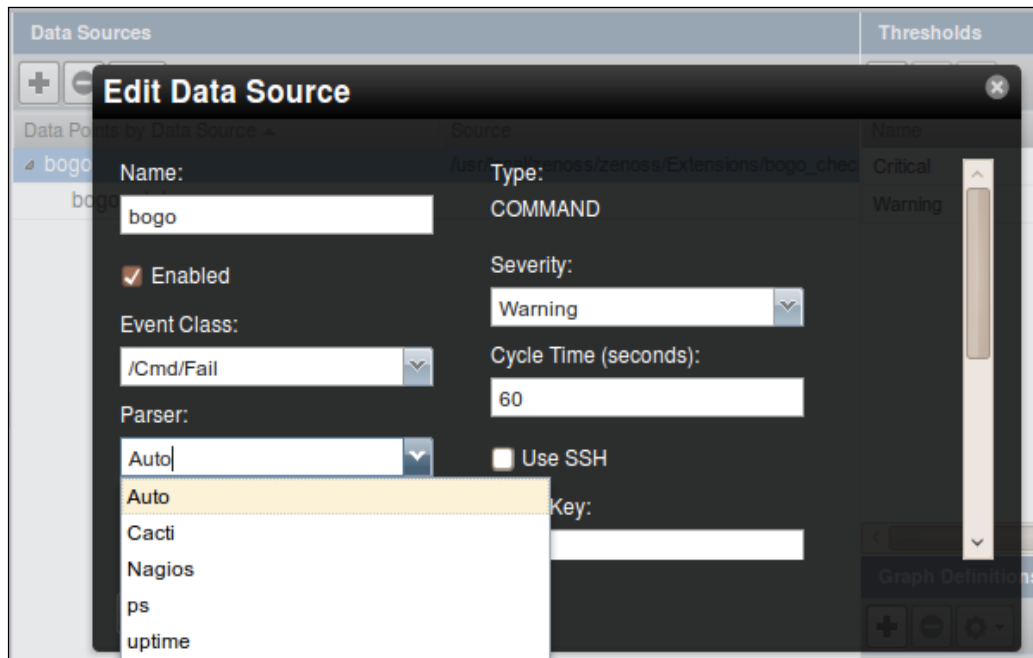
The Zenoss Core documentation is rampant with references that you can use a Cacti plugin or Cacti compatible plugin, but there do not appear to be any readily available examples.

To help us find an example that you can use to as a guide to creating your own plugin, we once again turn our attention to the Zenoss bug tracker to review ticket #4945 ([http://dev.zenoss.com/trac/attachment/ticket/4945/single\\_dp\\_cacti\\_plugin.py](http://dev.zenoss.com/trac/attachment/ticket/4945/single_dp_cacti_plugin.py)). This script is similar in function to the `bogo_check.py` plugin we just installed and is authored by Kells Kearney. This Cacti compatible plugin generates random data in a single datapoint value, which the Zenoss Core Cacti parser understands.

Everything we just did to add and configure a monitoring template can be done with a Cacti compatible plugin. Just feed the data the location of the script and you're on your way.

## Data Source parser

For each command-type data source, you have the option to set a parser. As you can see in the following screenshot, the choices are **Auto**, **Cacti**, **Nagios**, **ps**, and **uptime**. The default value is **Auto** and it's what we selected when we added the `bogo_check.py` plugin.



Cacti and Nagios obviously expect the output of a command to be in a specific format, and this chapter provides sample scripts for both. The `ps` and `uptime` values parse the output from their respective Unix commands.

It's helpful to know what your options are, but you probably won't have to change the parser type unless you're troubleshooting plugin problems.

## Summary

We learned how to extend our Zenoss Core monitoring environment with monitoring templates and collect performance data via SNMP and command data sources, including Nagios and Cacti plugins. In the process, we discovered that Zenoss Core leverages the ubiquitous RRDtool to graph time-series data. There's a lot to digest with monitoring templates and modifying your templates provides one of the most powerful customizations in Zenoss Core.

Up to this point we have enough information to discover, monitor, and tune the monitoring properties of our devices. So, in the next two chapters, we focus on events. All that monitoring we're doing, including our dummy bogo device, is generating events in Zenoss Core. It's time we understand what that means.



# 6

## Core Event Management

We've spent a considerable amount of time defining what devices, services, and processes we want to monitor and how we want to monitor them. All that work is for nothing if our monitoring system doesn't have a mechanism to let us know when problems occur. In Zenoss Core, events identify when something happens to one of the devices we're monitoring.

In this chapter, we will:

- Use the Event Console to process events
- Create event commands through the Event Manager
- Test events without affecting the actual device
- Map events based on monitoring activity

One word of caution before we begin: Don't confuse events with alerts. In Zenoss Core, alerts notify us, the human, when an event occurs. We'll configure alerting rules in *Chapter 7, Collecting Events*.

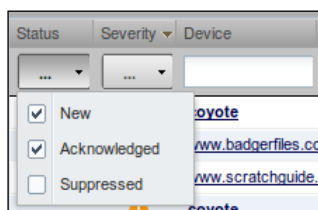


## Event Console

The **Event Console**, available from the **Events** menu, provides us with a single view of all the current events in the system. As the following screenshot depicts, we are presented with a lot of information about each event: **Status**, **Severity**, **Device**, **Event Class**, **Summary**, **First Seen**, **Last Seen**, and **Count**. And, naturally, we can change the information that is displayed, which we will cover later in this chapter when we talk about the Event Manager.

Status	Severity	Device	Component	Event Class	Summary	First Seen	Last Seen	Count
		192.168.1.0		/Status/Ping	192.168.1.0 sendto error [Erno 13] P	2010-09-10 22:53:27	2010-10-14 21:41:04	4303
		coyote	exim4	/Unknown	ALERT: exim paniclog /var/log/exim	2010-08-01 06:30:34	2010-10-14 06:29:28	49
		coyote	sshd	/Unknown	fatal: Timeout before authentication	2010-09-11 09:28:33	2010-09-17 21:28:20	14
		localhost		/Status/OSProcs	Device localhost does not publish H	2010-08-03 21:40:08	2010-10-14 21:45:38	34628
		192.168.1.1	snmp	/Status/Snmp	SNMP agent down	2010-09-10 22:55:52	2010-10-14 21:41:12	9776
		192.168.1.1	zenmodeler	/Cmd/Fail	User timeout caused connection fail	2010-09-10 22:52:45	2010-10-14 21:26:35	70

By default, the **Event Console** displays events with a minimum severity of information and minimum status of acknowledged. There are filters for **Status**, **Severity**, **Device**, **Component**, **Event Class**, **Summary**, **First Seen**, **Last Seen**, and **Count** that will expand or restrict the devices that display. For example, to filter by Severity or Status, select a new value from the drop-down list, and either check or uncheck a value. The view will automatically be updated:

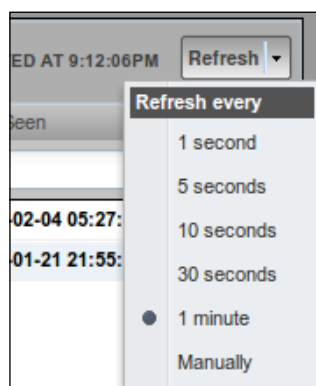


You can filter the list of events by typing a value in any of the column's text fields to filter the entire view based on the column. The following screenshot shows the events filtered by device:

Status	Severity	Device	Component
		coyote	exim4
		coyote	process

The **Event Console** automatically refreshes every minute, and the refresh will adhere to any filters you specify. So, if you filter the list for a specific device and a new event comes in for a different device, it won't show in the list until you remove the filter.

To change the refresh interval, select a new value from the **Refresh** drop-down list, as shown in the following screenshot:



If you don't like the order of the columns on the Event Console, you can click and drag the columns around to change the order.

## Event severities defined

The following table illustrates the available severities from the least severe to the most severe. There are no hard and fast rules to dictate how we apply event severities to our devices or monitoring environment. Zenoss Core makes some assumptions about an events default severity, but those assumptions are easily changed via the event class configurations, as well as at the device level.

Event severity	Integer value	Description
Clear	0	Correlates to a previous down event and moves the event to history. Represented by a green icon.
Debug	1	Used for troubleshooting. Does not indicate a problem. Represented by a gray icon.
Info	2	Used to mark an event in the system for informational purposes. Represented by a blue icon.
Warning	3	Indicates a potential problem. Represented by a yellow icon.
Error	4	The device or component is unavailable or is operating at dangerous performance levels. Represented by an orange icon.
Critical	5	The device or component is down. Represented by a red icon.

In the previous chapter we defined monitoring thresholds for our templates and one of the settings available to us was event severity. Anywhere we have an event, we have a severity.

Events become a key filter when we set up alerting rules, which we will do in *Chapter 7, Collecting Events*.

## Event statuses defined

An event status categorizes the current working status of an event. The following table lists each event status with a brief description of the state:

Event status	Integer value	Description
Suppressed	2	An event occurred, but it was sent directly to history.
Acknowledged	1	The event is still active and is being worked on by an admin.
Unacknowledged	0	Represents a new event that has not been acknowledged and is presumably not being worked on.

Zenoss Core provides several places for us to manipulate the status of an event, including event classes and event transformations.

## Acknowledging an event

Acknowledging an event signals to other team members and to Zenoss Core that you are aware of the event and, presumably, taking action. Acknowledging the event is good communication among your team, but Zenoss Core can also escalate event severities or alerts based on an event status.

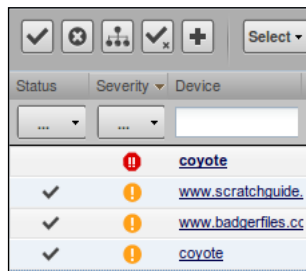
A common way to escalate an event or an alert is by the event count. For example, we can instruct Zenoss Core to escalate the event severity from Error to Critical if the event hasn't been acknowledged after a specified number of monitoring cycles. Or if we're dealing with alerts, Zenoss Core can be configured to alert the next person on call, in the event you fall asleep on the roof at 3 in the morning and don't realize the database server has been down for 15 minutes.

Make sense?

To acknowledge an event:

1. Select the event from the **Event Console** by clicking on it.
2. Click the on **Acknowledge Events** button (the icon that looks like a check mark).

A check mark will appear next to the event. See the following screenshot:



In the screenshot all the events are acknowledged, except the first one.

## Viewing an event log

Now that we know how to acknowledge an event, we should figure out how to view its details. That way we can get on to fixing the problem. The Event Console shows us a summary of information. However, there is a wealth of diagnostic data attached to each event, and we can view it by double clicking on an individual event.

Previous versions of Zenoss Core had a View Log link for each event, but in version 3.0, the only way to view the event log is to double click the event. Since it's really not intuitive to double-click in a web application, it bears some emphasis.

Status	Severity	Device	Component	Event Class
!!		coyote	exim4	/Unknowr
✓	!	www.scratchguide.		/Cmd/Fail
✓	!	www.badgerfiles.cc		/Cmd/Fail
✓	!	coyote	process	/Status/Sn
	!	localhost	gedit /home/mike/	/Status/O:
	!	fox	/usr/lib/gnome-se	/Status/O:
	!	Bogo2		/Perf/Snm
	!	localhost	memTotalFree	/Perf/Snm
	!	coyote	memTotalFree	/Perf/Snm

Error reading value for

Device: coyote

Component: memTotalFree

Event Class: /Perf/Snmp

Status: 0

Start Time: 2010/02/04 21:15:26.000

Stop Time: 2010/02/04 21:20:27.000

Count: 2

Show more details...

LOG

2010/02/04 16:44:07.000 Oh now. Not again. Somebody  
mike said: should trap this coyote.



In the event details, we have a place to log notes about the event. Each note is timestamped and associated with the logged in user, thereby creating documentation about the event.

To see more details about the event, click on the **Show more details** link. The event details window will now scroll vertically indicating there's a lot of information to be seen:

Hide details

prodState	1000
stateChange	2010/02/04 00:27:44.000
facility	mail
eventClassKey	exim4
agent	zensyslog
dedupid	coyote exim4   5 ALERT: exim paniclog /var/log/exim4 /paniclog has non-zero size, mail system possibly broken
manager	localhost
Location	
ownerid	
firstTime	2009/11/20 06:25:54.000
eventClass	/Unknown
message	ALERT: exim paniclog /var/log/exim4/paniclog has non-zero size, mail system possibly broken
DevicePriority	3
suppid	
monitor	localhost
priority	1
DeviceClass	/Server/Linux
eventState	0
evid	b257bb44-abb3-4dd1-a181-f64cec0376ba
eventClassMapping	
component	exim4
classid	

If Zenoss knows the value of a field, it's populated. However, there are some quirks. For example, many of the fields list the numeric representation of a value instead of the human friendly text descriptions. Also, the field names look more like variable names than labels.

But we're all smart people. We can cope with these interface nuances.

*Appendix A, Event Attributes* lists the event fields you see listed with a description of the field. We can also programmatically transform and access these event fields using Python code and TALES Expressions. We get to transformations in *Chapter 7, Collecting Events*.

## Events consoles are everywhere

If you browse the Zenoss Core interface long enough, it will seem like events spring up with every mouse click. Or maybe I'm just being dramatic.

You can find Event Consoles for each device, device class, and event class. The events are filtered based on the page you're looking at. So if you're viewing the `/Server/Linux` class, you will not see events for the devices in the `/Server/Windows` class.

## Closing events

If Zenoss Core monitors a device and finds that a previously detected error condition no longer exists, it automatically clears the event and moves it to History. When we create a test event later in this chapter, we will demonstrate clearing events; however, the *bogo* plugin we installed in *Chapter 5, Custom Monitoring Templates* is already automatically clearing events, which you can see in History.

You can also manually close an event, after you fix the problem that caused the initial error condition. Let's manually close an event.

1. Highlight the event from the **Event Console**.
2. Find the **Close selected events** icon and click on it. It's the button with the X inside a circle at the top of the Event Console. Closing an event has the same effect as when Zenoss Core clears the event.
3. The event disappears from the Event Console.

Don't worry, if the issue isn't really resolved, the event will reappear the next monitoring cycle.

## Displaying historical events

After we close an event, it gets moved to the Event History. You didn't think we'd just delete it, did you?

To view the historical events, click on **History** from the **Events** menu. If you installed and configured the `bogo_check` Nagios plugin with me in *Chapter 5, Custom Monitoring Templates*, then your Bogo device is generating lots of events, and you will have events in your history:

/Events / Event History

Zenoss server time: 22:33

Select

Export

Configure

LAST UPDATED AT 10:30:42PM

Refresh

Status	Severity	Device	Component	Event Class	Summary	First Seen	Last Seen	Count
		</						

The screenshot shows that the threshold exceeded events were automatically closed by subsequent threshold restored events.

## Event Manager

The **Event Manager** provides an interface that allows us to configure how events are stored, displayed, and acted on. We access the **Event Manager** from the **Events** menu:

**Zenoss CORE** DASHBOARD EVENTS INFRASTRUCTURE REPORTS ADVANCED mike

Event Console History Event Classes **Event Manager**

**ZenEventManager**

**Connection Information**

Backend Type: mysql

User Name: zenoss

Password: .....

Database: events

Hostname: localhost

Port: 3307

**Cache**

Cache Timeout: 20

Cache Clear Count: 20

History Cache Timeout: 300

History Cache Clear Count: 20

**Maintenance**

Event Aging Threshold (hours): 4

Don't Age This Severity and Above: Error

Delete Historical Events Older Than (days): 0

Default Availability Report (days): 7

Default Syslog Priority: 3

Save Changes

The **Edit** page displays when we first open the **Event Manager** and provides three configuration areas: **Connection Information**, **Cache**, and **Maintenance** (as shown in the previous screenshot). The following fields are available:

### 1. Connection Information

- **Backend Type:** Events are stored in a **MySQL** database.
- **User Name:** Database username. Default is **zenoss**.
- **Password:** Password for username. Default is **zenoss**.
- **Database:** Events database name. Default is **events**.
- **Hostname:** Database hostname. Default is **localhost**.
- **Port:** Database port number. Default is **3306**.



## 2. Cache

- **Cache Timeout:** Sets the event cache timeout in seconds. The lower the number, the more responsive the Event Console will be. The default is 20.
- **Cache Clear Count:** Sets a threshold to clear event cache counts. The default is 20.
- **History Cache Timeout:** Sets the history event cache timeout in seconds. The lower the number, the more responsive the history events views will be. The default is 300.
- **History Cache Clear Count:** Sets a threshold to clear history event cache counts. The default is 20.

## 3. Maintenance

- **Event Aging Threshold (hours):** If the event has not been acknowledged in the specified amount of time, move it to history. Default is **4** hours.
- **Don't Age This Severity and Above:** Events higher than the specified severity will not automatically go to history. Default is **error**.
- **Delete Historical Events Older than (days):** If you want to remove your event history, specify the number of days you want to keep event history. The default value is **0**, which means events are never deleted from history. And since it takes disk space to store these events in a MySQL database, you should consider removing event history, especially on larger installations.
- **Default Availability Report (days):** Specify the number of days to show data for the availability report. The default is **7**.
- **Default Syslog Priority:** Set the severity level for an event to generate an entry in the syslog. The default is **3**, which is an error.

The automatic event aging settings are interesting. If you check your event history, you will likely see lots of events with severities of information, clear, and maybe debug. These events show up on the Event Console, and then automatically clear after 4 hours, which is the default **Event Aging Threshold**. Clear events, go straight to history. We use alerting rules to determine what events we, the admins, receive notification of. This means that if informational events are important to you, you should set up an alerting rule so that you're notified when the event occurs.

## Event Fields

The **Fields** page of the **Event Manager** provides a way to add and remove fields from the **Event Consoles**. The page divides into two rows: **Default Result Fields** and **Device Result Fields**, as seen in the next screenshot:

The screenshot shows the **ZenEventManager** interface with the **Event Manager** tab selected. The left sidebar contains links for **Edit**, **Fields** (highlighted), **History Fields**, **Commands**, and **Modifications**. The main content area is titled **Event Fields** and includes the following elements:

- Default Sort:** A dropdown menu showing `severity desc, lastTime desc`.
- Default Result Fields:**
  - Available:** A list of fields including `dedupid`, `evid`, `eventKey`, `message`, `eventClassKey`, `eventGroup`, `stateChange`, `prodState`, `suppid`, and `mananer`.
  - Selected:** A list of fields including `eventState`, `severity`, `device`, `component`, `eventClass`, `summary`, `firstTime`, `lastTime`, and `count`. Arrows indicate movement between the available and selected lists.
- Device Result Fields:**
  - Available:** A list of fields including `dedupid`, `evid`, `device`, `eventKey`, `message`, `eventClassKey`, `eventGroup`, `stateChange`, `prodState`, and `sunnid`.
  - Selected:** A list of fields including `eventState`, `severity`, `component`, `eventClass`, `summary`, `firstTime`, `lastTime`, and `count`. Arrows indicate movement between the available and selected lists.
- SUBMIT** button at the bottom.

As the screenshot indicates, the page provides controls to move the fields from **Available** to **Selected** and vice versa. The up and down arrows sort the selected field in the list.

The fields assigned to the **Default Result Fields** display on the Event Console and the Events page of an Event Class. The fields assigned to the **Device Result Fields** display on the device's Events page.

You can change the fields of the event history via the **History Fields** link in the left sidebar. The fields assigned to the **Default Result Fields** display on the event class' **History** page, while the fields assigned to the **Device Result Fields** display on the device's **History** page.

Both the **Fields** and the **History Fields** pages provide a **Default Sort** field to control how the data is sorted. The default sort order for the **Fields** tab is descending by **severity**, and then descending by **lastTime**.



The screenshot shows the ZenEventManager interface with the 'Event Fields' tab selected. Below the tab, there is a 'Default Sort:' label followed by a text input field containing the text 'severity desc, lastTime desc'.

The **History Fields** are sorted in descending order by **lastTime**. The **lastTime** variable corresponds to the last time the event was seen by Zenoss Core.



The screenshot shows the ZenEventManager interface with the 'History Fields' tab selected. Below the tab, there is a 'Default Sort:' label followed by a text input field containing the text 'lastTime desc'.

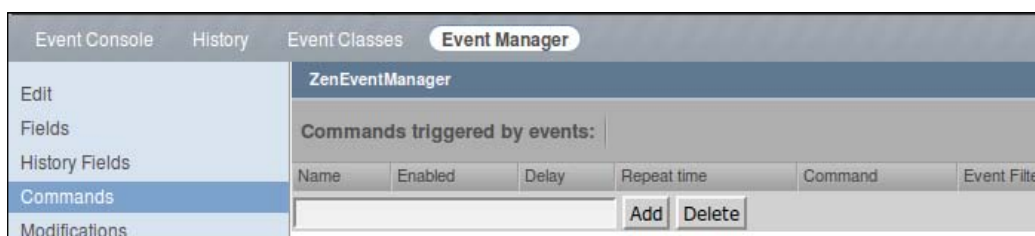
The syntax for the sort field is to list the field name to be sorted followed by the sort order. Multiple sort conditions (fields) are separated by a comma, as illustrated in the previous screenshots. As an example, if we want to sort our historical events in ascending order by count, we enter "count asc" as the **Default Sort**.

*Appendix A, Event Attributes* lists most of the events fields with descriptions. You also see the same information when viewing the event log.

## Event commands

The **Commands** page of the Event Manager (shown in the following screenshot), allows us to create shell commands that we can run in response to an event. Here are some possible uses for event commands:

- Integration with external services, such as Twitter
- Restart a service on a monitored device via SSH
- Push alerts through alternative mediums, such as instant messages



For our event command example, we'll write to a text file, which I admit is a contrived example. However, I want to demonstrate the feature, and we'll use this command later in the chapter to generate some events. If you want to do something more interesting, such as creating an RSS feed of your events, visit the Zenoss Core community and look up this tip: <http://community.zenoss.org/docs/DOC-7815>.

Now, let's create that file.

## Creating a command

To create a command, type a descriptive name (for example, **CreateFile**) in the text box and click the **Add** button. Click on the command name to display the Edit tab.

Let's modify our **CreateFile** command in the following way:

1. Set the **Enabled** field to **True**.
2. In the **Command** field, enter the following:
 

```
echo "The Event with ID ${evt/evid} is on fire!" >>
/tmp/SampleEventCommand\
```
3. In the **Clear Command** field, enter the following:
 

```
echo "${evt/evid} for ${dev/id} is no longer a burning issue" >>
/tmp/SampleEventCommand
```

4. In the **Where** field, define a filter for **Event Class** that begins with **/App**.
5. Save the changes:

ZenEventManager > Event Commands > CreateFile

State at time: 2010/10/17 16:55:25

Enabled

Default Command Timeout (secs)

Delay (secs)

Repeat Time (secs)

Command

```
echo "The Event with ID ${evt/evId} is on fire!" >> /tmp/SampleEventCommand
```

Clear Command

```
echo "${evt/evId} for ${dev/id} is no longer a burning issue" >> /tmp/SampleEventCommand
```

Where

Event Class

Add filter

Save

The variables inside the brackets {} are TALEs expressions. Zenoss uses TALEs expressions to substitute contextual information about an event or device into the command. When we test this command in a few pages, you will see the real-world result. *Appendix A, Event Attributes* and *Appendix B, Device Attributes* provide a listing of event and device attributes that can be substituted into our commands as TALEs expressions.

The following table lists each of the options on the command's edit page:

Property	Description
Enabled	Select True to enable the command and False to disable it.
Default Command Timeout	The time in seconds to wait for the command to complete. The default is 60. This can be adjusted as needed to account for lag or to give the command enough time to run.
Delay	The time in seconds Zenoss waits to execute the command from the time an event triggers the command. The default is 0, which means the command will run as soon as the event occurs.

---

Property	Description
Command	Enter the command to run when a new event matches the command filter. Accepts either Python statements or substitution via TALEs expressions.
Clear Command	Enter the command to run when a clear event matches the command filter. Accepts Python code or TALEs expressions.
Where	Add filters to modify the conditions that trigger the command.

---

Let's put our event command to use and work with events.

## Working with events

Zenoss Core creates events in response to a monitored condition, such as availability status, or performance.

For the most part, the event creation process is automatic, and we don't need to think about it, but we do have the opportunity to customize how events are processed in our individual monitoring environments.

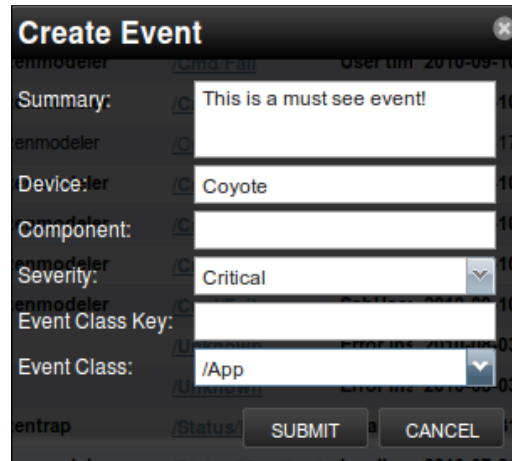
Zenoss Core provides us an easy way to manually generate events via the Event Console. We may want to manually generate an event so that we can test or troubleshoot mappings, transformations, or alerting rules without actually taking the monitored device out of service.

## Simulating an event

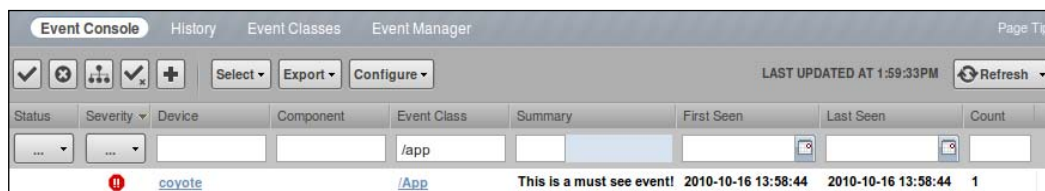
Let's create an event to run the `createFile` command that we created:

1. Go to the **Event Console** and click on the **Add Event** button (it looks like a plus sign).
2. In the **Create Event** dialog box, set the following fields:
  - Type a brief **Summary** message
  - Type the name of a **Device**
  - Select **/App** from the **Event Class**.

3. Click the **SUBMIT** button to generate the event.



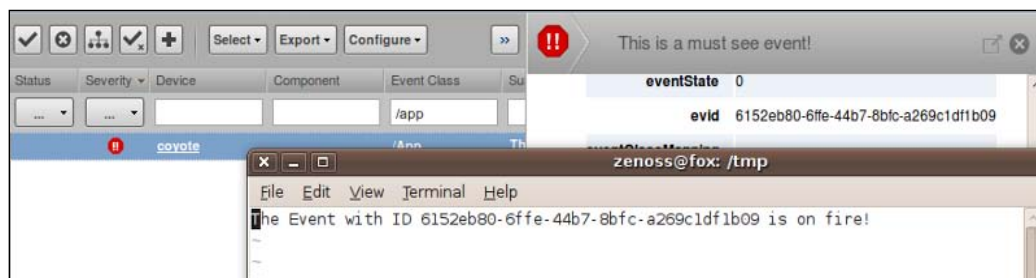
The event we just created displays in the **Event Console** and in the **Events** tab of the `/Events/App` class. Here's the screenshot to prove it:



Status	Severity	Device	Component	Event Class	Summary	First Seen	Last Seen	Count
		coyote		/App	This is a must see event!	2010-10-16 13:58:44	2010-10-16 13:58:44	1

Please note that no actual device, service, or process was hurt in the creation of this event.

Now, check the `/tmp/SampleEventCommand` file and verify that the event created an entry in the file. The following screenshot shows the Event Console, the event properties, and a terminal window with the contents of the `SampleEventCommand`:



The Event Console shows the event with the following properties:

- eventState: 0
- evid: 6152eb80-6ffe-44b7-8bfc-a269c1df1b09

The terminal window shows the command `zenoss@fox: /tmp` and the output:

```
The Event with ID 6152eb80-6ffe-44b7-8bfc-a269c1df1b09 is on fire!
```

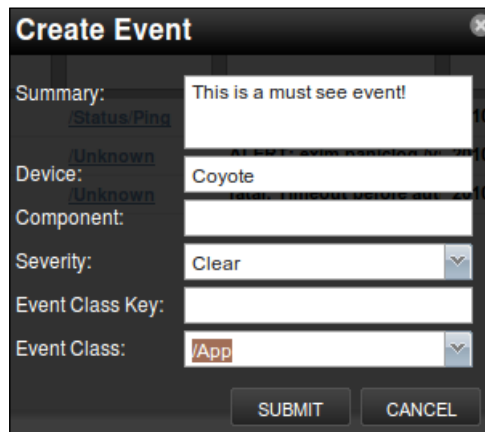
Notice the long alpha-numeric string and the name of the device appear in the command output. The alphanumeric string is the event ID, represented as {evt/id} in our event command that we created earlier in the chapter. Coyote is our device name and is represented by {dev/id} in the event command.

Do you think your command would run if you did not specify the Event Class when you created the test event?

Remember, we specified a filter for the command based on an event class that began with /App. So creating an event against /App/Info would trigger the command, but /Server would not.

## Clearing the event

Next, we will simulate a clear event. Add another test event via the Event Console, but this time choose **Clear** for the **Event Severity**. See the following screenshot:



When we submit the clear event, Zenoss correlates the current critical event with the clear event and moves both of them to History. All the event views update, and the /tmp/Sample/Event/Command file updates based on the clear command value we specified in the CreateFile command.

As long as we know the specific event condition we want to test, we can use the add event option to simulate a real event, thereby allowing us to test mapping rules, event commands, or notification rules.



## Event mapping

What did our event simulation and event command example show us, other than two really helpful features? If you said event mapping, you're reading closely. Let's see what that means.

Zenoss Core automatically maps an event to an event class based on the daemon reporting the event. The following table lists some Zenoss Core daemons and the default event class:

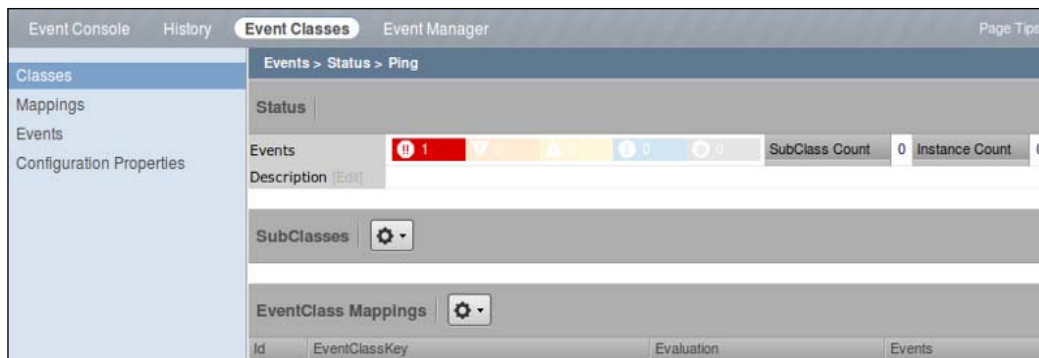
Daemon	Default event class	Type of event
Zenping	/Status/Ping	Device does not respond to a ping.
Zendisc	/Information	A new device is discovered.
Zenprocess	/Status/OSProcess	There is a problem with a monitored process.
Zenstatus	/Status/IpService	There is a problem with a monitored IP service.
Zenwin	/Status/WinServices	There is a problem with a Windows Service.
zenperfsnmp	/Status/SNMP	The device is not available via SNMP.

By knowing something as simple as the class that a type of event gets mapped to, we can customize our event handling by using the class `zProperties`. We could, for example, drop all ping events so they don't show in the Event Console or set the severity for all SNMP events to critical. Those are just a couple ideas to get you thinking.

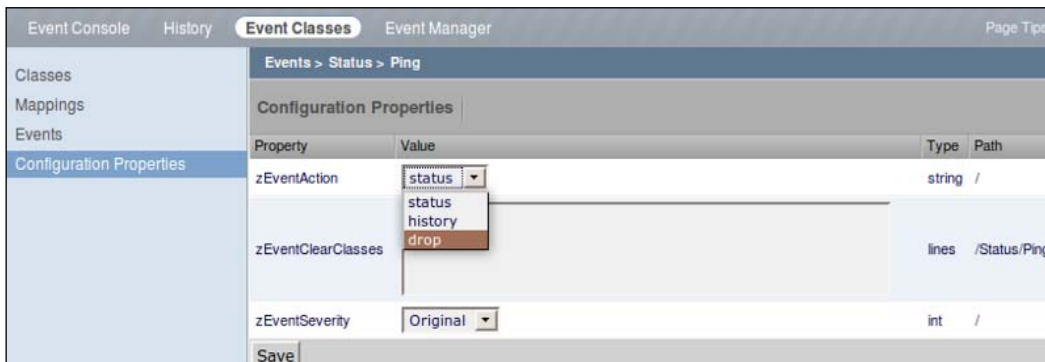
Let's take a look at event classes.

## Event Classes

Event classes establish a hierarchy of rules in the same way that we use device classes to inherit configuration properties. To view the classes, click on the **Events** menu and then **Event Classes**. To view the properties of a class, click on the name of the event class. I'm going to look at **/Status**:



Event Classes have two items that are of interest to us: mapping and configuration properties. For now, let's take a look at the **Configuration Properties**:



## Event class zProperties

After an event maps to an event class, the class configuration properties (a.k.a **zProperties**) are assigned. The event class properties will override any properties (for example, status or severity) that the event inherited from the device itself.

The available **zProperties** (shown in the previous screenshot) are outlined in the following table:

zProperty	Description
zEventAction	Specify the action to take on the event. The following options are available: <ul style="list-style-type: none"><li>• <b>Status:</b> Keep the event active and display it in the Event Console.</li><li>• <b>History:</b> Move the event straight to history. Does not show the event on the Event Console.</li><li>• <b>Drop:</b> Do not archive the event.</li></ul>
zEventClearClasses	Clear the event if the device generates an event that matches one of the specified event classes.
zEventSeverity	Specify the fail severity for the event. In descending order of severity, the available options are: <ul style="list-style-type: none"><li>• <b>Critical</b></li><li>• <b>Error</b></li><li>• <b>Warning</b></li><li>• <b>Info</b></li><li>• <b>Debug</b></li><li>• <b>Clear</b></li><li>• <b>Default</b></li></ul>

## Mapping an event

There may be times when an event pops into the Event Console in the /Unknown event class. Syslog events, for example, will map to the /Unknown event class.

We can change the mapping of an existing event to a new class. Whether you're mapping an unknown event or changing an existing mapping, the procedure is the same.

Let's take an example where we have an active event from an **exim4** mail server program event which we want to map to the **/App/Log** event class. Feel free to use your own example.

Status	Severity	Device	Component	Event Class	Summary	First Seen	Last Seen	Count
...	...							
II		192.168.1.0		/Status/Ping	192.168.1.0 sendto error [Err	2010-09-10 22:53:27	2010-10-16 20:16:11	4528
II		covote	exim4	/Unknown	ALERT: exim paniclog /var/l	2010-08-01 06:30:34	2010-10-16 06:29:17	51
II		covote	sshd	/Unknown	fatal: Timeout before auther	2010-09-11 09:28:33	2010-09-17 21:28:20	14

To map the event:

1. Select the event from the **Events** list.
2. Click the **Map to Event Class** button (looks like an org chart) to display the **Classify Events** dialog box.
3. In the **Classify Events** dialog box, select the **event class** (for example, **/App/Log**).
4. Click on **SUBMIT** to map the event.

After we map the event, Zenoss displays a confirmation message with a link to view the new mapping. Follow the link to view the new mapping:

Event Console
History
**Event Classes**
Event Manager
Page Tips

Events > App > Log > exim4

Status
Edit
Sequence
Configuration Properties
Events
Modifications

Status

Events

0
0
0
0
0

Total Event Count
0

EventClassInst

Event Class
exim4

Key

Sequence
7

Rule

Regex

Example
ALERT: exim paniclog /var/log/exim4/paniclog has non-zero size, mail system possibly broken

Transform

Explanation

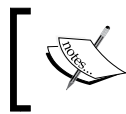
Resolution

You can also browse the `exim4` mapping by navigating to the `/Apps/Log` event class and then selecting the **Mappings** link in the sidebar.

Let's test our work by simulating an event from the Event Console. Give the event the following attributes:

- **Message:** This is a test event
- **Device:** Any device
- **Event Class Key:** **exim4**
- **Event Class:** Leave this blank

The event displays in the Event Console. To clear the event, we can move it to History.



When processing events generated by Zenoss daemons, the events classes are used. When the event comes from external sources, the event class key is used to process the event.

The **exim4** mapping in the preceding screenshot is one of several event mappings for the **/App/Log** class. We can view all the mappings associated with the class by clicking on the **Log** link in the page breadcrumb.

As seen in the following screenshot, the **EventClass Mappings** table displays some information about each mapping including event class, evaluation rule, and the number of active events for the mapping:

Event Console	History	Event Classes	Event Manager	Page Top
Classes	Events > App > Log	Status		
Mappings		Events	SubClass Count	Instance Count
Events		Description		
Configuration Properties		SubClasses		
		EventClass Mappings		
		Id	EventClassKey	Evaluation
		<input type="checkbox"/> <a href="#">CROND</a>	CROND	\((?P<username>\S+)\) CMD \((?P<command>.*?)\)
		<input type="checkbox"/> <a href="#">exim4</a>	exim4	ALERT: exim paniclog /var/log/exim4/paniclog has non-zero si
		<input type="checkbox"/> <a href="#">pmta</a>	pmta	
		<input type="checkbox"/> <a href="#">postfix/smtpd</a>	postfix/smtpd	warning: (?P<eventKey>\S+):
		<input type="checkbox"/> <a href="#">procmail</a>	procmail	Descriptor 1 was not open
				Events

To configure a mapping, click on the mapping name and then click on the **Edit** link. Let's edit the CROND mapping to see what information is available to us.

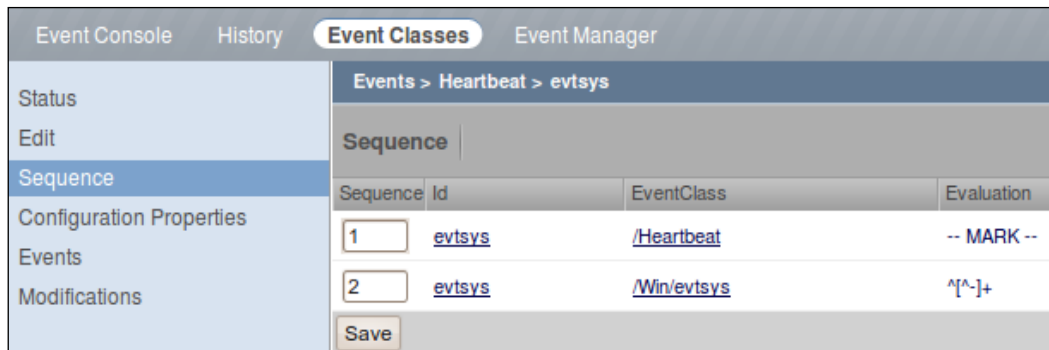
The following table outlines the available properties for an event class mapping:

Option	Description	Example
<b>Name</b>	A descriptive name for the mapping.	CROND
<b>Event Class Key</b>	Refers to the component name attached to the incoming event. Matches the event with the event class.	exim4
<b>Sequence</b>	Defines the order in which the Event Class Key is processed in relation to the other keys with the same name.	7
<b>Rule</b>	A Python statement and TALES expression that evaluates the current event context.	<code>evt.ipAddress == 192.168.1.132</code>  <code>evt.device != Coyote</code>
<b>Regex</b>	A Python regular expression used to match the current event details.	<code>\((?P&lt;username&gt;\S+)\) CMD \((?P&lt;command&gt;.*)\)</code>
<b>Example</b>	Sample event text. Click Save to validate the regular expression against the Example text. If the Regex command turns red, the expression does not match the Example.	<code>(root) CMD (run-parts /etc/cron.hourly)</code>
<b>Transform</b>	Manipulate an event using Python and TALES. For more information about TALES expressions, see Appendix B.	<code>evt.summary = 'Down again!'</code>
<b>Explanation</b>	A text description of the event mapping for Zenoss users.	
<b>Resolution</b>	Provides a spot to document fixes that can be used by other system administrators.	

As we saw when we created the exim4 mapping to the /App/Log event class, specifying the component name as the event class key is enough to map the event. However, you can have multiple mappings per class and multiple mappings for the same event class key. The **Rule**, **Regex**, and **Transform** values help you refine the mapping so that you can isolate specific events and filter the noise.

## Event mapping sequence

If we can have multiple event class keys across mappings, there must be some processing logic – and there is. To demonstrate, pull up the **evtsys** mapping for the **/Heartbeat** event class. Then, click on the **Sequence** link in the sidebar. See the following screenshot:



Events > Heartbeat > evtsys			
Sequence			
Sequence	Id	EventClass	Evaluation
1	<a href="#">evtsys</a>	<a href="#">/Heartbeat</a>	-- MARK --
2	<a href="#">evtsys</a>	<a href="#">/Win/evtsys</a>	^[^]+
Save			

The **Sequence** page displays the **Sequence**, **ID**, **EventClass**, and **Evaluation** for each instance of the event class mapping. If we look closely at the two instances of **evtsys**, we see that **evtsys** maps to two event classes. The **evaluation** field tells us what each mapping looks for.

If the **evtsys** event includes **--MARK--**, the event goes to **/Heartbeat** and Zenoss Core stops processing event mappings. If the first mapping fails, then mapping two is evaluated, and so on until all mappings are evaluated or a match is found.

To resequence the mappings, type the new order into **Sequence** column and click on **Save**.

## Event de-duplication


Zenoss Core continuously monitors devices and will continue to generate events for problems it has previously detected. However, Zenoss Core builds some intelligence into its event handling by recognizing duplicate events.

An example of a duplicate event might be a file system that's operating at 98% capacity on a server. We want Zenoss Core to notify us when that threshold is crossed, but we don't need a "reminder" every time Zenoss polls the device. That would clog up the Event Console and clog up our e-mail with dubious alerts.

Think about this. If Zenoss Core monitors a device every 60 seconds, it could generate 1,440 events a day for a single problem. Thankfully, Zenoss suppresses all that noise with event de-duplication.

If Zenoss determines that the event is a duplicate of an existing event, it increments the count on the existing event, rather than generate a new event. Events trigger alerts, and by suppressing duplicate events we avoid duplicate alerts, thereby reducing the volume of e-mails or pages we receive.

The de-duplication identification (dedupid) is a combination of the following fields: device, daemon, event class, event key, severity, and event message. We can see it in the event log of any event. Go to the Event Console and double click on an event to display the log.

 SshUserAuth: no password found -- has	
<b>dedupid</b>	coyote zenmodeler /Cmd/Fail  4 SshUserAuth: no password found -- has zCommandPassword been set?
<b>manager</b>	localhost
<b>Location</b>	

## Turning off event de-duplication

Zenoss Core wants to dedupe every event it receives, but you can override this behavior by adding a custom event mapping and event transformation for the event class. The following transformation comes from the Zenoss Core community documentation at <http://community.zenoss.org/docs/DOC-2445#HowdoIstopautomaticEventDeduplication>:

```
mydedupfix1 = getattr(evt, 'triggerLastOccur')
mydedupfix2 = getattr(evt, 'triggerEventName')
evt.eventKey = '%s - %s' % (mydedupfix1, mydedupfix2)
```

This code creates a unique event key for every instance of the event, thereby voiding Zenoss Core's deduplication efforts. We'll work with event transformations some more in *Chapter 7, Collecting Events*.



## Summary

Events are the fruit of our monitoring labor, and event handling is one of the primary ways we can turn the raw monitoring data into actionable items. For example, we review the event history of a device to identify trends. Event commands provide a way to automate actions when a certain event happens, and the breadth of what you can do with event commands is only limited by your imagination and programming knowledge. In the next chapter, we will also collect events from external sources and create alerting rules.

Event mappings route the event to the event class and in the process, the event inherits properties, such as severity. In the next chapter, we'll see how we can manipulate the properties of an event with transformations.

# 7

## Collecting Events

In *Chapter 6, Core Event Management* we took an in-depth look at how Zenoss Core processes events that result from its core monitoring activities. In other words, Zenoss Core went looking for problems. In this chapter, we'll let the problems come to Zenoss Core by way of non-Zenoss applications, such as syslog, Windows event log, scripting, and e-mail.

In this chapter, we will:

- Route syslog messages to Zenoss Core
- Monitor the Windows event log
- Incorporate event reporting into third-party scripts via zensendevent
- Create events by e-mail using zenpop3 and zenmail
- Configure alerting rules

Let's get started.

### Routing syslog messages to Zenoss Core

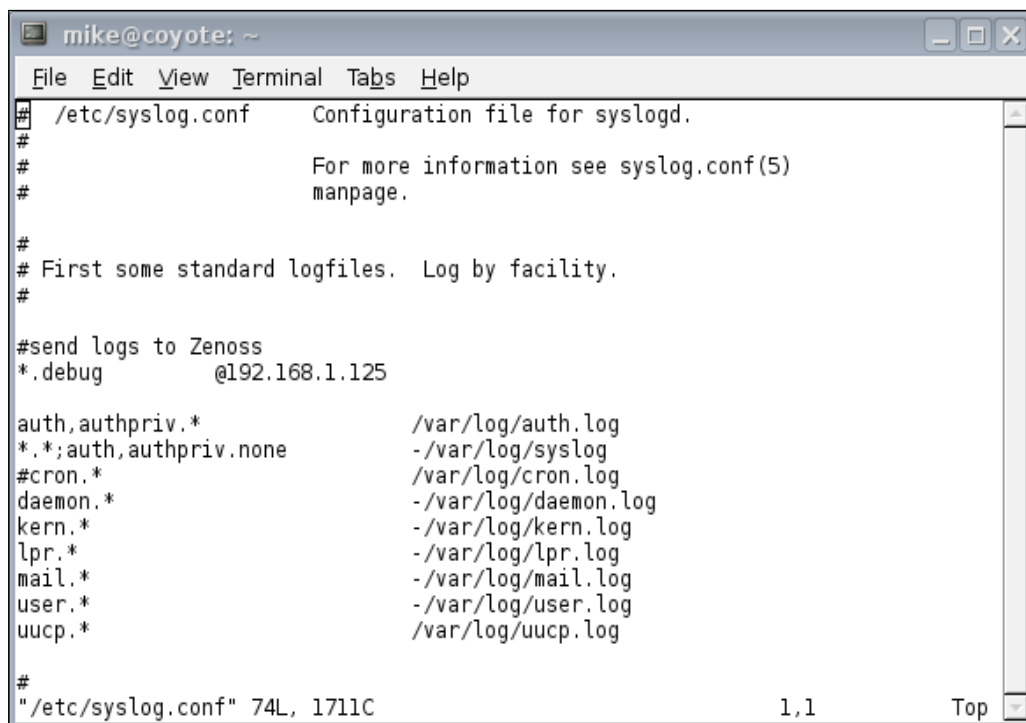
We have the capability to monitor syslog messages from Unix-based hosts on the network by turning Zenoss Core into a syslog server. The syslog is a standard logging format for Unix-based systems that allow administrators to analyze, troubleshoot, and debug the programs and services running on a device. Zenoss uses the `zensyslog` daemon to turn incoming syslog messages into events from any host on the network.

Before we configure our servers to send syslog messages to Zenoss Core, we need to determine the syslog facility and priority we want to monitor. The available facilities include `auth`, `authpriv`, `cron`, `daemon`, `ftp`, `kern`, `lpr`, `mail`, `news`, `syslog`, `user`, and `uucp`. The facility specifies the subsystem we want to monitor. For example, we specify the `lpr` facility to monitor print activity.

We specify one of the following priorities, listed from the lowest to the highest severity: `debug`, `info`, `notice`, `warning`, `err`, `crit`, `alert`, and `emerg`. As an example, we would choose `warning` to monitor logs with a priority of `warning`, `err`, `crit`, `alert`, and `emerg`.

Pick a Linux server from your inventory and let's work through an example by modifying its `syslog.conf` file:

1. Edit `/etc/syslog.conf` as root (refer to the following screenshot).
2. Add the following line where `192.168.1.125` is the IP address or the host name of the Zenoss Core server:  
`*.debug @192.168.1.125`
3. Restart the syslog service as root:  
`/etc/init.d/syslogd restart`



```
mike@coyote: ~
File Edit View Terminal Tabs Help
# /etc/syslog.conf Configuration file for syslogd.
#
# For more information see syslog.conf(5)
# manpage.
#
# First some standard logfiles. Log by facility.
#
#send logs to Zenoss
*.debug @192.168.1.125
auth,authpriv.* /var/log/auth.log
*,*,auth,authpriv.none -/var/log/syslog
#cron.* /var/log/cron.log
daemon.* -/var/log/daemon.log
kern.* -/var/log/kern.log
lpr.* -/var/log/lpr.log
mail.* -/var/log/mail.log
user.* -/var/log/user.log
uucp.* /var/log/uucp.log
#
"/etc/syslog.conf" 74L, 1711C 1,1 Top
```

Note the `*.debug` syntax in the syslog example. This sends all syslog facilities (represented with `*`) with a minimum priority of debug to Zenoss Core. I chose this setting for our example so we could get the events flowing. Dumping all your syslog messages to a the Zenoss Core event will likely cause a lot of events, and may be more than you want. So, some customization is warranted here and the natural question is, "What is an appropriate value?". However, I can't provide an answer that applies to you and your environment. Remember, we're telling individual servers what messages to send to Zenoss Core. If in doubt, cast a wide net and then narrow it down.

I, for example, really don't want to see events in Zenoss Core for anything less than an error. So at the very least, I modify the rule in my server's `syslog.conf` file to:

```
*.err @192.168.1.125
```

If I was configuring the syslog on a server whose main mission in life was as an FTP server, then I might be inclined to narrow the server's `syslog.conf` with this rule:

```
ftp.err@192.168.1.125
```

This will send all syslog messages concerning FTP that at least have a priority of error to the Zenoss Core server for processing by `zensyslog`. You can specify multiple rules, which helps us send just the right messages to Zenoss Core.

Feel free to set a more appropriate value. For the benefit of the non-Linux system administrators, you can gain more information about `syslog.conf`, by running the command at your nearest terminal:

```
man syslog.conf
```

Of course, Unix-based servers are not the only devices that have remote syslog capabilities. Many routers provide remote logging features. We'll overview the steps needed for a Cisco router. For other devices, consult the documentation.

## Collecting Cisco router syslogs

To forward a Cisco router's syslogs to Zenoss Core, we need to know the Zenoss Core host, the minimum log priority to collect, and the facility. The following priorities are available: emergency, alert, critical, error, warning, notice, informational, and debug. The available facilities include `local0` through `local7` (also available to syslog on Unix servers) and the default facility is `local7`.

To forward syslog messages from a Cisco IOS router to Zenoss Core, log into the router and follow these steps using privileged EXEC mode:

1. Enter the configuration mode with the command:  
`configure terminal`
2. Specify the Zenoss server by IP address or host name with the command:  
`logging 192.168.1.125`
3. Set the syslog priority:  
`logging trap warning`
4. Set the syslog facility:  
`logging facility local7`
5. Quit the configuration mode:  
`end`
6. Verify the logging information:  
`show logging`

## Testing syslog configuration with Logger

We can test our remote syslog configuration by using the command line tool "logger" to send a test syslog message of a specified facility and priority. To test, run the following commands from the Linux device that is logging its syslog messages to Zenoss Core:

```
logger -p cron.warn "This is a test"
logger -p mail.error "This is another test"
```

The logger command syntax is straightforward. The `-p` option specifies the facility and the priority which we follow with a message in quotes. Depending on the `syslog.conf` rules you wrote, the event may or may not be sent to Zenoss Core. Obviously, your examples should test both use cases.

To double check, click on the Event Console and verify that the syslog messages are being logged correctly.



The events from syslog will show with an unknown class. Use the procedures outlined in *Chapter 6, Core Event Management* to map the events to a class.

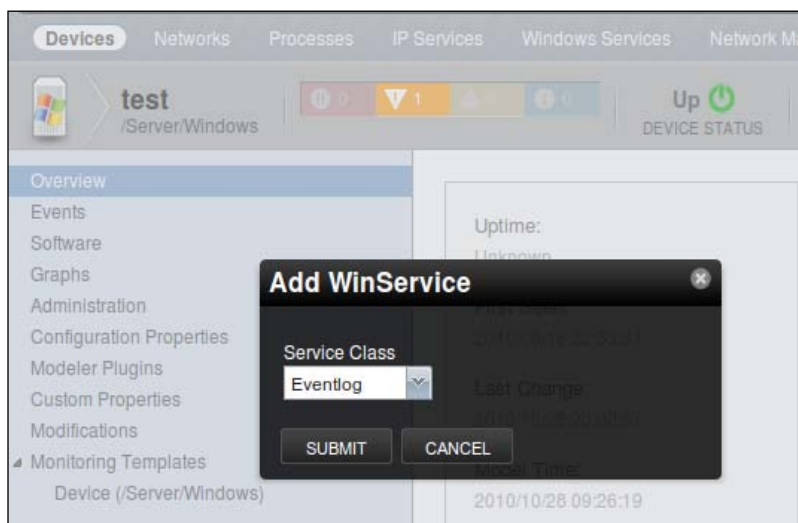
## Monitoring Windows event logs

In the *Configure SNMP and WMI for Windows* section of *Chapter 2, Discovering Devices* we discussed setting up Windows Management Instrumentation (WMI). If WMI is not yet installed, take a few moments to review the instructions in *Chapter 2, Discovering Devices*.

Unlike syslog, which logs messages directly to a remote host, Zenoss Core has to connect to the Windows server to pull entries from the Event Log—at least that's the default behavior we will explore in this section. There are third party applications that will log Windows event logs to remote syslog servers such as Zenoss Core. This allows zensyslog to process the messages, and you could use event mappings to make sure the events from the Windows server get associated with an appropriate event class.

If you have a Windows server available, open it in Zenoss Core so we can configure Event Log monitoring:

1. From the devices page, select **Add WinService** from the **Add Component** menu.
2. In the **Add WinService** dialog box, enter **Eventlog**:



3. Click **Submit** to add the service to the device. You will also notice that a new group of **Components** have been added to the device, called **Windows Services**.

- When you add the service it will inherit the service class' default monitoring value. So we need to ensure monitoring is enabled. Click on **Windows Services** from the device's **Components**. Then click on the **Eventlog** service to display its configuration.

The screenshot shows the configuration page for the 'Eventlog' service in Zenoss Core. The page has a tabbed interface with 'Windows Services' selected. The configuration fields are as follows:

Field	Value
Name:	Eventlog
Description:	Event Log
Service Keys:	Eventlog
Monitored Start Modes:	
Enable Monitoring? (zMonitor)	Set Local Value: Yes
Failure Event Severity (zFailSeverity)	Set Local Value: Error

- Under the heading **Enable Monitoring**, select **Yes** from the **Set Local Value** drop-down list.
- Click on **Save**.

Next, we need to configure the appropriate device zProperties (**Configuration Properties**) to connect to the Windows machine and monitor the event logs. From the device's overview page, select **Configuration Properties** and enter the following configuration:

- Set **zWinEventlog** to **True**.
- Set **zWinPassword** to the password of the **zWinUser**.
- Set **zWinUser** to a user who has administrative access to the Windows server.
- For a domain user, specify `DOMAIN\user`.
- For a local user, specify `\user`.
- Set **zWmiMonitorignore** to **False**.

## Windows event log severities

By default, Zenoss Core collects the Windows events with a minimum severity of warning. But we can change that by specifying a value in **zWinEventlogMinSeverity**. The following table shows the available event log severities:

Event Log Severity	Description
1	Error
2	Warning
4	Informational
8	Security Audit Success
16	Security Audit Failure

## Testing the event log configuration with Eventcreate

Windows provides a tool called `eventcreate.exe` that we can use to generate system events and test our Event Log setup. To test, run the following commands from a Windows device where Zenoss Core is monitoring the Event Log:

```
eventcreate /t error /l system /id 500 /d "test message"
eventcreate /t error /id 501 /d "another test message"
eventcreate /?
```

Let's look at the command syntax. We use the `/t` option to specify the severity, `/l` to specify either the application or the system message, `/id` to create an event ID, and `/d` to include a message. The first command creates a system error message with an ID of 500, while the second command creates an application error message with an ID of 501. The third command displays the `eventcreate.exe` help page.

## Incorporating event reporting into third-party scripts via zensendevent

Do you have a bunch of homegrown system administration scripts running on your network? Wouldn't it be nice to monitor the activities of those scripts? Zenoss Core ships with a stand alone script called `zensendevent` that we can integrate into our existing scripts in order to use Zenoss Core's event systems.



We can install `zensendevent` on any server (or desktop) as long as the system has Python installed.

A sample use would be to use `zensendevent` to create events to document the progress of a backup script by generating an informational event at the start and a clear event after the script completes.



We'll work up a simple example, but first let's use `zensendevent` to generate some events from a command line. To see what options are available, run the following command from a command line as the zenoss user:

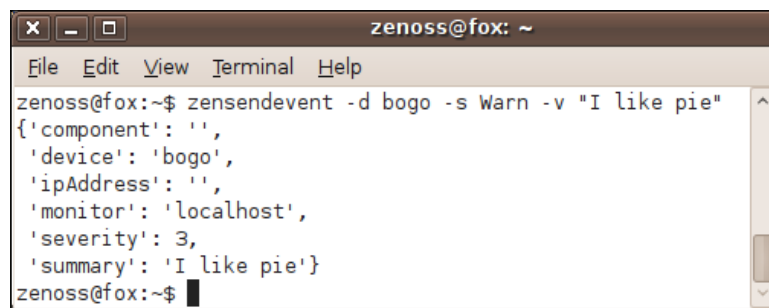
```
zensendevent --help
```

As you look over the usage summary, you should recognize most of the available options from our previous topics. Among the familiar items that we can recognize are `device`, `ipAddress`, `event key`, `event class`, and `component`.

Let's run a test command:

```
zensendevent -d bogo -s Warn -v "I like pie"
```

The following screenshot shows the output of the command:



```
zenoss@fox: ~  
File Edit View Terminal Help  
zenoss@fox:~$ zensendevent -d bogo -s Warn -v "I like pie"  
{'component': '',  
 'device': 'bogo',  
 'ipAddress': '',  
 'monitor': 'localhost',  
 'severity': 3,  
 'summary': 'I like pie'}  
zenoss@fox:~$
```

There are not really any surprises in this command. We created an event for the device named `bogo` with a severity of warning. The summary of the event is `I like pie`. Hey, it's getting lonely writing about network monitoring late at night. My mind likes to wander to pie. Feel free to substitute your own silliness. The `-v` option shows the verbose output, which is showing us the information that was sent to Zenoss Core.

Let's look at the events for our `bogo` device in Zenoss Core:



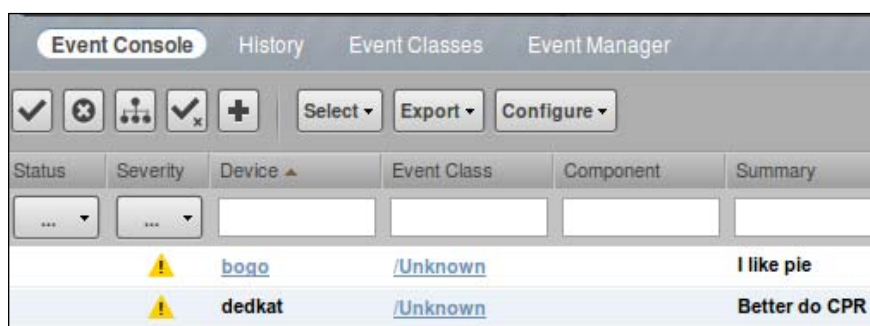
Once we get that event into Zenoss Core we can leverage Zenoss's event handling system, which we talked about in the previous chapter. Perhaps more importantly, we can now use Zenoss Core's alerting system, which we will set up at the end of this chapter. We could also transform the event, also coming up in this chapter.

Or we could just take a moment to reflect on the world of opportunity we've just unlocked:

Let's run one more test command using a device that doesn't exist in Zenoss Core's inventory:

```
zensendevent -d dedkat -s Warn -v "Better do CPR"
```

The following screenshot shows Zenoss Core's event console:



Note that the event name `dedkat` is not a clickable link, but it still shows up in the **Event Console**. It's not clickable because it's not a device in the inventory. The point of showing this was just to show that you can feed Zenoss Core a lot of information, which could be used to report some high level monitoring aspect not tied to a device or to have a new server automatically report system information to Zenoss Core for use in configuring the device in Zenoss Core. (Thank you to my technical reviewers, Jonny and Charlie for providing some ideas.)

## Simple backup script with zensendevent

Let's take a look at a real world use for `zensendevent` by incorporating it into a script that backs up our Zenoss Core data:

```
#!/bin/sh

zenbackup --save-mysql-access 2> $HOME/backups/zenbackup.log

STATUS=$(echo $?)
```

```
if [ "$STATUS" = "0" ]; then

    zensendevent -d fox -k backup -s Info -c /Status/Update "Zenoss
    backup successful"

else

    zensendevent -d fox -k backup -s Warn -c /Status/Update "Zenoss
    backup failed"
fi
```

The script itself is fairly simple. It runs a Zenoss file called `zenbackup` which will automatically write backups to `/home/zenoss/backups`. We use `zensendevent` to create an event. If the `zenbackup` command completes successfully, we set a severity of informational, and if it fails, we set the severity to warning.

The biggest shortcoming of this script is that it doesn't alert you if the backup fails to run, which could be a problem for some. Feel free to adapt this script as you see fit.

I suspect many readers probably have similar scripts that send e-mails if there is a problem, so you might wonder why we'd want to go through the trouble of directing the data to Zenoss Core. As we already discussed, Zenoss Core provides a centralized place to collect and alert on our events. Zenoss Core will also maintain an event history and we can write custom reports to view that history.

## Creating events by e-mail

Zenoss Core provides two not often talked about daemons to generate events from e-mails. The `zenmail` daemon allows us to start a Zenoss Core SMTP server that other programs can use as a mail server to send e-mail messages directly to Zenoss Core. Zenoss Core automatically turns the message into an event. The `zenpop3` daemon retrieves e-mails from a specified account and generates events based on the incoming e-mails.

To use either program in daemon mode, we need to edit the `$ZENHOME/bin/zenoss` configuration file, so that the daemons start when Zenoss Core starts.

As the `zenoss` user:

1. Open `$ZENHOME/bin/zenoss` in a text editor.
2. Find the line in the script that begins with `$ZENHOME/bin/zenfunctions` and uncomment or add the following lines (refer to the next screenshot):

```
C="$C zenmail"
C="$C zenpop3"
```

- Restart the Zenoss daemons with the command `zenoss restart`:

```
C="$C zenperfsnmp"
C="$C zencommand"
C="$C zenprocess"
C="$C zenpop3"
C="$C zenmail"
$ZENHOME/bin/python -c 'import pysamba' 2>/dev/null
if [ $? -eq 0 ]
then
    C="$C zenwin"
    C="$C zeneventlog"
fi
```

When we restart the Zenoss daemons, `zenmail` and `zenpop3` print warning messages that tell us that they were unable to find the configuration files in `/usr/local/zenoss/etc/`. To clear those messages up, run the following commands as the `zenoss` user:

```
zenmail genconf
```

```
zenpop3 genconf
```

The `genconf` option creates a configuration file in `$ZENHOME/etc` with all the available options for the daemon. Each Zenoss daemon accepts the `genconf` option. Now we're ready to configure `zenmail` and `zenpop3`.

## Zenmail

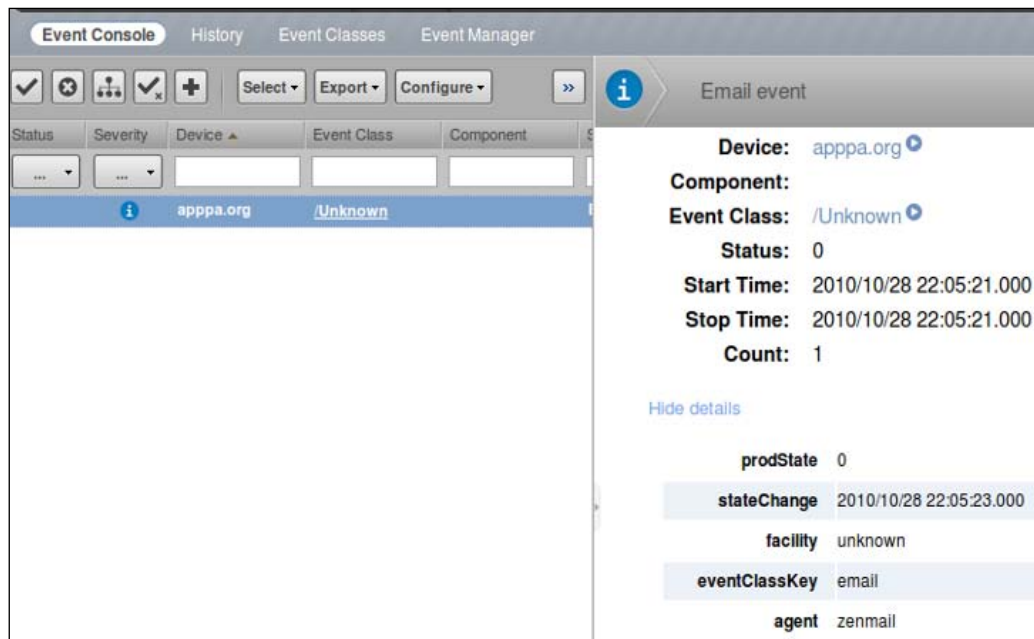
You may have some devices on your network that don't support other monitoring protocols, such as SNMP. However, they may be capable of sending e-mail notifications when there is a problem (for example, Low disk space on a network storage device). By setting up `zenmail` and configuring the device to relay the e-mail notifications through the IP of the Zenoss Core server, we can keep a history of events and use Zenoss Core to manage the events and alerts.

An easy way to test out `zenmail` is to configure a mail client to send e-mail via the host name or IP address of the Zenoss Core server. The default port will be 25, unless you already have a mail server running on port 25 of the Zenoss Core server, in which case `zenmail` will not function.

If we want to bind `zenmail` to a port other than 25, we can edit the `zenmail` configuration file and add the parameter `listenport` followed by the new port number. We examine the Zenoss Core daemons and their configurations in more detail in *Chapter 9, Extending Zenoss Core with ZenPacks*. Don't forget to restart the daemon after you make configuration changes:

```
zenmail restart
```

The following screenshot shows an event that was sent through zenmail along with part of the event log:



The device name is automatically determined by the *from* address and the event is created with a severity of information. The event log shows many interesting fields, including the **eventClassKey**. We covered the event class key in *Chapter 6, Core Event Management*.

All events have an **agent** field that lists the Zenoss Core daemon that is responsible for creating the event. In this case, the **agent** is **zenmail**.

If you use zenmail, then you will probably want to use an event transformation to fine tune how the event is generated. The device, eventClasskey, and agent are all great fields to use for an event transformation.

Most mail programs will require that you specify a *To:* address in your e-mail, but zenmail doesn't use that information.

For a list of all options of zenmail, run the command:

```
zenmail --help
```

## Zenpop3

As the name implies, `zenpop3` will check a POP3 account and turn the messages into events. Finding a use for `zenpop3` might take more creative work, but we'll explore it anyway.

In order to make `zenpop3` work, we need to specify the mail server, username, and password at a minimum:

1. In Zenoss Core, navigate to **Administration | Daemons**.
2. Edit the configuration for **zenpop3**.
3. Enter the following parameters followed by the correct POP3 server values (refer to the next screenshot):
  - **pophost**
  - **popuser**
  - **poppass**
  - **cycletime**
4. Save the configuration.
5. Restart **zenpop3**.

The screenshot shows the Zenoss Core Administration interface. The 'Daemons' section is selected in the left sidebar. The 'zenpop3' configuration is displayed in a table-like format with the following values:

Parameter	Value	Description
maxqueueulen	5000	Maximum number of events to queue
monitor	localhost	Name of monitor instance to use for configuration. Default is localhost.
nodelete	<input type="checkbox"/>	Leave messages on POP server
pophost	pop.zenoss.com	POP server to auth against
poppass	zenoss	POP password to auth using
popport	110	POP port to auth against
popuser	zenoss	POP user to auth using

To test the setup, we send an e-mail to the account we specified in the `zenpop3` configuration. If everything is successful, we get an unknown event in the **Event Console**. If you also retrieve this e-mail account from another e-mail client, make sure you check the **nodelete** option to leave the mail on the server.

Now that we're collecting all these events in Zenoss Core, let's take a look at how we proactively notify an admin.

## Configuring alerting rules

Alerts are the final piece of our core monitoring setup, if you need or want them. Events trigger alerts, and alerts notify a human about a problem. We attach alerts to users or groups of users. Configuring alerts for users and groups require the same basic steps, so we'll demonstrate the alerting rules for a user. If you opt not to set up e-mail or pager alerting rules, Zenoss Core will continue to monitor devices and generate events that will be visible in the Event Console.

To add an alerting rule, first edit the user. Select the **Alerting Rules** tab (see the following screenshot) while editing the username to display the list of rules assigned to the user.

1. From the **Advanced** menu, select **Settings** and then **Users** to display a list of users. You should see a list of users that includes the admin user and the user you defined during installation.
2. Edit a user (other than admin) by clicking on the name. The properties for the user account will be displayed:

The screenshot shows the Zenoss Core interface for editing a user named 'mike'. The 'Roles' dropdown menu is open, showing 'Manager', 'ZenManager', and 'ZenUser'.

3. From the sidebar, click on the **Alerting Rules** menu. A blank **Alerting Rules** page will be displayed.
4. Select **Add Alerting Rule** from the **Alerting Rules** menu. The **Add Alerting Rule** dialog box will be displayed.
5. In the **Add Alerting Rule** dialog box, type a name for the rule (for example, Test). Click the **OK** button to add the rule. At this point the rule is disabled.

Edit

Administered Objects

Event Views

Alerting Rules

ZenUsers > mike

Alerting Rules

Name	Delay	Repeat Time	Action	Enabled	Send Clear
<input type="checkbox"/> <u>Test</u>	0	Does not repeat	email	False	True

6. To enable the rule, click on the rule name to display the properties.
7. Select **True** from the **Enabled** drop-down list.
8. Click on **Save** to enable the alerting rule.

At this point we have an alerting rule with some default settings. The default rule sends an e-mail when any device in a **Production State** generates a new event with a **Severity** level equal to or greater than **Error**. Zenoss also sends an alert when the event clears.

We can do so much more. Let's take a look at the alerting rule properties:

The screenshot shows the 'Alerting Rules > DeLuxe' configuration page. At the top, it says 'State at time: 2011/02/06 11:37:56'. The settings are as follows:

- Delay (secs)**: 0
- Enabled**: True (dropdown)
- Action**: email (dropdown)
- Address (optional)**: mbadger@mojoactive.com
- Plain Text**: False (dropdown)
- Repeat Time (secs)**: 0
- Send clear messages**: True (dropdown)
- Where**:
  - Event State**: = (dropdown) New (dropdown)
  - Production State**: = (dropdown) Production (dropdown)
  - Severity**: >= (dropdown) Warning (dropdown)
  - Device**: is (dropdown) www.deluxebuildingsystems.com (dropdown)
- Add filter**: (dropdown)
- Save**: (button)

The properties are explained in the following table:

Property	Description
<b>Delay (secs)</b>	Delay sending the alert for the specified time. Default is <b>0</b> .
<b>Enabled</b>	Set to <b>True</b> to enable the alert. If the value is <b>False</b> , this rule does not send alerts.
<b>Action</b>	Choose either <b>email</b> or <b>pager</b> notifications.
<b>Address (optional)</b>	Specify any valid e-mail address. If left blank, the e-mail address specified for the user is used.
<b>Send clear messages</b>	Select <b>True</b> to send alerts when the event clears. Select <b>False</b> to suppress clear messages.
<b>Repeat Time (secs)</b>	Repeat the alerting rule for the specified time. Default is <b>0</b> .
<b>Where</b>	Select the alert filter criteria. Add and remove filters as needed.



By adding filters to the alerting rule, we can create very specific alerting conditions. Let's take a closer look at the alert filters.

## Alert filters

An alert filter consists of three parts: an event field, a comparison operator, and a value to compare to the event field. The following screenshot shows the default filters for a new rule:

The screenshot shows a configuration window for alert filters. It contains three rows of filter settings, each with an event field, a comparison operator, and a value. Below these rows is an 'Add filter' button and a 'Save' button.

Production State	=	Production	-
Severity	>=	Error	-
Event State	=	New	-

Below the table is an 'Add filter' button and a 'Save' button.

In the screenshot, **Production State**, **Severity**, and **Event State** are the event fields. The middle column of drop-downs with the = and >= signs are the comparison operators, and the third column shows the available values that can be assigned to the event fields.

The **Add filter** drop-down list contains a list of available event fields:

The screenshot shows the 'Add filter' drop-down list open, displaying a list of available event fields. The list includes: Agent, Component, Count, Device, Device Class, Device Groups, Device Priority, Event Class, Event Class Key, Event Key, Event State, Facility, IP Address, Location, Manager, Message, ntevid, Owner Id, and Priority.

You can find each of these items with a corresponding description listed in Appendix A Event Attributes.

The more filters we add to a rule, the more specific our alerting rule becomes. A common practice with alerts is to build an alert escalation. For example, we can create a rule that says, if a new event remains unacknowledged after five consecutive times, then trigger a new alert. Five is an arbitrary number.

Let's add an escalation for our default alerting rule.

## Alert escalations

Alert escalation is a broad term that commonly refers to the act of increasing the severity of an event or notifying a different person about an event. For a quick example, we'll create an alerting rule to notify a backup contact person regarding an unacknowledged event.

In our example, I'll assume you're adding a second rule named Escalate to your user account, but the rule can be added to any user or group.

In the **Alerting Rules** properties for the new rule, set the following fields:

- **Address:** The e-mail address of on-call admin
- **Add filter for Count.** For the comparison select **>=** and enter **5** for the value

The screenshot shows the configuration page for an alerting rule named 'Escalate' under the path 'ZenUsers > mike > Alerting Rules > Escalate'. The page has a left sidebar with links: 'Edit', 'Message', and 'Schedule'. The main content area shows the rule's state as '2010/10/30 15:36:01'. Configuration fields include: 'Delay (secs)' set to 0, 'Enabled' set to True, 'Action' set to email, 'Address (optional)' set to bossman@badgerfiles.com, 'Plain Text' set to False, 'Repeat Time (secs)' set to 0, and 'Send clear messages' set to True. Under the 'Where' section, there are four filters: 'Production State' = Production, 'Severity' >= Error, 'Count' >= 5, and 'Event State' = New. Each filter has a minus sign to its right. There is an 'Add filter' button and a 'Save' button at the bottom.

By adding the new filter, the rule will only be triggered if a event has been unacknowledged for five consecutive times.



It's important to note that Zenoss Core will only send one alert based on the alerting rule's criteria. If you want multiple alerts to be sent, then you need to set up a new alerting rule with the "escalation point".

By default the alerting rules will be active 24/7, but we have the ability to schedule the active time frame for each alert.

## Schedule

We may set a schedule for each alerting rule so that the rule sends alerts only during the specified period. When editing an alerting rule, click on the **Schedule** link (in the sidebar) to view the **Active Periods** table.

The **Active Periods** table displays a list of schedules sorted by **Name** with columns for **Start**, **Duration**, **Repeat**, and **Enabled**, as shown in the next screenshot:

Edit Message Schedule	ZenUsers > mike > Alerting Rules > Escalate				
	Active Periods 				
	Name	Start	Duration	Repeat	Enabled?

To add a schedule:

1. Select **Add Rule Window** from the **Active Periods** menu.
2. Enter a descriptive name when prompted. I'm using **Weekends** as my example.
3. Click **OK** to add the new schedule to the Active Periods table.
4. Click on the name of the schedule to display the Status page.
5. From the Status page, enable the schedule and enter the **Start**, **Duration**, and interval to **Repeat**. Sample values might include:
  - Start Date: **10/30/2010**
  - Start Time: **0700**
  - Duration: **3 Days**
  - Repeat: **Weekly**

6. Click on **Save**.

The screenshot shows a web interface for configuring an alerting rule. The breadcrumb trail is 'ZenUsers > mike > Escalate > manage > Weekends'. The state is 'State at time: 2010/10/30 15:48:12'. The rule name is 'Weekends'. The 'Enabled' checkbox is checked. The 'Start' date is '10/30/2010', with a 'select' button and dropdowns for '07' hours and '00' minutes. The 'Duration' is set to '3' days, '00' hours, and '00' minutes. The 'Repeat' interval is set to 'Weekly'. A 'Save' button is at the bottom left.

This rule starts at 0700 on 10/30/2010, which is a Friday. It will be active until Monday morning. Then, the alerting will become active again on each Friday in the future.

The following table lists the available schedule settings:

Property	Description
<b>Enabled</b>	Set to <b>True</b> to enable the alerting rule during the specified time and duration.
<b>Start</b>	Specify the start date, hour, and minute. The hours are specified in 24-hour time.
<b>Duration</b>	Enter the <b>Days</b> , <b>Hours</b> , and <b>Minutes</b> to keep the alerting rule active after it starts.
<b>Repeat</b>	Available intervals are: <ul style="list-style-type: none"> <li>• <b>Never</b></li> <li>• <b>Daily</b></li> <li>• <b>Every weekday</b></li> <li>• <b>Weekly</b></li> <li>• <b>Monthly</b></li> <li>• <b>First Sunday of the month</b></li> </ul>

We may add as many schedules to an alerting rule as we need to accommodate each user's work schedule.

## Alert messages

While editing our alerting rule, we have the ability to customize the text of the alert message. To view the message template, click on the **Message** link while viewing an alert.

We can specify the **Subject** and the **Body** for both the down alert and the clear alert. As the text at the bottom of the **Message** tab indicates, the "Message Format is a Python format string. Fields are specified as %(fieldname)s."

All the event fields are listed at the bottom of the page for reference:

ZenUsers > mike > Alerting Rules > Escalate

State at time: 2010/10/30 16:16:08

Message (or Subject)

[zenoss] %(device)s %(summary)s

Body

Device: %(device)s  
Component: %(component)s  
Severity: %(severityString)s  
Time: %(firstTime)s  
Message: %(message)s  
<a href="%(eventUrl)s">Event Detail</a>

Clear Message (or Subject)

[zenoss] CLEAR: %(device)s %(clearOrEventSummary)s

Clear Body

Event: '%(summary)s'  
Cleared by: '%(clearSummary)s'  
At: %(clearFirstTime)s  
Device: %(device)s  
Component: %(component)s  
Severity: %(severityString)s  
Message:

Save

Message Format is a python format string. Fields are specified as %(fieldname)s. The list of fields available in the event database is: dedupid, evid, device, component, eventClass, eventKey, summary, message, severity, eventState, eventClassKey, eventGroup, stateChange, firstTime, lastTime, count, prodState, suppid, manager, agent, DeviceClass, Location, Systems, DeviceGroups, ipAddress, facility, priority, ntevid, ownerid, clearid, DevicePriority, eventClassMapping, monitor.

If we set the alerting rule to send a page, we can only specify a subject line for the down and clear alerts because of likely character restrictions on the pager. The following screenshot shows the available message settings for a pager alert:

Message (or Subject)
<code>[zenoss] %(device)s %(summary)s</code>
Clear Message (or Subject)
<code>[zenoss] CLEAR: %(device)s %(clearOrEventSummary)s</code>
Save
<p>Message Format is a python format string. Fields are specified as <code>%(fieldname)s</code>. The list of fields available in the event database is: <code>dedupid, evid, device, component, eventClass, eventKey, summary, message, severity, eventState, eventClassKey, eventGroup, stateChange, firstTime, lastTime, count, prodState, suppid, manager, agent, DeviceClass, Location, Systems, DeviceGroups, ipAddress, facility, priority, ntevid, ownerid, clearid, DevicePriority, eventClassMapping, monitor</code>.</p>

Feel free to experiment with the messages as you see fit.

## Event transformations

Event transformations allow us to alter an event's properties with a Python expression. For example, we can use an event transformation to:

- Suppress, drop, or otherwise alter the state of an event.
- Assign the event properties, such as severity, state, or summary, based on a conditional check of another device or event.
- Process events by location.
- Sort the incoming syslog messages for further handling.
- ... and much, much more

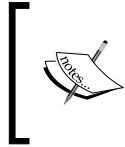
Event transformations are a flexible tool in your Zenoss Core toolbox, and how you use them will largely depend on your imagination, needs, and scripting abilities.

Zenoss Core is a Python-based application and as a result, knowing Python will drastically increase what you can do with event transformations. However, for the non-Python people among us, myself included, we can still use the event transformations to do some things. *Appendix A, Event Attributes* and *Appendix B, Device Attributes* contain event and device attributes that you can use in your transformations.

For example, if I wanted to suppress all ping events, I could write a simple transform at the `/Status/Ping` event class that looks like this:

```
evt.eventState = 2
```

We need to provide a context for the expression we're writing, so we specify `evt`. If we need to access a device attribute, the context is `device`. The attribute we want to assign a new value to is `eventState`. The `2` represents suppressed, so we'll suppress all events for the class.



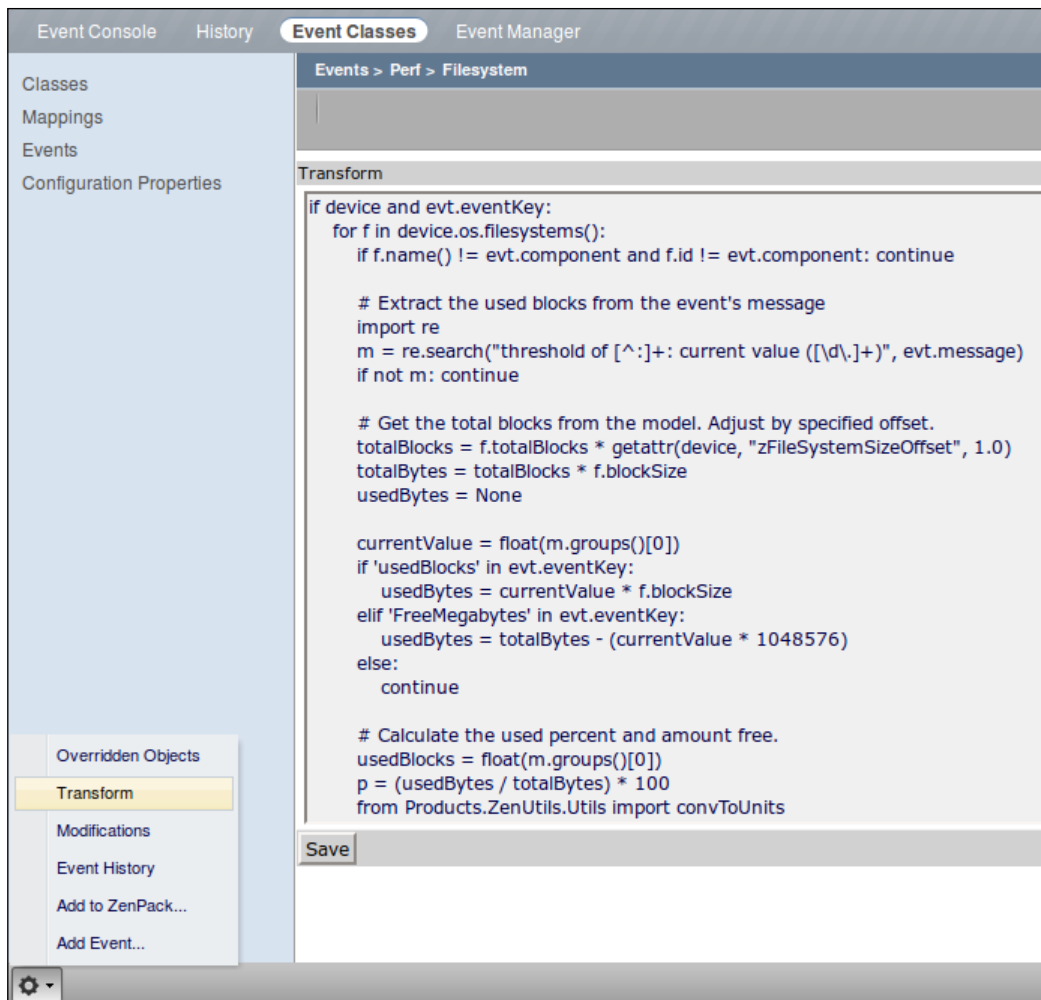
There are several event fields that you cannot alter with an event transformation, including `evid`, `dedupid`, `count`, `firstTime`, and `lastTime`. These fields are set when the event is added to the MySQL database, which is the last action in the event processing.

When Zenoss Core receives an event, it assigns properties to the event in the following order: device class, device properties, event class, and then event class mapping (if no event class can be assigned). We've already looked at the configuration properties for each of those steps already. Event transformations can be applied to either the event class or the event class mapping.

## Some event transformation examples

To pull off an event transformation, you need to know a bit of Python and how to access the Zenoss Core device and event attributes.

Let's start by reviewing an example that's included with Zenoss Core by default, by looking at the `/Perf/Filesystem` event class. To view the transform, navigate to the event class, and then select **Transform** from the Actions menu. See the following screenshot:



As you can see this transformation is a relatively complex Python script that manipulates filesystem events. As a Python coder or someone who longs to be, remember Python expects blocks to be indented with the same amount of whitespace.

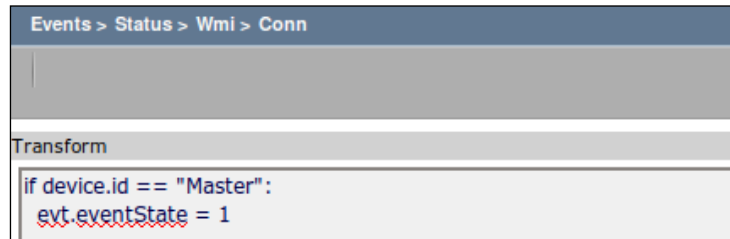
When you apply a transformation to an event class, subclasses will inherit the event transformation. Zenoss calls this: cascading event transformations.



If Zenoss Core displays your event transform code in red text, the code is invalid. The code is only evaluated after you save the transform.



And now, a simpler example. The following screenshot shows an event transformation that changes the event state for a device named **Master** to acknowledged (1) for the /Status/Wmi/Conn event class:



This transform looks to see if the device name attached to the event is "Master", and then changes the event state to acknowledged if the check is true.

The Zenoss Core community wiki provides some additional transformations that will give you more ideas: <http://community.zenoss.org/docs/DOC-2554>.

Zenoss Core provides a modified Python shell named `zendmd` that lets us interact with the Zenoss Core object database. We can use `zendmd` to test our python statements and to access the methods and attributes available to us from Zenoss Core.

## Programming in zendmd, an interactive shell

We can use `zendmd` to test our python statements and to access the methods and attributes available to us from Zenoss Core. From `zendmd`, we can write and test Python statements that manipulate the attributes of the devices:

You can use `zendmd` as a way to:

- Test event transformations
- Mass update device properties from a command line
- Test programming expressions when writing plugins or zenpacks

This section introduces the environment and provides some basic commands to get us started.

To start the `zendmd` shell, you'll need to run the following command as the `zenoss` user:

```
zendmd
```

The Zenoss dmd command shell opens and displays with a >>> prompt. Enter the following statements at the shell (exclude the commented text that begins with #):

```
zhelp()          # Display a list of objects
pprint(dir(dmd)) # Display methods available to dmd object
dir(devices)     # Display methods available to devices object
find('Coyote')   # Find the device by name
d = find('Coyote') # Assign the device to the variable d
d.deviceClass()  # Display the device class
```

The dmd object is the root of the Zenoss object database. When we execute the `dir(devices)` statement, one of the methods we return is `deviceClass()`, which we then use to print Coyote's device class. In this example, `d.deviceClass()` returns:

```
<DeviceClass at /zport/dmd/Devices/Server/Remote/devices/Coyote/
deviceClass/Remote>
```

The following script prints all the devices in the Zenoss object database with the corresponding device class. The zendmd command prompts are preserved:

```
>>> for x in dmd.Devices.getSubDevices():
...     print "%s, %s" % (x, x.getDeviceClassName())
... 
```

When working in the zendmd shell, the spacing of our code is important. Python requires that all lines of a block be indented the same number of spaces. If you make a mistake, the zendmd interpreter will display an **IndentationError: expected an indented block** error.

When you finish typing the python statement, hit enter on a blank line that's preceded by three periods (...) and the shell will evaluate the statement. The following is a sample output:

```
<Device at Crow>, /Network/Router
<Device at Fox>, /Server/Linux
<Device at Master>, /Server/Windows
<Device at Coyote>, /Server/Remote
<Device at Print Server>, /Printer
<Device at Bobcat>, /Workstation
<Device at badgerfiles.com>, /Web
```

If we want to print only the devices in the `/Server` device class, our Python statement becomes:

```
>>> for x in dmd.Devices.Server.getSubDevices():
...     print "%s, %s" % (x, x.deviceClass())
... 
```

If we want to print only the devices in the `/Server/Linux` device class, our Python statement becomes:

```
>>> for x in dmd.Devices.Server.Linux.getSubDevices():
...     print "%s, %s" % (x, x.deviceClass())
... 
```

As we run each statement, our results become very specific. We can also commit changes to the Zenoss object database from `zendmd`. Our next example finds the device named Bobcat and sets the production state to Production:

```
>>> x = find('Bobcat')
...     x.productionState = 1000
....
>>> commit()
```

The `commit()` method applies our changes to the Zenoss object database. When you commit changes from `zendmd`, the changes will be reflected in the Zenoss Core web interface, too.

For more information about using `zendmd`, check out the Zenoss Core community wiki at <http://community.zenoss.org/community/documentation/wiki/zendmd>. We'll dig deeper into `zendmd` when we review custom device reports in *Chapter 11, Writing Custom Device Reports*.

## Summary

If our toaster could plug into our network, Zenoss Core could receive events from it. In this chapter, we used Zenoss Core as a centralized place to manage events from many different sources, including syslog, Windows Event Log, shell scripts using `zensendevent`, and e-mail using `zenmail` and `zenpop3`.

Regardless of whether Zenoss Core generated the event or the event was generated by external applications, we can transform the event by modifying its attributes. This is powerful stuff and it means you can bend your monitoring environment to your will.

Alerts give events a way to get noticed, and we wrapped up our discussion on events by setting up alert notifications. That way, those pesky events, aka problems, can get fixed.

In the next chapter, we look at ways to extend our monitoring capabilities with add-on modules called `zenpacks`.

# 8

## Settings and Administration

Now that we've invested time in configuring our monitoring environment, it's wise for us to review some Zenoss Core administration tasks. Most of the features we need to discuss will be found under the Advanced menu in Zenoss Core, but we'll also jump out to the command line several times.

In this chapter, we will:

- Manage Zenoss Core users
- Create custom user commands
- Configure Zenoss Core's dashboard
- Backup and restore monitoring data
- Update Zenoss Core

Let's get started with users and groups to round out the alerting discussion from the end of *Chapter 7, Collecting Events*.

### Managing Zenoss Core users

We should set up a username for each person who will be using Zenoss Core, and all the users should log in using their user account, not as the default admin user. Individual users can be granted the same privileges as the admin account. However, working as a non-admin with an individual user account has several benefits:

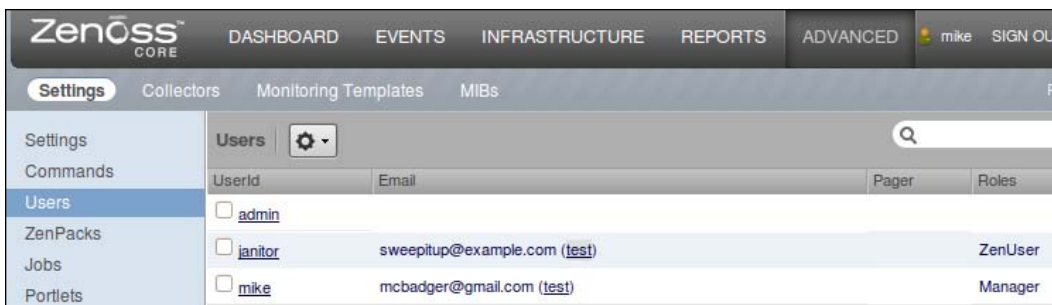
- Changes to settings are tracked via username
- Custom alerting rules can be defined per user
- Access permissions can be restricted per user

The initial setup created a user account and we've already configured that account with alerting rules in the previous chapter. Let's walk through the process of adding a user so we can review all the settings that are attached to a user.

Let's add a new user:

1. Select **Advanced** from the main menu.
2. Select **Users** from the side bar.
3. From the **Users** menu, select **Add New User**.
4. Enter the **User Name** and **Email address** in the **Add User** dialog box.
5. Click on **Submit** to create the user account.

The new username is added to the list of users (see the following screenshot) along with columns for **Email** address, **Pager**, **UserID**, and **Roles**.



Before a new user can log in, we must specify a password. To create a password and configure the account, edit the user account by clicking on the username from the **Users** table. The following table includes the fields we can set via the **Edit** Screen:

Property	Description
<b>Password</b>	Specify the new password in the first text field. Retype the password in the second box and click on <b>Save</b> to verify if the passwords match.
<b>Roles</b>	Specify a user role. Available options are Manager, ZenManager, and ZenUser.
<b>Groups</b>	If the user is a member of a defined group, select it. Groups are defined in <b>Settings</b>   <b>Users</b> .
<b>Email</b>	Enter an e-mail address if the user has to receive alerts via e-mail.
<b>Test</b>	Click the test link to send an e-mail to the e-mail address specified.
<b>Pager</b>	Enter a pager number if the user will receive alerts via a pager.
<b>Default Page Size</b>	Specify number of entries displayed in a grid listing. Default is 40.
<b>Default Admin Role</b>	Select the default role for administered objects.
<b>Default Admin Level</b>	This field is not currently used and is reserved for future use.

Property	Description
<b>Dashboard Refresh</b>	Enter the time in seconds between dashboard refreshes for the user. The default is 30 seconds.
<b>Dashboard Timeout</b>	Enter the time in seconds before the dashboard refresh times out. The default is 25 seconds.
<b>Dashboard Organizer</b>	Select the organizer view for the <b>Device Issues</b> dashboard portlet. The user can change or select a new organizer via the <b>Preferences</b> link. Available options include: <ul style="list-style-type: none"> <li>• Devices</li> <li>• Systems</li> <li>• Groups</li> <li>• Locations</li> </ul>
<b>Network Map Start Object</b>	Specify a default network from the monitored networks to map on the Network Maps view. For example, 192.168.1.1.

In order to save any changes, the user making the changes must type his password into the field labeled **Enter your password to confirm changes**.

We use roles to define a user's level of access to the system. The following table lists the available roles from the most to the least restrictive access:

Role	Access Privileges
ZenUser	View-only access to the system. Includes the Dashboard, Device List, Browse by organizers, and classes.
ZenManager	Access includes view, update, and delete. User is able to access the Management menu items and Event Console.

## Administered Objects

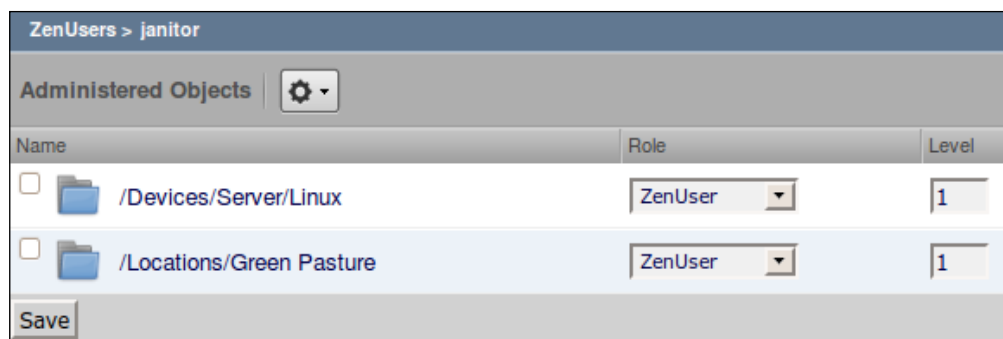
For each user, we can assign a list of administered objects, which includes devices, systems, groups, device classes, and locations. By matching users to administered objects, we have an easy way to identify who is responsible for the object. Any user with administrative rights can match users to administered objects.

To view or edit the administered objects for a user, click on the Administered Objects menu while editing a user account.

To add an object:

1. Choose the appropriate option from the **Administered Objects** menu.
2. If we add a device, the Add Device dialog box filters the list of devices as we type.
3. If we add a system, group, or location, we can choose the object from a drop-down list.

The following screenshot shows that our test user, **janitor**, is responsible for the **Linux** servers and the **Green Pasture** location.



A quirk with the administered objects is that if you assign a class to a user, the administered objects are not inherited (at least it's not displayed) at the device level. This means that if I have a device classified in `/Server/Linux`, the Administration page for the device will not show the user `janitor` (in keeping with this example) as an administrator.

If we click on the object name, Zenoss Core displays the status page for the device, system, group, class, or location. And on the **Details** page, there will be an **Administration** link. The following screenshot shows the **Administration** details for the `/Server/Linux` class and guess what you see under the **Administrators** heading?

**Linux** [SEE ALL](#)

- Devices (3)
- Events
- Configuration Properties
- Overridden Objects
- Custom Schema
- Administration**
- Modeler Plugins
- Modifications
- Monitoring Templates
  - Device (/Server/Linux)

**Define Commands** [Settings](#)

Name	Description	Command
<a href="#">DNS forward</a>	Name to IP address lookup	host \${device/id}
<a href="#">DNS reverse</a>	IP address to name lookup	host \${device/managerIp}
<a href="#">ping</a>	Is the device responding to ping?	ping -c2 \${device/managerIp}
<a href="#">snmpwalk</a>	Display the OIDs available on a device	snmpwalk -\${device/zSnmpVer} -c\${device/zSnmpCommunity} \${here/managerIp} system
<a href="#">traceroute</a>	Show the route to the device	traceroute -q 1 -w 2 \${device/managerIp}

**Maintenance Windows** [Settings](#)

Name	Start	Duration	Repeat	Start State	Stop State	Enabled?	M
------	-------	----------	--------	-------------	------------	----------	---

**Administrators** [Settings](#)

Name	Role	Level	Email
<a href="#">janitor</a>	ZenUser	1	sweepitup@example.com

The screenshot shows that the administrator for this device class is the username `janitor`, which is the assignment we just made. Now we know who to contact with questions about this device class. You may be wondering if you need to match up users with administered objects. Just because we have the capability doesn't mean we have the need. In my monitoring environments, I'm usually the only person who cares about monitoring. So, I administer everything. If you have a larger installation with multiple system administrators and want to use Zenoss Core as a way to track some detailed information about your device inventory, this might be an appropriate feature. The username in the Administrators list is a hyperlink that will take us back to an overview of the user.

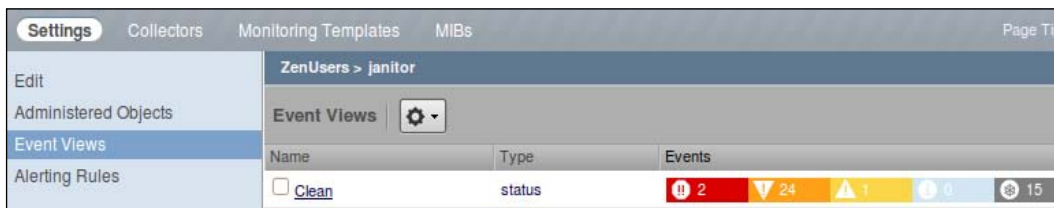
## Event Views

Zenoss Core enables each user to define a custom event view which will modify how that user sees the Event Consoles. We can filter the event view by a set of device and event-specific criteria, such as device class, production state, IP address, location, or severity, to name a few possibilities.

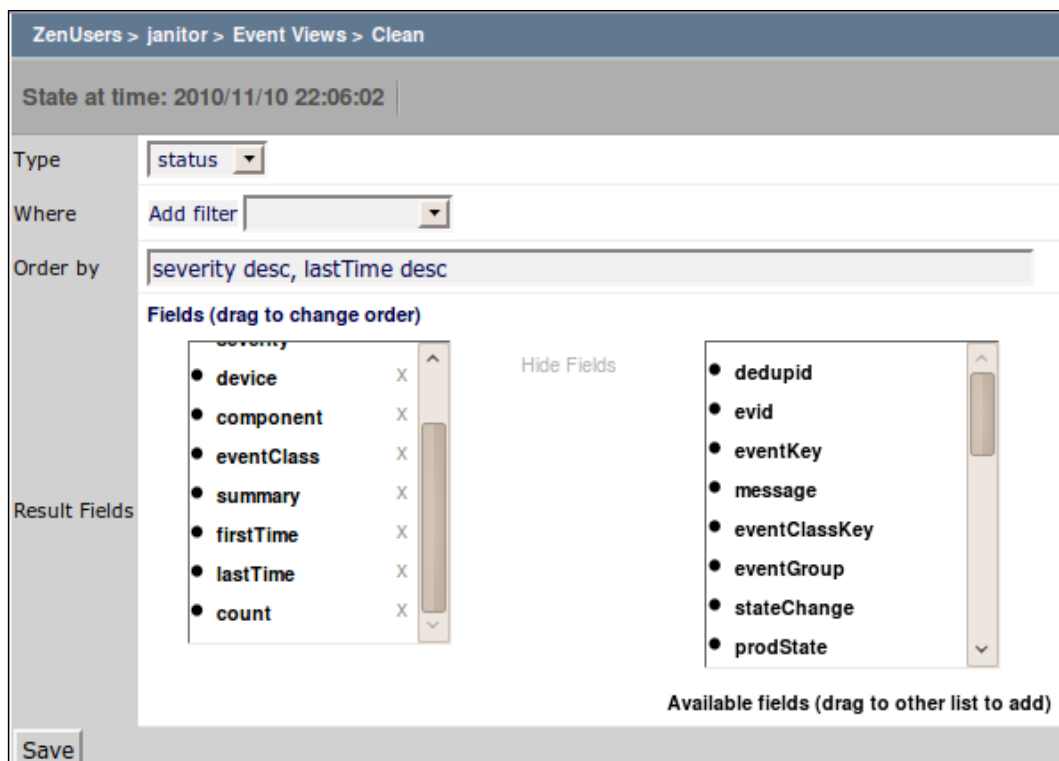


To define a custom event view for a user:

1. Edit the user account.
2. Select the **Event Views** link in the sidebar.
3. From the **Event Views** menu, select **Add Event View**.
4. Enter a descriptive name in the **Add Event View** dialog box.
5. Click on **Submit** to add the event view.



By default, the newly-created event view looks identical to the Event Console. Click on the event view name to edit the properties, as seen in the following screenshot:



The following table describes the available options for the Event Views:

Property	Description
<b>Type</b>	Select <b>status</b> to display active events and <b>history</b> to display cleared events.
<b>Where</b>	Build the filtering rules for the event view. For example, <b>Device is Coyote</b> .
<b>Order by</b>	Specify the default sort order. Sort orders are specified in pairs by field and order. Each sort order is comma separated. For example, if we specify a sort order equal to <b>severity desc, count asc</b> , the event view lists all the events from the most severe to the least severe. Within each severity, the view sorts by the count field in ascending order.
<b>Result Fields</b>	Add and remove fields to the event view.

## Groups


Everything that we have been configuring on a per user basis can also be configured on a group basis, including alerting rules, administered objects, and event views. Groups streamline our user management tasks by allowing us to configure a single set of properties and then assign users to the group. The users inherit the group's properties. You can override a group setting at the user level.

Configuring alerting rules, event views, and administered objects for a group is the same as for a user. Therefore, we won't step through any examples.

## Creating custom User Commands

User Commands are a convenient way to troubleshoot a device from the user interface and Zenoss Core includes a few such commands by default, such as **ping**, **snmpwalk**, and **traceroute**. By combining shell commands and TALES expressions, we can run commands against our devices from within the Zenoss Core web portal.

The following screenshot shows the defined commands, which you can see by navigating to **Advanced | Settings | Commands**:

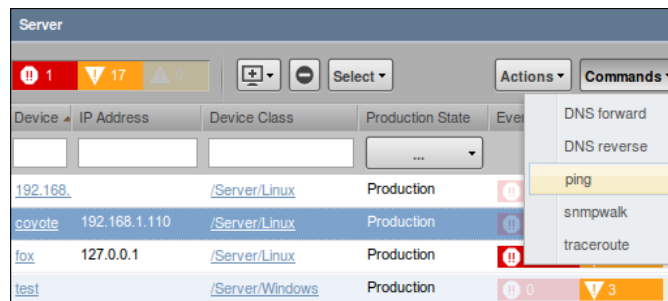


Name	Description	Command
<input type="checkbox"/> <a href="#">DNS forward</a>	Name to IP address lookup	host \${device/id}
<input type="checkbox"/> <a href="#">DNS reverse</a>	IP address to name lookup	host \${device/manageIp}
<input type="checkbox"/> <a href="#">ping</a>	Is the device responding to ping?	ping -c2 \${device/manageIp}
<input type="checkbox"/> <a href="#">snmpwalk</a>	Display the OIDs available on a device	snmpwalk -\${device/zSnmpVer} -c\${device/zSnmpCommunity} \${here/manageIp} system
<input type="checkbox"/> <a href="#">traceroute</a>	Show the route to the device	traceroute -q 1 -w 2 \${device/manageIp}

Although the terminology is similar, User Commands are different than Event Commands. We can run user commands against a device. An event command is a shell script that gets executed as a result of an event.

Let's examine the **ping** command. The actual command Zenoss Core executes against the device is **ping -c2 \${device/manageIp}**. The first half of the command construction (**ping -c2**) is a **ping** command that sends no more than two ping requests. The second half of the command (**\${device/manageIp}**) is a TALES expression that substitutes the IP address of the device into the command. The device portion defines the currently-selected device, while the part after the slash (/) gets the attribute of the device. You can find a list of the device attributes in *Appendix B, Device Attributes*.

As an example, let's ping a device in our inventory. I'm going to ping my favorite server, Coyote. So, find the device in the **Devices** view of the **Infrastructure** menu and select it. When you select the device, the **Commands** menu will become active. From the list of commands, select **Ping**. Refer to the following screenshot:



Device	IP Address	Device Class	Production State	Event
192.168.		/Server/Linux	Production	
coyote	192.168.1.110	/Server/Linux	Production	
fox	127.0.0.1	/Server/Linux	Production	
test		/Server/Windows	Production	



To run a command on more than one device at a time, select multiple devices from the list, and then run the command.

The command will run against each selected device and Zenoss Core will display the results, as seen in the following screenshot:

```

ping
===== coyote =====
ping -c2 192.168.1.110
PING 192.168.1.110 (192.168.1.110) 56(84) bytes of data:
64 bytes from 192.168.1.110: icmp_seq=1 ttl=64 time=0.537 ms
64 bytes from 192.168.1.110: icmp_seq=2 ttl=64 time=0.532 ms
--- 192.168.1.110 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.532/0.534/0.537/0.023 ms
===== fox =====
ping -c2 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data:
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.060 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.058 ms
--- 127.0.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 0.058/0.059/0.060/0.001 ms
===== 192.168.1.1 =====
ping -c2
Usage: ping [-LRUbdnqrVvAa] [-c count] [-i interval] [-w deadline]
[-p pattern] [-s packetsize] [-t ttl] [-I interface or address]
[-M mtu discovery hint] [-S sndbuf]
[-T timestamp option] [-Q tos] [hop1 ...] destination
===== test =====
ping -c2
Usage: ping [-LRUbdnqrVvAa] [-c count] [-i interval] [-w deadline]
[-p pattern] [-s packetsize] [-t ttl] [-I interface or address]
[-M mtu discovery hint] [-S sndbuf]
[-T timestamp option] [-Q tos] [hop1 ...] destination

```

As you can see, I have some device issues, including devices without IP addresses.

## Adding a User Command

Adding a new command is straight forward. Let's walk through the process of adding a new `nmap` command. We use `nmap` to determine open ports and available services on a machine. From the **Commands** page:

1. Select **Add User Command** from the **Commands** menu.
2. Enter a descriptive name (for example, **nmap**) in the **Add User Command** dialog box.
3. Click on **Submit** to add the command and display the command properties.
4. Type a short explanation in the **Description** field, such as **Display interesting ports on a device**.
5. Enter the following in the **Command** field: `nmap -v ${device/manageIp}`.
6. Enter your password to confirm the changes.
7. Click on **Save**.

The screenshot shows the 'Commands' page in Zenoss Core, specifically the 'Define Commands' tab. The form has the following fields and content:

- Name:** nmap
- Description:** Display interesting ports on a device
- Command:** nmap -v \${device/manageIp}
- Confirm Your Password:** (empty field)
- Save:** (button)

To test the command, you can run it against a device in the same way that we tested the ping command.

Let's step away from the user-level administration and settings and look at our administration options for the Zenoss Core application itself.

## System settings

The **Settings** screen, available from the **Advanced** menu, includes important information such as mail server, paging services, and conversion for severities and priorities. Before Zenoss Core can send alerts, we need to configure the SMTP and SNPP hosts' information, depending on which notification method we use.

The following table lists the available settings:

Property	Description
<b>SMTP Host</b>	The address of the SMTP server.
<b>SMTP Port</b>	The SMTP port. The default is <b>25</b> .
<b>SMTP Username</b>	If the SMTP server requires authentication to send mail, specify the username to send the mail.
<b>SMTP Password</b>	If necessary, specify the password for the SMTP username.
<b>From Address for Emails</b>	Alerts will come from the specified e-mail address.
<b>Use TLS?</b>	If the SMTP host uses Transport Layer Security, check this box.
<b>Page Command</b>	Specifies the path to the page command. The default value is <code>\$ZENHOME/bin/zensnpp localhost 444 \$RECIPIENT</code> where <code>zensnpp</code> is the Zenoss Core Simple Network Paging Protocol application and <code>\$RECIPIENT</code> is the contact of the user receiving the page.
<b>Dashboard Production State Threshold</b>	The dashboard displays devices with a threshold equal to or greater than the specified value. Default is <b>1000</b> .
<b>Dashboard Priority Threshold</b>	The dashboard displays devices with a priority equal to or greater than the specified value. Default is <b>2</b> .
<b>State Conversions</b>	<p>In descending order, Zenoss includes the following device states by default:</p> <ul style="list-style-type: none"> <li>• <b>Production: 1000</b></li> <li>• <b>Pre-Production: 500</b></li> <li>• <b>Test: 400</b></li> <li>• <b>Maintenance: 300</b></li> <li>• <b>Decommissioned: -1</b></li> </ul> <p>Some places within Zenoss use the text description while other places use the numeric state.</p>

Property	Description
Priority Conversions	<p>In descending order, Zenoss uses the following device priorities:</p> <ul style="list-style-type: none"><li>• Highest: 5</li><li>• High: 4</li><li>• Normal: 3</li><li>• Low: 2</li><li>• Lowest: 1</li><li>• Trivial: 0</li></ul> <p>Some places within Zenoss use the text description while other places use the numeric priority.</p>
Administrative Roles	Create user-defined roles. Not currently used in Zenoss Core for event processing.
Google Maps API Key	Enter the Google Maps API key to map locations on the dashboard.

Sending SMS pages via SNPP provides a unique problem in that Zenoss Core provides a utility, zensnpp, which will send the page. However, zensnpp needs network connectivity to function. And, as you can surmise, if network connectivity to the outside world is down, no SMS pages will get out. The same is true for e-mail, obviously.

There is a community contributed solution that talks about setting up paging alerts via a modem. I am including this as information only, as I've not tried the solution personally.


## Configuring Zenoss Core's Monitoring Dashboard

When a user logs in, Zenoss Core opens to a Dashboard, which contains a list of configurable, drag-and-drop portlets. Portlets are widgets that provide an overview of our monitoring environment.

We can choose from the following portlets:

- Locations (Google Maps)
- Device Issues
- Zenoss Issues
- Top Level Organizers

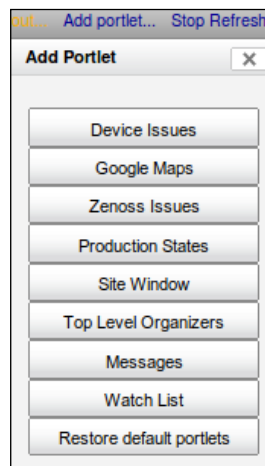
- Watch List
- Production States
- Site Window
- Messages


 For some tips on writing a custom dashboard portlet, see the Zenoss Core developers guide at [http://community.zenoss.org/community/documentation/official\\_documentation/zenoss-dev-guide](http://community.zenoss.org/community/documentation/official_documentation/zenoss-dev-guide).

The **Location**, **Device Issues**, and **Site Window (Welcome)** portlets display by default, but we can remove any of them by clicking on the asterisk at the top-right corner of the portlet to show a settings panel, as seen in the following screenshot. From the settings panel, choose the **Remove Portlet** link:



To add a portlet, click on the **Add Portlet** link at the top of the Dashboard. From the **Add Portlet** dialog box that is displayed, select the portlet you want to see on the Dashboard (refer to the following screenshot):





To arrange the portlets, click on the **Configure Layout** link at the top of the Dashboard to display the **Column Layout** dialog box. We can choose from various combinations of one, two, and three column arrangements. After we choose a layout, we can rearrange the order of the portlets on the screen by dragging and dropping a portlet to a new position on the screen.

You've probably noticed by now that the Locations portlet is complaining about a missing Google Maps API key. Let's set that up now and then we'll review each of the available portlets.

## Locations portlet with Google Maps

The **Locations** portlet makes great eye candy as it will display a point on the map for each location, link the locations, and show the highest severity event at each location.

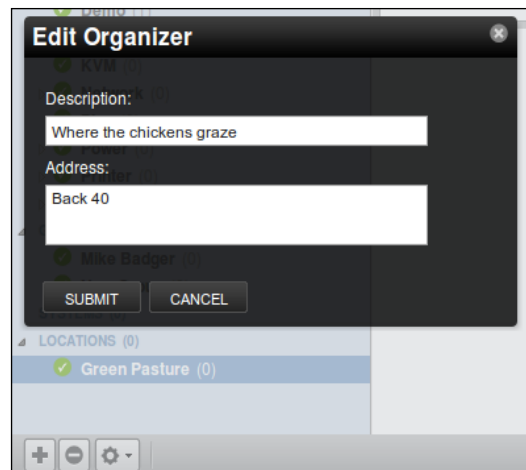
In order to make the **Locations** portlet work, we need to add a Google Maps API key to the Zenoss settings. To acquire a free Google Maps API key:

Visit <http://www.google.com/apis/maps/signup.html> and follow the registration procedure:

1. When prompted for a site URL, specify the URL of your Zenoss Core server, including the port number (for example, `http://localhost:8080`).
2. Copy the key.
3. In Zenoss Core, click on the **Advanced** menu and paste the key into the **Google API** field on the **Settings** page.
4. Save the changes.

Now when you navigate back to the Dashboard and view the Locations portlet, a map displays. Each location organizer we add to Zenoss Core has an Address field. We added locations in *Chapter 3, Device Setup and Administration*.













To edit the address of a location, select the name of the location from the **Devices** page, which can be found under the **Infrastructure** menu. Then from the **Actions** menu, select **Edit** to display the **Edit Organizer** dialog box, seen in the following screenshot:



Specify an **Address** that Google Maps will understand, which means it's a good idea to test the address out in Google Maps so you can see what Google returns for the address you enter.

## Device Issues portlet


The **Device Issues** portlet displays a list of all devices with an event using a color-coded status. Each device name is a hyperlink that links to the device's main status page. Likewise, clicking on the event redirects us to the event page for each device.

Device Issues		
Device	Events	
 fox	 1	 15
 test	 0	 3
 192.168.1.1	 0	 2
 192.168.1.0	 1	 1

We can modify the portlet **Title** and **Refresh Rate** from the settings pane.

## Zenoss Issues portlet

Zenoss Core not only monitors our network but it monitors itself. If one of the daemons has a problem, Zenoss Core displays that problem in the **Zenoss Issues** portlet:

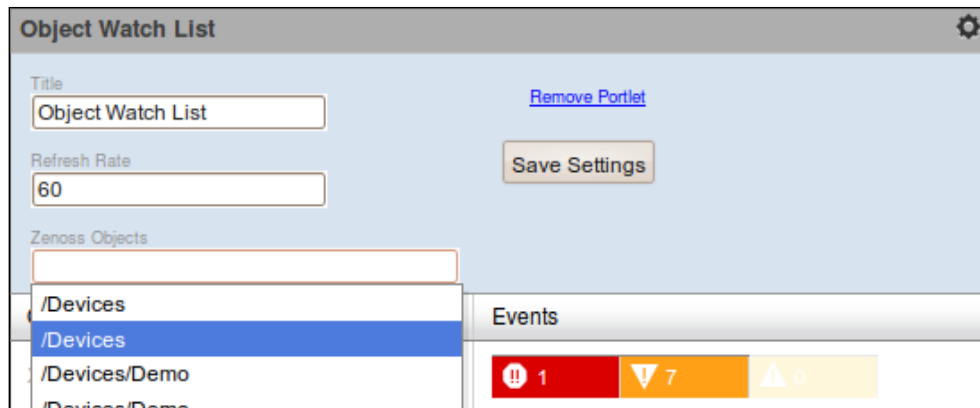
Zenoss Issues 		
Device	Daemon	Seconds
<a href="#">fox</a>	zenmail	1056208


Like the Device Issues portlet, we can only change the portlet **Title** and **Refresh Rate**.

## Watch List portlet

With the **Watch List** portlet we can monitor the status of a device class hierarchy. If a device in the selected class generates an event, the status updates on the **Watch List** portlet.

To watch a device class, select the class from the Zenoss Objects drop-down menu that appears in the portlet settings pane. We can also change the **Title** and **Refresh Rate**. The following screenshot shows the **Watch List** portlet with the settings pane expanded:






Object Watch List 

Title:  [Remove Portlet](#)

Refresh Rate:

Zenoss Objects:

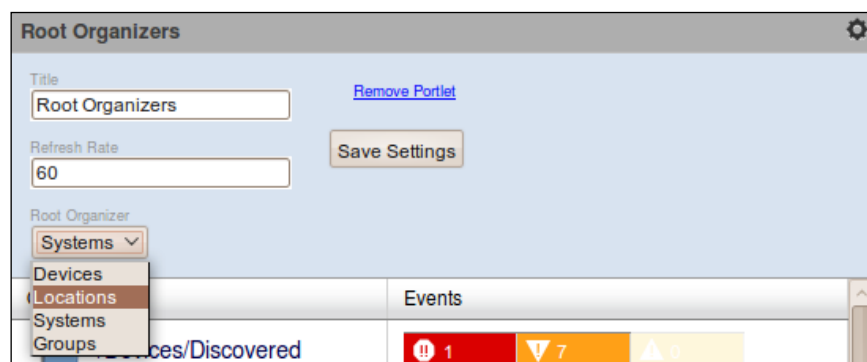
Events:  1  7  0

The **Zenoss Objects** field is a drop-down list of device classes.

## Root Organizers portlet

The **Root Organizers** portlet displays the status for the grouping we choose (locations, systems, groups, and devices). The default organizer is devices.

If you want to select a new **Root Organizer**, choose the new organizer from the settings pane of the portlet, as seen in the following screenshot:

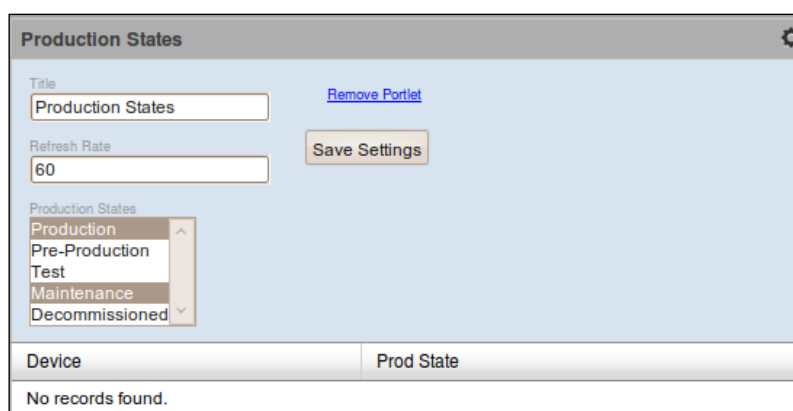


We can also change the portlet **Title** and **Refresh Rate**.

## Production States portlet

The **Production States** portlet displays the **Devices** assigned to the selected **Production State**. Default **Production States** are **Production**, **Pre-Production**, **Test**, **Maintenance**, and **Decommissioned**.

Select the **Production States** to display from the settings pane. To monitor multiple states, hold down the *Ctrl* key while selecting the states.



You may also change the portlet **Title** and **Refresh Rate**.

## Portlet permissions

We can restrict which users see which dashboard portlets by setting permissions on the **Portlets** page in the **Advanced** menu. We can choose from three levels:

- **Users with Manage DMD permission**
- **Users with View permission**
- **Users with ZenCommon permission**

Settings	Available Portlets	
Commands	Device Issues	Users with ZenCommon permission ▾
Users	Google Maps	Users with View permission ▾
ZenPacks	Zenoss Issues	Users with Manage DMD permission ▾
Jobs	Production States	Users with ZenCommon permission ▾
Portlets		
Daemons		

The three permission levels correspond to available Zenoss Core user roles: Manager, ZenManager, and ZenUser.

If you want to restrict users within the ZenUsers role from seeing a dashboard portlet, assign the portlet **Users with Manage DMD permission**. Users who are members of either the Manager or ZenManager role will be able to see all the device portlets regardless of the set permission.

## Meet the Zenoss Daemons

A daemon is a process that runs in the background on Unix systems and is comparable to what Windows calls a service. To see a list of Zenoss Core daemons, navigate to **Advanced | Settings | Daemons**. For each daemon, we see the process ID (PID), **Log File**, **Configuration**, **State**, and **Actions**, as shown in the following screenshot:

Settings	Collectors	Monitoring Templates	MIBs	Page Tips		
Settings	Zenoss Daemons					
Commands	Zenoss Daemon	PID	Log File	Configuration	State	Actions
Users	zeoctl	3625	<a href="#">view log</a>	<a href="#">view config</a>	<a href="#">edit config</a>	<div><div></div><div>Restart</div><div>Stop</div></div>
ZenPacks	zopectl	3636	<a href="#">view log</a>	<a href="#">view config</a>	<a href="#">edit config</a>	<div><div></div><div>Restart</div><div>Stop</div></div>
Jobs	zenhub	3672	<a href="#">view log</a>	<a href="#">view config</a>	<a href="#">edit config</a>	<div><div></div><div>Restart</div><div>Stop</div></div>
Portlets	zenjobs	3719	<a href="#">view log</a>	<a href="#">view config</a>	<a href="#">edit config</a>	<div><div></div><div>Restart</div><div>Stop</div></div>
Daemons	zenping	3791	<a href="#">view log</a>	<a href="#">view config</a>	<a href="#">edit config</a>	<div><div></div><div>Restart</div><div>Stop</div></div>
Versions						
Backups						

We've been working with these daemons from the very beginning through our actions within the Zenoss UI. As we look over the list of daemons, we can speculate about what some of these processes are responsible for. For example, **zensyslog** processes syslogs, **zenmodeler** creates the model of our devices based on the plugins defined for each device, and **zenping** monitors device availability.

We usually turn to the daemons when we're curious or troubleshooting. Click on the view log link to display the log file for each daemon. We can also find the logs by browsing the `$ZENHOME/log` directory.

If we want to override the default daemon behavior, we can edit the configuration by clicking on the **edit config** link, and naturally, the **view config** link displays the current configuration. To view the available options for each daemon, open a command line environment and type the name of the daemon followed by the word `help`. To see `zenmodeler`'s options, we type:

```
zenmodeler help
```

The syntax we use to enter command parameters and values via the web interface varies from the way we specify options on the command line. As an example, let's increase `zenmodeler`'s logging level from the default `INFO` to a more verbose `DEBUG`. On the command line, we use the following command as the Zenoss User:

```
zenmodeler restart --logseverity=4
```

If we use the **edit config** link via the **Daemons** page, all the possible options, with descriptions are displayed by the interface. To, set zenmodeler to log in debug mode, find the **logseverity** option, and choose **Debug** from the drop-down list. See the following screenshot:

logseverity	Debug	Info	Logging severity threshold
maxbackuplogs	3	3	Max number of back up log files; default 3
maxlogsize	10240	10240	Max size of log file in KB; default 10240
maxqueueelen	5000	5000	Maximum number of events to queue
monitor	localhost	localhost	Name of monitor instance to use for configuration. Default is localhost.

Don't forget to click the restart button for the zenmodeler daemon to pass the new configuration to the daemon. To view the results of our change, click on the view log link and scroll to the bottom. We see the results of our action and our logs now show debugging messages.

Each of the daemon configuration files can be found in the `$ZENHOME/etc` directory, and we can edit them with a text editor if we so choose.

## Maintenance Windows

If we plan to take a device out of service for maintenance or other scheduled down time, we can set up a maintenance window so that Zenoss Core does not alert us of a problem when our scheduled maintenance starts. Using a maintenance window also helps ensure that the availability reports show accurate numbers. We'll cover reports in *Chapter 10, Reviewing Built-in Reports*. We define maintenance windows via the Administration properties of devices, device classes, systems, groups, or locations.

Let's walk through a quick example using a device of your choice:

1. While viewing the device, click on the **Administration** link on the device's overview page.
2. From the **Maintenance Windows** table, select **Add Maint Window**.
3. In the **Add Maintenance Window** dialog box, enter a descriptive name: **Upgrade**.

4. Click **OK** to add the maintenance window.
5. Next, click on the name of the maintenance window to display the properties. We need to define the window.
6. The following screenshot shows the configurable options for a maintenance window.

The following table outlines the available maintenance window properties:

Property	Description
<b>Enabled</b>	Select true to activate the Maintenance window. Default is <b>False</b> .
<b>Start</b>	Enter the start date and time.
<b>Duration</b>	Specify the duration in <b>Days</b> , <b>Hours</b> , and <b>Minutes</b> .
<b>Repeat</b>	Select the interval to repeat the maintenance window. Available intervals are: <ul style="list-style-type: none"> <li>• Never</li> <li>• Daily</li> <li>• Every weekday</li> <li>• Weekly</li> <li>• Monthly</li> <li>• First Sunday of the month</li> </ul>



Property	Description
Start Production State	Specify the production state to move the devices into, once the maintenance window starts. Default is <b>Maintenance</b> .
Stop Production State	Specify the production state for the device after the maintenance window ends. The default selection is <b>Original</b> .

By setting the production state to **Maintenance**, Zenoss Core continues to monitor the device; however, it will not send any alerts.

## Adding MIBs

We haven't talked about Management Information Database's files since *Chapter 2, Discovering Devices* but at some point, we may need a MIB that Zenoss Core does not provide. If we see OID numbers (for example, .1.3.6.1.4.1.311.1.1.3.1.3) in our events, then that indicates that we need to update our MIBs. To find a MIB and its dependencies, we can search the following resources:

- Vendor's support site
- Web search for the OID
- MIB search sites, such as <http://www.mibsearch.com>

We will use the `MSFT-MIB.mib` file (found at <http://mibsearch.com>) to demonstrate how to register a MIB with Zenoss Core. First, copy the `MSFT-MIB.mib` to `/usr/local/zenoss/common/share/mibs/site/`. Second, run the following command as the Zenoss user:

```
zenmib run
```

If the command is successful, we will see the following in our command output:

```
INFO:zen.zenmib:Loaded mib MSFT-MIB
```

We can see the result of our action by logging into the Zenoss Core UI and selecting **MIBs** from the **Advanced** menu. Our newly-registered MIB displays in the table along with the number of nodes mapped by the MIB. Click on the name to display the contents of the MIB file. Refer to the following screenshot, which shows the OID mappings of an imported MIB file:

Mibs > MSFT-MIB

State at time: 2010/11/13 15:03:22

Name: MSFT-MIB

Language: SMIV1

Contact:

Description:

OID Mappings

Name	OID	Type
<input type="checkbox"/> <a href="#">dc</a>	1.3.6.1.4.1.311.1.1.3.1.3	node
<input type="checkbox"/> <a href="#">microsoft</a>	1.3.6.1.4.1.311	node
<input type="checkbox"/> <a href="#">os</a>	1.3.6.1.4.1.311.1.1.3	node
<input type="checkbox"/> <a href="#">server</a>	1.3.6.1.4.1.311.1.1.3.1.2	node
<input type="checkbox"/> <a href="#">software</a>	1.3.6.1.4.1.311.1	node
<input type="checkbox"/> <a href="#">systems</a>	1.3.6.1.4.1.311.1.1	node
<input type="checkbox"/> <a href="#">windows</a>	1.3.6.1.4.1.311.1.1.3.2	node
<input type="checkbox"/> <a href="#">windowsNT</a>	1.3.6.1.4.1.311.1.1.3.1	node
<input type="checkbox"/> <a href="#">workstation</a>	1.3.6.1.4.1.311.1.1.3.1.1	node

1 of 9 [dc](#) [show all](#) Page Size

By looking at the contents of the MIB, we can see the human friendly name each node (OID) maps to.

## Backing up and restoring monitoring data

Most of us believe in backing up our data and Zenoss Core provides some tools to make the task a bit easier. For example, we can create an on-demand backup from the Zenoss Core web interface. Navigate to **Advanced** | **Settings** | **Backups** to find it.

Create New Backup

Include MySQL events database in backup ☒

Include MySQL login information in the backup ☒

Timeout (seconds)

[Create Backup](#)

Backups

File Name	Size	Date
<input type="checkbox"/> zenbackup_20101029.tgz	10.26 MB	Fri 29 Oct 2010 08:08:51 PM
<input type="checkbox"/> zenbackup_20101029_1.tgz	10.26 MB	Fri 29 Oct 2010 08:12:04 PM
<input type="checkbox"/> zenbackup_20101029_2.tgz	10.26 MB	Fri 29 Oct 2010 08:13:25 PM

Here, you can create a backup and view your available backup files. The backup files are stored in `$ZENHOME/backups`.

As you see in the screenshot, you can choose whether or not you want to include MySQL events or include MySQL login information in the backup. The default for both options is to include the information. If you do include the login information, your database connection will be included in the backup in a plain text file.

When you create a backup, the following directories in `$ZENHOME` are included:

- `/bin`: Zenoss utilities and commands
- `/etc`: Configuration files
- `/perf`: RRD files to graph the performance of devices and daemons
- `/ZenPacks`: Add-on modules that you installed in Zenoss Core

Snagging a backup from the interface is convenient, especially if you do it just prior to upgrading. However, you probably want a more regular backup plan. Zenoss Core provides the `zenbackup` command that we can use to automate the process.

## Automating backups with zenbackup

We can schedule regular backups and the `zenbackup` command with `cron`, a Unix-based daemon.

In order for the `zenoss` user to use `crontab`, the username `zenoss` must either appear in the `/etc/cron.allow` file or must not appear in the `/etc/cron.deny` file depending on the system configuration. As an example, we'll add `zenoss` to `/etc/cron.allow` by adding a new line with the username `zenoss` on it. If `/etc/cron.allow` does not exist, we can create it as root using our favorite text editor.

When we define a `crontab` entry, we define the minute, hour, day of the month, month, or day of the week followed by the command to run. The following table shows valid values for each time, day, and date field:

Time intervals (listed in the order they appear in crontab)	Valid values
Minute	0 - 59
Hour	0 - 23
Day of Month	1 - 31
Month	1 - 12 (January = 1)
Day of Week	0 - 6 (Sunday = 0)

Let's set our `zenoss_daily` script to run at 11:30 P.M. daily. As the `zenoss` user, invoke the `crontab` editor with the following command:

```
crontab -e
```

Make the following entry into the `crontab` editor and save it:

```
30 23 * * * zenbackup --save-mysql-access
```

In this example, we're not specifying a day of the month, month, or day of the week, so we use an asterisk (\*) for those fields.

The default behavior of `zenbackup` mimics the web interface, except that we need to specify the option `--save-mysql-access` in order for `zenbackup` to include the MySQL login information with the backup. The events are included by default and the files are stored in `$ZENHOME/backups`.

Want to do more with `zenbackup`? Append the `--help` option to the command to see available options. It's probably a good idea to backup `$ZENHOME/backups` to your central backup location.

## Restoring backups with `zenrestore`

In order to restore the backup file, we need to know the backup filename and the events database password. If our system uses a non-Zenoss default events database name and credentials, then we need to specify that information in our `zenrestore` command. To see a list of all the available options, append the `--help` option to the `zenrestore` command.

To restore a backup file, run the following commands as the `zenoss` user:

```
zenoss stop
```

```
zenrestore --file=$ZENHOME/backups/zenbackup__20101113.tgz
```

```
zenoss start
```

The sequence of events here should be fairly obvious. We stop all the Zenoss Core daemons, restore the backup, and then restart the Zenoss Core daemons. We can exclude certain parts of the backup by adding options to the `zenrestore` command:

- `--no-zobd`: Do not restore the ZODB (device data)
- `--no-eventsdb`: Do not restore the events
- `--no-perfdata`: Do not restore the performance data

## Updating Zenoss Core

We won't dwell on the updating process as it may vary over time and depend on the installation method. For each new release, the Zenoss team will publish any specific steps you need to take to update your installation.

Zenoss updates frequently and as I write this sentence, the current version is 3.03.

To view version information about the current Zenoss Core installation, navigate to **Advanced | Settings | Versions** (see the following screenshot):

Software Component Versions	
Zenoss	Zenoss 3.0.1
OS	Linux (i686) 2.6.32 (Linux fox 2.6.32-25-generic #45-Ubuntu SMP Sat Oct 16 19:48:22 UTC 2010 i686)
Zope	Zope 2.12.1
Python	Python 2.6.2
Database	MySQL 5.0.45 (Ver 5.0.45)
RRD	RRDtool 1.3.9
Twisted	Twisted 8.1.0
NetSnmp	NetSnmp 5.4.1
PyNetSnmp	PyNetSnmp 0.28.14
WMI	Wmi 1.3.11

Uptimes	
Zope	13 days 22 hours 49 min 31 sec

Check For Updates	
Available Zenoss Version	3.0.3
Last Checked	2010/11/12 21:05:31.000
Check For Updates Daily	<input checked="" type="checkbox"/>
Send Anonymous Usage Info with Daily Update Check	<input checked="" type="checkbox"/>
Server Key	942f23e0-9c47-11df-9211-0019d1e547d2
<a href="#">Check Zenoss Version Now</a>	

The **Versions** tab shows us version information about Zenoss Core, its core components, and the host operating system. In the **Check For Updates** table, we can click the **Check Zenoss Version Now** button to get the latest version number, which is then reported as the **Available Zenoss Version**. You'll note that I'm two maintenance releases behind, but I refuse to update until rewrites.

Zenoss does not upgrade automatically. We need to download the update from <http://www.zenoss.com/download/>. The update procedure depends on whether or not we are using a DEB, RPM, source, or virtual appliance install.

## Summary

Now that we have concluded our Zenoss Core administration discussion, we have all the tools to implement and maintain a highly customizable monitoring solution. We looked at the various ways to control system-wide monitoring properties through daemons, system settings, and custom user commands. And yes, we reviewed all of the important backup procedures.

In the next chapter, we'll take a look at extending Zenoss Core's functionality through its ZenPack architecture.



# 9

## Extending Zenoss Core with ZenPacks

In the previous chapters, we've explored the out-of-the-box functionality of Zenoss Core. In this chapter, we'll explore ZenPacks. ZenPacks are add-on modules and are the most common way for the Zenoss community to extend Zenoss Core's capabilities. In this chapter, we will learn how to:

- Install `HttpMonitor`—a community ZenPack to monitor the response time and content of a web page
- Create a ZenPack using the `bogo_check` plugin that we configured in *Chapter 5, Custom Monitoring Templates*
- Package and distribute a ZenPack

The examples in this chapter provide a solid understanding of how you would install or create a ZenPack. As with all the other examples in this book, it's up to you to apply the concepts to your individual monitoring environments.

Let's start by installing `HttpMonitor`.

### Installing community ZenPacks

My goal will not be to review all the ZenPacks currently available or try to anticipate your exact monitoring needs. Rather, we're going to walk through the process with a relatively simple example to monitor websites with the `HTTPMonitor` ZenPack.



Find ZenPacks online at <http://community.zenoss.org/community/zenpacks>.





Many ZenPacks are community contributed and are reviewed by the Zenoss team prior to being made available for download. Still, it would be wise to install a ZenPack on a test installation of Zenoss Core before you commit an unknown change to your monitoring environment.

As of December 2010, Zenoss Core 3.0 has been out for six months, but not all ZenPacks are compatible with the latest version of Zenoss Core. Check the ZenPack download page to ensure 3.0 compatibility.


Installing a ZenPack consists of three general steps:

1. Download the ZenPack from the Zenoss ZenPack Project Site.
2. Install the ZenPack.
3. Configure the devices to use the ZenPack.


Let's demonstrate the process with the HttpMonitor ZenPack, which monitors the status and response time of a website.

## **Monitoring websites with HttpMonitor**

Always check the documentation that accompanies a ZenPack for clues about functionality and additional configuration steps. You should also make sure you meet any prerequisites. The following screenshot shows the download page for our HttpMonitor ZenPack.

 **HTTP MONITOR** VERSION 7

Created on: Sep 14, 2009 1:32 PM by [John Hamilton](#) - Last Modified: Jul 15, 2010 11:43 AM by [Matt Ray](#)

Average User Rating  
  
(1 rating)

---

**Submitted by:** Zenoss Core

---

**Description:**  
This ZenPack provides status and response time monitoring of HTTP URLs.  
  
This ZenPack is documented in the [Zenoss Extended Monitoring Guide](#).

---

**REQUIREMENTS:**  
**Zenoss Version:** 2.5 for file ending in "py-2.4.egg" and 3.0 for file ending in "py-2.6.egg"  
**ZenPack Dependencies:**  
**External Dependencies:**  
**Installation:**



---

**Source:** <http://dev.zenoss.org/trac/browser/trunk/zenpacks>


---

**Report issues:** <http://dev.zenoss.org/trac/>

---

**Attachments:**  
 [ZenPacks.zenoss.HttpMonitor-2.0.1-py2.4.egg.zip](#) (11.0 K)  
 [ZenPacks.zenoss.HttpMonitor-2.0.3-py2.6.egg.zip](#) (14.4 K)

---

54,831 Views  Tags: [network](#), [http](#), [web](#)

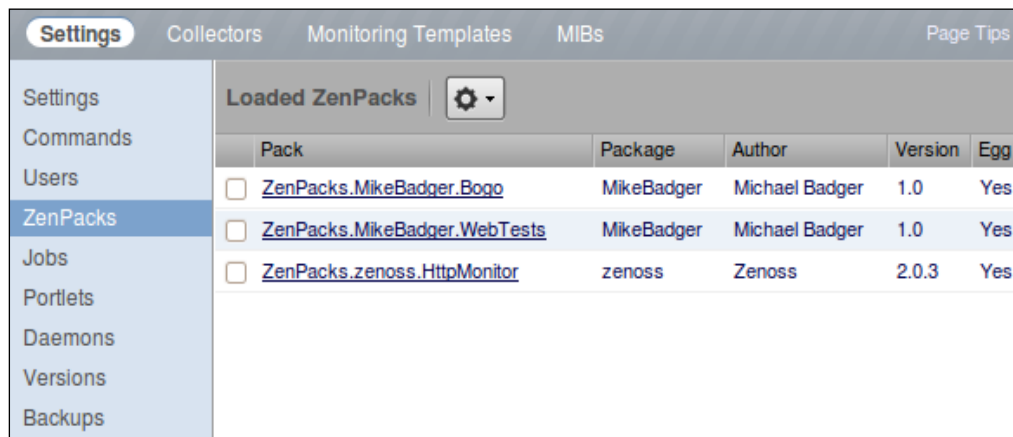
As the screenshot shows, ZenPacks are available in multiple versions. Zenoss Core 3.0 ZenPacks will have a py2.6 in the file name while ZenPacks for earlier versions of Zenoss Core will have py2.4 in the file name. The nomenclature refers to the Python version. Zenoss Core 3.0 is based on Python 2.6 and previous versions were based on Python 2.4.

Let's begin by installing the HttpMonitor ZenPack:

1. Download the HttpMonitor package from the ZenPack project site and unzip the file. When you unzip the file, you will have a Python package in .egg format.
2. From the **Advanced** menu, navigate to **Settings | ZenPacks**.
3. From the **Loaded ZenPacks** table menu, select **Install ZenPack**.
4. Browse for and select the HttpMonitor ZenPack you downloaded.


5. Click **OK** to install the ZenPack.
6. Restart Zope by running the command `zopectl restart`.

After the ZenPack installs, Zenoss displays the results of the ZenPack installation in the browser window, as shown in the following screenshot:



Pack	Package	Author	Version	Egg
<input type="checkbox"/> <a href="#">ZenPacks.MikeBadger.Bogo</a>	MikeBadger	Michael Badger	1.0	Yes
<input type="checkbox"/> <a href="#">ZenPacks.MikeBadger.WebTests</a>	MikeBadger	Michael Badger	1.0	Yes
<input type="checkbox"/> <a href="#">ZenPacks.zenoss.HttpMonitor</a>	zenoss	Zenoss	2.0.3	Yes

As you can see from the screenshot, it's possible to have multiple ZenPacks installed.



Sometimes we may have problems (for example, timeouts) installing a ZenPack from the web interface. We can also install with the `zenpack` command. Using our `HttpMonitor` example, here is the command:

```
zenpack --install ZenPacks.zenoss.HttpMonitor-2.0.3-py2.6.egg
```

```
zenoss restart
```

Note that we install the ZenPack after we unzip it.

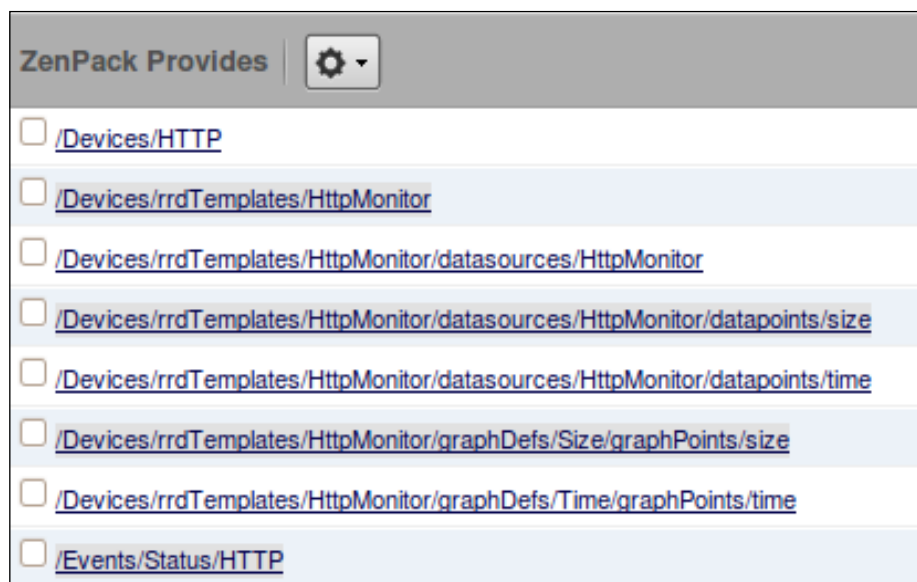
Installing the `HttpMonitor` ZenPack gives us access to new monitoring functionality but we won't be able to use that functionality until we configure a device, device class, and monitoring template to monitor with the ZenPack.

We can get a good idea of the capabilities of any ZenPack we install by viewing the list of objects that the ZenPack provides.

The `HttpMonitor` ZenPack is distributed by the Zenoss team and is now documented in the Zenoss Core Extended Monitoring Guide. For that reason, we won't cover this ZenPack feature-by-feature.

## Viewing a list of installed ZenPack objects

After we install the ZenPack, we can view its details by clicking on the package name in the list of **Loaded ZenPacks**. Among the meta information, you'll see a list of files and a list of objects provided. It's the **ZenPack Provides** list that gives us the most insight into its functionality:



Based on the names of the items in this list, we know that the ZenPack installs a new device and event classes, **/Devices/HTTP** and **/Events/Status/HTTP** respectively. The other objects include a monitoring template called **HttpMonitor** that adds a new data source to graph size and time values. You can click on each item to go directly to its configuration screen.

We've covered all the concepts required to use the ZenPack in previous chapters: device classes in *Chapter 3, Device Setup and Administration*, event classes in *Chapter 6, Core Event Management*, and templates in *Chapter 4, Monitor Status and Performance* and *Chapter 5, Custom Monitoring Templates*.

One of features of this ZenPack is that it adds its own configuration screen. Let's take a look.

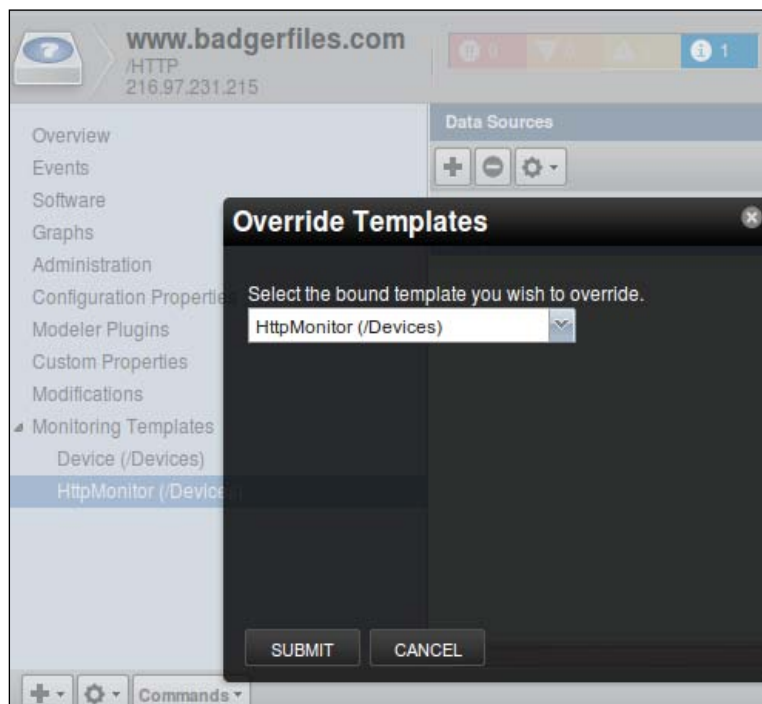
## Configuring HttpMonitor

Pick a website to monitor and add it as a device to the **/Devices/Http** class. You're now monitoring the availability and response time for the website. It doesn't get much easier than that. Add as many websites as you care to monitor.

Monitoring response time may be helpful to you. For me, it's not my primary concern. A more important use case for me is to monitor my websites to ensure the Google Analytics tracking code is always present. Sometimes during site updates the tracking code gets nuked. Among other things, HttpMonitor can check the contents of the web page for text and generate an event if the text isn't found.

In order to monitor unique content for each website, we need to override the monitoring template for each device. We learned how to override templates in *Chapter 5, Custom Monitoring Templates*. To override the default template, navigate to the device and select **Override Template** from the **Actions** menu.

The following screenshot shows the **Override** dialog box for a website that I'm monitoring. After you select the template you want to override, submit the change. Now we can configure the locally defined version of the template, and any changes we make will be isolated to the selected device, which provides us the flexibility to monitor unique properties for each site.



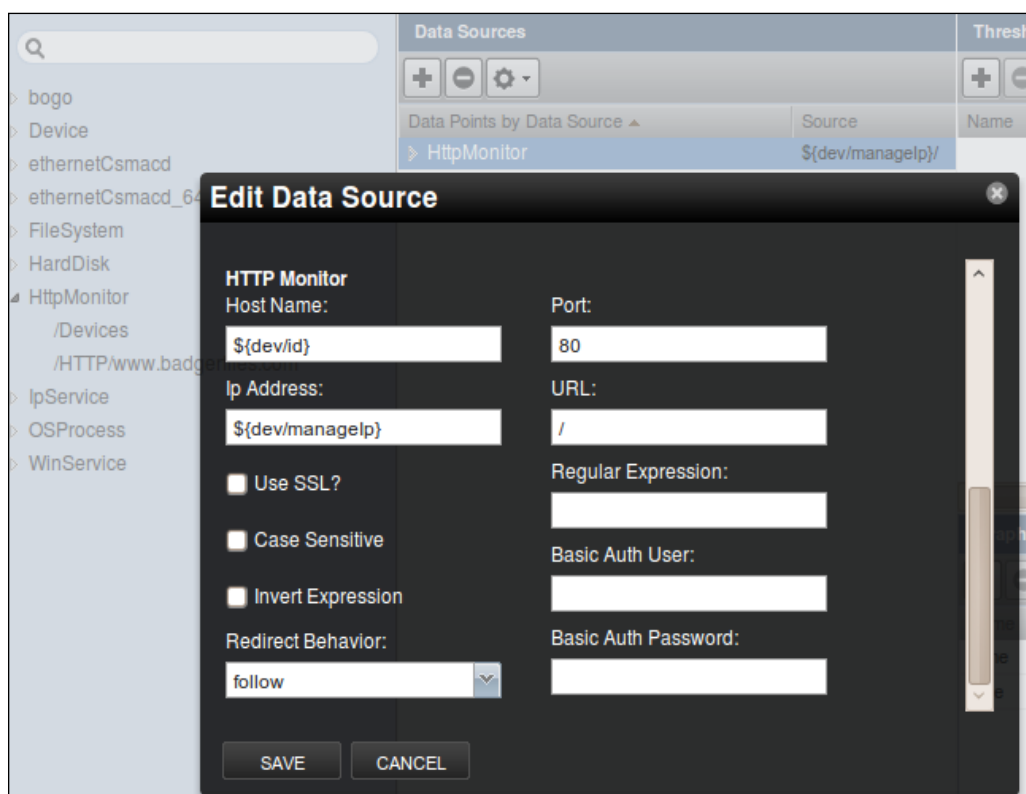
A new copy of the template will be created for the device and listed as a subitem of the HttpMonitor Template.

## Configuring HttpMonitor settings

The HttpMonitor ZenPack adds its own interface to configure specific monitoring properties, such as authentication, regular expressions, and page redirects.

Let's take a look. I'll be using my website `http://www.badgerfiles.com`:

1. Select the overridden template you just created in the last section.
2. Select the **HttpMonitor** data source and select **View and Edit Details** from the **Data Sources** menu. The **Edit Data Source** dialog box displays.
3. The first half of the **Edit Data Source** window shows us the normal configuration options that we reviewed in *Chapter 5, Custom Monitoring Templates*. If you scroll down the window, however, you will notice that we have an entire section dedicated to HttpMonitor. The following screenshot shows the options:



These options should be self-evident and more information can be found in the *Zenoss Core 3.0 Extended Monitoring Guide*. However, I'll call out a few items of interest – or at least things I want to draw your attention to.

The **Host Name** and **Ip Address** fields use TALES expressions to reference the device. That way when HttpMonitor polls the device, it just substitutes the device ID for the host name. That's why you enter your device as `www.example.com`.

The **Regular Expression** field lets us search for a specific piece of text on the page. Select the **Invert Expression** checkbox to check if the defined regular expression is not present.

HttpMonitor will generate an event when it monitors the device and finds a condition that is not true with regard to the settings.

Assuming you have something unique to monitor for all your websites, you'll need to create a local template for each device.



Want to see the code that implements the graphical interface for the settings? You can start with the `interfaces.py` and `editHttpMonitorDataSource.pt` files for real-life programming examples. To find the path to these files, view the **Files in ZenPack** section of the interface, available from **Advanced** | **Settings** | **ZenPack** | **ZenPacks.zenoss.HttpMonitor**.

## Creating a ZenPack

There may be a couple of different reasons for creating a ZenPack. Perhaps, you want to package a group of custom settings to share with other locations or test environments. Or you may want to contribute a custom monitoring solution to the Zenoss Core community.

To demonstrate the creation and distribution process, we're going to package the `bogo_check` plugin we installed in *Chapter 5, Custom Monitoring Templates*. That will give us an opportunity to show you how to distribute preconfigured objects with the ZenPack, as we saw with the HttpMonitor example earlier in this chapter.

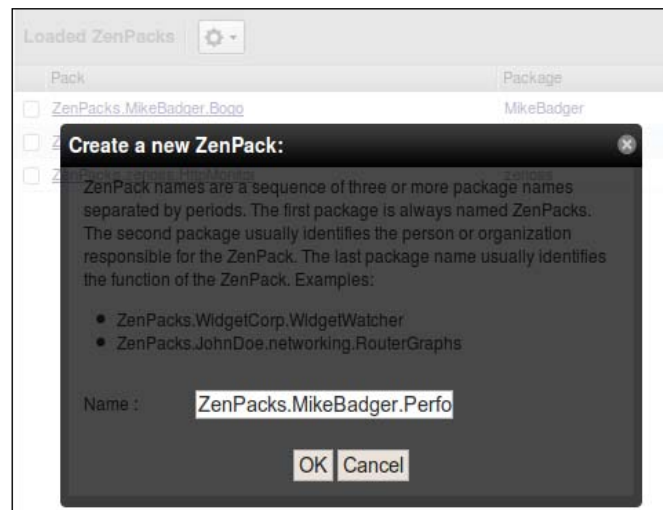
And even though we didn't write the `bogo_check` plugin, it still serves as an excellent programming example for a Nagios compliant plugin. It shows us how we can package a custom script into a ZenPack.

Creating ZenPacks is a three step process:

1. Create the ZenPack.
2. Add objects and files to the ZenPack.
3. Package the ZenPack for distribution.

Please note that even though we're going to walk through an example that adds a custom script, the procedures discussed also apply to users who may only want to create a ZenPack to distribute a report or a set of graphs. Let's get right to it and create the ZenPack:

1. From the **Advanced** menu, navigate to **Settings | ZenPacks**.
2. From the **Loaded ZenPacks** menu, choose **Create New ZenPack** to display the **Create a new ZenPack** dialog box.
3. The **Create a new ZenPack** dialog box asks you to specify a **Name** according to a Zenoss Core convention. View the following screenshot for an example. Then we'll talk about that convention.



As the dialog box describes, the name consists of three fields separated by periods:

- Field 1: **ZenPacks** is a default value.
- Field 2: Identify the name or organization of the author (for example, **MikeBadger**).
- Field 3: Identify the function of the ZenPack (for example, **PerformanceMonitorDemo**).



The naming convention helps keep ZenPack naming problems to a minimum by creating unique name spaces by the author.

After you name the ZenPack, click **OK** to create it. The properties (meta data) of the ZenPack will display, as seen in the following screenshot:

The screenshot shows the ZenPackManager interface for the ZenPack `ZenPacks.MikeBadger.PerformanceMonitorDemo`. The interface is divided into three main sections: Metadata, Dependencies, and Files in ZenPack.

**Metadata Section:**

- Name:** `ZenPacks.MikeBadger.PerformanceMonitorDemo`
- Version:** `1.0`
- Author:** `Mike Badger`
- Save** button

**Dependencies Section:**

Required?	Name	Version(s)
<input type="checkbox"/>	Zenoss	<code>3.0</code>
<input type="checkbox"/>	ZenPacks.MikeBadger.Bogo	
<input type="checkbox"/>	ZenPacks.MikeBadger.WebTests	
<input type="checkbox"/>	ZenPacks.zenoss.HttpMonitor	

**Files in ZenPack Section:**

- `/usr/local/zenoss/zenoss/ZenPacks/ZenPacks.MikeBadger.PerformanceMonitorDemo/ZenPacks/MikeBadger/PerformanceMonitorDemo/__init__.py`
- `/usr/local/zenoss/zenoss/ZenPacks/ZenPacks.MikeBadger.PerformanceMonitorDemo/ZenPacks/MikeBadger/PerformanceMonitorDemo/datasources/__init__.py`

**Save** button

Let's make a few observations about the configuration of the ZenPack. First, you can set the ZenPack **Version** number and **Author** in the Metadata section.

The **Dependencies** section will show you a list of all your installed ZenPacks and allow you to choose one of them as a dependency of your new ZenPack. Obviously, you only choose this option if your ZenPack builds upon functionality contained in another ZenPack.

The meta data and dependency information are displayed on the ZenPack's download page on the Zenoss Core website. It lets users know what they need in order to use your ZenPack.

You will also notice that the ZenPack does not contain any objects. That's our next task.

## Adding files and objects to the ZenPack

Now that we have the ZenPack directory structure defined, let's move our `bogo_check` plugin into the ZenPack directory structure. We'll need to work from the command line as the `zenoss` user to copy the plugin from the `$ZENHOME/Extensions` directory where we installed it in *Chapter 5, Custom Monitoring Templates*, into the correct ZenPack directory.

Remember in the previous section, we discussed that the unique ZenPack naming convention created name spaces that prevent community naming collisions? Well, that unique naming convention creates an ugly directory structure that we need to traverse.

Generally speaking, `$ZENHOME/ZenPacks` is our root ZenPack directory with directories that correspond to the name of the ZenPack. From there, the directory structure follows the naming convention: ZenPacks, Author, and Description.

In our example, the ZenPack is named `ZenPacks.MikeBadger.PerformanceMonitorDemo`. If you are using a different example, substitute the name of your ZenPack where appropriate in the following commands.

Here are the steps:

```
su - zenoss
```

```
cd $ZENHOME
```

```
mkdir ZenPacks/ZenPacks.MikeBadger.PerformanceMonitorDemo \
    /ZenPacks/MikeBadger/PerformanceMonitorDemo/libexec
```

```
cp Extensions/bogo_check.py \
    ZenPacks/ZenPacks.MikeBadger.PerformanceMonitorDemo \
    /ZenPacks/MikeBadger/PerformanceMonitorDemo/libexec/
```

When we modify the monitoring template, we'll find a slightly better shorthand way to reference the `bogo_check.py` plugin.

As you see, we created a `libexec` directory to store our plugin because the Zenoss Core Development Guide suggests that we keep our collector plugins in the `libexec` directory.

## Adding a new data source to the monitoring template

Next we need to update the data source of the Demo monitoring template to the new path of the `bogo_check.py` plugin:

1. Navigate to the **Monitoring Templates** page and select the **bogo/Demo** template.
2. Select the bogo data source.
3. From the **Data Source Edit** menu, select **View and Edit Details** to display the **Edit Data Source** dialog box.

Replace the original command in the **Command Template** field with the following (replace `ZenPackName` with the actual name of your ZenPack):

```
${here/ZenPackManager/packs/ZenPackName/path}/libexec/bogo_check.py
```

Refer to the following screenshot.

To test the command, enter **bogo** in the **Test Against a Device** field. Then click on **Test**. You should expect an output similar to:

```
bogo_check CRITICAL datapoint 92 | bogopoint=92%;70;90;0;100
```

Click on **SAVE** to update the data source.

**Edit Data Source**

Data Sources

Command Template:

```
${here/ZenPackManager/packs/ZenPacks MikeBadger PerformanceMonitorDemo/path}/libexec/bogo_check.py
```

**Test Against a Device**

Device Name:

bogo

TEST

SAVE CANCEL

You should notice that we didn't specify the absolute path for our command. Instead, we used another TALES expression to reference the path of `bogo_check.py` file. The TALES expression is `${here}/ZenPackManager/packs/ZenPackName/path` where `ZenPackName` is the actual name of the ZenPack. When Zenoss Core evaluates that expression, the absolute path is substituted.

This is Zenoss' preferred convention and should help prevent broken monitoring templates in future Zenoss Core versions, and it should allow the ZenPack to run regardless of where Zenoss Core is installed on the system.



You can also use the `zentestcommand` to test the data source against a device from the command line. Here's the equivalent of using **Test Against a Device** field in the **Edit Data Source** window:

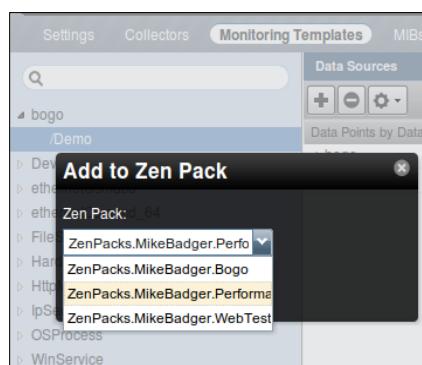
```
zentestcommand --device=bogo --data source=bogo
```

## Adding objects to a ZenPack

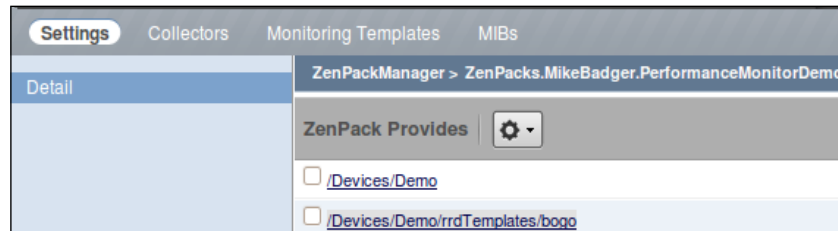
Now that we've updated the Demo monitoring template with the new data source, it's time to add the template to the ZenPack. The Zenoss Core interface makes it easy to add classes, templates, graphs, and reports to our ZenPack. It only requires that we navigate to each object to add them.


Because we just modified the Demo monitoring template, let's add the template to our ZenPack:

1. With the **Demo** template selected, select **Add to Zen Pack** from the **Actions** menu.
2. In the **Add to Zen Pack** dialog box, use the drop-down list to select the name of our ZenPack. See the next screenshot.
3. Click on the **Submit** button to add the template to the ZenPack.



4. It doesn't get much easier than that, eh? Repeat the steps for each object, such as the **Demo** device class, you wish to add to the ZenPack.
5. Want to check your work? Go back to the ZenPack screen in Zenoss Core and click on the ZenPack name to view the settings. Scroll to the bottom and view the **ZenPack Provides** section. You'll see the name of the objects listed, as seen in the following screenshot:



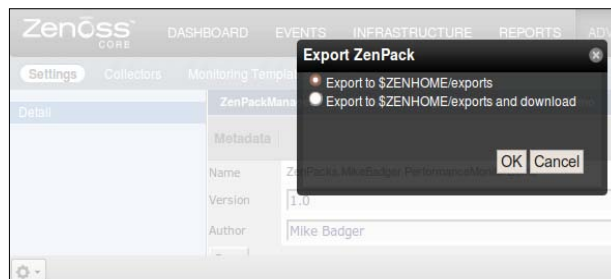
[  You can't add a device to a ZenPack because devices are unique to each installation. ]

6. We're now ready to package the ZenPack for distribution.

## Packaging the ZenPack

The first step to distributing your ZenPack is to export the ZenPack.

1. To export the ZenPack, go to the ZenPack's **Detail** page.
2. Select **Export ZenPack** from the **Actions** menu.
3. From the **Export ZenPack** dialog box, choose an export option:
  - **Export to \$ZENHOME/exports**
  - **Export to \$ZENHOME/exports and download**
4. Click on **OK** to export the ZenPack.



If you didn't choose the option to download the ZenPack, you can retrieve the ZenPack as a Python egg file from `$ZENHOME/export`.

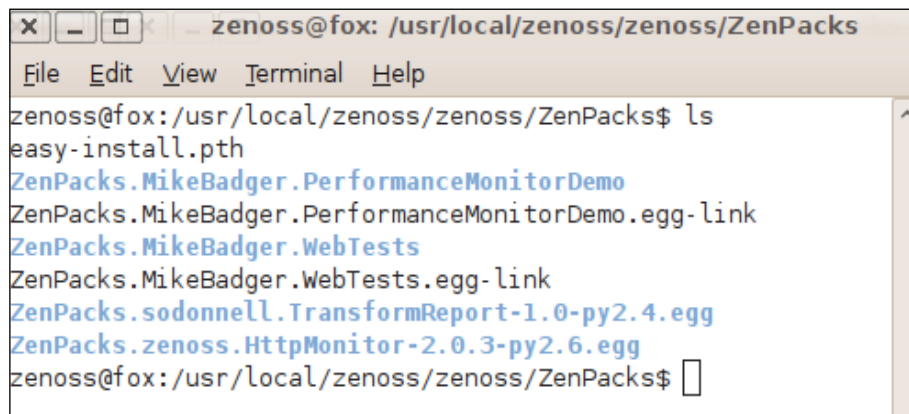
Our ZenPack is ready to distribute to other users.

## ZenPack development mode

When users install an `.egg` file, they will be unable to update the contents of the ZenPack. If you were watching closely when you added the monitoring template and the device class to the PerformanceMonitorDemo ZenPack, you noticed that HttpMonitor was not an available option, meaning we can't add objects to it.

We can, however, work around this limitation by placing the ZenPack into development mode. Depending on your point of view, this may actually be a feature in that you don't allow users to edit your ZenPack.

ZenPack development mode is a euphemism for copying some source files into the ZenPack's `.egg` directory. If we browse to the `$ZENHOME/ZenPacks` directory, and view the contents we can make some observations about the directory names. Refer to the following screenshot:



```
zenoss@fox: /usr/local/zenoss/zenoss/ZenPacks
File Edit View Terminal Help
zenoss@fox:/usr/local/zenoss/zenoss/ZenPacks$ ls
easy-install.pth
ZenPacks.MikeBadger.PerformanceMonitorDemo
ZenPacks.MikeBadger.PerformanceMonitorDemo.egg-link
ZenPacks.MikeBadger.WebTests
ZenPacks.MikeBadger.WebTests.egg-link
ZenPacks.sodonnell.TransformReport-1.0-py2.4.egg
ZenPacks.zenoss.HttpMonitor-2.0.3-py2.6.egg
zenoss@fox:/usr/local/zenoss/zenoss/ZenPacks$
```

By looking at the directory names in my ZenPack directory, we know that HttpMonitor is a ZenPack that we installed as a Python egg, as evidenced by the `.egg` extension of the directory name. The PerformanceMonitorDemo ZenPack we created does not have `.egg` in its name because that's a ZenPack we created and developed.

Let's assume that we want to add objects to the HttpMonitor ZenPack. To place the ZenPack in development, we must copy the ZenPackTemplate files to our HttpMonitor directory. Copy the following list of files to `$ZENHOME/ZenPacks/ZenPacks.zenoss.HttpMonitor-2.0.3-py2.6.egg`:

- `$ZENHOME/Products/ZenModel/ZenPackTemplate/setup.py`
- `$ZENHOME/Products/ZenModel/ZenPackTemplate/INSTALL.txt`
- `$ZENHOME/Products/ZenModel/ZenPackTemplate/MANIFEST.in`
- `$ZENHOME/Products/ZenModel/ZenPackTemplate/README.txt`

This is basically the contents of the ZenPackTemplate folder minus the `CONTENT` directory.

Now, when you choose the **Add to ZenPack** option from the **Actions** menu, the HttpMonitor ZenPack will be an option. To preserve your changes, export the ZenPack.

## Developer resources

The Zenoss team publishes a Community ZenPack Development page at [http://community.zenoss.org/community/developers/zenpack\\_development](http://community.zenoss.org/community/developers/zenpack_development). Here you'll find examples and techniques.

The Zenoss Developer's Guide provides an overview of Zenoss Core's architecture, data structure, programming techniques, and API documentation. You'll also find good reference material for zendmd, event processing, and more.

Developing for Zenoss Core and ZenPacks are topics that deserve their own book. We're only touching the fringes of those topics here.

You can find the Developer's Guide at [http://community.zenoss.org/community/documentation/official\\_documentation](http://community.zenoss.org/community/documentation/official_documentation).

## Summary

Now you have a well rounded view of ZenPacks. You can install a community ZenPack, create and distribute a ZenPack, or update an existing ZenPack. This chapter didn't make you a Zenoss developer but it did provide the knowledge to create/distribute customizations.

In the next chapter, we review the available reports.

# 10

## Reviewing Built-in Reports

Zenoss Core includes several reports that allow us to view and export status and performance information about the devices and components we monitor. Using these reports, we can troubleshoot problems, brief management, provide justification for system upgrades, or view data over time for further analysis.

In this chapter, we'll review each of the included reports, as well as create a custom graph and multi-graph reports.

### Report overview

To see a list of the default report classes, select **Reports** from the navigation menu. The report classes we will cover are:

- **Device Reports**
- **Event Reports**
- **Graph Reports**
- **Multi-Graph Reports**
- **Performance Reports**
- **User Reports**



The following screenshot shows the available reports:



As you see from the screenshot, the report classes display in the sidebar and expand to display the individual reports contained in the class.

Most of the reports share common functionality. For example, all reports include a search box that allows you to find specific information based on the data the report shows. Each report also includes an export all button that creates a CSV file of the current report view. As we move through the reports, you'll notice that many of the reports include report-specific selection criteria.

## Device Reports

The **Device Reports** class contains reports that aggregate information from all the devices. Zenoss Core includes the following device reports by default, as of version 3.0.2:

- **New Devices**
- **Device Changes**
- **Model Collection Age**
- **Software Inventory**
- **SNMP Status Issues**
- **Ping Status Issues**
- **All Devices**
- **All Monitored Components**

## New Devices

The New Devices report shows a list of devices that have been added to the Zenoss Core inventory within the past seven days. The report lists the device's **Name** and **Class** with timestamps for the **First Seen**, **Collection**, and **Change** dates:

New Devices <input type="text"/>				
Name	Class	First Seen	Collection	Change
<a href="http://www.scratchguide.com">www.scratchguide.com</a>	/HTTP	2010/12/04 13:38:23	1969/12/31 19:00:00	2010/12/04 13:38:41
<a href="http://www.badgerfiles.com">www.badgerfiles.com</a>	/HTTP	2010/12/04 13:38:22	1969/12/31 19:00:00	2010/12/04 13:38:40
<a href="#">export all</a>				

## Device Changes

When Zenoss Core models a device and detects a change, it records the date of the change. The **Device Changes** report displays all devices that have changed within the past day. The report lists the devices by **Name** and **Class**. We also see the date and time when the device was **First Seen**, last modeled, and changed.

## Model Collection Age

If a non-decommissioned device has not been updated for 48 hours, it displays on the **Model Collection Age** report. The report includes the same data as the Device Changes report: **Name**, **Class**, **First Seen**, **Collection**, and **Change**:

Model Collection Age <input type="text"/>				
Name	Class	First Seen	Collection	Change
<a href="#">192.168.1.1</a>	/Server/Linux	2010/09/10 05:48:39	1969/12/31 19:00:00	2010/09/10 05:48:39
<a href="#">Hotdog</a>	/	2010/09/10 22:32:44	1969/12/31 19:00:00	2010/09/10 22:32:44
<a href="#">bogo</a>	/Demo	2010/10/05 19:29:36	1969/12/31 19:00:00	2010/10/05 19:29:36
<a href="#">export all</a>				

This report excludes devices that have the `zSnmplibIgnore zProperty` set to `True`.

## Software Inventory

The **Software Inventory** report pulls data from two sources and organizes the information by **Manufacturer** and **Product**. If we specify the **OS Manufacturer** and **OS Product** fields on a device's overview page, that information will display on the report. The report also includes the software listed on the device's Software page, which may be collected via normal monitoring:



Manufacturer	Product	Count
<a href="#">Ubuntu</a>	<a href="#">Linux 2.6.17-10-server</a>	1

export all

The **Count** column on the report provides the total number of instances that the software product shows up.

If we click on the manufacturer link, the manufacturer overview page displays information specific to the vendor, including associated products. If we click on the product name, the product overview page displays a list of all the devices associated with that product.

## Manufacturers and Products

Zenoss Core includes a default list of Manufacturers with associated products; however we can add or remove items at will. To view the list, first click on the **Infrastructure** menu and then the **Manufacturers** menu:



Manufacturers		
<input type="checkbox"/> <a href="#">Broadcom</a>	<a href="http://www.broadcom.com/">http://www.broadcom.com/</a>	8
<input type="checkbox"/> <a href="#">Cisco</a>	<a href="http://www.cisco.com">http://www.cisco.com</a>	623
<input type="checkbox"/> <a href="#">Citrix</a>	<a href="http://www.citrix.com/">http://www.citrix.com/</a>	9
<input type="checkbox"/> <a href="#">Debian</a>	<a href="http://www.debian.org/">http://www.debian.org/</a>	7
<input type="checkbox"/> <a href="#">Dell</a>	<a href="http://www.dell.com/">http://www.dell.com/</a>	22
<input type="checkbox"/> <a href="#">EMC</a>	<a href="http://www.emc.com/">http://www.emc.com/</a>	3

As the screenshot shows, we see a list of **Manufacturers** by name, URL, and number of products attached to each listing.

In addition to organizing a list of products, we can document contact information for each manufacturer, as the following screenshot shows. To view the details of a manufacturer, click on its name from the list:

Manufacturers > ATI

**Manufacturer**

Name:  Phone:

URL:

Street 1:

City:  State/Province:

Country:  Postal/Zip Code:

**Products**

Name	Type	Product Key	Count
<input type="checkbox"/> <a href="#">ATI Display Driver</a>	Software	ATI Display Driver	0
<input type="checkbox"/> <a href="#">ATI Multimedia Center 8.2.0.0</a>	Software	ATI Multimedia Center 8.2.0.0	0

## SNMP Status Issues

The **SNMP Status Issues** report is similar to the Ping Status Issues report, except that it reports devices that have an SNMP status other than up. For devices that have an SNMP status equal to down, the report includes a count of unsuccessful SNMP connection attempts to the device.

SNMP Status Issues <input type="text" value="🔍"/>					
Name	Class	Product	State	Ping	Snmp
<a href="#">fox</a>	/Server/Linux	PowerEdge 1850	Production	Up	1
<a href="#">router2</a>	/Network/Router		Production	Up	None
<a href="#">192.168.1.1</a>	/Discovered		Production	Up	11567
<a href="#">www.badgerfiles.com</a>	/HTTP		Production	Up	None
<a href="#">www.scratchguide.com</a>	/HTTP		Production	67	None
<a href="#">export all</a>					

Because the SNMP Status Issues report shows the SNMP status for each device in one place, it makes a great report to identify all system-wide SNMP issues. The SNMP issues may be normal system downtime, but they might also correspond to improperly configured devices or monitoring setups.

## Ping Status Issues

The **Ping Status Issues** report shows a list of devices that currently have a ping status other than up. In addition to the device **Name** and **Class**, the report lists the hardware **Product** description, **State**, **Ping** status, and **Snmp** status. If the ping status is down, a count of failed ping attempts is displayed in the **Ping** status column.

Ping Status Issues					
Name	Class	Product	State	Ping	Snmp
<a href="#">192.168.1.0</a>	/Discovered		Production	6926	Up
<a href="#">www.scratchguide.com</a>	/HTTP		Production	70	None
export all					

Like the SNMP Status Issues report, the **Ping Status Issues** report shows a system-wide view of all the ping problems. The problems might be caused by device downtime, but it might also identify problems with the monitoring or device setup.

## All Devices

The **All Devices** report (refer to the following screenshot) lists each device's **Name** with additional columns for **Class**, **Product**, **State**, **Ping** status, and **Snmp** status:

All Devices					
Name	Class	Product	State	Ping	Snmp
<a href="#">192.168.1.0</a>	/Discovered		Production	6926	Up
<a href="#">192.168.1.1</a>	/Server/Linux		Production	Up	Up
<a href="#">192.168.1.1</a>	/Discovered		Production	Up	11568
<a href="#">Hotdog</a>	/		Production	Up	Up

The **All Devices** report differs from the SNMP Status Issues and Ping Status Issues reports in that it displays all devices regardless of the SNMP or ping status.

## All Monitored Components

The **All Monitored Components** report lists all the interfaces, processes, services, file systems, and routes that are being monitored for each device.

All Monitored Components <input type="text"/>				
Device	Component	Type	Description	Status
<a href="#">coyote</a>	<a href="#">/</a>	FileSystem	/	Up
<a href="#">coyote</a>	<a href="#">/boot</a>	FileSystem	/boot	Up
<a href="#">coyote</a>	<a href="#">/home</a>	FileSystem	/home	Up
<a href="#">coyote</a>	<a href="#">/proc/bus/usb</a>	FileSystem	/proc/bus/usb	Up
<a href="#">coyote</a>	<a href="#">0</a>	CPU	0	Up

The report includes the **Device** name, **Component** name, **Type**, **Description**, and component **Status**. Click on the device name to view the device's status page or click on the component name to view the properties screen for the component.

## Event Reports

The Event Reports give us a system-wide view of event classes, mappings, and heartbeats. We'll review the following reports:

- All Event Classes
- All Event Mappings
- All Heartbeats

## All Event Classes

To see a list of all the event classes defined in the system, we view the **All Event Classes** report. For each event class, the report includes the number of **SubClasses**, **Instances** of the class within the system, and the number of current **Events**:

All Event Classes <input type="text"/>			
Name	SubClasses	Instances	Events
<a href="#">/App</a>	16	71	0
<a href="#">/App/Citrix</a>	0	4	0
<a href="#">/App/Conn</a>	1	6	0

## All Event Mappings

The **All Event Mappings** report displays a list of all the event mappings currently defined in the Zenoss system. For each event mapping, the report lists the **EventClassKey**, the **Evaluation** text, and the number of current **Events**:

All Event Mappings <input type="text"/>			
Name	EventClassKey	Evaluation	Events
<a href="#">/App/Citrix/IMAService_3615</a>	IMAService_3615		0
<a href="#">/App/Citrix/IMAService_3622</a>	IMAService_3622	The server running MetaFrame Presentation Server failed to c	0
<a href="#">/App/Citrix/MetaFrameEvents_1103</a>	MetaFrameEvents_1103	An error occurred while retrieving client printer properties	0

## All Heartbeats

Heartbeats monitor the health of the Zenoss Core daemons, and the **All Heartbeats** report displays the list of current heartbeat failures by **Device** and **Component**. On the report, the **Components** column corresponds to the available daemons, such as **zenactions** and **zenstatus**. The report provides the duration of the heartbeat failure in **Seconds**:

All Heartbeats <input type="text"/>		
Device	Component	Seconds
<a href="#">fox</a>	zenmail	3473133
<a href="#">fox</a>	zenperfsnmp	120
<a href="#">fox</a>	zenwin	112

## Graph Reports

Graph reports allow us to create custom graphs based on existing performance graphs. You might recall from our ongoing discussions that graphs show up for multiple components, such as interfaces, processes, file systems, memory, and CPU. The HttpMonitor ZenPack we added in *Chapter 9, Extending Zenoss Core* with ZenPacks also added graphs to track the response time of the web page.

The value of a graph report is that we can create one view with multiple, related graphs instead of going to each device to view graphs in isolation. Zenoss Core does not include any graph reports by default.

As an example, we're going to create a report to list all the response time graphs for the websites we're monitoring with the HttpMonitor ZenPack. Let's get started.

1. Select **Reports** from the navigation menu.
2. Select the **Graph Reports** class from the sidebar.
3. We need to create the report, so click the **Add** button at the bottom of the **Report Classes** sidebar and choose **Add Graph Report**.
4. Enter a descriptive name (for example, Website Response Time) in the **ID** field of the **Add Graph Report** dialog box.
5. Click **OK** to add the report. The graph's configuration page displays.
6. From the **Device** field, select all the websites you're monitoring with HttpMonitor.
7. Select **Size** from the list of graphs.
8. Click the **Add Graph to Report** button to save the graph.

Seq	Name	Device	Component	Graph
0	<input type="checkbox"/> <a href="#">www.badgerfiles.com Time</a>	www.badgerfiles.com		Time
1	<input type="checkbox"/> <a href="#">www.scratchguide.com Time</a>	www.scratchguide.com		Time



After we click the **Add Graph to Report** button, a graph for each device is added to the Graphs table, as seen in the screenshot. Because the websites do not have any components, no values show up in the **Components** field.



The list of components will dynamically display based on the selected devices. And the list of graphs you can choose from is based on the combined device and component selection.

The prominent field on this edit screen that needs some explanation is the **Comments** field (refer to the following screenshot). As you can see, it's a block of customizable HTML that allows you to add a logo, set the report name and time.

Reports > Graph Reports > Website Response Time

**Graph Report**

Name: Website Response Time

Title:

Number of Columns: 1

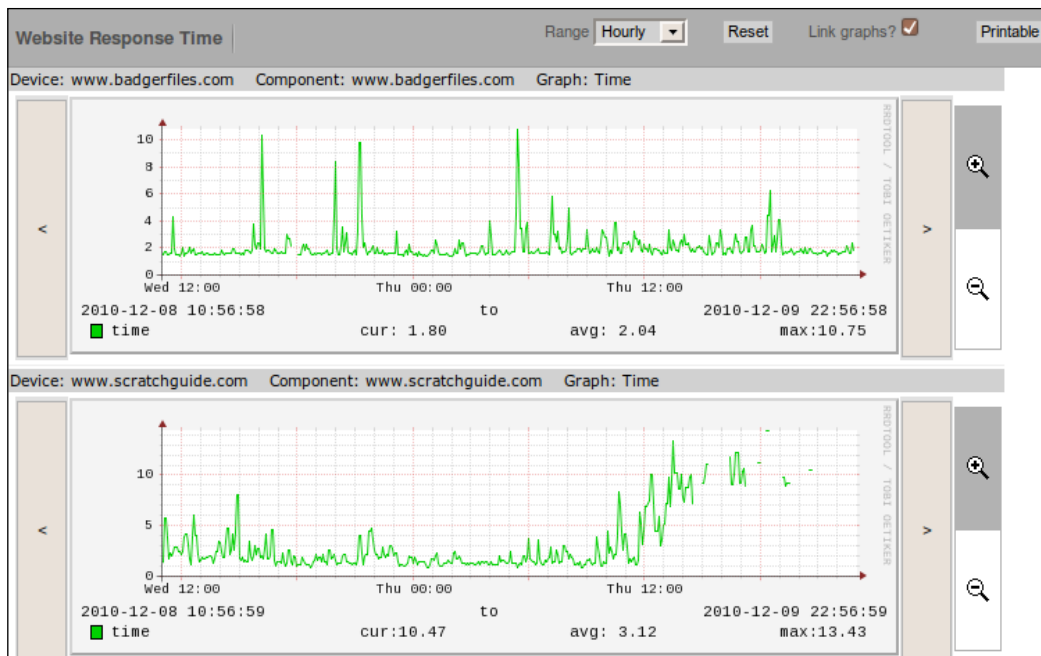
Comments: 

```
<div style="float: right;"></div>
<div style="font-size: 16pt;">${report/id}</div>
<div style="font-size: 12pt;">${now/aDay} ${now/aMonth} ${now/day}, ${now/year}<br />
${now/AMPMMinutes}
</div>
```

Save

This information will display on the printed version of the report, and if you're so inclined, you can edit the information.

To view our new report, click the **View Report** link in the sidebar or if you've already clicked out of the edit view, navigate to **Reports | Graph | Website Response Times**.



To print this report, click the **Printable** button at the top-left of the report window.

## Multi-Graph Reports

Multi-Graph reports are similar to graph reports in that we can create custom graphs. Graph reports, however, restrict us to existing performance graphs, whereas the multi-graph reports allow us to define our own data points. In addition, we can graph multiple devices and components on a single graph.

As you will see, creating a multi-graph report is much more complicated than creating a graph report, but it also gives us more flexibility. If you know the name of the data point that's collecting data from your monitoring template, you can graph it in a multi-graph report, regardless of whether or not the monitoring template is configured to show a graph.

Since we're already familiar with HttpMonitor, let's continue to work with it by turning the data points into a multi-graph report.

1. Select **Reports** from the navigation menu.
2. Click the **Add** button at the bottom of the **Report Classes** sidebar and choose **Add Multi-Graph Report**.

3. Enter a descriptive name (for example, **Website Monitoring Report**) in the **ID** field of the **Add Multi-Graph Report** dialog box.
4. Click **OK** to add the report and display the graph's **Edit** page. See the following screenshot:

The screenshot shows the 'Edit Report' page for a 'Website Monitoring Report'. The page has a left sidebar with 'View Report' and 'Edit Report' (selected). The main content area has a breadcrumb 'Reports > Multi-Graph Reports > Website Monitoring Report' and a 'State at time: 2010/12/12 18:21:51' label. Below this are input fields for 'Name' and 'Title', both containing 'Website Monitoring Report', and a 'Number of Columns' dropdown set to '1'. A 'Save' button is below these fields. Further down are sections for 'Collections', 'Graph Definitions', and 'Graph Groups', each with a gear icon. The 'Collections' section has a table with columns 'Name' and 'Number of Items'. The 'Graph Definitions' section has a table with columns 'Name', 'Graph Points', 'Units', 'Height', and 'Width'. The 'Graph Groups' section has a table with columns 'Seq', 'Name', 'Collection', and 'Graph Definition'.

## Adding Collections

Naming the report is easy. To make the graph display data, we need to create **Collections**, **Graph Definitions**, and **Graph Groups**. **Collections** allow us to filter the list of device classes, systems, groups, locations, or specific devices/components.

To add a new collection:

1. Select **Add Collection** from the **Collections** table menu.
2. Enter a descriptive name (for example, **HTTP**) in the ID field of the **Add Collection** dialog box. The actual name you use doesn't matter.
3. Click **OK** to add the collection and display the properties.
4. In the **Add to Collection** table, select an **Item Type** (for example, **Device Class**).
5. From the list of available selections, select the item (for example, **/HTTP**).
6. Set **Include Suborganizers** to **True** to recursively include all suborganizers for the selected Item Type.

7. Click the **Add to Collection** button (see the following screenshot).

Reports > Multi-Graph Reports > Website Monitoring Report > HTTP

State at time: 2010/12/12 18:28:24

Name:

**Add To Collection**

Item Type:

Device Class:

- /Discovered
- /Network
- /Server
- /Printer
- /Power
- /KVM
- /Ping
- /Demo
- /HTTP
- /Web

Include Suborganizers?:

**Collection Items**

Seq	Name	Item Description	Number of Devices/Components
<input type="text" value="0"/>	<input type="checkbox"/> <a href="#">Item</a>	<a href="#">/Devices/HTTP</a> and suborganizers	2

The **Collection Items** table updates to include a description of the item we added along with the number of devices selected.



We can add as many item types to an individual collection as necessary, and we can add multiple collections to the report.

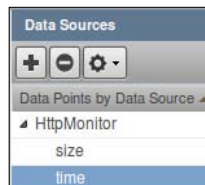
## Adding Graph Definitions

Next, we add graph definitions, which tell the report what data to retrieve for each device in the collection. We add the graph definitions from the edit report screen, which you can get back to by clicking on the report name (Website Monitoring Report) in the breadcrumb navigation at the top of the page.

1. Select **Add Graph** from the **Graph Definition** menu.
2. Enter a descriptive name (for example, **Time**) in the **ID** field of the **Add a New Graph** dialog box. The actual name of the graph doesn't matter.
3. Click **OK** to display the **Graph Definition** edit page. Now we need to add the actual graph points, which will get us access to the data.
4. From the **Graph Points** table menu, select **Add DataPoint** to display the **Add Graph Point** dialog box.
5. In the **Add GraphPoint** dialog box, type **HttpMonitor\_time** into the **datapoint** field. Then, Click **OK** to add the data point to the **Graph Points** table.



The graph point **HttpMonitor\_time** is derived from the **Data Sources** section of the HTTP monitoring template. **HttpMonitor** is the data source and **time** is the data point we want to graph. To create the graph point for the multi-graph report we separate the two values with an underscore. The following screenshot shows the **Data Sources** of the HTTP monitoring template:



The next screenshot shows the **Graph Definition** properties for the time graph point. If you recall, the **HttpMonitor** ZenPack also collects the size of the web page. I'm going to add a second graph definition for size. Feel free to do the same.

Reports > Multi-Graph Reports > Website Monitoring Report > Time

State at time: 2010/12/12 19:18:09

Name	Time
Height	100
Width	500
Units	
Logarithmic Scale	False
Base 1024	False
Min Y	-1
Max Y	-1
Has Summary	True

Save

Graph Points

Seq	Name	Type	Description
0	<input type="checkbox"/> time	DataPoint	HttpMonitor_time

For this example, we're only going to add data points to the graph, but we have the option to display thresholds on the graph just as we can create and display thresholds on the monitoring templates. To include a threshold, select **Add Threshold** from the **Graph Point** menu.

## Adding Graph Groups

Finally, we need to add a graph group. This is where we answer the all important question: Do we want to see a single graph or one graph for each device?

Navigate back to the report's edit page to add the graph group:

1. From the **Graph Groups** menu, select **Add Graph Group**.
2. Enter a descriptive name in the **ID** field of the **Add Group dialog** box (for example, **Time**).
3. Click **OK** to display the **Graph Group** properties.

4. Select the **Collection** and **Graph Definition** to apply to the graph group, which will be **HTTP** and **Time** respectively. Remember, we can have multiple collections and graph definitions set up.
5. In the **Method** drop-down list, choose how you want the graphs to display. I should point out that if you select **Separate graph for each device**, you will duplicate the Graph Report we created earlier. That's not our goal here, so choose **All devices on single graph**.
6. Click on **Save**.
7. If you're also going to graph the size data point, don't forget to add a graph group.

Reports > Multi-Graph Reports > Website Monitoring Report > Time

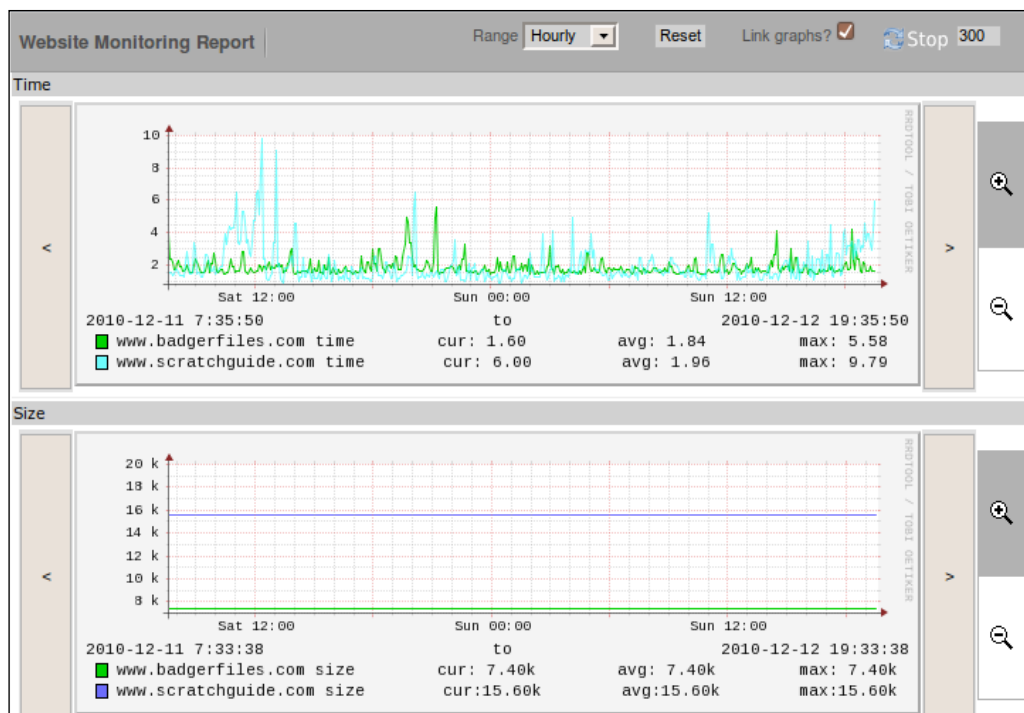
State at time: 2010/12/12 19:12:23

Name	Time
Collection	HTTP ▾
Graph Definition	Time ▾
Method	All devices on single graph ▾

Save

Our multi-graph example is simple in that we didn't add multiple collections and graph definitions; however, we have enough background to experiment with more complex multi-graphs to correlate device performance.

To view the multi-report, select the **Multi-Graph Reports** class from the **Reports** page, and then click on the report name. If you followed my example, you have a report that looks like the following screenshot:



By creating a single graph, we can compare the response time of all the websites in relation to one another at the same moment in time. Likewise for size.

When you view this graph online, you'll notice that each device on the graph has a unique color.

That should get you started with multi-graph reports. I like them for comparing data, but you might want to create report dashboards that show the file system utilization for all devices in a location for further review. You know the drill by now. It's up to you to apply this information to your specific needs.

## Performance Reports

The performance reports include a mix of the following graphs and text-based reports:

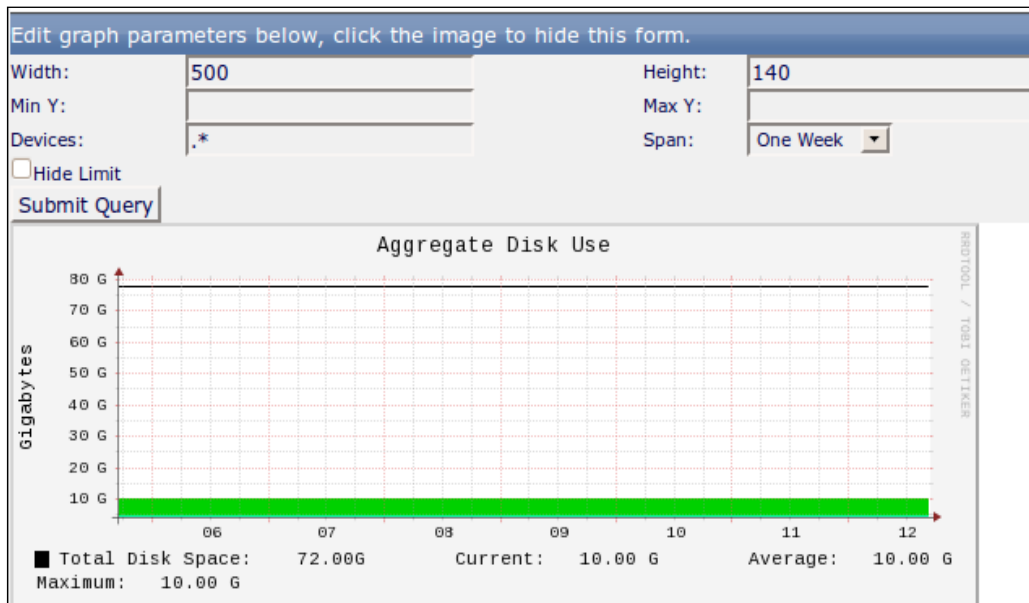
- Aggregate
- Availability
- CPU Utilization



- Filesystem Utilization
- Interface Utilization
- Memory Utilization
- Threshold Summary

## Aggregate Report

The **Aggregate Report** shows the performance graphs that combine data from all the devices into one graph, which allows us to view the cumulative performance for a component. The available aggregate reports include CPU Use, Free Memory, Free Swap, and Network Input/Output. We can customize how each graph is displayed by changing the graph parameters. To make changes, click on the graph image to open the parameters window (as shown in the following screenshot):



We can control the size of the graph by specifying a new **Width** and **Height** in pixels. In addition, we can set new minimum and maximum values for the y-axis that correspond to the unit of measurement for each graph. By default, all the devices are included, but we can view the graph for a single device by entering the device name in the **Devices** field.

The default time **Span** for the graph is **One Week**, but we may choose one day, two weeks, one month, or one year. After making the selections, press the **Submit Query** button to redraw the graph based on the new parameters.

## Availability

The **Availability** report lists each device in the inventory along with its **Systems** organizer. The availability is calculated for the selected **Event Class** and **Severity**:

Device:		Component:																	
Start Date:	12/05/2010 <span>select</span>	End Date:	12/12/2010 <span>select</span>																
Event Class:	/Status/Ping	Severity:	Error																
<span>Update</span>																			
<div>Device Availability <span>Q</span></div> <table border="1"> <thead> <tr> <th>Device</th> <th>Component</th> <th>Systems</th> <th>Availability</th> </tr> </thead> <tbody> <tr> <td><a href="#">192.168.1.0</a></td> <td></td> <td></td> <td>4.370%</td> </tr> <tr> <td><a href="#">www.scratchguide.com</a></td> <td></td> <td></td> <td>23.883%</td> </tr> <tr> <td><a href="#">www.badgerfiles.com</a></td> <td></td> <td></td> <td>29.670%</td> </tr> </tbody> </table>				Device	Component	Systems	Availability	<a href="#">192.168.1.0</a>			4.370%	<a href="#">www.scratchguide.com</a>			23.883%	<a href="#">www.badgerfiles.com</a>			29.670%
Device	Component	Systems	Availability																
<a href="#">192.168.1.0</a>			4.370%																
<a href="#">www.scratchguide.com</a>			23.883%																
<a href="#">www.badgerfiles.com</a>			29.670%																

The default report gives us the availability percentage for the past seven days for the **/Status/Ping** event class based on a **Severity** of **Error**. We can change the reporting criteria based on the following options:

Report Filter	Description
<b>Device</b>	Enter a device name to limit the report to a single device.
<b>Component</b>	Enter a component name from the device OS tab. Zenoss returns devices that match with the specified component.
<b>Start Date</b>	Specify the first day of the report.
<b>End Date</b>	Specify the last day of the report
<b>Event Class</b>	Select the type of the event to report on. For example: /Status/SNMP.
<b>Severity</b>	Select the event severity to use when calculating availability.

After we enter the report criteria, we can click on the **Update** button to view the new report.

## CPU Utilization

The CPU Utilization provides the **Load Average** and the **Percent Utilization** for each device. If Zenoss Core is not collecting CPU performance statistics for a device, the **Load Average** and **Percent Utilization** values display as **N/A** (refer to the following screen shot):

Device Class:	/	Device Filter:	
Start Date:	12/05/2010	select	End Date: 12/12/2010
Summary Type:	Average		select
Update			
CPU Utilization			
Device	Load Avg	% Util	
coyote	0.0	1.0	
device01	N/A	N/A	

By default, the report displays the previous seven days. However, we can specify a custom **Start Date** and **End Date** for the report. We can also choose one of the following for **Summary Type: Maximum** or **Average**. Maximum displays the maximum Load Average and Percent Utilization for the date range while the Average summary type provides average Load Average and Percent Utilization calculations.

## Filesystem Utilization

All monitored file systems are included in the **Filesystem Utilization** report. For each file system **Mount** point, the report includes the **Device**, **Total bytes**, **Used bytes**, **Free bytes**, and **Percent Utilization** (as seen in the next screenshot). If Zenoss Core does not know a value, it populates the report values with **N/A**.

Device Class:

/

Device Filter:

Start Date:

12/05/2010

select

End Date:

12/12/2010

select

Summary Type:

Average

Update

Filesystem Utilization

Device	Mount	Total bytes	Used bytes	Free bytes	% Util
<a href="#">coyote</a>	<a href="#">/home</a>	18.3GB	5.1GB	13.2GB	28
<a href="#">coyote</a>	<a href="#">/boot</a>	896.8MB	80.2MB	816.6MB	9
<a href="#">coyote</a>	<a href="#">/</a>	53.2GB	4.1GB	49.0GB	8

The default date range of the report includes the previous seven days, but we can specify our own **Start** and **End Dates**. We can further filter the report output by showing the Maximum or Average usage statistics by choosing the appropriate option from the **Summary Type**.

This report gives us a single view of file system utilization site-wide or across a single class, which may help identify the need for more disk space.

## Interface Utilization

The **Interface Utilization** Report (seen in the following screenshot) includes all monitored interfaces. For each **Interface**, the report includes the **Device**, **Speed**, **Input**, **Output**, **Total** throughput, and **Percent Utilization**. The report lists N/A for any unknown values.

Device Class:	/	Device Filter:				
Start Date:	12/05/2010 <a href="#">select</a>	End Date:	12/12/2010 <a href="#">select</a>			
Summary Type:	Average					
<a href="#">Update</a>						
<b>Interface Utilization</b>						
Device	Interface	Speed	Input	Output	Total	% Util
<a href="#">covote</a>	<a href="#">eth0</a>	10.0MB	1.3KB	1.2KB	2.5KB	0.0
<a href="#">export all</a>						

The default date range of the report includes the previous seven days, but we can specify our own **Start** and **End Dates**. We can further filter the report output by showing the Maximum or Average usage statistics by choosing the appropriate option from the **Summary Type**.

## Memory Utilization

The **Memory Utilization** Report (see the next screenshot) includes all the devices and provides the following memory statistics: **Total**, **Available**, **Cached**, **Buffered**, and **Percent Utilization**. Like several of the performance reports, known values are displayed, while N/A is displayed for unknown values.

Device Class:

Device Filter:

Start Date:

End Date:

Summary Type:

Memory Utilization

Device	Total	Available	Cache Memory	Buffered Memory	% Util
<a href="#">coyote</a>	248.6MB	100.8MB	32.3MB	64.0MB	59.5
device01	0.0B	N/A	N/A	N/A	N/A

The default date range of the report includes the previous seven days, but we can specify our own **Start Date** and **End Date**. We can further filter the report output by showing the Maximum or Average usage statistics by choosing the appropriate option from the **Summary Type**.

## Threshold Summary

To see a list of the devices that have crossed their performance threshold, we run the **Threshold Summary** Report (refer to the following screenshot). For each **Component** listed, the report includes the **Device**, **Event Class**, and a **Count** of the threshold violations, the **Duration**, and **Percent Utilization**:

Start Date:	08/01/2010	select	End Date:	12/12/2010	select	Event
Class:	/Perf					
Update						
Threshold Summary						
Device	Component	Event Class	Count	Duration	%	
export all						

The report displays the previous seven days by default, but we can specify a custom **Start Date** and **End Date**. The default **Class** is **/Perf**, which includes all the performance class events. However, we can limit the report to the following event subclasses: **/Perf/CPU**, **/Perf/Memory**, **/Perf/Filesystem**, **/Perf/Interface**, **/Perf/Snmp**, and **/Perf/XmlRpc**.

## User Reports

The **User Reports** organizer includes user-centric reports. We'll review the **Notification Schedules** report.

## Notification Schedules

The **Notification Schedules** report (see the next screenshot) displays each alerting rule by name along with the assigned user. The other fields on the report include alert delays, **Active** status, alert **Duration**, and **Next Active** window:

Notification Schedules

User	Rule	Delay	Active?	Next Active	Duration	Repeat
<a href="#">janitor</a>	<a href="#">sdf</a>	0	False	<a href="#">Never</a>	Forever	Never
	severity >= 4 and eventState = 0 and prodState = 1000					
<a href="#">mike</a>	<a href="#">Test</a>	0	False	<a href="#">Never</a>	Forever	Never
	severity >= 4 and eventState = 0 and prodState = 1000					
<a href="#">mike</a>	<a href="#">Escalate</a>	0	True	<a href="#">Now</a>	01:00	Never
	(prodState = 1000) and (eventState = 0) and (count >= 6) and (severity >= 4)					
<a href="#">export all</a>						

Each alert includes two rows on the report, and on the second row, we see the actual alert criteria, which makes this an ideal report to troubleshoot problems with alert notifications.

## Summary

I know, some of you find reports boring. But for the critical thinkers among us, reports provide the opportunity to learn something from our monitoring activities. This chapter shows us where to find a well rounded view of your monitoring activities, including devices, events, and performance. We build custom graph and multi-graph reports so that we can compare monitoring performance among devices.

In the next chapter, we take Zenoss Core reporting to the next level and review custom device reports.



# 11

## Writing Custom Device Reports

In the previous chapter, we reviewed each of Zenoss Core's canned reports, and even though we could customize some of them, we were still constrained as to what each report could contain. In this chapter, we're going to query the Zenoss Core data model to create custom reports. And we'll do it from the graphical interface provided by the Custom Device Report interface. Our topics include:

- Custom Device Reports and how to use `zendmd` to explore the data model
- Scheduled report delivery with `reportmail`

Let's start with a simple custom device report.

### Creating Custom Device Reports

When you view the **Reports** page in Zenoss Core, you should notice an empty report class called **Custom Device Reports**. The custom device report provides an interface that lets us query Zenoss Core for devices. Each device that matches the query is listed as a row on the report.

Let's get started with a simple report to show uptime values for all devices, and then we'll talk through the options.

With the **Custom Device Reports** organizer selected:

1. Select **Add Custom Device Report** from the **Add Report** button (at the bottom of the **Report Classes** sidebar).
2. In the **Create Custom Device Report** dialog, type a name for your report in the **ID** field. I'm going to use **Uptime**. When you click on **Submit**, the edit screen is displayed.



3. Enter the following values for the Uptime report, as seen in the following screenshot. Leave all other fields set to their default values.
  - **Title: Uptime**
  - **Sort Column: Name**
  - **Columns: uptimeStr, getId**
4. To view the report, click the **Save** button, and then click the **View Report** link (in the sidebar).

The following screenshot shows the edit screen of our report:

Name	Uptime
Title	Uptime
Path	/
Query	
Sort Column	Name
Sort Sense	asc
Columns	getId uptimeStr
Column Names	Name Uptime

The next screenshot shows what our report looks like when we view it:

REPORT CLASSES (24)

Device Reports (9)

Custom Device Reports (2)

Hardware Inventory by Group

Uptime

Graph Reports (1)

Multi-Graph Reports (1)

Performance Reports (7)

User Reports (1)

Event Reports (3)

Device Report

Name	Uptime
<a href="#">device01</a>	Unknown
<a href="#">router02</a>	Unknown
<a href="#">web03</a>	Unknown
<a href="#">fox</a>	Unknown
<a href="#">router2</a>	Unknown
<a href="#">192.168.1.1</a>	Unknown
<a href="#">Hotdog</a>	Unknown
<a href="#">bogo</a>	Unknown
<a href="#">test</a>	Unknown
<a href="#">192.168.1.0</a>	Unknown
<a href="#">192.168.1.1</a>	Unknown
<a href="#">www.badgerfiles.com</a>	Unknown
<a href="#">www.scratchguide.com</a>	Unknown
<a href="#">covote</a>	71d:17h:59m:21s

1 of 14

<< < > >>

Unknown

show all

export all

Page Size 40

As you see, we created a simple report that lists each device along with its uptime value. In this report, we use the **Column** field to do the heavy lifting. We specified the device attributes of `uptimeStr` and `getId`, which give us a human readable value of the device's uptime and the name of the device, respectively.



A good place to find device attributes is in *Appendix B, Device Attributes* which lists common attributes you can use with TALES expressions.

Our report shows columns for **Name** and **Uptime**, which are the values we specified in the **Column Names** field. For each attribute specified in the **Columns** field, we can define a corresponding friendly **Column Name**. List the column names in the same sequence as the columns.

Now that we've seen how easy it is to get started with a simple report, we'll preface our more involved examples by reviewing the available report fields.

## Custom Device Report fields

The following table shows the available fields on the custom device reports:

Field	Description	Example
Name	Descriptive name for the report. The name doesn't display on the report.	Uptime
Title	The title displays on the report.	Uptime Report
Path	Specify the device class to query. The default is <code>/</code> , which will include all devices.	<code>/Server/Linux</code>
Query	Use the query field to filter and select data. Accepts a python or TALES expression.	<code>here.hw.cpus.countObjects() &gt; 0</code> <code>here.sysUpTime() &gt; -1</code>
Sort Column	Leave blank to select all devices. Specify the column that the report is sorted on. Can either be the Column or Column Name value.	<code>here.comments != ""</code> Name <code>getId</code>

Field	Description	Example
Sort Sense	Apply the specific sort order to the value in Sort Column. Can either be ascending or descending.	asc desc
Columns	Specify the data to display for each device returned in the query. This can take a device attribute or a python expression. Python expressions must be prefaced with <code>python:</code> . Enter one value per line.	UptimeStr python:dev.hw.totalMemoryString()
Column Names	Define the human-friendly names that correspond to each value you entered in Columns.  Enter one value per line.	Uptime Total Memory

There is minimal error checking done on the data you enter. If you enter a query with an invalid syntax, the **View Report** option will tell you about the error.

If you enter more column names than you specify columns, the interface will turn all the column names red when you save the report.

Let's refine our uptime report a bit more, so we can figure out exactly how we can build meaningful queries.

## Building Custom Device Report queries

Our current version of the uptime report returns the uptime value for each device. Because we didn't specify a query, the report includes all devices and shows us values for devices that aren't reporting an uptime. That's what all the "Unknown" values are on the report.

To improve our report, let's exclude those devices from our report. The query that will exclude devices with an unknown system uptime is:

```
here.uptimeStr() != "Unknown"
```

Let's examine the query from left to right. The `here` portion evaluates the expression on the currently selected device. The custom device reports will retrieve all devices that are included in the specified **Path**.

The `here.uptimeStr() != "Unknown"` query returns the system uptime value for the selected device and checks to see if it is not equal to "Unknown". If the statement evaluates to true, then the device is included in the report.

But how do you know which query to use when building a report? To help us build queries, we turn to the interactive shell `zendmd`.

## Using `zendmd` to test report queries

We got our first look at `zendmd` in *Chapter 7, Collecting Events* as a resource for determining event transformations, and it came up again in *Chapter 9, Extending Zenoss Core with ZenPacks* when we talked about programming ZenPacks. We're going to explore the `zendmd` shell a bit deeper this time to help us figure out how we can identify the devices with unknown system uptime values.

As the `zenoss` user, run `zendmd`. Let's start with a device that we know is reporting uptime. In my test network, that device name is `Coyote`. We'll run through the command and then we'll talk about what happened.

From `zendmd`, run the following commands:

```
d = find("Coyote")
```

Hopefully, you remembered to use a device on your network. otherwise, your assignment will fail. You can check to see what's assigned to the variable `d` with the command:

```
print d
```

If that command returns the value `None`, you need to retry your `find` command using a correct device name. Don't forget to wrap the device name you want to find in quotes.

Now we need to figure out which object holds the uptime value. `zendmd` has a built-in auto-complete feature. So, if you type this command, you will see all the available methods that you can run against the device:

```
d.<tab> <tab> <y>
```

Pressing the *Tab* key twice is the way to get `zendmd` to auto-complete the command. In this scenario, it will ask if you want to display all 605 possibilities. Pressing `y` for yes will page through the results.

Unless you're bored, you may not want to page through hundreds of pages of possible objects. It's probably more fruitful to guess at the first letter of the object you need. In our uptime report, we use the `uptimeStr` attribute to display the system uptime value. So, give it a try:

`d.u<tab> <tab>`

The output is as follows:

```
>>> d.u
d.undoable_transactions(      d.updateProcesses(
d.unindex_ips(                d.uptimeStr(
d.unindex_object(            d.urlLink(
d.unlock(                     d.userCanTakeOwnership(
d.unrestrictedTraverse(       d.userCommands(
d.unsetSendEventWhenBlockedFlag( d.userdefined_roles(
d.upToOrganizerBreadCrumbs(    d.users_with_local_role(
d.updateDevice(
>>> d.u
```

This command provides several methods that begin with the letter "u" and `uptimeStr` is one of the options. Let's finish out the command:

`d.uptimeStr()`

The result of the command is the uptime value for the device, as seen in the following screenshot:

```
>>> d.uptimeStr()
'79d:17h:02m:27s'
>>>
```

So far so good, but we're actually looking for the opposite condition. Let's see what an unknown uptime value looks like. In my network, the device `Fox` is not reporting system uptime, so I can start the process over again:

```
d = find("Fox")
d.uptimeStr()
```

This results in an **Unknown** message, which is what we want to see:

```
>>> d.uptimeStr()
'Unknown'
>>>
```

At this point we have what we need to write our query for the custom device report or at least test it out. Edit the uptime report and enter this new value in the **Query** field: `here.uptimeStr() != "Unknown"`.

Save and view the report. It works. All the devices that have an unknown system uptime value are excluded from the report.

What if we mistype the query and enter this instead: `here.uptimeStr != "Unknown"`. Go ahead and give it a try.

The query brings back all devices, which means it's wrong. If you pop back into zendmd and run `d.uptimeStr`, you'll notice that you get an entirely different result than we did before. The lesson here is simple. The parentheses matter, so when in doubt, test the output of your functions with and without the parentheses.

Incidentally, this would be an equivalent query to exclude all unknown values: `here.sysUpTime() > -1`.

The `sysUpTime` function returns the uptime value in an integer format and `-1` is equal to `Unknown`. The `uptimeStr` function translates the `sysUpTime` value to a friendly format. In other words, there may often be more than one way to get your data and a little exploration may lead to enlightenment.

What if you wanted to create a report that listed all devices with a serial number? What would that query look like? Try this:

```
here.hw.serialNumber != ""
```

Let's explore this command using auto-complete in zendmd. Type `d.hw <tab><tab>`. The following screenshot shows the result:

```
>>> d.hw(
__builtins__  dmd      login      setLogLevel  version
app           edit      logout      sh            zhelp
cleandir      find      me          shell_stderr  zport
commit       getFacade pprint      shell_stdout
d            grepdir   printNets   socket
devices      history   reindex     sync
>>> d.hw(
```

Notice that there is an open parenthesis after **hw**. This is our cue in zendmd that we may be able to access other objects, so we replace the opening parenthesis with a dot and type the following command:

```
d.hw.s <tab><tab>
```

The output is as follows:

```
>>> d.hw.s
d.hw.saveCustProperties(      d.hw.setSiteManager(
d.hw.saveZenProperties(      d.hw.setTitle(
d.hw.security                d.hw.setZenProperty(
d.hw.selectedRoles           d.hw.smallRolesWidget
d.hw.sendEventWhenResultBlocked( d.hw.snmpIgnore(
d.hw.serialNumber            d.hw.snmpIndex
d.hw.setDescription(         d.hw.sub_meta_types
d.hw.setProduct(             d.hw.superValues(
d.hw.setProductKey(
>>> d.hw.s
```

We see that **serialNumber** is one of our options. The other place we can find useful device data is in the `os` object, such as this report query:

```
here.os.filesystems.countObjects() > 0
```

This query will return all devices with a known file system, as seen in the Components section of the device overview. If we take that knowledge back to zendmd, we can interact with the `os` object to discover the data that might be available to us:

```
>>> d.os.filesystems.c
d.os.filesystems.changeOwnership( d.os.filesystems.checkValidId(
d.os.filesystems.checkRelation(    d.os.filesystems.countObjects(
>>> d.os.filesystems.c
```

As the screenshot shows, the `countObjects` function is available from zendmd.

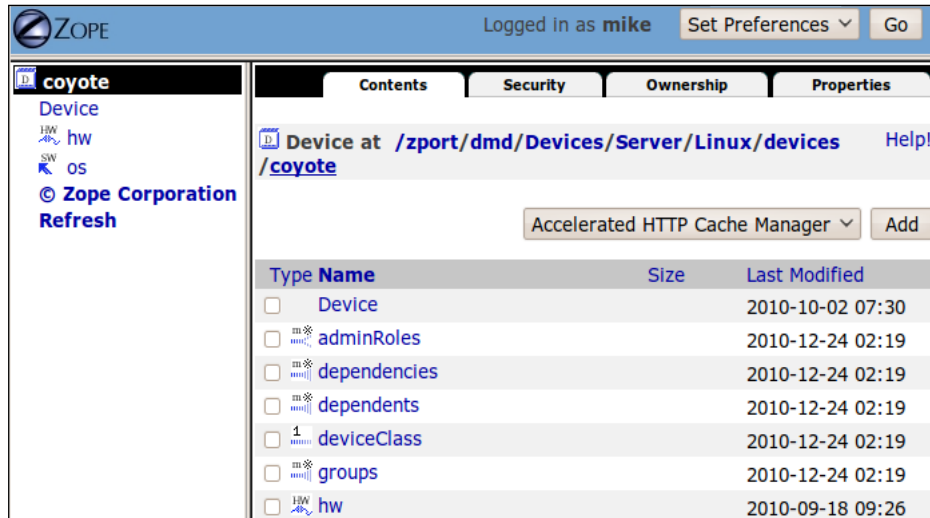
Are you wondering how we knew to look in the `os` and `hw` objects? To answer that question, let's turn back to the Zenoss Core web interface and briefly explore Zope, the framework that drives Zenoss Core.

## Exploring data in Zope

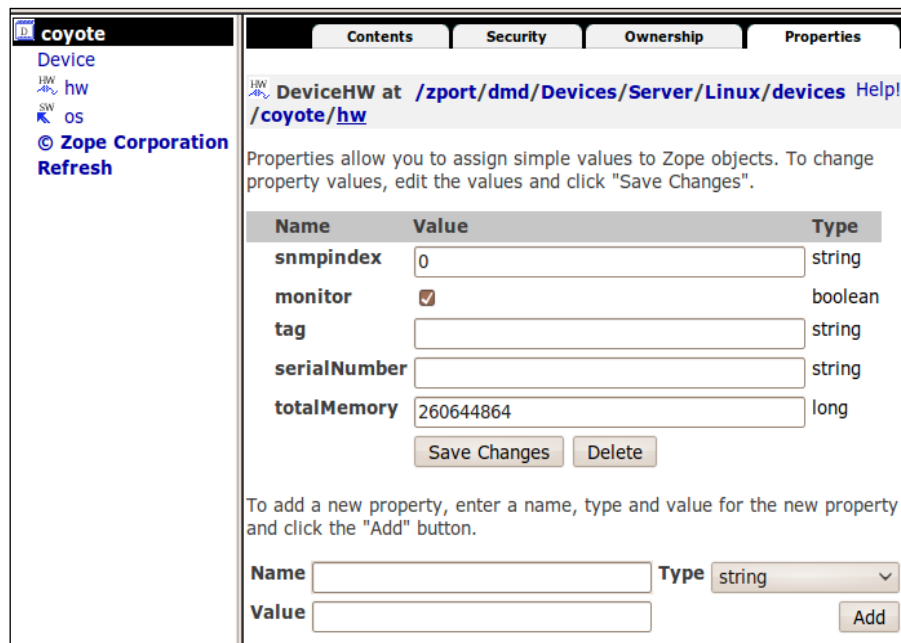
Navigate to a device and note the URL. When I navigate to my favorite device, Coyote, here is the URL I see: **[http://localhost:8080/zport/dmd/Devices/Server/Linux/devices/coyote/devicedetail#deviceDetailNav:device\\_overview](http://localhost:8080/zport/dmd/Devices/Server/Linux/devices/coyote/devicedetail#deviceDetailNav:device_overview)**.

Your hostname and device name will vary, but you should see something similar. Replace **`/devicedetail#deviceDetailNav:device_overview`** with **`/manage`** so that the new URL looks like this: **<http://localhost:8080/zport/dmd/Devices/Server/Linux/devices/coyote/manage>**.

When you press return, should you see a screen that resembles the following screenshot:



If we direct our attention to the left sidebar, we see the data structure that we just explored from zendmd. We see that **hw** and **os** are children of **Device**. To view the final piece of this puzzle, click on the **hw** link, and then click on the **Properties** tab:





As we see in the screenshot, **serialNumber** is a property of **hw**.

Using Zope is just a different way to visualize the Zenoss Core data. We happen to be talking about custom device reports, but this could be useful for other Zenoss Core programming tasks as well.

Let's get back to the custom device reports, so we can review how to evaluate python expressions in the column data.

## Using Python expressions in the columns

The custom device report allows us to specify a Python expression for each column on the report. The expression must be prefaced with `python:`.

To demonstrate, let's run a report that pulls all devices with monitored memory values, using this query:

```
here.hw.totalMemory > 0
```

Enter the following values for Columns:

- `getId`
- `python:dev.hw.totalMemory`
- `python:convToUnits(dev.hw.totalMemory)`

For the **Column Names**, specify **Name**, **Memory**, and **Total Memory**, respectively.

Query	<code>here.hw.totalMemory &gt; 0</code>
Sort Column	<code>Name</code>
Sort Sense	<code>asc</code>
Columns	<code>getId</code> <code>python:dev.hw.totalMemory</code> <code>python:convToUnits(dev.hw.totalMemory)</code>
Column Names	<code>Name</code> <code>Memory</code> <code>Total Memory</code>

Save and then view the report.

When we run the report, we see that for each device with a `totalMemory` property, we're printing the total memory value as an integer and in a human readable format, as seen in the following screenshot:



Name	Memory	Total Memory
coyote	260644864	248.6MB

Before we talk about the `convToUnits` function, note how we reference the device in the Python expression. In the context of the report columns, we use the variable `dev` to access the device. Recall that we use the `here` variable when writing a query, but this expression will fail as a column expression:

```
python: here.hw.totalMemory # not correct
```

In place of the variable `dev`, we could also use `device`, but why type three extra letters if we don't have to?

Now, what about `convToUnits`? When we work with custom device reports, we have access to a couple of "convenience" functions.

## Convenience functions

In the preceding report, we use the `convToUnits` function to turn the `totalMemory` value into something human readable. Sample usage follows.

### convToUnits

The `convToUnits` function converts a number to its human-readable format (for example, 4GM, 128MB, 2.1GHz, and so on). We can supply it with the number to convert, the number to divide by, and a unit string:

```
convToUnits(number=0, divby=1024, unitstr="B")
```

If we give the function a number, such as the `totalMemory` value, it will automatically divide the number by 1024 and append a "B" to the units. We can change all three values if we need to, as the following usage shows:

```
convToUnits(123456789, 1000, "Hz")
```

The `divby` number should be self-evident and answers the question, "what value do I need to divide my number by in order to obtain a readable result?". The `unitstr` value needs some further explanation.

By default, the function will evaluate the unit in terms of 'K','M','G','T', and 'P'. The `unitstr` value you specify is appended to the evaluated unit. So if you specify `Hz` as the `unitstr`, then your potential units become `KHz`, `MHz`, `GHz`, and so on when the function evaluates the number.



The `convToUnits` function is also available to event transformations.  
The function is defined in `$ZENHOME/Products/ZenUtils/Utils.py`.

## Scheduling reports for e-mail delivery

Zenoss Core includes the `reportmail` command in `$ZENHOME/bin` that enables us to send any report, including custom device reports, via e-mail. No graphical interface is available which means that we must work from the command line as the `zenoss` user.

We need to specify the URL of the report, the username/password for a user, and a from address for the e-mail. Run the command `reportmail --help` to get a full list of options.

To get the URL of the report, open the report you want to mail from the Zenoss UI and copy the URL from the browser. Using the Uptime Report we created earlier in this chapter, I would use the command:

```
reportmail -U userName -p password \  
-u http://localhost:8080/zport/dmd/reports#reporttree:.zport.dmd.  
Reports.Custom%20Device%20Reports.Uptime
```

As you can see, the URL is quite large and causes the above command to wrap over three lines. It's one of those things that doesn't transfer well to print. And even though it's ugly, it's relatively simple. We feed `reportmail` a username and password. Then we specify in the above command the URL to the report.

The report will be sent to the e-mail address associated with the user account. Use the `-a` option to change the recipient.

## Sending a CSV report

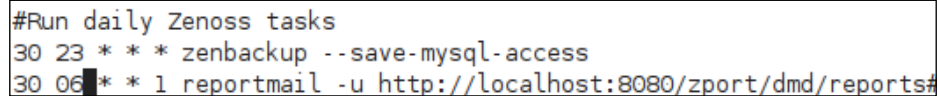
The default report that gets sent from reportmail will be readable and will likely serve its purpose. However, you may want to e-mail the CSV version of the report instead. To get the CSV version of any report, append `?doExport` to the report URL.

## Scheduling a cron job

If we want to e-mail the report out on a recurring basis, we can schedule it as a cron job. We talked about setting up cron jobs in the *Automating backups with zenbackup* section of *Chapter 8, Settings and Administration*. To add the `reportmail` command, edit the crontab as the `zenoss` user with the command:

```
crontab -e
```

The following screenshot shows my updated crontab:



```
#Run daily Zenoss tasks
30 23 * * * zenbackup --save-mysql-access
30 06 * * * 1 reportmail -u http://localhost:8080/zport/dmd/reports#
```

Not all of the `reportmail` command is shown in the screenshot because it's too long.

Just be aware that `reportmail` requires you to specify the username/password for a Zenoss Core account. If you do report mail in a script, change the ownership on the script so only the `zenoss` user can read it. You might also want to set up a less privileged account in Zenoss Core account with a role of `ZenUser` for use with `reportmail`.

## Summary

As you can see, custom device reports make writing custom reports in Zenoss Core tolerable. It keeps us from pulling our hair out while trying to write complicated scripts from scratch. If, however, you want to write your own scripts from scratch, the *Zenoss Core Administrators Guide* provides a brief introduction. And with this chapter as background information, you should at least have a good starting position.

That concludes our introduction to Zenoss Core. Now, go forth and monitor.





# Event Attributes

Throughout the course of this book, we've interacted with event attributes in many different contexts. As you might guess, these attributes define the details of an event in Zenoss Core. However, not all attributes are applicable to each event. The most obvious place to witness the event attributes and how they are applied is in the event log, which we covered in *Chapter 6, Core Event Management*. Also in *Chapter 6, Core Event Management* we reviewed event views, which determine how events show up in the event console—all predicated on the event attributes listed in this appendix.

We can also use the event attributes to process events in relation to:

- Event transformations
- Event mappings
- Event commands
- ZenPack programming

Your programming context determines how you access the event attributes. We can substitute the event attributes in our Python statements via TALEs expressions, as we saw with the discussion of event commands in *Chapter 6, Core Event Management*.

An example of a TALEs expression to access an event attribute looks like this:

```
${evt/attribute}
```

The attribute can be any of the event fields defined in the following table. So if we wanted to write a TALEs expression to access the message body of an event, the expression would look like this: `${evt/message}`. Python evaluates the expression and substitutes the value into the current Python statement.

In other programming contexts where you might not use TALEs, you would use the following syntax to access the event attribute:

```
evt.attribute
```

The following table lists the available event attributes in Zenoss Core.

Event Attribute	Description
dedupid	Identifies the event so that Zenoss can deduplicate events. Takes the form of device, component, eventClass, eventKey, and severity.
evid	A unique identifier for the event.
device	Specifies the device attached to the event.
component	The Zenoss daemon reporting the event.
eventClass	The event class that the event maps to.
eventKey	A user-defined way to map events. Event keys can be sequenced to aid the event class mapping of events from a common source to different event classes.
summary	Summary of the event.
message	Message body for the event. May be the same as summary.
severity	A Numeric representation of the event: <ul style="list-style-type: none"><li>• 5 = Critical</li><li>• 4 = Error</li><li>• 3 = Warning</li><li>• 2 = Info</li><li>• 1 = Debug</li><li>• 0 = Clear</li></ul>
eventState	Numeric representation of the event state: <ul style="list-style-type: none"><li>• 0 = New</li><li>• 1 = Acknowledged</li><li>• 2 = Suppressed</li></ul>
eventClassKey	Maps the event to an event class.
eventGroup	Event source group: For example, syslog, process, and ping.
stateChange	Time stamp when the event state changed.
firstTime	Time stamp when the event first occurred.
lastTime	Time stamp when the event last occurred.
count	The total number of times the event has occurred based on the dedupid.

Event Attribute	Description
prodState	The production state of the device. The Zenoss defaults are: <ul style="list-style-type: none"> <li>• 1000 = Production</li> <li>• 500 = Pre-Production</li> <li>• Test = 400</li> <li>• Maintenance = 300</li> <li>• Decommissioned = -1</li> </ul>
suppid	If the event is suppressed, this is the ID of the suppressing event.
manager	The fully qualified domain name of the event collector that generated the event.
agent	Reports the Zenoss daemon responsible for generating the event.
DeviceClass	The device class.
Location	The location organizer assigned to the device.
Systems	The system organizer assigned to the device.
DeviceGroups	The group organizer assigned to the device.
ipAddress	The IP address of the device.
facility	The syslog subsystem that generated the event (for example, cron, mail, lpr, auth, authpriv, daemon, ftp, kern, mark, news, syslog, user, uucp, local0 through local7).
priority	The priority of the syslog event.
ntevid	The Event ID field of the Windows NT event log.
ownerid	The ID number of the event owner.
clearid	The ID number of the event that cleared this event.
DevicePriority	The priority as assigned in the device's Edit page: <ul style="list-style-type: none"> <li>• 5 = Highest</li> <li>• 4 = High</li> <li>• 3 = Normal</li> <li>• 2 = Low</li> <li>• 1 = Lowest</li> <li>• 0 = Trivial</li> </ul>
eventClassMapping	The event class mapping used to evaluate and map the event.





# B

## Device Attributes

Device attributes describe each device, and like the event attributes in *Appendix A, Event Attributes* not all attributes will apply to every device. We use device attributes to extract bits of information in the following places:

- User commands
- Event commands
- Monitoring template commands
- Event transformations
- ZenPack programming
- Custom device reports

Your programming context determines how you access the device attributes. For example, we can substitute the device attributes in our Python statements via TALEs expressions.

An example of a TALEs expression to access a device attribute looks like this:

```
${dev/attribute}
```

So if we wanted to write a TALEs expression to access the name of the device, the expression would look like this: `${dev/getId}`. Python evaluates the expression and substitutes the value into the current Python statement.

In other programming contexts where you might not use TALEs, you would use the following syntax to access the event attribute:

```
device.attribute
```

The following table includes a list of the attributes that we may use when working with our devices. If you recall our work with custom device reports in *Chapter 11, Writing Custom Device Reports*, you know this is not an exhaustive list of device attributes that we access. In *Chapter 11, Writing Custom Device Reports*, we used `zendmd` to explore and find device attributes. However, the attributes listed correspond closely to the fields found on the device's Configuration Properties:

Device Attributes	Description
<code>id</code>	The device name, which is not necessarily the fully qualified domain name.
<code>manageIp</code>	The IP address of the device.
<code>productionState</code>	The numeric value of the device's production state: <ul style="list-style-type: none"><li>• 1000 = Production</li><li>• 500 = Pre-Production</li><li>• 400 = Test</li><li>• 300 = Maintenance</li><li>• -1 = Decommissioned</li></ul>
<code>productionStateString</code>	The device's production state as a human-readable string.
<code>priority</code>	The numeric priority value: <ul style="list-style-type: none"><li>• 5 = Highest</li><li>• 4 = High</li><li>• 3 = Normal</li><li>• 2 = Low</li><li>• 1 = Lowest</li><li>• 0 = Trivial</li></ul>
<code>priorityString</code>	The device's priority as a human-readable string.
<code>locationName</code>	The location organizer assigned to the device.
<code>systemNames</code>	The list of system organizers assigned to the device.
<code>groupNames</code>	The list of group organizers assigned to the device.
<code>snmpDescr</code>	The SNMP Description.
<code>snmpOID</code>	The OID from SNMP.
<code>snmpContact</code>	The SNMP contact value.
<code>snmpSysName</code>	The system name from SNMP.
<code>snmpLastCollection</code>	The last time Zenoss collected SNMP data for the device.
<code>comments</code>	User-entered comments on the device.
<code>uptimeStr</code>	The uptime values for the device.

Device Attributes	Description
pingStatusString	The device's ping status: <ul style="list-style-type: none"><li>• 0 = Up</li><li>• 1 = Down</li><li>• 2 = None</li></ul>
snmpStatusString	The device's SNMP status: <ul style="list-style-type: none"><li>• 0 = Up</li><li>• 1 = Down</li><li>• 2 = None</li></ul>
osVersion	The operating system version.
osProductName	The software product name defined on the device's edit page.
osManufactureName	The operating system manufacturer name defined on the device's edit page.
hwProductName	The hardware product name defined on the device's edit page.
hwManufacturerName	The hardware manufacturer name defined on the device's edit page.



# C

## Example snmpd.conf

In *Chapter 2, Discovering Devices* we configured the `/etc/snmp/snmpd.conf` file, which allows Zenoss Core to retrieve monitoring data from the server. A sample `snmpd.conf` file is listed here as a reference:

```
#####
# Access Control

#####

# YOU SHOULD CHANGE THE "COMMUNITY" TOKEN BELOW TO A NEW KEYWORD ONLY
# KNOWN AT YOUR SITE.  YOU *MUST* CHANGE THE NETWORK TOKEN BELOW TO
# SOMETHING REFLECTING YOUR LOCAL NETWORK ADDRESS SPACE.

# By far, the most common question I get about the agent is "why won't
# it work?", when really it should be "how do I configure the agent to
# allow me to access it?"

#

# By default, the agent responds to the "public" community for read
# only access, if run out of the box without any configuration file in
# place.  The following examples show you other ways of configuring
```

#### *Example snmpd.conf*

---

```
# the agent so that you can change the community names, and give
# yourself write access as well.
#
# The following lines change the access permissions of the agent so
# that the COMMUNITY string provides read-only access to your entire
# NETWORK (EG: 10.10.10.0/24), and read/write access to only the
# localhost (127.0.0.1, not its real ipaddress).
#
# For more information, read the FAQ as well as the snmpd.conf(5)
# manual page.
# First, map the community name "public" into a "security name"

#      sec.name  source          community
com2sec notConfigUser  default      public

####

# Second, map the security name into a group name:
#      groupName      securityModel securityName
group  notConfigGroup v1          notConfigUser
group  notConfigGroup v2c          notConfigUser

####

# Third, create a view for us to let the group have rights to:
# Make at least snmpwalk -v 1 localhost -c public system fast again.
#      name          incl/excl    subtree      mask(optional)
```

```
view      systemview      included      .1

####

# Finally, grant the group read-only access to the systemview view.

#      group              context sec.model sec.level prefix read   write
notif

access notConfigGroup ""      any      noauth      exact  systemview
none none

#####
# System contact information

#

# It is also possible to set the sysContact and sysLocation system
# variables through the snmpd.conf file:

syslocation Unknown (edit /etc/snmp/snmpd.conf)

syscontact Root <root@localhost> (configure /etc/snmp/snmp.local.conf)

# Added for support of bcm5820 cards.

pass .1 /usr/bin/ucd5820stat

#####
# Further Information

#

# See the snmpd.conf manual page, and the output of "snmpd -H".

trapcommunity public

# trapsink default

trapsink 127.0.0.1 public 162
```





# Index

## Symbols

`$(evt/message)` expression 265  
`$ZENHOME/log` directory 201  
`$ZENHOME/ZenPacks` directory 225  
`/App/Log` event class 153  
`/Apps/Log` event class 151  
`--deviceclass` option 41, 42  
`{dev/id}` 147  
`.egg` directory 225  
`/Events/App` class 146  
`{evt/id}` 147  
`--help` option 110  
`/Information` class 148  
`--no-eventsdb` option 207  
`--no-perfdata` option 207  
`--no-zobd` option 207  
`-p` option 160  
`/Server/Cmd` class 66, 103  
`/Server/Linux` class 101, 103  
`/Server/Linux` device class 90  
`/Server/Linux` template  
    editing 103-105  
`/Server/Scan` class 103  
`/Server/SSH` class 103  
`/Server/Windows` class 103, 137  
`/Status/IpService` class 148  
`/Status/OSProcess` class 148  
`/Status/Ping` class 148  
`/Status/SNMP` class 148  
`/Status/WinServices` class 148  
`/tmp/Sample/Event/Command` file 147  
`/tmp/SampleEventCommand` file 146  
`/Unknown` event class 150

## A

**Actions** menu 83  
**Add a Single Device** window  
    about 43  
    device attributes, adding to 43  
**Add Components** button 92, 93  
**Add Device Options**  
    attributes 44-46  
**Add OSProcess** menu 93  
**Add Single Device** option 42  
**administered objects**  
    about 185  
    editing, for users 185, 187  
**Advanced | Collectors** menu 76  
**agent event attribute** 267  
**aggregate report** 244, 245  
**alert escalation** 173, 174  
**alert filter** 172, 173  
**alerting rules**  
    alert escalation 173, 174  
    alert filters 172, 173  
    alert messages 176, 177  
    configuring 170, 171  
    schedule 174, 175  
**alert messages** 176, 177  
**alerts** 170  
**All Devices** report 232  
**All Event Classes** report 233  
**All Event Mappings** report 234  
**All Heartbeats** report 234  
**All Monitored Components** report 13, 232  
**auth facility** 158  
**authpriv facility** 158  
**availability report** 245  
**Available RRD Variables** property 125

## B

### backups

- automating, with zenbackup 206
- restoring, with zenrestore 207

### binding 100

### bogo\_check plugin 108-111

### bogo\_check.py command

- about 108
- installing 109

### bogo\_check.py plugin 128

### bogo device 164

### Bogo template

- device, adding for monitoring 126

## C

### Cacti 10

### Cacti plugins

- data source parser 128
- monitoring with 127

### Chunk Size property 78

### Cisco router syslogs

- collecting 159, 160

### class

- about 56
- devices, assigning to 58

### clearid event attribute 267

### CMDB 9, 16

### collection layer

- about 17
- device management daemons 18
- event daemons 19
- performance daemons 18, 19

### collections

- adding 238, 239

### collectors

- about 76, 77
- performance collector, configuring 77-79

### color property 125

### command

- creating 143, 144

### command line discovery 48

### Commands page, Event Manager

- about 143
- commands, creating 143, 144

### comments device attribute 270

### commit() method 182

### community ZenPack

- installing 211, 212

### Community ZenPack Development page

- URL 226

### component event attribute 266

### components

- monitoring 79

### components, monitoring

- about 79
- add components button 92, 93
- component details, editing 93
- component details, viewing 93
- file systems 89
- interfaces 80, 81
- network routes 90
- OS processes 81
- services 85

### Config Cycle Interval property 78

### configuration, HttpMonitor settings 217, 218

### Configuration Management Database. *See* CMDB

### Configuration Reload Interval property 78

### convToUnits function 261, 262

### count event attribute 266

### countObjects function 258

### CPU utilization report 246

### CreateFile command 147

### creatFile command 145

### cron facility 158

### cron job

- scheduling 263

### CSV report

- scheduling 263

### custom device report fields

- about 253
- column names 254
- columns 254
- name 253
- path 253
- query 253
- sort column 253
- sort sense 254
- title 253

### custom device report queries

- building 254
- testing, zendmd used 255-258

### custom device reports

- about 14, 251
- creating 252, 253
- fields 253, 254
- Python expressions, using in columns 260, 261

- custom event view**
  - defining, for users 188

- custom user commands**
  - creating 189-191

- Cycle Interval property 78**

## D

- daemon 200**

- daemon facility 158**

- daemons, Zenoss Core**

- zendisc 148
  - zenperfsnmp 148
  - zenping 148
  - zenprocess 148
  - zenstatus 148
  - zenwin 148

- Dashboard**

- configuring 194

- data**

- backing up, with zensendevent 165
  - exploring, in Zope 258-260

- data collection**

- troubleshooting 62

- data layer 16**

- data point**

- adding 116, 117

- DataPoint property 124**

- data source**

- adding, to monitoring templates 222, 223

- data sources, Nagios plugin**

- adding 114-116
  - data point, adding 116, 117
  - RRDtool data point configurations 117, 118

- data sources, SNMP**

- monitoring 101
  - data sources, SNMPtemplates, overriding 101-103

- data sources, Zenoss Core 101**

- dedkat 165**

- dedupid 155**

- dedupid event attribute 266**

- de-duplication identification. *See* dedupid**

- Default Discovery Networks property 78**

- development mode, ZenPack 225, 226**

- device administration tasks 67**

- device attributes**

- Add a Single Device window, adding to 43

- device attributes, Zenoss Core**

- about 269
  - comments 270
  - groupNames 270
  - hwManufacturerName 271
  - hwProductName 271
  - id 270
  - locationName 270
  - managelp 270
  - osManufactureName 271
  - osProductName 271
  - osVersion 271
  - pingStatusString 271
  - priority 270
  - priorityString 270
  - productionState 270
  - productionStateString 270
  - snmpContact 270
  - snmpDescr 270
  - snmpLastCollection 270
  - snmpOID 270
  - snmpStatusString 271
  - snmpSysName 270
  - systemNames 270
  - uptimeStr 270

- Device Changes report 229**

- device classes**

- about 57
  - list 57
  - templates, binding to 125, 126

- DeviceClass event attribute 267**

- device event attribute 266**

- DeviceGroups event attribute 267**

- device information**

- gathering, modeler plugin used 60, 61

- Device Issues portlet 197**

- device management 9**

- device management daemons, Zenoss Core**

- about 18
  - zendisc 18
  - zenmodeler 18

**DevicePriority event attribute** 267

**Device Reports class**

about 228

All Devices report 232

All Monitored Components report 232

Device Changes report 229

Model Collection Age report 229

New Devices report 229

Ping Status Issues report 232

SNMP Status Issues report 231

Software Inventory report 230

**devices**

adding 35

assigning, to class 58

autodiscovering 37, 38

deleting 70

locking 67

organizing, in Zenoss Core 52-54

preparing, for monitoring 22

renaming 68

searching, manually 35

unlocking 67

**devices, organizing**

by groups 54

by location 52-54

by systems 54

**device status page** 9

**dmd command** 181

## E

**Edit Graph Point screen**

options 124, 125

**Edit Graph Point screen, options**

Available RRD Variables property 125

color property 125

consolidation property 124

DataPoint property 124

format property 125

legend property 125

limit property 125

line type property 125

line width property 125

name property 124

RPN property 125

stacked property 125

type property 124

**e-mail**

events, creating by 166

**e-mail delivery**

reports, scheduling for 262, 263

**Errors graph** 94

**eth0 interface** 94

**event attributes, Zenoss Core**

agent 267

clearid 267

component 266

count 266

dedupid 266

device 266

DeviceClass 267

DeviceGroups 267

DevicePriority 267

eventClass 266

eventClassKey 266

eventClassMapping 267

eventGroup 266

eventKey 266

eventState 266

evid 266

facility 267

firstTime 266

ipAddress 267

lastTime 266

Location 267

manager 267

message 266

ntevd 267

ownerid 267

priority 267

prodState 267

severity 266

stateChange 266

summary 266

suppid 267

Systems 267

**event classes**

about 148, 149

properties 149

**event classes, Zenoss Core**

/Information 148

/Status/IpService 148

/Status/OSProcess 148

/Status/Ping 148

- /Status/SNMP 148
- /Status/WinServices 148
- eventClass event attribute** 266
- eventClassKey event attribute** 168, 266
- event class mapping**
  - properties 153
- eventClassMapping event attribute** 267
- event class mapping, properties**
  - Event Class Key 153
  - Example 153
  - Explanation 153
  - Name 153
  - Regex 153
  - Resolution 153
  - Rule 153
  - Sequence 153
  - Transform 153
- Event Console**
  - about 12, 132
  - event log, viewing 135-137
  - events, closing 137
  - event severities, defining 133, 134
  - event statuses, defining 134
  - working 132, 133
- Eventcreate**
  - event log configuration, testing with 163
- eventcreate.exe** 163
- event daemons, Zenoss Core**
  - about 19
  - zeneventlog 19
  - zensyslog 19
  - zentrap 19
- event de-duplication**
  - about 154
  - turning off 155
- eventGroup event attribute** 266
- eventKey event attribute** 266
- event log**
  - viewing 135-137
- event log configuration**
  - testing, with Eventcreate 163
- Event Log Cycle Interval property** 77
- event log severities, Windows** 162, 163
- event logs, Windows**
  - monitoring 161, 162
- event management** 11, 12
- Event Manager**
  - about 132, 138
  - Commands page 143
  - Edit page 139, 140
  - Fields page 141, 142
- event mapping** 148-152
- event mapping sequence** 154
- event reporting**
  - incorporating, into third-party scripts 163, 165
- Event Reports**
  - about 233
  - All Event Classes report 233
  - All Event Mappings report 234
  - All Heartbeats report 234
- events**
  - about 16
  - acknowledging 134
  - clearing 147
  - closing 137
  - creating, by e-mail 166
  - event classes 148, 149
  - event mapping sequence 154
  - mapping 150-152
  - simulating 145-147
  - working with 145
- event severities**
  - defining 133, 134
- Events menu** 132
- eventState event attribute** 266
- event status**
  - defining 134
  - event, acknowledging 134
- event transformations**
  - about 177, 178
  - examples 178-180
- event views**
  - about 187
  - options 189
- evid event attribute** 266
- EXEC mode** 160
- exim4 mapping** 151, 152
- Expect Regex field** 88

## F

- facility event attribute** 267
- Fields page, Event Manager** 141, 142

## **files**

adding, to ZenPack 221

## **file systems**

about 89

ignoring, with zProperties 89, 90

**Filesystem Utilization report** 246, 247

**firstTime event attribute** 266

**format property** 125

**ftp facility** 158

## **G**

**genconf option** 167

**getId attribute** 253

**Google Maps portlet** 53

## **graph**

configuring 123

data, representing graphically 120-122

RRDtool graph point configurations 123

## **graph definitions**

adding 240, 241

## **graph group**

adding 241-243

## **graph reports**

about 234

creating 235-237

**groupNames device attribute** 270

**groups** 189

**groups organizer** 54

## **H**

## **historical events**

displaying 138

## **HttpMonitor**

installing 211, 212

settings, configuring 217, 218

websites, monitoring with 212-217

## **HttpMonitor settings**

configuring 217, 218

## **HttpMonitor ZenPack**

installing 213, 214

**hwManufacturerName device attribute** 271

**hwProductName device attribute** 271

## **I**

**ICMP monitoring** 10

**id device attribute** 270

**Infrastructure | IP Services** 86

**installation, community ZenPack** 211, 212

**installation, HttpMonitor** 211, 212

**installation, HttpMonitor ZenPack** 213, 214

**installation, Zenoss Plugins** 29

**installation, ZenPacks** 211, 212

**installed ZenPack objects**

list, viewing 215

**interface template** 95, 96

**Interface Utilization report** 247

**IP address**

resetting 69

**ipAddress event attribute** 267

**Iptables** 31

## **K**

**kern facility** 158

## **L**

**lastTime event attribute** 266

**legend property** 125

**libexec directory** 221

**limit property** 125

**line type property** 125

**line width property** 125

**Link graphs checkbox** 97

## **Linux**

SNMP, configuring on 25, 26

## **Linux firewalls**

configuring 31

## **Linux server**

memTotalFree, monitoring for 104, 105

**listenport parameter** 167

**localhost monitor** 76

**Location event attribute** 267

**locationName device attribute** 270

**location organizers** 52-54

**Locations portlet** 196, 197

**log file**  
  discovering, from device discovery job 40-42  
**logger**  
  syslog configuration, testing with 160  
**lpr facility** 158

## M

**mail facility** 158  
**maintenance window**  
  about 202  
  properties 203  
**Manage Graph Points dialog box** 123  
**manageIp device attribute** 270  
**Management Information Bases.** *See* MIBs  
**manager event attribute** 267  
**Maximum Ping Failures property** 78  
**Memory Utilization report** 248  
**memTotalFree**  
  monitoring, for Linux server 104, 105  
**message event attribute** 266  
**MIBs**  
  about 24  
  adding 204, 205  
**Model Collection Age report** 229  
**model devices** 59  
**modeler plugin**  
  assigning 62  
  device information, gathering 60, 61  
**monitoring data**  
  backing up 205  
  restoring 205  
**monitoring solutions, Zenoss Core**  
  about 8  
  availability monitoring 10  
  device management 9  
  event management 11, 12  
  performance graphs 10  
  plugin architecture 12, 13  
  system architecture 14, 15  
  system reports 13  
**monitoring templates**  
  about 100  
  data source, adding to 222, 223  
  Nagios plugin, adding to 112-114

**monitoring thresholds** 118-120  
**multi-graph reports**  
  about 237  
  collections, adding 238, 239  
  graph definitions, adding 240, 241  
  graph group, adding 241-243

## N

**Nagios** 10  
**Nagios plugin**  
  about 138  
  adding, to monitoring templates 112-114  
  data, representing graphically 120-122  
  data sources, adding 114-116  
  monitoring thresholds 118-120  
  monitoring with 108-110  
  performance data 111, 112  
  return codes 111  
  working with 111  
**name property** 124  
**Net-SNMP** 25  
**network routes** 90  
**networks** 91  
**New Devices report** 229  
**news facility** 158  
**nmap command** 192  
**Notification Schedules report** 249  
**ntevid event attribute** 267

## O

**Object Identifier.** *See* **OID**  
**objects**  
  adding, to ZenPack 221-224  
**OID**  
  about 23  
  searching, for SNMP monitoring 105-108  
**OpenSSH** 28  
**organizers**  
  editing 56  
  moving 56  
  working with 54, 55  
**osManufactureName device attribute** 271  
**OS processes**  
  about 81  
  adding 81



- configuration properties 82, 83
- details, adding 82
- details, editing 82
- monitoring 83, 84
- osProductName device attribute 271**
- osVersion device attribute 271**
- ownerid event attribute 267**

## P

- Packets graph 94**
- Pattern field 82**
- performance collector**
  - configuring 77-79
- performance daemons, Zenoss Core**
  - about 18
  - zencommand 19
  - zenperfsnmp 19
  - zenping 19
  - zenprocess 19
  - zenstatus 19
- performance data, Nagios plugin 111, 112**
- performance graphs**
  - about 96
  - performance thresholds, monitoring 98
  - working with 97
- performance reports**
  - about 243
  - aggregate reports 244, 245
  - availability report 245
  - CPU utilization 246
  - filesystem utilization 246, 247
  - interface utilization 247
  - memory utilization 248
  - threshold summary 248
- ping command 190**
- Ping Status Issues report 232**
- pingStatusString device attribute 271**
- Ping Timeout property 78**
- Ping Tries property 78**
- plugin architecture 12, 13**
- POP3 account 169**
- portlets permissions 200**
- portlets, Zenoss Core**
  - about 194, 195
  - device issues 197
  - locations 196, 197

- permissions 200
- production states 199, 200
- root organizers 199
- watch list 198
- Zenoss issues 198
- port scan 30**
- priority device attribute 270**
- priority event attribute 267**
- priorityString device attribute 270**
- Process Cycle Interval property 78**
- Process Parallel Jobs property 78**
- prodState event attribute 267**
- productionState device attribute 270**
- Production States portlet 199, 200**
- productionStateString device attribute 270**
- Python expressions**
  - using, in columns 260, 261

## R

- Remodel Device option 83**
- Render URL property 78**
- Render User property 79**
- report classes, Zenoss Core**
  - about 227
  - device reports 228
  - event reports 233
  - graph reports 234-237
  - multi-graph reports 237, 238
  - performance reports 243, 244
  - user reports 249
- reportmail command 262, 263**
- reports**
  - about 227, 228
  - scheduling, for e-mail delivery 262, 263
- Reset button 97**
- return codes, Nagios plugin**
  - 0 111
  - 1 111
  - 2 111
  - 3 111
  - about 111
- Root Organizers portlet 199**
- Round Robin Database. *See* RRD**
- router syslogs, Cisco**
  - collecting 159, 160
- RPN property 125**

RRD 16  
RRDtool 99  
RRDtool Data Point configurations 117, 118  
RRDtool graph point configurations 123

## S

SampleEventCommand 146  
schedule 174, 175  
SCMP monitoring 10  
serialNumber property 260  
service  
  about 85  
  configuration properties 87  
  exceptions, monitoring 87, 88  
  IP services, monitoring 88  
  monitoring, enabling 85-87  
severity event attribute 266  
Simple Network Management Protocol. *See* SNMP  
SNMP  
  about 9-23  
  configuring, on Linux 25, 26  
  configuring, on Windows 26-28  
  versions 24  
snmpContact device attribute 270  
SNMP data sources  
  monitoring 101  
  /Server/Linux template, editing 103-105  
SNMP data sources, monitoring  
  templates, overriding 101-103  
snmpd.conf file  
  example 273-275  
snmpDescr device attribute 270  
snmpget utility 25  
SNMP Informant  
  URL 28  
SNMP issues  
  on Windows 64  
  troubleshooting 62  
SNMP issues, troubleshooting  
  about 62  
  snmpwalk command, running 63  
snmpLastCollection device attribute 270  
SNMP monitoring  
  OIDs, searching for 105-108  
snmpOID device attribute 270

SNMP Performance Cycle Interval property 77  
SNMP Status Issues report 231  
snmpStatusString device attribute 271  
snmpSysName device attribute 270  
snmptrap utility 25  
SNMP v3 24  
SNMP versions 24  
snmpwalk command  
  about 63, 105  
  running 63  
SNPP 194  
Software Inventory report  
  about 230  
  manufacturers 230  
  products 230  
SSH 9  
stacked property 125  
stateChange event attribute 266  
Status Cycle Interval property 78  
summary event attribute 266  
suppid event attribute 267  
syslog 157  
syslog.conf file 158, 159  
syslog configuration  
  testing, with logger 160  
syslog facility 158  
syslog messages  
  routing, to Zenoss Core 157-159  
system architecture 14, 15  
systemNames device attribute 270  
system reports  
  about 13  
  custom device reports 14  
system settings, Zenoss Core 193, 194  
Systems event attribute 267  
systems organizer 54  
sysUpTime function 23, 257

## T

TALES expression  
  about 144  
  example 265  
Telnet 28  
templates  
  about 99

- binding, to device class 125, 126
- monitoring 100
- overriding 101-103
- Threshold Summary report** 248
- Throughput graph** 94
- totalMemory property** 261
- Twisted** 15
- type property** 124

## U

- uptimeStr attribute** 253, 256, 270
- user command**
  - adding 192
- user facility** 158
- user layer** 15
- user reports**
  - about 249
  - notification schedules 249
- users**
  - adding 184
  - administered objects, editing for 185, 187
  - custom event view, defining for 188
  - managing 183-185
- uucp facility** 158

## V

- virtualMemory option** 90
- VMware** 8
- VMware Server**
  - URL, for downloading 22

## W

- Watch List portlet** 198
- web portal** 9
- websites**
  - monitoring, with HttpMonitor 212-217
- Windows**
  - event log severities 162, 163
  - SNMP, configuring on 26-28
  - SNMP issues 64
  - WMI, configuring on 26-28
- Windows event logs**
  - monitoring 161, 162
- Windows firewall**
  - configuring 32

- Windows Management Instrumentation.**
  - See* WMI
- Windows Service Cycle Interval property** 78
- Windows WMI Batch Size property** 78
- Windows WMI query timeout property** 78
- WMI**
  - about 21, 22
  - configuring, on Windows 26-28
- WMI issues**
  - troubleshooting 64
- WMI issues, troubleshooting**
  - Zeneventlog 65

## Z

- zAlertOnRestart property** 83
- zCollectorClientTimeout property** 71
- zCollectorDecoding property** 71
- zCollectorLogChanges property** 71
- zCollectorPlugins property** 71
- zCommandCycleTime property** 71
- zCommandExistenceTest property** 71
- zCommandLoginTimeout property** 71
- zCommandLoginTries property** 71
- zCommandPassword property** 71
- zCommandPath configuration property** 115
- zCommandPath property** 71
- zCommandPort property** 71
- zCommandProtocol property** 71
- zCommandSearchPath property** 71
- zCommandTimeout property** 71
- zCommandUsername property** 72
- zCountProcs property** 83
- zDeviceTemplates property** 72
- zen** 17
- zenbackup command**
  - backups, automating with 206
- zenbatchload command**
  - about 46
  - devices, importing with 47
- zenbatchload file** 48
- zenbatchload import utility** 47
- zencommand daemon** 19, 111
- zendisc daemon**
  - about 18, 41, 48, 148
  - command line discovery 48

- zendmd command** 270
  - custom device report queries, testing 255-258
  - using 180-182
- Zeneventlog** 65
- zeneventlog daemon** 19
- ZenHub** 16
- zenmail daemon** 166-168
- zenmib command** 106
- zenmodeler daemon** 18, 201, 202
- Zenoss**
  - component 79
- zenoss.cmd.darwin plugin** 62
- Zenoss Core**
  - about 7
  - Add Single Device option 42
  - alerting rules, configuring 170, 171
  - collection layer 17
  - community ZenPacks, installing 211, 212
  - custom device reports, creating 251-253
  - custom user commands, creating 189-191
  - daemon 200-202
  - Dashboard, configuring 194
  - data layer 16
  - data sources 101
  - device administration tasks 67
  - device attributes 269-271
  - device classes 57
  - devices, organizing in 52-54
  - devices, preparing for monitoring 22
  - event attributes 266, 267
  - Event Console 132, 133
  - event de-duplication 154
  - Event Manager 138-140
  - event mapping 148
  - events 145
  - events, generating with e-mails 166
  - event transformations 177, 178
  - event views 187-189
  - features 7
  - groups 189
  - HttpMonitor settings, configuring 217, 218
  - HttpMonitor ZenPack, installing 213, 214
  - maintenance window 202, 203
  - MIBs, adding 204, 205
  - model devices 59
  - monitoring data, backing up 205
  - monitoring data, restoring 205
  - monitoring solutions 8
  - monitoring templates 100
  - portlets 194-196
  - port requisites 30
  - report classes 227
  - reports 227, 228
  - /Server/Cmd class 66
  - set up wizard 33-38
  - syslog messages, routing to 157-159
  - system settings 193, 194
  - templates, monitoring 100
  - updating 208, 209
  - user layer 15
  - users, adding 184
  - websites, monitoring with HttpMonitor 212, 213
  - ZenPack, creating 218-220
  - zProperties 71-74
- Zenoss Core 3.0** 212
- Zenoss Core daemons** 200-202
- Zenoss Core data collection**
  - troubleshooting 62
- Zenoss Core setup wizard**
  - about 33
  - device inventory example 38, 40
  - devices, specifying to monitor 34-38
  - users, setting up 33, 34
- Zenoss Core users**
  - managing 183-185
- Zenoss Core web interface**
  - device class, adding 112
- Zenoss Developer's Guide**
  - URL 226
- Zenoss Enterprise** 7
- Zenoss Issues portlet** 198
- Zenoss Plugins**
  - about 28
  - installing 29
  - troubleshooting 66
- ZenPack**
  - about 10, 13, 211, 212
  - creating 218-220
  - development mode 225, 226
  - exporting 224
  - files, adding to 221
  - installing 211, 212

- objects, adding to 221-224
- packaging 224
- URL 211
- ZenPackName** 223
- ZenPacks**
- zenperfsnmp** daemon 19, 148
- zenping** 201
- zenping** daemon 19, 148
- zenpop3** 169
- zenprocess** daemon 19, 148
- zenrestore** command
  - about 207
  - backups, restoring with 207
  - options 207
- zensendevent**
  - data, backing up with 165
  - event reporting, incorporating into third-party scripts 163, 165
- zenstatus** daemon 19, 148
- zensyslog** daemon 19, 157, 201
- zentrap** daemon 19
- zenwin** daemon 148
- zEventAction** property 150
- zEventClearClasses** property 150
- zEventSeverity** property 150
- zFailSeverity** property 83
- zFailSeverity, Zproperty** 87
- zFileSystemMapIgnoreNames** field 90
- zFileSystemMapIgnoreNames** property 72
- zFileSystemMapIgnoreTypes** property 72, 90
- zFileSystemSizeOffset** property 72
- zHardDiskMapMatch** property 72
- zHideFieldFromList, Zproperty** 87
- zIcon** property 72
- zIfDescription** property 72
- zInterfaceMapIgnoreNames** property 72
- zInterfaceMapIgnoreTypes** property 72
- zIpServiceMapMaxPort** property 72
- zKeyPath** property 72
- zLinks** property 72
- zLocalInterfaceNames** property 72
- zLocalIpAddresses** property 72
- zMaxOIDPerRequest** property 72
- zMonitor** property 83
- zMonitor, Zproperty** 87
- zNmapPortscanOptions** property 72
- ZODB** 16
- Zope**
  - about 15
  - data, exploring in 258, 260
- zopectl restart** command 214
- Zope Object Database.** *See* ZODB
- zPingMonitorIgnore** property 72
- zProdStateThreshold** property 73
- zProperties**
  - about 71, 149
  - zCollectorClientTimeout** 71
  - zCollectorDecoding** 71
  - zCollectorLogChanges** 71
  - zCollectorPlugins** 71
  - zCommandCycleTime** 71
  - zCommandExistenceTest** 71
  - zCommandLoginTimeout** 71
  - zCommandLoginTries** 71
  - zCommandPassword** 71
  - zCommandPath** 71
  - zCommandPort** 71
  - zCommandProtocol** 71
  - zCommandSearchPath** 71
  - zCommandTimeout** 71
  - zCommandUsername** 72
  - zDeviceTemplates** 72
  - zEventAction** 150
  - zEventClearClasses** 150
  - zEventSeverity** 150
  - zFailSeverity** 87
  - zFileSystemMapIgnoreNames** 72
  - zFileSystemMapIgnoreTypes** 72
  - zFileSystemSizeOffset** 72
  - zHardDiskMapMatch** 72
  - zHideFieldFromList** 87
  - zIcon** 72
  - zIfDescription** 72
  - zInterfaceMapIgnoreNames** 72
  - zInterfaceMapIgnoreTypes** 72
  - zIpServiceMapMaxPort** 72
  - zKeyPath** 72
  - zLinks** 72
  - zLocalInterfaceNames** 72
  - zLocalIpAddresses** 72
  - zMaxOIDPerRequest** 72
  - zMonitor** 87
  - zNmapPortscanOptions** 72

- zPingMonitorIgnore 72
- zProdStateThreshold 73
- zPythonClass 73
- zRouteMapCollectOnlyIndirect 73
- zRouteMapCollectOnlyLocal 73
- zRouteMapMaxRoutes 73
- zSnmpAuthPassword 73
- zSnmpAuthType 73
- zSnmpCommunities 73
- zSnmpCommunity 73
- zSnmpMonitorIgnore 73
- zSnmpPort 73
- zSnmpPrivPassword 73
- zSnmpPrivType 73
- zSnmpSecurityName 73
- zSnmpTimeout 73
- zSnmpTries 73
- zSnmpVer 73
- zSshConcurrentSessions 73
- zStatusConnectTimeout 73
- zSysedgeDiskMapIgnoreNames 73
- zTelnetEnable 74
- zTelnetEnableRegex 74
- zTelnetLoginRegex 74
- zTelnetPasswordRegex 74
- zTelnetPromptTimeout 74
- zTelnetSuccessRegexList 74
- zTelnetTermLength 74
- zWinEventLog 74
- zWinEventLogMinSeverity 74
- zWinPassword 74
- zWinUser 74
- zWmiMonitorIgnore 74
- zXmlRpcMonitorIgnore 74

- Zproperty
- zPythonClass property 73
- zRouteMapCollectOnlyIndirect property 73
- zRouteMapCollectOnlyLocal property 73
- zRouteMapMaxRoutes property 73
- zSnmpAuthPassword property 73
- zSnmpAuthType property 73
- zSnmpCommunities property 73
- zSnmpCommunity property 73
- zSnmpMonitorIgnore property 73, 229
- zSnmpPort property 73
- zSnmpPrivPassword property 73
- zSnmpPrivType property 73
- zSnmpSecurityName property 73
- zSnmpTimeout property 73
- zSnmpTries property 73
- zSnmpVer property 73
- zSshConcurrentSessions property 73
- zStatusConnectTimeout property 73
- zSysedgeDiskMapIgnoreNames property 73
- zTelnetEnable property 74
- zTelnetEnableRegex property 74
- zTelnetLoginRegex property 74
- zTelnetPasswordRegex property 74
- zTelnetPromptTimeout property 74
- zTelnetSuccessRegexList property 74
- zTelnetTermLength property 74
- zWinEventLogMinSeverity property 74
- zWinEventLog property 74
- zWinPassword property 74
- zWinUser property 74
- zWmiMonitorIgnore property 74
- zXmlRpcMonitorIgnore property 74





## Thank you for buying **Zenoss Core 3.x Network and System Monitoring**

### About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.packtpub.com](http://www.packtpub.com).

### About Packt Open Source

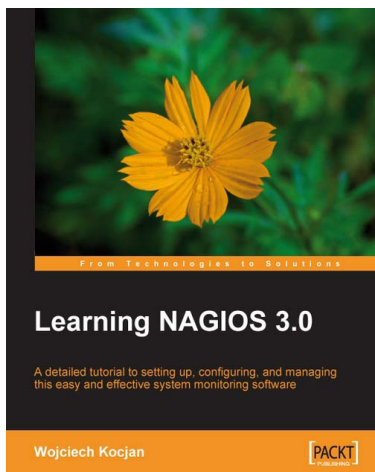
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

### Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.





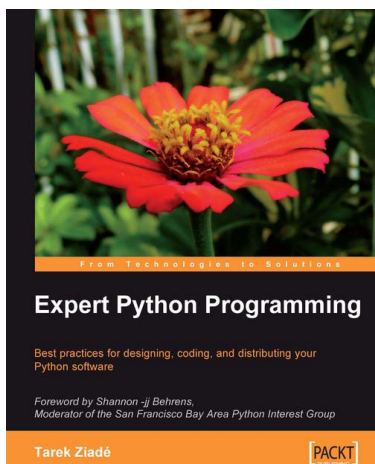
## Learning Nagios 3.0

ISBN: 978-1-847195-18-0

Paperback: 316 pages

A comprehensive configuration guide to monitor and maintain your network and systems

1. Secure and monitor your network system with open-source Nagios version 3
2. Set up, configure, and manage the latest version of Nagios
3. In-depth coverage for both beginners and advanced users



## Expert Python Programming

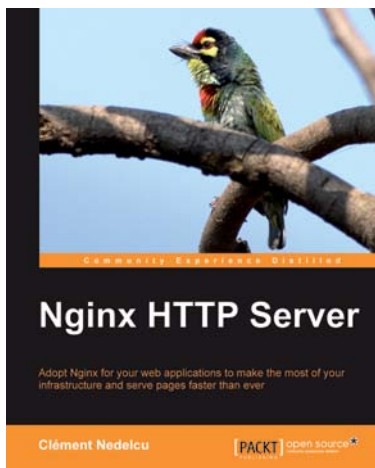
ISBN: 978-1-847194-94-7

Paperback: 372 pages

Best practices for designing, coding, and distributing your Python software

1. Learn Python development best practices from an expert, with detailed coverage of naming and coding conventions
2. Apply object-oriented principles, design patterns, and advanced syntax tricks
3. Manage your code with distributed version control
4. Profile and optimize your code

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles

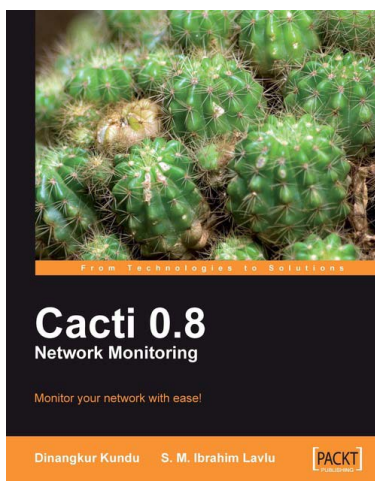


## Nginx HTTP Server

ISBN: 978-1-849510-86-8      Paperback: 348 pages

Adopt Nginx for your web applications to make the most of your infrastructure and serve pages faster than ever

1. Get started with Nginx to serve websites faster and safer
2. Learn to configure your servers and virtual hosts efficiently
3. Set up Nginx to work with PHP and other applications via FastCGI
4. Explore possible interactions between Nginx and Apache to get the best of both worlds



## Cacti 0.8 Network Monitoring

ISBN: 978-1-847195-96-8      Paperback: 132 pages

Monitor your network with ease!

1. Install and setup Cacti to monitor your network and assign permissions to this setup in no time at all
2. Create, edit, test, and host a graph template to customize your output graph
3. Create new data input methods, SNMP, and Script XML data query
4. Full of screenshots and step-by-step instructions to monitor your network with Cacti

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles



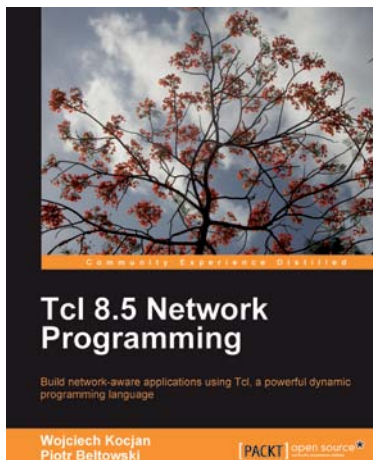
## FreeSWITCH 1.0.6

ISBN: 978-1-847199-96-6

Paperback: 320 pages

Build robust high-performance telephony systems using FreeSWITCH

1. Install and configure a complete telephony system of your own even if you are using FreeSWITCH for the first time
2. In-depth discussions of important concepts like the dialplan, user directory, and the powerful FreeSWITCH Event Socket
3. Best practices and expert tips from the FreeSWITCH experts, including the creator of FreeSWITCH, Anthony Minessale



## Tcl 8.5 Network Programming

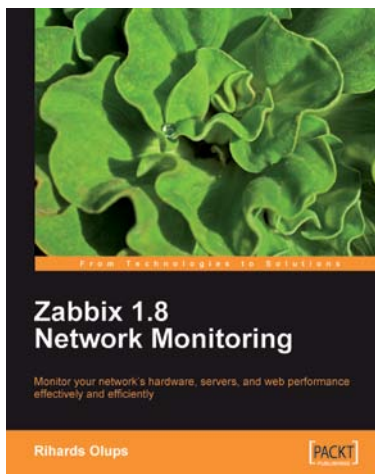
ISBN: 978-1-849510-96-7

Paperback: 588 pages

Build network-aware applications using Tcl, a powerful dynamic programming language

1. Develop network-aware applications with Tcl
2. Implement the most important network protocols in Tcl
3. Packed with hands-on-examples, case studies, and clear explanations for better understanding

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles

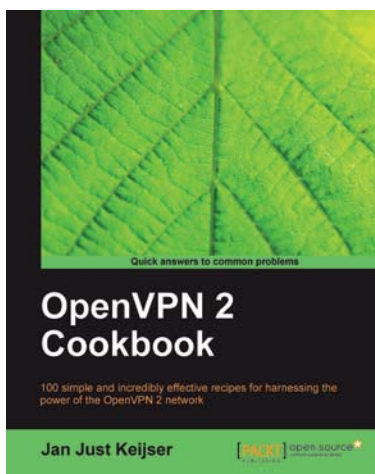


## **Zabbix 1.8 Network Monitoring**

ISBN: 978-1-847197-68-9      Paperback: 428 pages

Monitor your network hardware, servers, and web performance effectively and efficiently

1. Start with the very basics of Zabbix, an enterprise-class open source network monitoring solution, and move up to more advanced tasks later
2. Efficiently manage your hosts, users, and permissions
3. Get alerts and react to changes in monitored parameters by sending out e-mails, SMSs, or even execute commands on remote machines



## **OpenVPN 2 Cookbook**

ISBN: 978-1-84951-010-3      Paperback: 356 pages

100 simple and incredibly effective recipes for harnessing the power of the OpenVPN 2 network

1. Set of recipes covering the whole range of tasks for working with OpenVPN
2. The quickest way to solve your OpenVPN problems!
3. Set up, configure, troubleshoot and tune OpenVPN
4. Uncover advanced features of OpenVPN and even some undocumented options

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles