Zenoss Labs Documentation

Release

Zenoss Labs

September 21, 2015

Contents

Herein lies documentation created by Zenoss Labs. This will have to do until a better home can be found.

Warning: The zenpacklib documentation has replaced the ZenPack Development Guide that used to be here.

CHAPTER 1

Contents

1.1 ZenPack Documentation

ZenPacks must be documented using reStructuredText. The minimum documentation requirement is that each ZenPack have a `README.rst` located in its top-level directory.

An optional top-level `docs/` directory containing at least one file named `index.rst` can also be created to suplement the `README.rst`. This would be the recommended approach if a ZenPack's documentation requires the additional complexity of additional structure, files, or Sphinx extensions.

Contents:

1.1.1 ZenPack Standards Guide

This document describes all requirements and recommendations for ZenPack development. The intended audience is all Zenoss, Inc. employees who create or modify ZenPacks including engineering, services and support. The intended audience also includes any third-parties that create or modify ZenPacks that are delivered to Zenoss customers by Zenoss, Inc.

Assumptions

Some assumptions are made in this document regarding access to test facilities.

Product Inclusion

ZenPacks must always be developed with the understanding that the potential exists for them to become part of the product. Even in cases were ownership of the ZenPack does not rest with Zenoss this must be observed because ownership can change in the future. For this reason, special attention must be paid to document the inclusion of any sensitive company data in ZenPacks.

Access to Test Environment

In cases where the ZenPack developer(s) do not have access to endpoints necessary for integration and testing work some standard operating procedures must be suspended.

File Locations

The location of specific files within a ZenPack's directory structure is technically mandated in some circumstances, and open to the developer's desires in others. To make it easier for other developers to more easily get up to speed with the ZenPack in the future, the following recommendations for file locations should be used.

- `ZenPacks.<namespace.PackName>/`
- `ZenPacks.<namespace.PackName>/ZenPacks/namespace/PackName/`
  - `analytics/`

    The analytics bundle in .zip format: `analytics-bundle.zip`
  - `browser/` (Note: Pre-ZPL only. See resources/ below)
    - `configure.zcml`

      All ZCML definitions related to defining browser views or wiring browser-only concerns. This `configure.zcml` should be included by the default `configure.zcml` in its parent directory.
    - `resources/`
      - `css/` - All stylesheets loaded by users' browsers.
      - `img/` - All images loaded by users' browsers.
      - `js/` - All javascript loaded by users' browser.
  - `resources/` (Note: ZPL, See browser/ above)

    Any javascript code that modifies views go here. Especially note these JS file correlations:
    - `device.js` - Modifies the default device page.
    - `ComponentClass.js` - Modifies the component ComponentClass page.

    Folders inside `resources` have the following properties:
    - `icon/` (Note: ZPL)

      All images and icons loaded by the browser. Within this folder note the following name correspondence:
      - `DeviceClass.png` ` - Icon used in top left corner.
      - `ComponentClass.png` ` - Icon used in Impact diagrams for component.
  - `datasources/`

    All datasources plugin files. Ensure your datasource has a descriptive name that closely correlates to the plugin name.
  - `lib/`

    Any third-party modules included with the ZenPack should be located in this directory. In the case of pure-Python modules they can be located directly here. In the case of binary modules the build process must install them here. See the section of License Compliance below for more information on how to properly handle third-party content.

- `libexec/`

  Any scripts intended to be run by the zencommand daemon must be located in this directory.

- `migrate/`

  All migration code.

- `modeler/`

  All modeling plugins.

- `objects/`

  There should only ever be a single file called objects.xml in this directory. While the ZenPack installation code will load objects from any file in this directory with a `.xml` extension, the ZenPack export code will dump all objects back to `objects.xml` so creating other files only creates future confusion between installation and export.

- `parsers/`

  All custom parsers.

- `patches/`

  All monkeypatches. Note: your patches/__init__.py must specify patch loading.

- `protocols/`

  AMQP schema: Javascript code is read into the AMQP protocol to modify queues and exchanges.

- `services/`

  Custom collector services plugins.

- `service-definition/` (Note: 5.X+)

  Service definitions for 5.X services containers.

- `skins/`

  All TAL template skins in .pt format. These change the UI look.

- `tests/`

  All unit tests.

- `facades.py`

  All facades (classes implementing `Products.Zuul.interfaces.IFacade`) should be defined in this file. In ZenPacks where this single file becomes hard to maintain, a facades/ directory should be created containing individual files named for the group of facades they contain.

- `info.py`

  All info adapters (classes implementing `Products.Zuul.interfaces.IInfo`) should be defined in this file. In ZenPacks where this single file becomes hard to maintain, an `info/` directory should be created containing individual files named for the group of info adapters they contain.

- `interfaces.py`

  All interfaces (classes extending `zope.interface.Interface`) should be defined in this file. In ZenPacks where this single file becomes hard to maintain, an `interfaces/` directory should be created containing individual files named for the group of interfaces they contain.

  - `routers.py`

    All routers (classes extending `Products.ZenUtils.Ext.DirectRouter`) should be defined in this file. In ZenPacks where this single file becomes hard to maintain, a `routers/` directory should be created containing individual files named for the group of routers they contain.

License Compliance

All ZenPack content must be compliant with the license of the ZenPack being developed. If you intend to include a third-party module with a GPL license, the ZenPack must also carry a GPL license and not include any other code that would violate the GPL license. Always run third-party module inclusion through legal to make sure there is no conflict.

Coding Standards

All code and configuration in ZenPacks should be developed according to the following public style guides.

- Python
  - PEP 8 – Style Guide or Python Code
  - PEP 257 – Docstring Conventions
- ZCML
  - Zope's ZCML Style Guide

Monitoring Template Standards

Performance templates are one of the easiest places to make a real user experience difference when new features are added to Zenoss. Spending a very small amount of time to get the templates right goes a long way towards improving the overall user experience. For this reason, the following checklist should be used to determine if your monitoring template is acceptable.

Templates

1. Is the template worthwhile? Should it be removed?
2. Is the template at the correct point in the model?
3. Does the template have a description? Is the description a good one?

Data Sources

1. Can your datasource be named better?

1. Is it a common metric that is being collected from other devices in another way? If so, name yours the same. This makes global reporting much easier.
2. camelCaseNames seem to be the standard. Use them.

2. Never use absolute paths for COMMAND datasource command templates. This will end up causing problems on one of the three platforms we deal with. Link your plugin into zenPath('libexec') instead.

Data Points

1. Using a COUNTER? You might want to think otherwise.
1. Unnoticed counter rollovers can result in extremely skewed data.
2. Using a DERIVE with a minimum of 0 will record unknown instead of wrong data.
2. Enter the minimum and/or maximum possible values for the data point if you know them.
1. This again will allow unknown to be recorded instead of bad data.

Data Point Aliases

1. Include the unit in the alias name if it is in any way not obvious. For example, use `cpu_percent` instead of `cpu_usage`.
2. Use an RPN to calculate the base unit if the data point isn't already collected that way. For example, use `1024,*` to convert a data point collected in KBytes to bytes.

Thresholds

1. Don't include a number in your threshold's name.
1. This makes people have to recreate the threshold if they want to change it.

Graph Definitions

1. Have you entered the units? Do it!
1. This will become the y-axis label and should be all lowercase.
2. Always use the base units. Never kbps or MBs. bps or bytes are better.
2. Do you know the minimum/maximum allowable values? Enter them!
1. Common scenarios include percentage graphing with minimum 0 and maximum 100.
3. Think about the order of your graph points. Does it make sense?
4. Are there other templates that show similar data to yours?
1. If so, you should try hard to mimic their appearance to create a consistent experience.

Graph Points

1. Have you changed the legend? Do it!
2. Adjust the format so that it makes sense.
1. %5.2lf%s is good for values you want RRDTool to auto-scale.
2. %6.2lf%% is good for percentages.
3. %4.0lf is good for four digit numbers with no decimal precision or scaling.
3. Should you be using areas or lines?
1. Lines are good for most values.

2. Areas are good for things that can be thought of as a volume or quantity.
4. Does stacking the values to present a visual aggregate makes sense?

ETL Standards

ETL is an acronym for *Extract, Transform, Load*. When writing ETL adapters you're defining how Zenoss model data is extracted and transformed into the *Zenoss Analytics* schema. The following guidelines should be used to keep reporting consistent.

1. The `reportProperties` implementation in `IReportable` adapters must include the units in the name if not immediately obvious. For example, use `cpu_used_percent` instead of `cpu_used`.

Documentation

ZenPacks must be documented according to the ZenPack.example.Name template. The ZenPacks.zenoss.SolarisMonitor documentation can be used as an example of a ZenPack that has been documented using this template.

Code Documentation

Python code must be documented in docstrings in the locations specified in PEP-8 and according to the style of PEP-257. Links to these standards can be found in the *Coding Standards* section. Inline code comments should also be used when the code isn't obvious.

Testing

The following types of testing must be performed. All test results should be recorded in the ZenPack's test result matrix. The matrix will have the ZenPack version on one axis and the Zenoss version on the other axis. At the intersection will be the result of unit testing, internal integration testing and live integration testing.

Unit Tests

Unit tests must be written for all public interfaces of ZenPack-specific code. Unit tests will be the only mechanism for automated regression testing in some cases, and the primary source in all others.

Internal Integration Testing

ZenPacks must be tested internally using the packaged .egg that is will be delivered to the customer. The test server must be the exact same version of Zenoss being used by the customer. The test environment must match the customer's environment as closely as possible. The only exception to internal integration testing is cases where it is not possible to replicate the test environment internally.

Live Integration Testing

ZenPacks must be tested in their live deployment environment. A development or staging instance of Zenoss that matches the production environment as closely as possible should be used.

Versioning

The first feature-complete ZenPack delivered to a customer should be version 1.0.0. Subsequent versions must increment the micro version if they contain only bugfixes or tweaks (i.e. 1.0.1.) Subsequent versions must increment the minor version if the contain new features (i.e. 1.1.0.)

A ZenPack's version must be incremented each time it is delivered to a customer if there has been any change to it whatsoever.

Reviews

Peer review is a strong mechanism to catch potential issues before integration testing is performed. To that end the following reviews must be performed.

Design Review

The initial design of a ZenPack must be peer reviewed before coding begins.

Code Review

All code, including updates, must be peer reviewed before being committed to the mainline development branch or any stable release branch.

Packaging & Delivery

All ZenPacks must be delivered in their packaged .egg format. If arrangements have been made for the customer to also get the source for the ZenPack it should be provided in addition to the packaged egg as a tarball of the development directory.

ZenPacks must be built using the same environment that the customer will be installing them into. If the customer is installing into multiple environments a separate egg should be built and delivered for each environment. In this context the same environment is defined as the following.

- Exact same version of Zenoss
- Same major version of operating system
- Same architecture (i.e. i386 or x86_64)

All files including documentation must be delivered to customers in a ZenDesk ticket.

1.1.2 ZenPack.example.Name

About

Introduction to the ZenPack and any potentially foreign concepts such as the target that is being monitored or integrated with.

Features

High-level overview of the ZenPack's features and functionality. Focus on the value proposition. Target audience would be people trying to understand Zenoss' capabilities, not Zenoss administrators.

Prerequisites

Requirements and dependencies that must be satisfied. This section should cover the following at a minimum and where applicable.

- Minimum required Zenoss version
- Maximum supported Zenoss version if known
- ZenPack dependencies (with specific versions if known)
- Other installation dependencies if known (e.g. operating system packages)
- Supported versions of the monitoring or integration target

Limitations

Note any shortcomings that the user might otherwise be left curious about. This section is optional.

Usage

The target audience for the entire Usage section is a Zenoss administrator.

Installing

Standard installation steps plus any other installation steps or notes specific to this ZenPack.

Using

One or more ad-hoc usage related sections. This (or these) sections will likely contain the bulk of the ZenPack's custom documentation. The section(s) will not necessarily be called `Using`.

Removing

Standard ZenPack removal steps plus any removal steps or notes specific to this ZenPack. Be especially careful to cover anything that will result in data loss such as removal of device classes and their contained devices.

Troubleshooting

Document common problems users of the ZenPack may run into such as what happens in the result of authentication failures or other configuration mistakes.

Appendix

The two examples appendixes below will very commonly be used. Additional reference material can be made available in additional appendixes. The `Appendixes` section can only be omitted if the ZenPack installs no items as described below, and requires no non-platform daemons as described below.

Appendex A: Installed Items

Detail the items installed by the ZenPack. Items include the following.

- Device Classes
- Configuration Properties
- Modeler Plugins
- Command Parsers
- Monitoring Templates
- Process Classes
- IP Service Classes
- Windows Service Classes
- Event Classes
- Event Mappings
- MIBs
- Reports

Appendex B: Related Daemons

Detail the daemons outside of the core platform required to take advantage of all of the ZenPack's functionality. The core platform daemons listed below *should not* be explicitly listed.

- `zeoctl`
- `zeneventserver`
- `zeneventd`
- `zenhub`
- `zenjobs`
- `zendisc`
- `zenmodeler`
- `zenimpactserver`
- `zenimpactgraph`
- `zenimpactstate`
- `zenjserver`

Daemons such as the following, plus any daemons delivered with the ZenPack *should* be listed.

- `zencommand`
- `zenperfsnmp`
- `zenprocess`

1.1.3 ZenPacks.zenoss.SolarisMonitor

About

The SolarisMonitor ZenPack enables Resource Manager to use Secure Shell (SSH) to monitor Solaris hosts. Resource Manager models and monitors devices placed in the /Server/SSH/Solaris device class by running commands and parsing the output. Parsing of command output is performed on the Resource Manager server (if using a local collector) or on a distributed collector. The account used to monitor the device does not require root access or special privileges.

In addition to the previously described modeling and monitoring features this ZenPack also enables Resource Manager to model and monitor Sun Solaris LDOM servers. Resource Manager will model devices utilizing the Simple Network Management Protocol (SNMP) to collect LDOM information when a device resides in either the /Server/Solaris or /Server/SSH/Solaris device classes. The discovered LDOM information will be displayed as components of the LDOM host server.

Features

The SolarisMonitor ZenPack provides:

- File system and process monitoring
- Network interfaces and route modeling
- CPU utilization information
- Hardware information (memory, number of CPUs, and model numbers)
- OS information (OS-level, command-style information)
- Pkginfo information (such as installed software)
- LDOM monitoring

Prerequisites

Prerequisite	Restriction
Zenoss Platform	3.1 or greater
Installed ZenPacks	ZenPacks.zenoss.SolarisMonitor
Firewall Acccess	Collector server to 22/tcp and 161/udp of Solaris server
Solaris Releases	OpenSolaris 5.11, Solaris 9 and 10

Limitations

The SolarisMonitor ZenPack does not support monitoring in Solaris Zones or systems containing Solaris Zones. (Implemented with Solaris 10, Solaris Zones act as isolated virtual servers within a single operating system instance.)

Usage

Installation

This ZenPack has no special installation considerations. Depending on the version of Zenoss you're installing the ZenPack into, you will need to verify that you have the correct package (.egg) to install.

- Zenoss 4.1 and later: The ZenPack file must end with `-py2.7.egg`.

- Zenoss 3.0 - 4.0: The ZenPack file must end with `-py2.6.egg`.

To install the ZenPack you must copy the `.egg` file to your Zenoss master server and run the following command as the `zenoss` user:

```
zenpack --install <filename.egg>
```

After installing you must restart Zenoss by running the following command as the `zenoss` user on your master Zenoss server:

```
zenoss restart
```

If you have distributed collectors you must also update them after installing the ZenPack.

Configuring

Depending on the version of Solaris you may be able to monitor the server using either SSH or SNMP. For OpenSolaris and Solaris 10, you can choose to use either SSH or SNMP monitoring. For Solaris 9, only SSH monitoring is supported.

Configuring SSH Monitoring Use the following steps to configure Zenoss to monitor your Solaris server(s) using SSH.

1. Navigate to the `/Server/SSH/Solaris` device class' configuration properties.
2. Verify that the `zCommandUsername` and `zCommandPassword` are set to valid login credentials.
3. Add your Solaris server(s) to the `/Server/SSH/Solaris` device class.

Configuring SNMP Monitoring Use the following steps to configure Zenoss to monitor your Solaris server(s) using SNMP.

1. Verify that the snmpd process is running on your Solaris server(s).
2. Navigate to the `/Server/Solaris` device class' configuration properties.
3. Verify that your Solaris server(s) SNMP community strings are listed in the `zSnmpCommunities` property.
4. Add your Solaris server(s) to the `/Server/Solaris` device class.

Configuring LDOM Monitoring For OpenSolaris and Solaris 10 servers you will also get support for monitoring LDOMs if they're used on the server. However, this monitoring is always performed using SNMP. If you're already monitoring your Solaris server using SNMP there is no additional configuration required to monitor its LDOMs. If you configured Zenoss to monitor your Solaris server using SSH you should take the following steps to monitor LDOMs.

1. Verify that the snmpd process is running on your Solaris server(s).
2. Navigate to the `/Server/SSH/Solaris` device class' configuration properties.
3. Verify that your Solaris server(s) SNMP community strings are listed in the `zSnmpCommunities` property.
4. Remodel your Solaris server(s) if they're already in the system. Otherwise add them to the `/Server/SSH/Solaris` device class.

Removal

Use caution when removing this ZenPack

- Will **permanently** remove devices located in `/Server/SSH/Solaris` device class.
- Will **permanently** remove LDOM modeled components for devices located in `/Server/Solaris`.
- Will **permanently** remove associated monitored data for LDOM components.
- Will **permanently** remove the `/Server/SSH/Solaris` device class.

To remove this ZenPack you must run the following command as the `zenoss` user on your master Zenoss server:

```
zenpack --remove ZenPacks.zenoss.SolarisMonitor
```

You must then restart the master Zenoss server by running the following command as the `zenoss` user:

```
zenoss restart
```

Troubleshooting

Resolving CHANNEL_OPEN_FAILURE Issues The zencommand daemon's log file (`$ZENHOME/collector/zencommand.log`) may show messages stating:

```
ERROR zen.SshClient CHANNEL_OPEN_FAILURE: Authentication failure WARNING:zen.SshClient:Open of commar
```

If the sshd daemon's log file on the remote device is examined, it may report that the MAX_SESSIONS number of connections has been exceeded and that it is denying the connection request. In the OpenSSH daemons, this MAX_SESSIONS number is a compile-time option and cannot be reset in a configuration file.

To work around this sshd daemon limitation, use the configuration property zSshConcurrentSessions to control the number of connections created by zencommand to the remote device:

1. **Navigate to the device or device class in the Resource Manager interface.**
    - **If applying changes to a device class:**
        (a) Select the class in the devices hierarchy.
        (b) Click Details.
        (c) Select Configuration Properties.
    - **If applying changes to a device:**
        (a) Click the device in the device list.
        (b) Select Configuration Properties.
2. Set the `zSshConcurrentSessions` property. Try 10 first, and 2 if that doesn't resolve the problem.

Resolving Command Timeout Issues The zencommand daemon's log file ($ZENHOME/collector/zencommand.log) may show messages stating:

```
WARNING:zen.zencommand:Command timed out on device device_name: command
```

If this occurs, it usually indicates that the remote device has taken too long to return results from the commands. To increase the amount of time to allow devices to return results, change the configuration property `zCommandCommandTimeout` to a larger value.

1. **Navigate to the device or device class in the Resource Manager interface.**

- **If applying changes to a device class:**
  (a) Select the class in the devices hierarchy.
  (b) Click Details.
  (c) Select Configuration Properties.
- **If applying changes to a device:**
  (a) Click the device in the device list.
  (b) Select Configuration Properties.

2. Increase the `zCommandCommandTimeout` property incrementally to a maximum of 240 until the timeout is resolved.

Appendixes

Appendix A: Installed Items

Type	Name	Location
Device Class	/SSH/Solaris	/Devices/Server
Modeler Plugin	df_ag	zenoss.cmd.solaris
Modeler Plugin	kstat	zenoss.cmd.solaris
Modeler Plugin	memory	zenoss.cmd.solaris
Modeler Plugin	netstat_an	zenoss.cmd.solaris
Modeler Plugin	netstat_r_vn	zenoss.cmd.solaris
Modeler Plugin	pkginfo	zenoss.cmd.solaris
Modeler Plugin	process	zenoss.cmd.solaris
Modeler Plugin	uname_a	zenoss.cmd.solaris
Modeler Plugin	hostid	zenoss.snmp.solaris
Modeler Plugin	ldommap	zenoss.snmp.solaris
Monitoring Template	Device	/Server/SSH/Solaris
Monitoring Template	FileSystem	/Server/SSH/Solaris
Monitoring Template	OSProcess	/Server/SSH/Solaris
Monitoring Template	ethernetCsmacd	/Server/SSH/Solaris
Monitoring Template	LDOM	/Server
Monitoring Template	LDOMVcpu	/Server
Monitoring Template	LDOMVds	/Server
Event Class	/Status/LDOM	/
Event Class	/Status/LDOM/vCPU	/
Event Mapping	ldomStateChange	/Change
Event Mapping	ldomVCpuChange	/Change
Event Mapping	ldomVccChange	/Change
Event Mapping	ldomVconsChange	/Change
Event Mapping	ldomVdiskChange	/Change
Event Mapping	ldomVdsChange	/Change
Event Mapping	ldomVmemChange	/Change
Event Mapping	ldomVnetChange	/Change
Event Mapping	ldomVswChange	/Change
Event Mapping	ldomCreate	/Change/Add
Event Mapping	ldomDestroy	/Change/Remove
MIB	SUN-LDOM-MIB	/

Monitoring Templates Device *(/Server/SSH/Solaris)*

- *Data Points*
  - cpu_ssCpuIdle
  - cpu_ssCpuInterrupt
  - cpu_ssCpuSystem
  - cpu_ssCpuUser
  - io_read
  - io_written
  - percent_memory_percentMemUsed
  - percent_swap_percentSwapUsed
  - uptime_laLoadInt1
  - uptime_laLoadInt5
  - uptime_laLoadInt15
  - uptime_sysUpTime
- *Thresholds*
  - CPU Utilization
  - high load
- *Graphs*
  - Load Average
  - CPU Utilization
  - Memory Utilization
  - IO

FileSystem *(/Server/SSH/Solaris)*

- *Data Points*
  - disk_availBlocks
  - disk_availNodes
  - disk_percentInodesUsed
  - disk_totalBlocks
  - disk_totalInodes
  - disk_usedBlocks
  - disk_usedInodes
- *Thresholds*
  - high_disk_usage
- *Graphs*
  - Utilization
  - Inode Utilization

OSProcess *(/Server/SSH/Solaris) - Data Points*

- ps_count
- ps_cpu
- ps_mem
- *Graphs*
  - CPU Utilization
  - Memory
  - Process Count

ethernetCsmacd *(/Server/SSH/Solaris) - Data Points*

- intf_ifInErrors
- intf_ifInPackets
- intf_ifOutErrors
- intf_ifOutPackets
- intf_octets_ifInOctets
- intf_octets_ifOutOctets
- *Thresholds*
  - Utilization 75 perc
- *Graphs*
  - Throughput
  - Packets

LDOM *(/Server)*

- *Data Sources*
  - ldomOperState
- *Thresholds*
  - operational state

LDOMVcpu *(/Server)*

- *Data Sources*
  - ldomVcpuOperationalStatus
  - ldomVcpuUtilPercent
- *Threshold*
  - operational status
- *Graph Definition*
  - CPU Utilization

LDOMVds *(/Server)*

- *Data Source*
  - ldomVdsNumofAvailVolume

  - ldomVdsNumofUsedVolume
- *Graph Definition*
  - Volumes

Appendix B: Required Daemons

In addition to the core platform daemons the following optional daemons are required for this ZenPack to fully function.

- zenperfsnmp
- zencommand

1.2 ZenPack Taxonomy

ZenPacks can be used to override almost any functionality of the Zenoss platform, or to add any new features. This wide-open extensibility is much like Firefox's add-on system and has the natural result of people building some surprising things. For this reason it can be difficult to answer the question, "What is a ZenPack?"

This document will outline the various ways ZenPacks can be classified to make it easier to find the ZenPack you need, describe a ZenPack that already exists, and to serve as an example for what is possible.

1.2.1 ZenPack Classifications

Every ZenPack will be classified using one or more of the elements listed under each classifier property. Some classifier properties have mutually exclusive elements, and some do not. This distinction will be made in the definition of each classifier property. For examples of how some existing representative ZenPacks are classified see *Example ZenPack Classifications*.

Functionality

Functionality classifies the high-level type of feature(s) provided. Specifically its type of interaction with the environment outside of the Zenoss platform. It is highly recommended that these categories be treated as mutually exclusive. While it is possible for a ZenPack to build a ZenPack that delivers functionality in more than one of the following categories, this can lead to less modularity and confusion about what a ZenPack does.

Monitoring

Monitoring is one or more of status, event and performance collection. The collection can be through active polling, passive receiving or both. A monitoring ZenPack provides functionality to perform this collection for a specific target technology.

Example *ZenPacks.zenoss.ApacheMonitor*

Integration

Integration is defined as being any interaction with systems outside of Zenoss not deemed to be a *Monitoring* interaction. Examples include pushing or pulling non-monitoring data to or from an external system, or causing action in a remote system or allowing a remote system to cause action within Zenoss.

Example *ZenPacks.zenoss.RANCIDIntegrator*

Platform Extension

A Zenoss platform extension is defined as any functionality that doesn't interact with outside systems. The provided functionality is instead used directly by users or by other parts of the Zenoss platform, or by other ZenPacks.

Example *ZenPacks.zenoss.DistributedCollector*

Maintainer

The maintainer of a ZenPack is the organization or individual that controls the code repository for a ZenPack and is the gate for all changes including defect and enhancement resolution. The following categories are mutually exclusive.

Zenoss Engineering

Maintained by the product engineering organization at Zenoss, Inc. Ongoing support is provided by Zenoss, Inc. at the same level as the Zenoss platform software for customers with an active subscription.

Example *ZenPacks.zenoss.Impact*

Zenoss Services

Maintained by the services organization at Zenoss, Inc. Ongoing support is provided under a statement of work.

Example *ZenPacks.zenoss.ServiceNowIntegrator*

Zenoss Community

Maintained by a member of the Zenoss community. Ongoing support is subject to the individual maintainers' control, and is typically provided in the community web forums, mailing lists and IRC channel.

Example *ZenPacks.community.ZenODBC*

Availability

Who has access, license and permission to use the ZenPack. The following elements are mutually exclusive.

Open Source

ZenPack source and packages are available as free open source. Designed to function properly on a Zenoss system with or without commercial-only ZenPacks installed.

Example *ZenPacks.zenoss.ApacheMonitor*

Available with Zenoss Subscription

ZenPack packages are available at no extra cost to anyone with a Zenoss subscription, but are not installed by default. May have dependencies on *Open Source* ZenPacks or other ZenPacks that are *Available with Zenoss Subscription*.

Example *ZenPacks.zenoss.DatabaseMonitor*

Additional Cost with Zenoss Subscription

ZenPack packages are available at an additional cost on top of an existing Zenoss subscription. May have dependencies on *Open Source* ZenPacks, ZenPacks that are *Available with Zenoss Subscription*, or other ZenPacks that are *Additional Cost with Zenoss Subscription*.

Example *ZenPacks.zenoss.Impact*

QA Level

The level of automated, manual and field testing A ZenPack has. The elements are mutually exclusive.

Untested

Insufficient automated testing to qualify as *Automatically Tested*, and insufficient manual testing to qualify as *Q.A. Tested*.

Example *ZenPacks.zenoss.OpenStack*

Automatically Tested

Standard automated testing passes plus a minimum of 90% unit test code coverage with all tests passing.

Q.A. Tested

Tested, and passed, by the quality assurance group of Zenoss, Inc.

Complexity

Defined by the technical difficulty of implementing specific types of functionality within the ZenPack. The elements are not mutually exclusive, and most ZenPacks will implement multiple types of functionality as defined below. A rough total complexity score could be created for each ZenPack by summing the complexity score of all implemented elements.

Configuration

Built entirely in the web interface. No programming knowledge required.

Complexity 1

Skills Zenoss

Example *ZenPacks.zenoss.IISMonitor*

Scripts

Scripts can be written in any language and do anything. Since all Zenoss customizations should be packaged as ZenPacks, they're only included in ZenPacks as a packaging mechanism. They might not have any direct interaction with the Zenoss platform.

Complexity 2

Skills Scripting (Any Language)

Example *ZenPacks.zenoss.RANCIDIntegrator*

Command DataSource Plugins

Command datasource plugins can be written in any language and executed either on the Zenoss server, or remotely using SSH. Without writing a custom parser (see next item) they must write to STDOUT using either the Nagios or Cacti output formats and exit using the appropriate Nagios or cacti exit code.

Complexity 2

Skills Scripting (Any Language)

Example *ZenPacks.zenoss.ApacheMonitor*

Event Class Transforms and Mappings

Built in the web interface. Basic Python knowledge required.

Complexity 2

Skills Zenoss, Basic Python

Example *ZenPacks.zenoss.OpenStack*

Command DataSource Parsers

Command datasource parsers must be written in Python and conform to the Zenoss *CommandParser* API. These parsers must be written to extract extended data from the output of command datasource plugins (see previous item), or to handle output that doesn't conform to the Nagios or Cacti output formats.

Complexity 3

Skills Zenoss, Python

Example *ZenPacks.zenoss.SolarisMonitor*

DataSource Types

When a new datasource is added in the web interface you must choose the type. Creating a DataSource type in a ZenPack is a way to add new types to this list. The *ApacheMonitor* ZenPack listed as the example below adds the ability to collect performance metrics from an Apache httpd server using *mod_status*.

New DataSource types are written in Python and must subclass `RRDDataSource` or one of its existing subclasses. Additionally an API adapter must also be written in Python to define the user interface to the datasource properties.

Complexity 4

Skills Zenoss, ZCML, Python

Example *ZenPacks.zenoss.ApacheMonitor*

Impact Adapters

There are three types of impact adapters. All are written in Python and added to the system configuration through ZCML directives.

The first is a state provider. These implement the `IStateProvider` interface and allow manipulation of how a given node type's state within the impact graph is calculated.

The second is a relations provider. These implement the `IRelationshipDataProvider` interface and allow manipulation of what other nodes a given node type impacts, and what other nodes impact it.

The third is a triggers provider. These implement the `INodeTriggers` interface and allow manipulation of the default impact policies set on a given type of node.

Complexity 5

Skills Zenoss, ZCML, Python

Example *ZenPacks.zenoss.ZenVMware*

ETL Adapters

ETL is used to export model, performance and event data from a Zenoss instance to a Zenoss Analytics instance. However, ETL adapters only need to be written to manipulate the *model* data that is exported. There are two types of ETL adapters. They're both written in Python and added to the system configuration through ZCML directives.

The first type is a reportable. These implement the `IReportable` interface and allow precise control over which properties of an object type are exported, and how they're named and manipulated for export.

The second type is a reportable factory. These implement the `IReportableFactory` interface and all manipulation of which objects are considered for export. By default all devices and components are considered for extraction so a reportable factory is usually only used when fine-grained control over the relationships between these objects is needed.

Complexity 4

Skills Zenoss, ZCML, Python

Example *ZenPacks.zenoss.ZenVMware*

User Interface

Modifications to the existing user interface, or entirely new sections of user interface. The difficulty of these changes varies considerably. See the *Skills* field below for the range of skills that could be required to make these kinds of changes.

The *ServiceNowIntegrator* example given below adds a new button to the event console that pops up a new dialog box with some custom options available. Only ZCML and JavaScript were required for this type of change.

TAL is usually only required when editing or creating old-style pages that aren't entirely built using ExtJS.

Complexity 5

Skills Zenoss, ZCML, TAL, JavaScript, ExtJS

Example *ZenPacks.zenoss.ServiceNowIntegrator*

Modeler Plugins - SNMP, COMMAND, WMI

Modeler plugins provide the mapping between data collected from the environment and the Zenoss model. In the case where the data can be collected using SNMP, COMMAND (run a command remotely via SSH) or WMI, there is existing infrastructure to make these tasks easier. However, the modeler plugins are still written in Python.

If collecting using SNMP the `SnmpPlugin` class can be extended to do the hard parts of SNMP gets or walks for you. If collecting by running a command on a remote system via SSH, the `CommandPlugin` class can be extended to do the hard parts of SSH and output parsing for you. If collecting from a Windows system using WMI, the `WmiPlugin` class can be extended to do the hard parts of WQL querying for you.

The only significant logic that must be implemented in these cases is turning the returned data structures into `ObjectMap` and `RelationshipMap` objects to apply to the Zenoss model.

Complexity 6

Skills Zenoss, Python, (SNMP, Scripting or WMI)

Example *ZenPacks.zenoss.SolarisMonitor*

Modeler Plugins - Python

See *Model Extensions* above for what modeler plugins are. Python modeler plugins only differ in that you extend the `PythonPlugin` class, and must implement the collection logic in addition to the processing logic.

The `collect` method implementation may return data normally, or it may return a Twisted `deferred` to take advantage of the asynchronous modeling engine. It is recommended to use the deferred approach whenever possible to avoid blocking the *zenmodeler* daemon while the `collect` method executes.

Complexity 7

Skills Zenoss, Python, Twisted

Example *ZenPacks.zenoss.OpenStack*

Model Extensions

When the standard model of the Zenoss platform doesn't cover an object or property you need in your ZenPack, the model can be extended. Existing model classes such as Device, FileSystem or IpInterface can be extended, and entirely new types of components can be created.

The typical requirements for extended the model include at least the following steps.

1. Create a Python class
2. Create an API interface and adapter
3. Wire up the API with ZCML
4. Write JavaScript to tailor the display of your component
5. Write a *modeler plugin*

   Complexity 8

   Skills Zenoss, ZCML, Python, JavaScript

   Example *ZenPacks.zenoss.OpenStack*

Daemons

A new daemon must be written only if none of the existing daemons can perform the task required by your ZenPack. The `zencommand` daemon is the usual last resort for custom collection requirements if none of the more specialized daemons will work. See *Command DataSource Plugins* and *Command DataSource Parsers* for what can be done by `zencommand`.

There is a common collector framework that should be used to perform much of the typical daemon functionality such as configuration and scheduling in a consistent way. To use this you should create a `CollectorDaemon` object, configure it with a class that implements the `ICollectorPreferences` interface and create a task class that implements the `IScheduledTask` interface.

In almost all cases you will also need to create a ZenHub service to build the configuration for your new daemon. This service should subclass `HubService` or one of its existing more specialized subclasses.

Complexity 9

Skills Zenoss, Python, Twisted

Example *ZenPacks.zenoss.ZenVMware*

Platform Extension

Platform extensions are any implementations added to a ZenPack that doesn't fall into any of the previously-defined complexity elements. Due to the flexibility of ZenPacks, these could be almost anything.

The *DistributedCollector* example given below falls into this category because it extends the simple flat collector structure in the core Zenoss platform to be a tiered hub and collector structure. It also adds extensive hub and collector management capabilities.

Complexity 10

Skills Zenoss, ZCML, Python, JavaScript, etc.

Example *ZenPacks.zenoss.DistributedCollector*

1.2.2 Example ZenPack Classifications

ZenPacks.zenoss.ApacheMonitor

Classification	Value
Functionality	*Monitoring*
Maintainer	*Zenoss Engineering*
Availability	*Open Source*
QA Level	*Q.A. Tested*
Complexity	*Configuration* *Command DataSource Plugins* *DataSource Types*

ZenPacks.zenoss.IISMonitor

Classification	Value
Functionality	*Monitoring*
Maintainer	*Zenoss Engineering*
Availability	*Available with Zenoss Subscription*
QA Level	*Q.A. Tested*
Complexity	*Configuration*

ZenPacks.zenoss.DistributedCollector

Classification	Value
Functionality	*Platform Extension*
Maintainer	*Zenoss Engineering*
Availability	*Available with Zenoss Subscription*
QA Level	*Q.A. Tested*
Complexity	*Configuration* *User Interface* *Platform Extension*

ZenPacks.zenoss.RANCIDIntegrator

Classification	Value
Functionality	*Integration*
Maintainer	*Zenoss Engineering*
Availability	*Available with Zenoss Subscription*
QA Level	*Q.A. Tested*
Complexity	*Configuration* *Event Class Transforms and Mappings* *Scripts*

ZenPacks.zenoss.DatabaseMonitor

Classification	Value
Functionality	*Monitoring*
Maintainer	*Zenoss Engineering*
Availability	*Available with Zenoss Subscription*
QA Level	*Q.A. Tested*
Complexity	*Configuration* *Command DataSource Plugins* *DataSource Types*

ZenPacks.zenoss.ZenVMware

Classification	Value
Functionality	*Monitoring*
Maintainer	*Zenoss Engineering*
Availability	*Available with Zenoss Subscription*
QA Level	*Q.A. Tested*
Complexity	*Configuration* *Event Class Transforms and Mappings* *DataSource Types* *User Interface* *Impact Adapters* *ETL Adapters* *Model Extensions* *Daemons*

ZenPacks.zenoss.SolarisMonitor

Classification	Value
Functionality	*Monitoring*
Maintainer	*Zenoss Engineering*
Availability	*Available with Zenoss Subscription*
QA Level	*Q.A. Tested*
Complexity	*Configuration* *Command DataSource Plugins* *Command DataSource Parsers* *Modeler Plugins - SNMP, COMMAND, WMI*

ZenPacks.zenoss.Impact

Classification	Value
Functionality	*Platform Extension*
Maintainer	*Zenoss Engineering*
Availability	*Additional Cost with Zenoss Subscription*
QA Level	*Q.A. Tested*
Complexity	*Configuration* *User Interface* *Impact Adapters* *Daemons* *Platform Extension*

ZenPacks.zenoss.OpenStack

Classification	Value
Functionality	*Monitoring*
Maintainer	*Zenoss Engineering*
Availability	*Open Source*
QA Level	*Untested*
Complexity	*Configuration* *Event Class Transforms and Mappings* *Command DataSource Plugins* *Command DataSource Parsers* *User Interface* *Impact Adapters* *Modeler Plugins - Python* *Model Extensions*

ZenPacks.zenoss.ServiceNowIntegrator

Classification	Value
Functionality	*Integration*
Maintainer	*Zenoss Services*
Availability	*Available with Zenoss Subscription*
QA Level	*Q.A. Tested*
Complexity	*Configuration* *User Interface* *Model Extensions* *Daemons*

ZenPacks.community.ZenODBC

Classification	Value
Functionality	*Platform Extension*
Maintainer	*Zenoss Community*
Availability	*Open Source*
QA Level	*Q.A. Tested*
Complexity	*DataSource Types* *Modeler Plugins - Python*

CHAPTER 2

Documentation Formats

This documentation is available in the following formats.

- HTML
- PDF
- Epub
- Manpage

CHAPTER 3

Contribution

This documentation is generated automatically every time a change is made to the zenosslabs repository. The most direct route for contribution would be to fork that repository, make the desired change to the documentation and send a pull request.

See GitHub's Fork a Repo documentation if you're unfamiliar with this process. Otherwise, email the address below to have someone incorporate your changes for you.

CHAPTER 4

Contact

Questions and comments related to this documentation should sent to labs@zenoss.com.