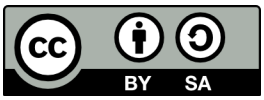
Zenoss

ZenPack Developers' Guide

Version 1.0.1

Revision Date: September 20, 2016



Synopsis

Zenoss is the global leader in hybrid IT monitoring and analytics software, providing complete visibility for cloud, virtual and physical IT environments for more than 40,000 global organizations.

ZenPacks are the extension mechanism provided by Zenoss to build new functionality and also to easily port customization from one Zenoss server to another. This document provides detailed ZenPack design practices, coding techniques and debugging hints, based around a number of sample ZenPacks.

The sample ZenPacks explore:

- Zenoss architecture
- ZenPack architecture
- Creating new object classes and relationships
- The zenpacklib tool
- Creating new collector modeler plugins to populate the new classes with data
- Converting old, non-zenpacklib ZenPacks to use zenpacklib
- Creating code for web pages for new types of objects, both JavaScript and TAL
- Creating new performance datasources and data templates
- Converting COMMAND-based ZenPacks to use the PythonCollector
- Incorporating new event classes, triggers and notifications in ZenPacks
- Creating and extending menus
- Extending functionality using routers and facades
- Logging and debugging
- The process of ZenPack creation, GitHub and ZenPack submission to Zenoss

Another objective of the document is to provide examples of good practice. These tips are highlighted throughout the document with a green tick symbol.

The final main objective is to offer deeper insights into the architecture and functionality provided in the standard Zenoss code, especially with reference to how the ZenPack developer uses and extends this code. These sections should probably be skipped initially by someone first starting out with ZenPacks. There are a number of complete sections and they are prefaced with * . Smaller, in-depth points throughout the paper are highlighted with a yellow asterisk.

At the time of writing (Spring 2016) Zenoss 4.2.5 is the main Zenoss production platform. Many Zenoss 3 implementations are still in use and Zenoss 5.x is emerging among early adopters. The document will major on 4.2.5 practices but will also cover differences for Zenoss 3 and Zenoss 5.

It is assumed that the reader is familiar with basic SNMP and Linux concepts and with standard Zenoss configuration techniques.

This document is based mainly on Zenoss 4.2.5 with zenup fix 457, on a CentOS 6.3 platform. The hostname of the Zenoss 4 server is *zen42.class.example.org*. There are examples specifically for Zenoss 5 (5.0.7), where the Zenoss 5 server is *zen50.class.example.org*.

The Zenoss open source community is fundamental to Zenoss. Jane Curry, an open source community advocate, is the author of this document and we actively invite and encourage collaboration and feedback moving forward. The document will never be "finished"; always a "work-in-progress" as the product and knowledge of the product expands.

Notations

Throughout this document:

- Text to be typed, file names and menu options, are highlighted by *italics*.
- Important points to take note of are shown in **bold.**
- Points of particular note are highlighted by an icon.
- Points of good practice are highlighted with a tick icon
- Subtle or more advanced points are denoted with a star icon.
- Comments to be examined are prefaced with TODO:

ZenPack samples

All the ZenPack samples can be found on GitHub at https://github.com/ZenossDevGuide

Table of Contents

Document history

April 13 2016	Original draft 1.0
April 23 2016	Draft 1.0.3 printed for GalaxZ 2016
June 30 2016	Version 1.0.0. First published version with updates from Zenoss and community.
September 20 2016	Version 1.0.1. Left/right page formatting corrected. Document history added.

1.0 Zenoss concepts

1.1 Background to Zenoss

Zenoss Core first appeared in 2006 as the brainchild of Erik Dahl; an Open Source package for systems and network management to compete with IBM Tivoli, HP OpenView, BMC Patrol and CA UniCenter.

Written largely in Python, Zenoss has an object model at it's heart that encapsulates all the manageable entities such as devices and their components, users, events and monitoring configurations. Zenoss offers:

- Device and component discovery
- Availability monitoring
- Performance data collection, thresholding and graphing
- Collection of events and generation of alerts and corrective automation
- Reporting
- Central dashboard

From the outset, Zenoss has been unashamedly an **agentless** technology. This has strengths and weaknesses but the removal of the requirement to acquire, distribute, update and test agents is very powerful. This means that Zenoss leverages operating system built-in “agents” such as Simple Network Management Protocol (SNMP) agents, Secure Shell (ssh), Windows Management Instrumentation (WMI) and latterly Windows Remote Management (WinRM). Common Zenoss add-ons increase this list with http, JMX, the Python Twisted asynchronous communications library, SQL, Lightweight Directory Access Protocol (LDAP) and various Application Programming Interfaces (APIs) such as Amazon and VMware.

The Graphical User Interface (GUI) for Zenoss is built on the **Zope** web application environment.

A chargeable version of Zenoss appeared around 2008, known as Zenoss Enterprise. Initially this offering was largely a way of providing support to Zenoss Core users. Latterly extra functionality has been built in to the chargeable version and it has been renamed to Zenoss Service Dynamics, with the central product being Zenoss Resource Manager.

1.2 Devices, components, object classes and device classes

Zenoss discovers and monitors various entities:

- **Devices** - typically physical or logical hardware with an IP address; examples are Linux and Windows servers, network routers, switches and firewalls, printers and power supplies. Devices may also be application-like, such as HTTP websites.
- **Components** - many devices have components. Most have one or more interfaces. Servers have file systems and CPUs; network devices have routing tables and some have VLAN components. Components may be more application-like, for example a server might have an Operating System process as a component.

All devices and components are represented in the Zenoss Object Database (**ZODB**) as objects, with various attributes such as id and IP address, and can have one or more relations

(note that the terms *relations* and *relationships* are used interchangeably throughout Zenoss documentation). For example, most devices have a relation that associates them with their interface component objects and each interface component object has a relationship back to its parent device.

Different device and component types are implemented as python object classes. The file *Device.py* contains the definition of the *Device* object class that defines the fundamental attributes, relations and methods that all devices have. A method is a piece of Python code that "does something" to an object; for example, set the *manageIp* attribute to a new value.

An object class provides a template for each item and an *instance* is a manifestation of the class. The *Device* object class specifies attributes such as *manageIp*, *title*, *snmpSysName. T*he device instance *zen50.class.example.org* is an example of that class where the attributes are populated with discovered values.

Object classes are typically hierarchical. I*pInterface.py* defines the *IpInterface* component class, which inherits attributes, relations and methods from its parent *OSComponent* class, which inherits from two parent classes, *DeviceComponent* and *ManagedEntity*. An object becomes more specialized the further down the object tree it is. A class can define new attributes, relations and methods and can also redefine inherited characteristics. For example the *DeviceComponent* class defines a method *manage_deleteComponent* this is redefined in the *OSComponent* class and redefined again in the *IpInterface* class.

Each device has a zProperty, *zPythonClass*, which specifies the Python object class that defines this device instance. The default is the fundamental *Device*, shipped as standard with the Zenoss code. ZenPacks often create new device object classes and then reassign *zPythonClass* to match.

Devices are organized in a *Device Class hierarchy*. Fundamentally, a device can only be associated with a single Device Class. For example, */Network/Router/Cisco* or */Server/Microsoft/Windows*.

DeviceClasses are used to house the monitoring configuration for the Devices that are contained in the hierarchy below it. Monitoring configuration such as MonitoringTemplate definitions, MonitoringTemplate bindings, modeler plugin selection, configuration property values (*zProperties* and *cProperties*) can all be set at the DeviceClass level and will apply to all Devices below the DeviceClass where they are set, recursively through sub-DeviceClasses, unless overridden.

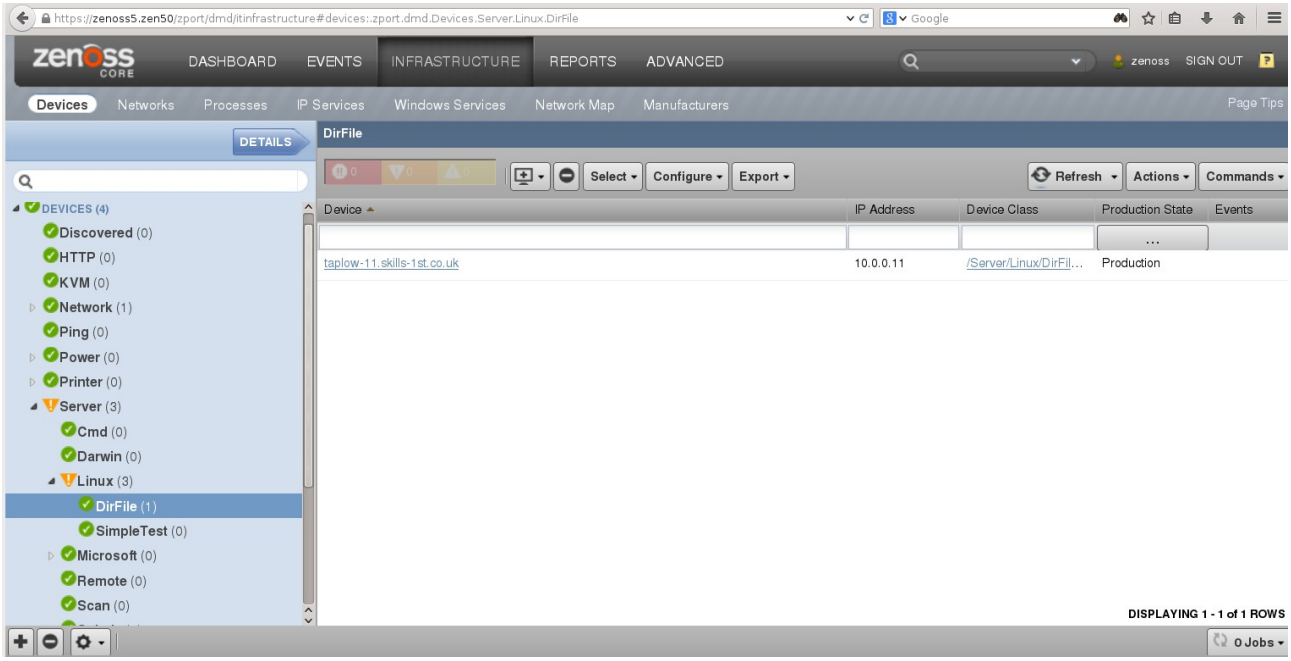


Figure 1: Device class hierarchy

The device class hierarchy is a specialized implementation of an **Organizer** hierarchy, that provides file system like paths such as */Devices/Servers*. Device Organizers have a containment relation called *children* to access its members. Figure 1 demonstrates the DeviceClass hierarchy where each device class "child" is shown indented one level from its "parent"; the main part of the window displays the members of the currently highlighted device class, including any children of the highlighted class; thus highlighting */Server/Linux* will show all devices under */Server/Linux,* */Server/linux/DirFile* and */Server/Linux/SimpleTest*.

Zenoss Systems, Groups and Locations are also examples of Organizer hierarchies that Devices can belong to although they are not used to house monitoring configuration and only provide additional context for the Devices and the events they generate.

1.2.1 Zenoss monitoring functionality

Zenoss delivers the following:

Discovery of devices

- Performed by the **zendisc** daemon either on demand or periodically by using a script (possibly run through the cron scheduler). Typically *ping* is the initial discovery mechanism, although other mechanisms can be used. Devices are inserted into the ZODB database with an initial Device Class and the characteristics that are implied by that DeviceClass.

Modeling

- **Configuration** polling of previously discovered devices. This can include attributes (such as amount of memory on a Server type device) and will often encompass the discovery / updating of a device's components. Attributes, relations and instances of relations are added / updated in the ZODB object model. This is performed by one or more **modeler plugins,** run by the **zenmodeler** daemon, either on a periodic basis or on demand. By default, zenmodeler runs every 12 hours from when the daemon is started.

- Configuration properties, also known as **zProperties**, can be used to configure the way the data is collected. Examples include the SNMP community name, the username for access through ssh or a Windows protocol, and interfaces to ignore, for example *loopback*.

Configuration inheritance

- The object-oriented inheritance paradigm enables child objects (Devices) to inherit or acquire attributes from their "parents". For example the DeviceClass that contains the Device. This applies to all monitoring configuration so a modeler plugin configured for DeviceClass */Server* will, by default, be inherited by the DeviceClass */Server/Linux* and all its children. This also includes the list of performance templates to be applied to a device. Any attribute can be overridden at a lower level down the hierarchy.

Device availability

- Typically this is achieved by the **zenping** daemon pinging every device (every 1 minute, by default).

Monitoring

- Monitoring can be applied to devices and components of devices. This is **performance** monitoring and typically takes place every 5 minutes. **Monitoring Templates** specify what metrics to monitor, how to retrieve the data, and how frequently to poll for the data. Threshold values can be specified for a metric and graphs can be configured to display combinations of the metrics.
- Performance monitoring is performed by various daemons, for example:
  - SNMP data by **zenperfsnmp**
  - ssh data by **zencommand**
  - **zenpython** retrieves data using python programs
  - **zenprocess** checks for the availability of discovered processes, by default every 3 minutes.
  - Many ZenPacks extend these standard daemons and can create additional collection daemons.

Collectors

- A collector is the logical collection of Zenoss daemons that gathers information from a specified group of target Devices to populate the central Zenoss databases. For example, *zenping*, *zenperfsnmp*, and *zenpython*.
- A typical Zenoss Core installation has **hub** functionality and a single “localhost” collector on the Zenoss server. Extra collectors can be deployed on separate servers.
- Zenoss Service Dynamics can have multiple hubs and multiple collectors on different machines.
- While distributing performance monitoring load is one reason for deploying extra collectors, sometimes network performance or firewall constraints will require that collector be deployed closer (in a networking sense) to the devices it monitors.

Events

- These are typically a record of a change of status of an object. Often events indicate problems with the Devices or their environment(s). For example, a performance Monitoring Template can set a threshold on disk space at 80% used, generating an event when the threshold is exceeded. A clear event will be generated when the disk space metric returns to less than 80% used.

Notifications

- A message, typically to a user, that an event has occurred. Several methods of sending notifications are available in the product. Although email is the most commonly used, paging, automation commands and SNMP TRAPs are also supported. Fields from the causal event can be used to generate the body of the notification.

- In Zenoss 4 and 5 a Notification is driven by a **Trigger** that is a set of criteria that determines **when** to send the notification. The trigger is a combination of required values for different fields of an event; for example *Device Production Status = Production AND severity greater than Warning AND status = New AND EventClass startswith '/DirFile'*.

- In Zenoss 3, Notifications and Triggers were implemented by user-specific **Alerting Rules**.

1.2.2 Standard conventions for Zenoss code and ZenPacks

By default, recent versions of Zenoss are installed under */opt/zenoss*. The installation directory is known as ***$ZENHOME*** and this environment variable should be made available to the *zenoss* user.

1.3 Zenoss Daemons

Zenoss uses many daemons, grouped into four areas:

- Daemons to present the Graphical User Interface (blue in Figure 2)
- Central hub daemons (red in the Figure 2)
- Daemons to manage the flow of events (green in the Figure 2)
- Data collection daemons (purple in the Figure 2)

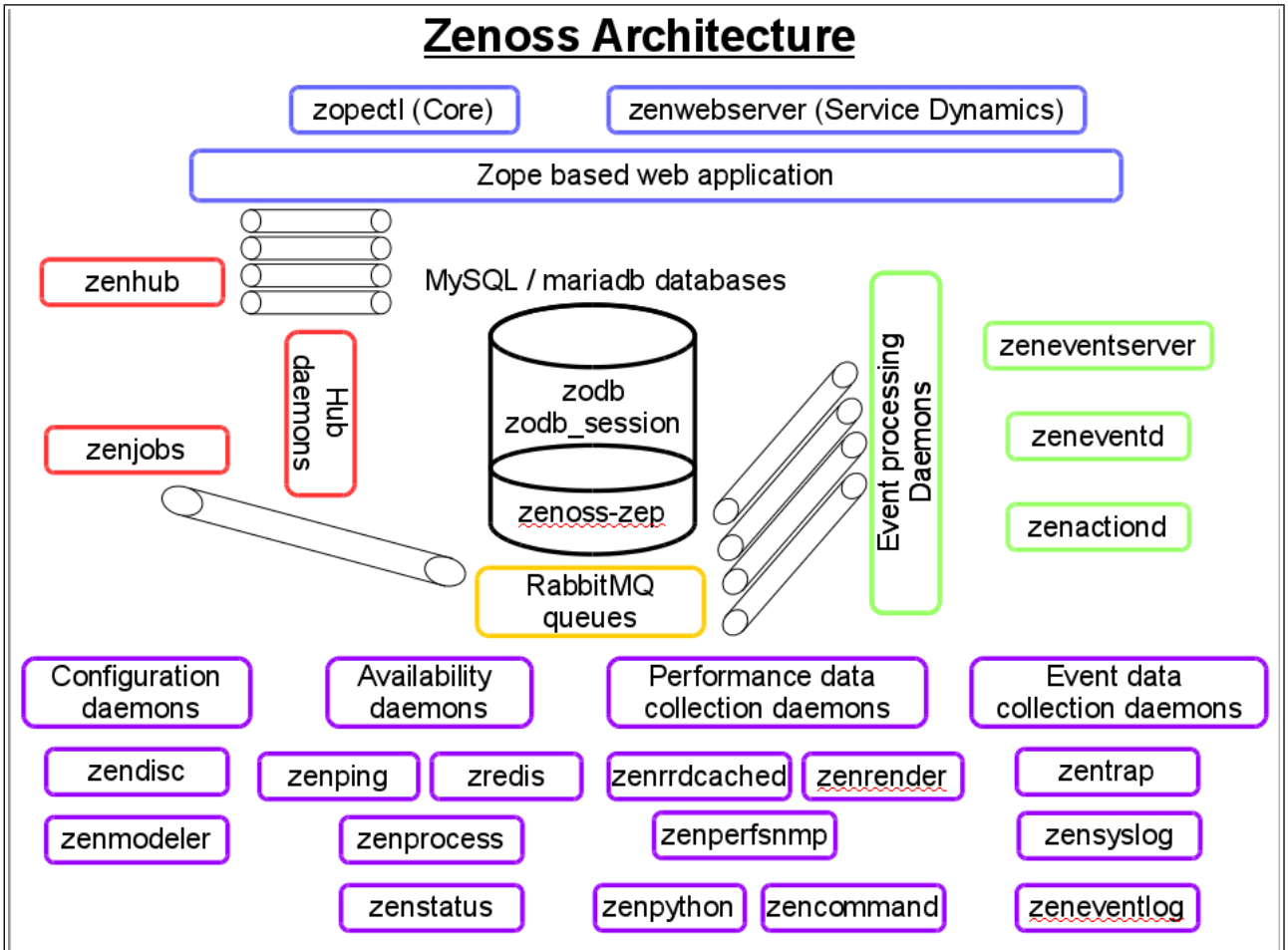


Figure 2: Zenoss daemon architecture

Note that Zenoss Service Dynamics replaces Core's **zopectl** daemon with **zenwebserver** (simply **webserver** in Zenoss SD 5.x). It manages Zope instances and controls the NGINX load balancer managing traffic to and from the Zope instance(s). Service Dynamics has an extra **zencatalogservice** daemon that maintains the Lucene index into the ZODB database to provide enhanced performance.

zenhub is the central daemon that brokers communication between collection daemons and other Zenoss processes. It can have assistant **zenhubworker** processes to increase performance.

zenrrdcached and **zenrender** will not exist in Zenoss 5 implementations as performance data is saved in an OpenTSDB database instead of Round Robin Database (rrd) files.

The **zredis** daemon provides a shared repository for all *zenping* daemons and facilitates correlation of *Ping Down* events.

In Figure 2, the daemons shown in purple are typically those that are active on a Zenoss collector.

A database system is required to store configuration and event data. In Zenoss Service Dynamics, this is an SQL database, known as *zends*. In Zenoss 5, this function is provided by a pair of mariadb databases.

From Zenoss 4 onward, RabbitMQ is used to provide Message Queueing technology for data to pass between various daemons, largely in the events subsystem:

Description	Between daemons
celery	zenhub and zenjobs
rawevents	zenhub and zeneventd
zenevents	zeneventd and zeneventserver
archive	zeneventserver and the zenoss_zep databases
migrated	zeneventserver and the zenoss_zep databases
heartbeat	zenhub and zeneventserver
modelchange	zenhub and zeneventserver
signal	zeneventserver and zenactiond

The relationship between the queues, databases and the event processing subsystem is shown in Figure 3.

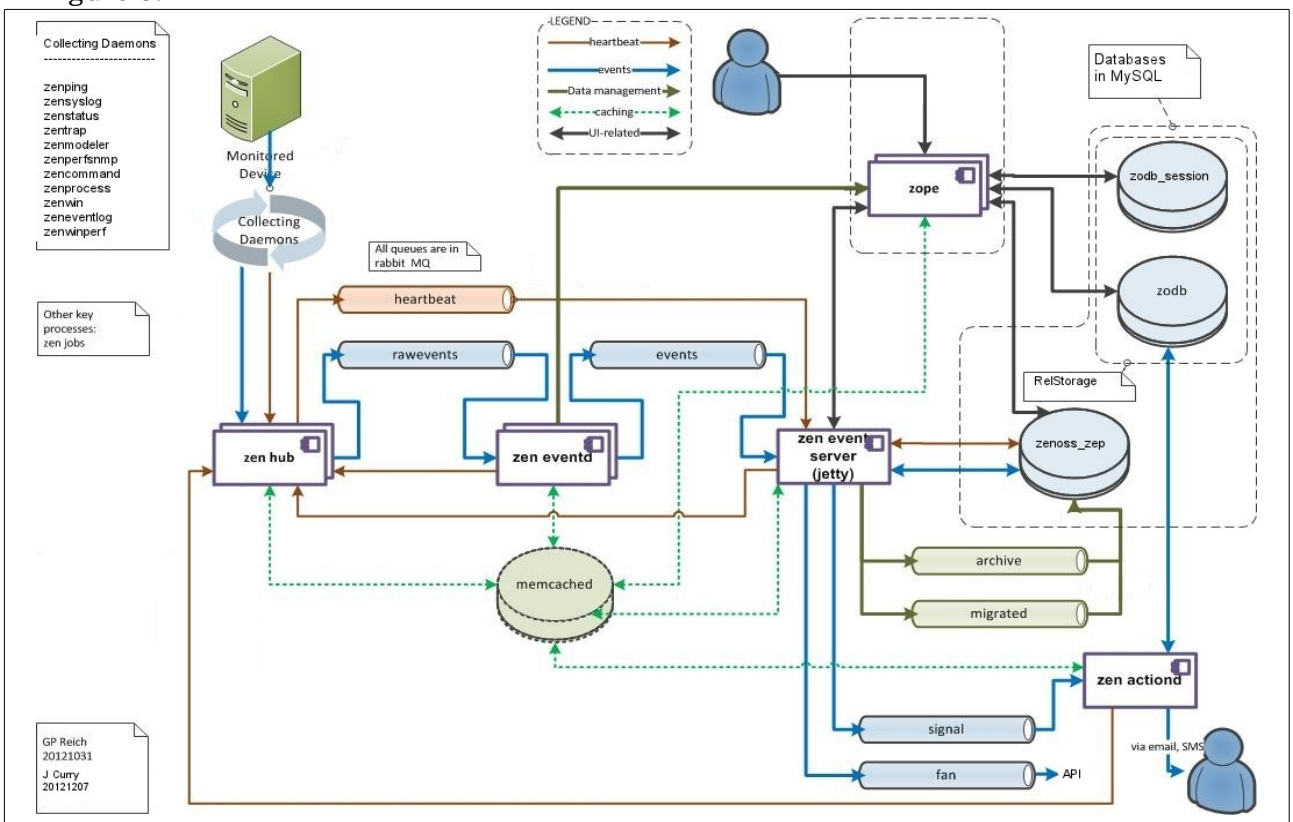


Figure 3: Event architecture showing daemons, queues and databases. With thanks to Georges Reichs.

1.4 Zenoss 5 docker architecture

Zenoss 5 architecture includes new functionality that is provided by docker and containerization. The following documents provide additional information:

- Zenoss Documentation Guides at https://www.zenoss.com/resources/documentation that include:
  - *Resource Manager Installation Guide*
  - *Resource Manager Upgrade Guide*
  - *Resource Manager Planning Guide*
  - *Resource Manager Administration Guide*
- *Zenoss Service Dynamics Architecture Overview* – a technical white paper available by request from http://pages.zenoss.com/WCZSD-WP.html
- Useful documents from the Zenoss KB repository:
  - *Virtualization and Docker Containerization for Poets* (https://support.zenoss.com/hc/en-us/articles/202254069)
  - *Introduction to Zenoss Control Center* (https://support.zenoss.com/hc/en-us/articles/206278353)

TODO: This needs extending.

1.5 Extending Zenoss out-of-the-box functionality

Although Zenoss ships with a rich set of functionality, there are always additional devices and applications required by various users. **ZenPacks** are a method to build add-on functionality to the core Zenoss offering.

ZenPacks might contain extra objects for the ZODB database; new event classes, SNMP MIBs or customized performance templates. Such ZenPacks can be created entirely through the GUI and require no programming expertise.

Alternatively, a ZenPack can deliver management for new types of devices, with new types of components, and provide new menus to help manage those devices with new methods of collecting and displaying the data. Such ZenPacks require some Python programming skills and an underlying knowledge of the Zenoss architecture.

2.0 What are ZenPacks?

ZenPacks are the method of extending standard Zenoss functionality. There are four different sources of ZenPacks. The http://wiki.zenoss.org/ZenPack_Catalog provides a list of available ZenPacks for download.

2.1 Sources for ZenPacks

2.1.1 Free ZenPacks developed by Zenoss

These are developed and maintained by Zenoss and are available to both Zenoss Core and Zenoss Service Dynamics users.

With Zenoss 5 and some deployments of Zenoss 4, various core ZenPacks are installed by default when Zenoss is installed. The following is an example list of ZenPacks for a new Zenoss Core 5.0.7 installation:

ZenPacks.zenoss.ApacheMonitor	ZenPacks.zenoss.Dashboard
ZenPacks.zenoss.DellMonitor	ZenPacks.zenoss.DeviceSearch
ZenPacks.zenoss.DigMonitor	ZenPacks.zenoss.DnsMonitor
ZenPacks.zenoss.FtpMonitor	ZenPacks.zenoss.HPMonitor
ZenPacks.zenoss.HttpMonitor	ZenPacks.zenoss.LDAPMonitor
ZenPacks.zenoss.LinuxMonitor	ZenPacks.zenoss.Microsoft.Windows
ZenPacks.zenoss.MySqlMonitor	ZenPacks.zenoss.NtpMonitor
ZenPacks.zenoss.PythonCollector	ZenPacks.zenoss.ZenJMX
ZenPacks.zenoss.ZenMail	

There are additional free, Zenoss-developed ZenPacks available at http://wiki.zenoss.org/Free_ZenPacks_by_Zenoss, some of which are good examples of writing ZenPacks, such as *ZenPacks.zenoss.AWS*, *ZenPacks.zenoss.RabbitMQ* and the two OpenStack ZenPacks.

2.1.2 Community developed ZenPacks

There are a vast number of ZenPacks developed by the community and made freely available. Although support for such offerings will vary, as open source developments, anyone has the potential to correct, update and maintain them. Start here - http://wiki.zenoss.org/Community_ZenPacks - for community ZenPacks. Be aware that some of these ZenPacks are old and may not function correctly with later versions of Zenoss - check documentation carefully.

2.1.3 Chargeable Zenoss ZenPacks

Zenoss have some ZenPacks that they only make available to Service Dynamics customers. Information can be found at http://wiki.zenoss.org/Commercial_ZenPacks_by_Zenoss but, unlike the other categories, you cannot download code from here.

2.1.4 Write your own ZenPack!

If you need to extend Zenoss in a way that has not yet been done, you can write your own ZenPack and contribute it back to the collection of community ZenPacks. The rest of this document provides guidance.

Alternatively, Zenoss Professional Services offer consultancy to customers to implement new ZenPacks.

2.2 ZenPack basics

ZenPacks are packaged as Python **eggs**. This is a standard Python mechanism consisting of a zipped file containing all the files that make up the package. To see what a .egg file contains, unzip it.

This ZenPack packaging is performed automatically for you and you don't need to get into the details of eggs.

Typically ZenPacks are stored on **GitHub** in repositories owned by developers. Zenoss repositories are at https://github.com/zenoss. A community ZenPack can be under the developers name, for example https://github.com/jcurry .

ZenPacks are often used for three main reasons:

- Adding monitoring support for new types of devices and components
- Porting Zenoss configuration from one Zenoss server to another (with care)
- Modifying or extending the Graphical User Interface

Many of the standard Zenoss Graphical User Interface (GUI) menus have an *Add to ZenPack* option; thus event classes, event commands, user commands, device classes, service classes, process classes, reports and product definitions as well as the data sources, graphs and thresholds of performance templates, can be simply added to a ZenPack using the GUI (a **simple** ZenPack).

A ZenPack can also add daemons, new device types and user interface features such as menus but this requires programming effort (a **complex** ZenPack). A complex ZenPack generally requires some Python code and might require JavaScript, HTML / XML and bash skills.

With Zenoss 4.2 and Zenoss 5 the **zenpacklib** utility (sometimes abbreviated to **ZPL**), provided by Zenoss, can dramatically reduce the amount of formal code that needs to be developed.

Consult the *Zenoss 4 Administration Guide* or *Zenoss 5 Administration Guide* for a short introduction to ZenPacks.

The Zenoss GUI provides a way to create, examine, export and delete ZenPacks from the A*DVANCED -> Settings -> ZenPacks* menu (provided you have at least **ZenManager** authority). Note that Zenoss 5 no longer allows ZenPack creation or deletion through the GUI.

However trivial the ZenPack is, it should always have a *README.rst*, a documentation file in **reStructuredText** format; see http://docutils.sourceforge.net/rst.html . This file should be in the top-level directory of the ZenPack.

2.3 Existing ZenPack documentation

Documentation for ZenPacks exists in many places, in many forms and with varying degrees of currency, accuracy and detail.

2.3.1 High-level documentation

Some ZenPack documentation is made available by Zenoss. The site https://zenosslabs.readthedocs.org/en/latest/ offers guidance on ZenPack documentation and taxonomy. It is also available in pdf format at http://media.readthedocs.org/pdf/zenosslabs/latest/zenosslabs.pdf .

The documentation section starts with a mandate that **every** ZenPack should have a *README.rst* (in reStructuredText format) in the top-level directory, and documents the proposed layout of that file.

The taxonomy section offers some interesting insights into ZenPack **complexity** on a scale of 1 - 10 where 1 is a ZenPack requiring no coding and 10 is fundamentally extending the Zenoss platform . This section also documents the skills required to develop the different elements of a ZenPack, such as Zenoss knowledge, Python, scripting, Twisted libraries, JavaScript, ZCML, etc.

The last part of this document gives example ZenPack classifications (many of which are only available to Service Dynamics customers) but which are still interesting as a way of understanding complexity.

There is a useful page on the Zenoss wiki at http://wiki.zenoss.org/ZenPack_Development_Guide including links to excellent, publicly-available sample ZenPacks. There is also a link to ZenPack web-based training.

http://wiki.zenoss.org/ZenPack_Development_Guide/Development_Environment provides Zenoss wiki tips for creating a ZenPack development environment.

2.3.2 zenpacklib documentation

The “readthedocs” page has a link to http://zenpacklib.zenoss.com/en/latest/ . zenpacklib emerged during 2015 as a library designed to simplify the coding of ZenPacks. Its documentation pages change earlier, more specific documentation that used some excellent examples written by Zenoss's Chet Luther.

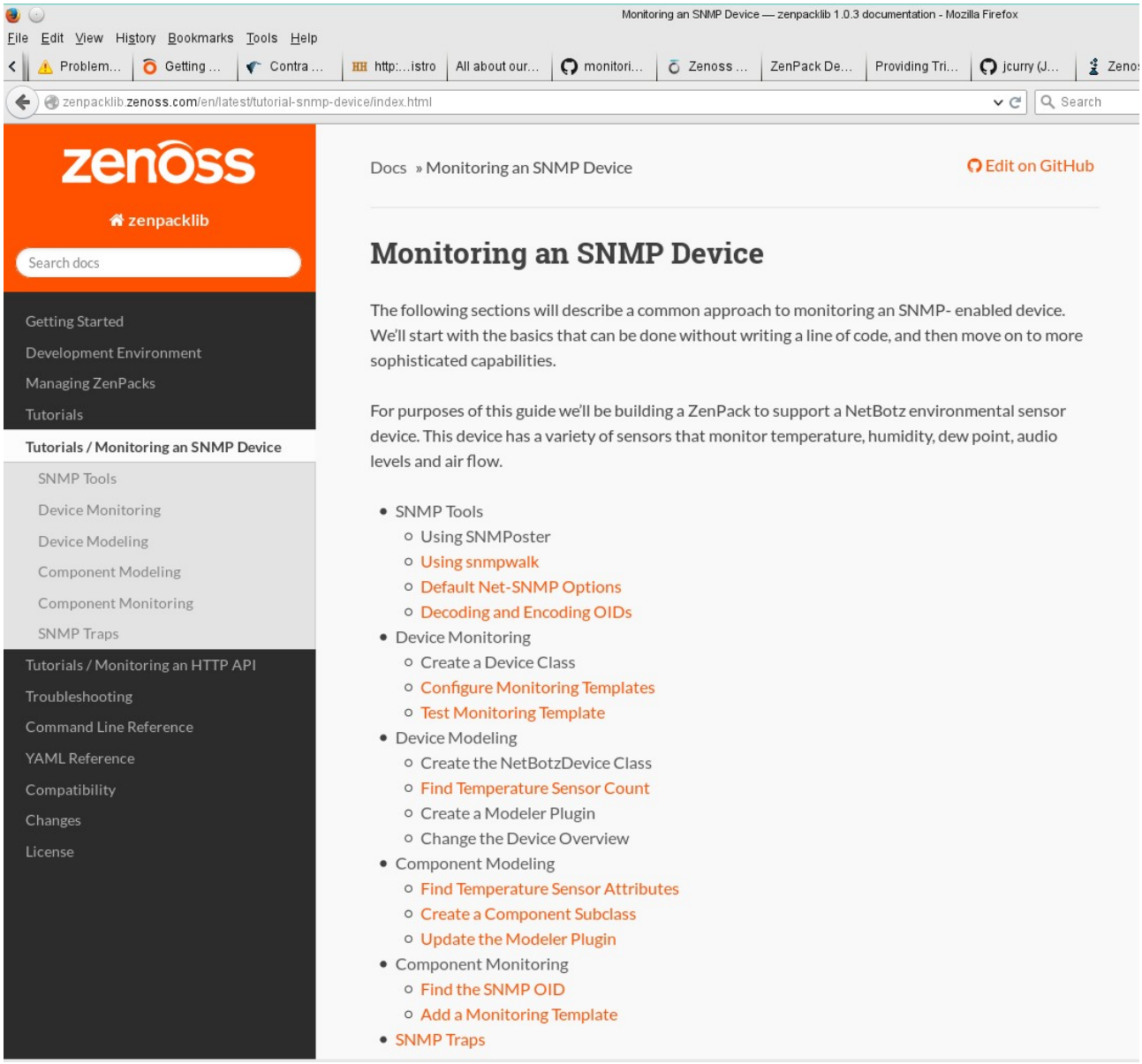


Figure 4: zenpacklib documentation

zenpacklib does not address all issues. For example, it does not simplify writing modeler plugins or custom datasources. This means it is necessary to thoroughly understand the constructs that zenpacklib simplifies for you.

2.3.3 Standard Zenoss documentation

Each version of the Zenoss product Administration Guide, including Zenoss 5, contains basic ZenPack information. For official Zenoss documentation, visit : https://www.zenoss.com/ and navigate to:
Support > Documentation.

Zenoss 3 provided a Zenoss Developer's Guide, published in 2010, with much more in-depth information about coding ZenPacks but, although much of this information is still relevant, much is also out of date. This document appears to have disappeared from Zenoss's own websites but I found it in October 2015 at
http://docs.huihoo.com/zenoss/3/Zenoss_Developers_Guide_08-102010-3.0-v01.pdf .

2.3.4 Community ZenPack documentation

The wiki site has a complete category for ZenPacks, with documentation and download links. See http://wiki.zenoss.org/Category:ZenPacks .

ZenPacks are typically stored on GitHub where the associated *README.rst* should be the best specific documentation. The Zenoss area is reached at https://github.com/zenoss . As an example of community GitHub ZenPacks, my github home is https://github.com/jcurry .

There is a large paper on "Creating Zenoss ZenPacks" at http://www.skills-1st.co.uk/papers/jane/zenpacks/ and a more specific paper on "Zenoss Datasources through the eyes of the Python Collector ZenPack" at http://www.skills-1st.co.uk/papers/jane/PythonZenPacks.pdf .

David Buler wrote an excellent *ZenPack Development Procedures* guide in 2010 that is very useful as an introduction to using GitHub to store and share ZenPacks.

3.0 The mechanics of building a ZenPack

3.1 ZenPack development environment

Use of developed ZenPacks can be dangerous! Even a trivial ZenPack developed entirely through the GUI, can cause unexpected issues, especially if exported to a different system. For this reason an organization should **always** have a test environment.

ZenPack work should always be performed as the *zenoss* user with all ZenPack directories and files owned by user *zenoss* and, typically, group *zenoss*. File permissions vary slightly depending on the ZenPack creation mechanism but either :

```
-rw-rw-r-- 1 zenoss zenoss 2418 Oct 14  2014 HPCPUMap.py          or
-rw-r--r-- 1 zenoss zenoss 2418 Oct 14  2014 HPCPUMap.py
```

is acceptable (zenpacklib defaults to -rw-rw-r-- and other creation methods default to -rw-r--r--). Note that files in any *libexec* directory should also be executable by the *zenoss* user.

3.1.1 Zenoss 4 and earlier

For simple ZenPack development through the GUI, no special environment is required.

When developing a complex ZenPack that involves writing code, it is recommended as a best practice to restart all Zenoss daemons whenever the ZenPack is removed/installed/updated.

NOTE: While it is possible that some daemons may not be affected by changes made in ZenPack development, until the implications of the changes are fully understood and the daemons that are affected, restart all daemons.

Often only a small set of daemons is required for development and testing. For example , the *zenjmx* daemon isn't required to test a ZenPack that doesn't involve java/jmx.

Two files are required to specify a minimal set of Zenoss daemons in *$ZENHOME/etc*:

- *DAEMONS_TXT_ONLY* must exist, probably zero-length
- *daemons.txt* one daemon per line

Typically, *DAEMONS_TXT_ONLY* (note all in capitals) is created with the *touch* command.

The *daemons.txt* file has one daemon per line for each daemon to be activated. A minimal list of daemons for Zenoss Core is:

- zeneventserver
- zopectl
- zeneventd
- zenhub
- zenjobs

Zenoss Service Dynamics also requires ***zencatalogservice***.

Lines can be commented out with #.

NOTE: To make this file unavailable and revert to all daemons operational, rename the file to *hide_ DAEMONS_TXT_ONLY*. *D*o not rename it by adding a suffix to the file name because some versions of Zenoss **will find and use** a file with a suffix, such as *DAEMONS_TXT_ONLY_hide*.

Note that *daemons.txt* is used for all *zenoss* commands - *zenoss start, zenoss stop, zenoss status*.

For additional debugging, the core daemons can be run in the foreground, leaving just *zeneventserver* in *daemons.txt* to run as a daemon. Start the daemons in separate command windows with the following commands:

```
zopectl fg
zeneventd run -v10
zenhub run -v10 --workers=0
zenjobs run -v10 --cycle
```

The "*-v10*" provides full debugging output to stdout and will create very verbose logs.

3.1.2 Zenoss 5

See http://zenpacklib.zenoss.com/en/latest/development-environment-5.html for excellent advice on developing ZenPacks in a Zenoss 5 environment.

Because Zenoss 5 is built on top of **Control Center,** a **docker** implementation, extra steps must be taken to attach to each isolated container. The Zenoss environment consists of many docker **containers** which run one or more processes; each Zenoss daemon has its own isolated container.

Control Center provides a snapshot mechanism to provide a **snapshot** backup facility before performing ZenPack installation. The procedure is outlined here; see the Zenoss *Resource Manager Administration Guide* for additional information on performing snapshots.

1. Log in to the Control Center browser interface.
2. In the Applications table, click on the name of the Zenoss Core instance to modify.
3. Stop the *Zenoss* service, and then verify its subservices are stopped.
4. Create a snapshot.
   a. Log in to the Control Center base host as a user with Control Center CLI privileges (typically the *zenoss* user).

b. Create a snapshot

```
serviced service snapshot Zenoss.core
```

c. The serviced command returns the ID of the new snapshot on completion.

5. Restart the Zenoss service.

TODO: Do we actually need to stop the Zenoss service before taking the snapshot?

3.1.2.1 zenoss user

ZenPack development should be performed as the *zenoss* user, which exists in various containers but does not typically exist on the Zenoss server base host. It will become necessary to share files between the base host and containers so standardizing the *zenoss* user throughout is useful.

To create a *zenoss* user on the base host, you will probably need root privilege. You should also create a *zenoss* user group.

```
groupadd --gid=1206 zenoss
adduser --uid=1337 --gid=1206 zenoss
```

It is **essential** that the *gid* and *uid* are the same as in the containers. The numbers given here are the usual defaults from a standard Zenoss installation.

The *zenoss* user will need to be able to run *sudo* commands and *docker* commands so should be added to the ***wheel*** and ***docker*** user groups.

```
usermod -a -G wheel zenoss
usermod -a -G docker zenoss
```

The *zenpack* command must be run from within the **Zope** container. Other commands need to be run from other containers. It is extremely tedious to attach to a container, switch user, run a command and exit the container. The *zenoss* user's *.bashrc* file (run whenever the user logs in) can be used to create aliases that make this process much more streamlined. Add the following to the end of */home/zenoss/.bashrc* on the base host (note the leading dot on bashrc):

```
alias zope='serviced service attach zope su zenoss -l'
alias zenhub='serviced service attach zenhub su zenoss -l'

z () { serviced service attach zope su zenoss -l -c "cd /z;$*"; }
zp () { serviced service attach zope su zenoss -l -c "cd /z/zenpacks;$*"; }
zenbatchload () { z zenbatchload $*; }
zendisc () { z zendisc $*; }
zendmd () { z zendmd $*; }
zenmib () { z zenmib $*; }
zenmodeler () { z zenmodeler $*; }
zenpack () { zp zenpack $*; }
zenpacklib () { zp ./zenpacklib.py $*; }
zenpython () { z zenpython $*; }
zencommand () { z zencommand $*; }
zenperfsnmp () { z zenperfsnmp $*; }
zenactiond () { z zenactiond $*; }
zeneventd () { z zeneventd $*; }
```

```
zeneventserver () { z zeneventserver $*; }
```

The *zenoss* user will need to relogin before these aliases become active. These are the common daemons; it may be useful to add others, depending on your environment and the ZenPacks that are installed.

- The first two lines attach to the running *zope* and *zenhub* containers, respectively. Further commands can then be run from that container environment.
- The "z" line provides:
  - A function to attach to the *zope* container as the *zenoss* user
  - Change directory to */z* (a directory shared between containers and the base host - see later)
  - Run the command name, passing any parameters that were provided
  - Exit back to the base host
- The "zp" line is similar to *z* but changes directory to */z/zenpacks*. This assumes that ZenPack development work will be performed under */z/zenpacks*.
- The rest of the lines define commands that can now be executed from the base host, to be implemented in the zope container, in the context of the current directory being */z* or */z/zenpacks*, as appropriate.

The *zenoss* user is configured, by default, as an account that cannot be directly logged in to (and always has been from very early versions of Zenoss); the *zenoss* account was accessed using *su - zenoss*. With Zenoss 5, it is advantageous to change that, simply by setting a password for the *zenoss* user (for which you will probably require root or sudo privilege):

```
passwd zenoss
```

Note that the *zenoss* user on the base host does **not** share the same home directory as that in the containers. This includes differing *.bashrc* scripts and differing *.ssh* directories. A consequence is that any direct ssh communications tests that will update the *zenoss* user's *.ssh/known_hosts* file **must** be conducted from within a container.

3.1.2.2 Common directory between containers and the base host - /z

Moving data between the base host and containers is non-trivial. To create a common directory, */z,* on the base host, as the root user, run:

```
mkdir -p /z
chown -R zenoss:zenoss /z
```

The */z* directory is now owned by the *zenoss* user, *zenoss* group.

To make */z* available to all containers as well as the base host, configure the Control Center's ***serviced*** daemon by editing */lib/systemd/system/serviced.service* on the base host adding a mount argument to the end of the *ExecStart* line:

```
ExecStart=/opt/serviced/bin/serviced --mount *,/z,/z
```

The configuration must be reloaded and the service restarted:

```
systemctl daemon-reload
systemctl restart serviced
```

Once */z* has been created and made universally available, further subdirectories can be created following a local convention; for example:

```
/z/zenpacks                         for ZenPack development
/z/packages                         for Operating System packages
/z/scripts                          for local scripts
```

To test that both the host and containers can read and write files in */z,* use the following on the base host:

```
su - zenoss # becomes zenoss user on host
touch /z/host
serviced service attach zenhub # attach to a container
su - zenoss # becomes zenoss user in container
rm /z/host
touch /z/container
exit # back to container root user
exit # back to host zenoss user
rm /z/container
exit # back to host root user
```

3.1.2.3 Configuring the service for a development minimum

Out of the box, at least in Zenoss.resmgr, Zope is configured to run a minimum of two instances. This is problematic when you insert a breakpoint (*pdb.set_trace()*) in code run by Zope because you can't be sure the breakpoint will occur in the instance of Zope you happen to be running in the foreground.

Run the following command to edit the Zope service definition. This will open *vi* with Zope's JSON service definition.

```
serviced service edit Zope
```

Search this file for *"Instances"* (with the quotes). You should see a section that looks something like the following. Change *Instances*, *Min*, and *Default* to 1. Then save and quit.

```
"Instances": 6,
"InstanceLimits": {
 "Min": 2,
 "Max": 0,
 "Default": 6
},
```

Restart Zope with:

```
serviced service restart Zope
```

As with earlier versions of Zenoss, a development environment probably does not need all the standard Zenoss daemons running. To prevent running unwanted daemons in unwanted serviced containers, edit the appropriate service definition file; for example:

```
serviced service edit zenping
```

Search this file for *"Launch"* (with the quotes). You should see a section that looks like the following. Change *auto* to *manual*. Then save and quit.

```
"Launch": "auto",
```

This won't stop *zenping* if it was already running, but it will prevent it from starting up next time you start *Zenoss.core* or *Zenoss.resmgr*.

Good candidates for setting to manual launch are:

- zencommand
- zenjmx
- zenmail (defaults to manual)
- zenmodeler
- zenperfsnmp
- zenping
- zenpop3 (defaults to manual)
- zenprocess
- zenpython
- zenstatus
- zensyslog
- zentrap

The Enterprise Resource Manager product has extra daemons. The following may usefully be set to manual mode:

- zenjserver
- zenpropertymonitor
- zenucsevents
- zenvsphere

The actual services on the system will depend on what ZenPacks are installed. The rule of thumb should be that any services under the *Collection* tree can be set to manual except for *zenhub*, *MetricShipper*, *collectorredis*, and *zminion*.

3.1.2.4 Useful references for managing a Zenoss 5 environment

Zenoss 5 is a very different environment from previous versions given its use of docker containers. This provides some new challenges, especially for those who are already familiar with Zenoss 4.

Some useful Knowledge Base articles are emerging which provide assistance:

- "Virtualization and Docker Containerization for Poets" - https://support.zenoss.com/hc/en-us/articles/202254069-Virtualization-and-Docker-Containerization-for-Poets
- "How to troubleshoot Resource Manager 5.x services that fail to start" - https://support.zenoss.com/hc/en-us/articles/207348996-How-to-troubleshoot-Resource-Manager-5-x-services-that-fail-to-start

- "How to Recover Control Center from Hardware Failure" - https://support.zenoss.com/hc/en-us/articles/204643769-How-to-Recover-Control-Center-from-Hardware-Failure
- "How to add tools or scripts into a Resource Manager 5.x Docker Container" - https://support.zenoss.com/hc/en-us/articles/207610516-How-to-add-tools-or-scripts-into-a-Resource-Manager-5-x-Docker-Container
- "Introduction to Zenoss Control Center" - https://support.zenoss.com/hc/en-us/articles/206278353-Introduction-to-Zenoss-Control-Center

3.2 ZenPack creation

The method for creating a ZenPack varies depending on the Zenoss version. Although Zenoss 4 and earlier provides a GUI menu but no command-line method and Zenoss 5 offers a command-line interface but no GUI, the *zenpacklib* tool provides a common method for both.

3.2.1 What's in a name?

When creating a new ZenPack, the first thing to decide is the ZenPack name. ZenPack names are a sequence of, typically, three package names separated by periods. The first part of the name is always **ZenPacks**. The second part usually identifies the person or organization responsible for the ZenPack. The last part of the name usually identifies the function of the ZenPack.

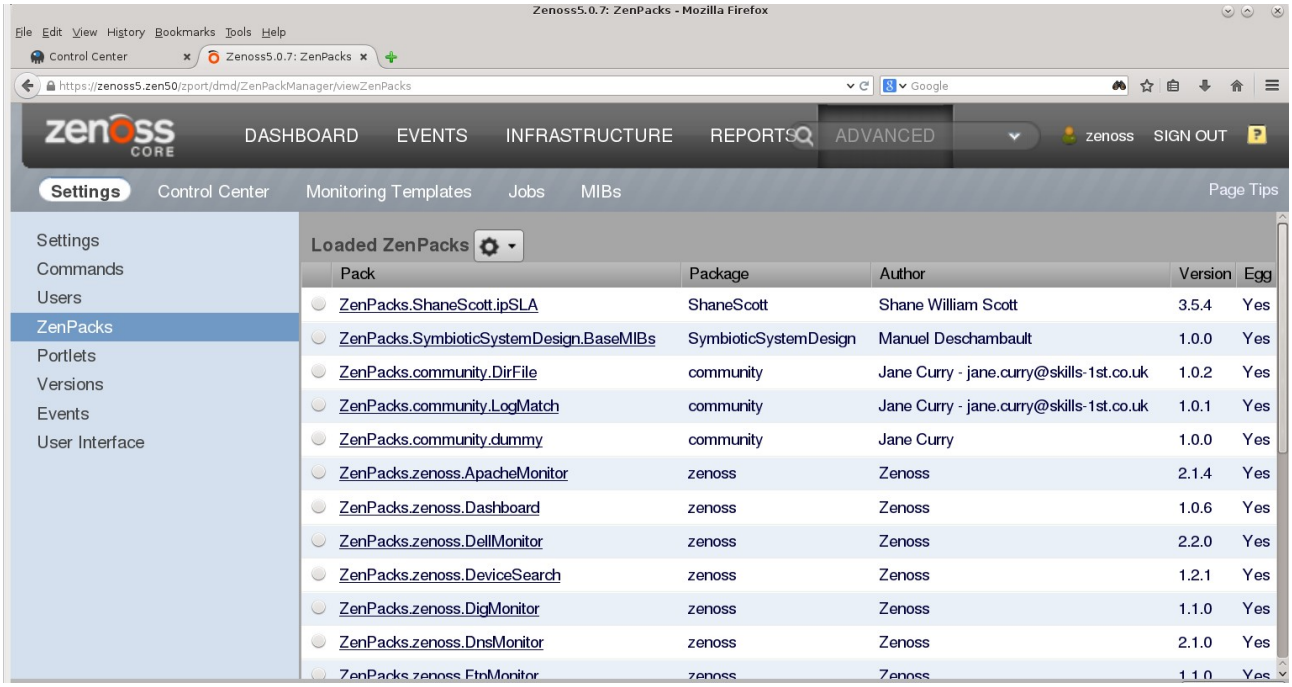


Figure 5: ZenPack page from Zenoss 5 - note the structure of ZenPack names

It is possible to have names with more than three segments, for example: *ZenPacks.zenoss.Microsoft.Windows*

ZenPack names **must** be unique.

ZenPack names must **not** overlap. For example, if *ZenPacks.zenoss.Microsoft.Windows* exists, then *ZenPacks.zenoss.Microsoft* is illegal.

3.2.2 ZenPack directory hierarchy

When creating a new ZenPack, the only initial requirement is the name. After creation, you can then specify other parameters, such as the Zenoss version dependency or other co-requisite ZenPacks. You should also specify an author, a version and a license for the ZenPack. It is good practice to supply contact details in the author field; for example *Joe User - joe.user@amazingzenosstech.com*. These details are configured using the GUI.

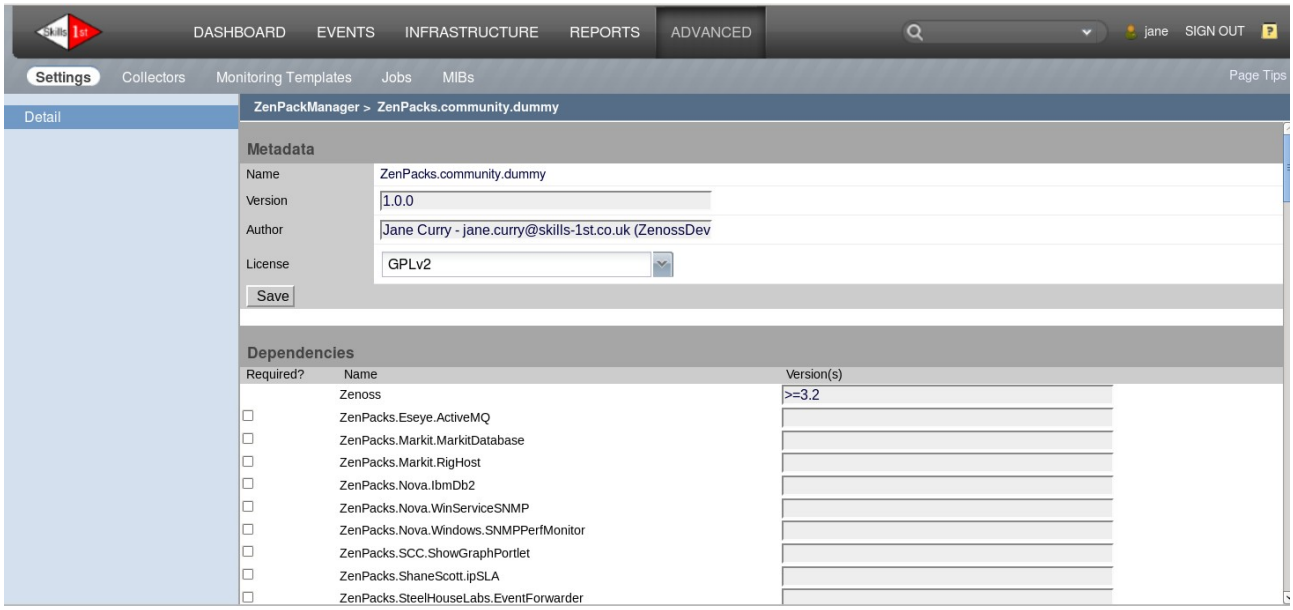


Figure 6: Creation details for a ZenPack

When a ZenPack is created, a directory hierarchy is automatically created under *$ZENHOME/ZenPacks*. For the example *ZenPacks.community.dummy* under *$ZENHOME/ZenPacks*, the hierarchy is:

- *ZenPacks.community.dummy*
- *ZenPacks.community.dummy/ZenPacks*
- *ZenPacks.community.dummy/ZenPacks/community*
- *ZenPacks.community.dummy/ZenPacks/community/dummy*

In this document, *ZenPacks.community.dummy/ZenPacks/community/dummy* is referred to as the **base directory** for the ZenPack. This is where the files and directories that actually do the work of the ZenPack, are placed.

There will also be a *ZenPacks.community.dummy.egg-link* file under *$ZENHOME/ZenPacks*; this file simply contains the name of the **top-level** directory where the code actually resides; in this case, */opt/zenoss/ZenPacks/ZenPacks.community.dummy*.

In Zenoss 5, the egg-link file is in */var/zenoss/ZenPacks* in the Zope container.

This top-level directory has a *setup.py* file containing the name, author, prerequisite and license information entered on the GUI.

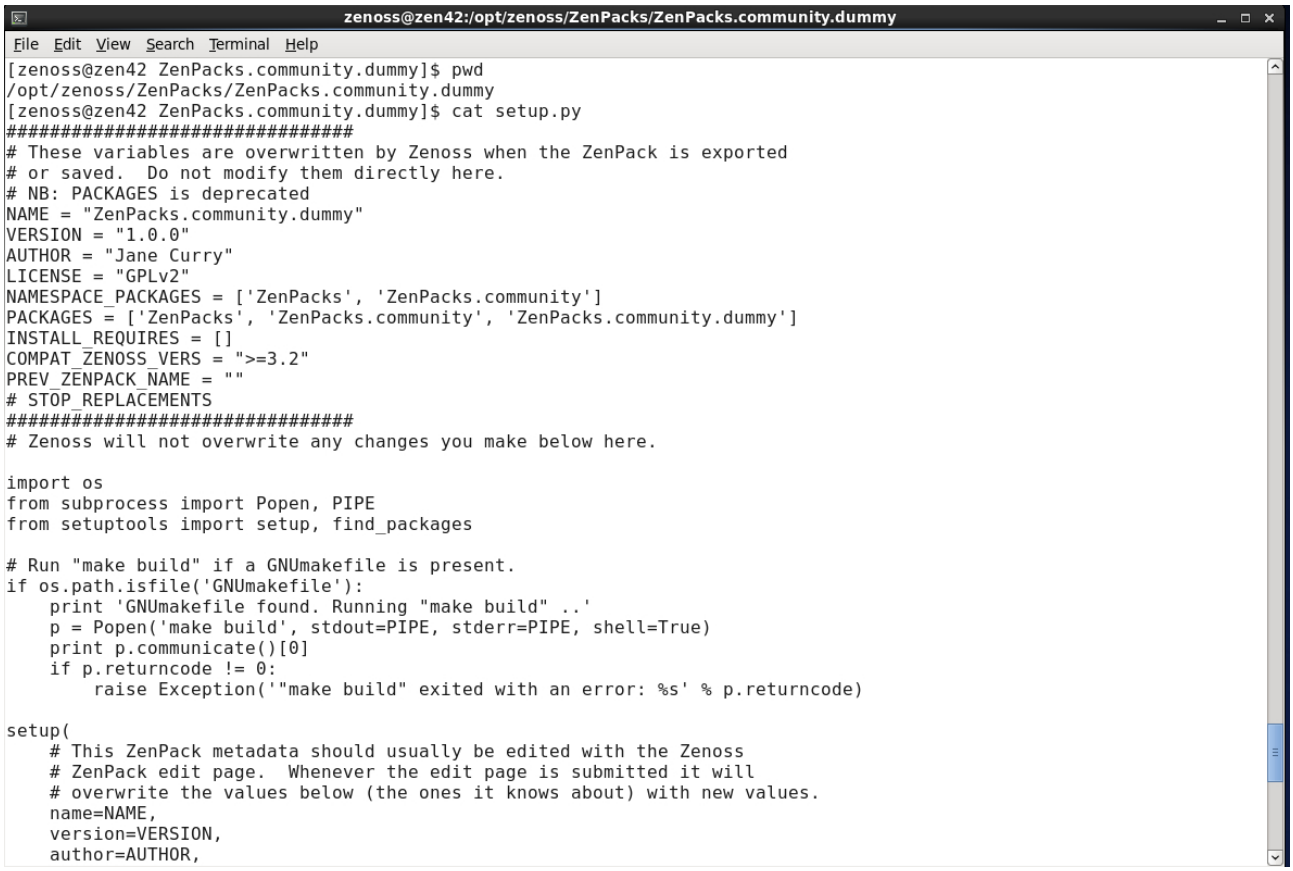


Figure 7: setup.py in the top-level directory of a ZenPack

Each of the directories, other than the top-level, has a sparsely populated file called ***__init__.py*** that is required. It does not require modification for simple ZenPacks.

NOTE: It is **essential** that all directories and subdirectories below the top-level that are part of the path to a python source file, have an *__init__.py* file, even if it is a zero-length file.

Provided the ZenPack is not created using the zenpacklib command, the base directory has several example files plus a directory hierarchy for creating the various elements of a ZenPack.

Points to consider:

- Some of these files and directories may be redundant and should ideally be removed if not required.
- Other directories may be desirable, for example *facades, routers, parsers.*
- Don't change or invent directory names.
- Most default files contain helpful comments that describe their function.
- At creation time, these sample files are generally harmless and items can be added to the ZenPack using the GUI; however it is good practice to remove any unwanted sample files.

- ZenPacks can be built by a combination of adding objects using the GUI and writing code.

Consult the guide at https://zenosslabs.readthedocs.org/en/latest/zenpack_standards_guide.html for a good overview of standard ZenPack files, directories and best practices.

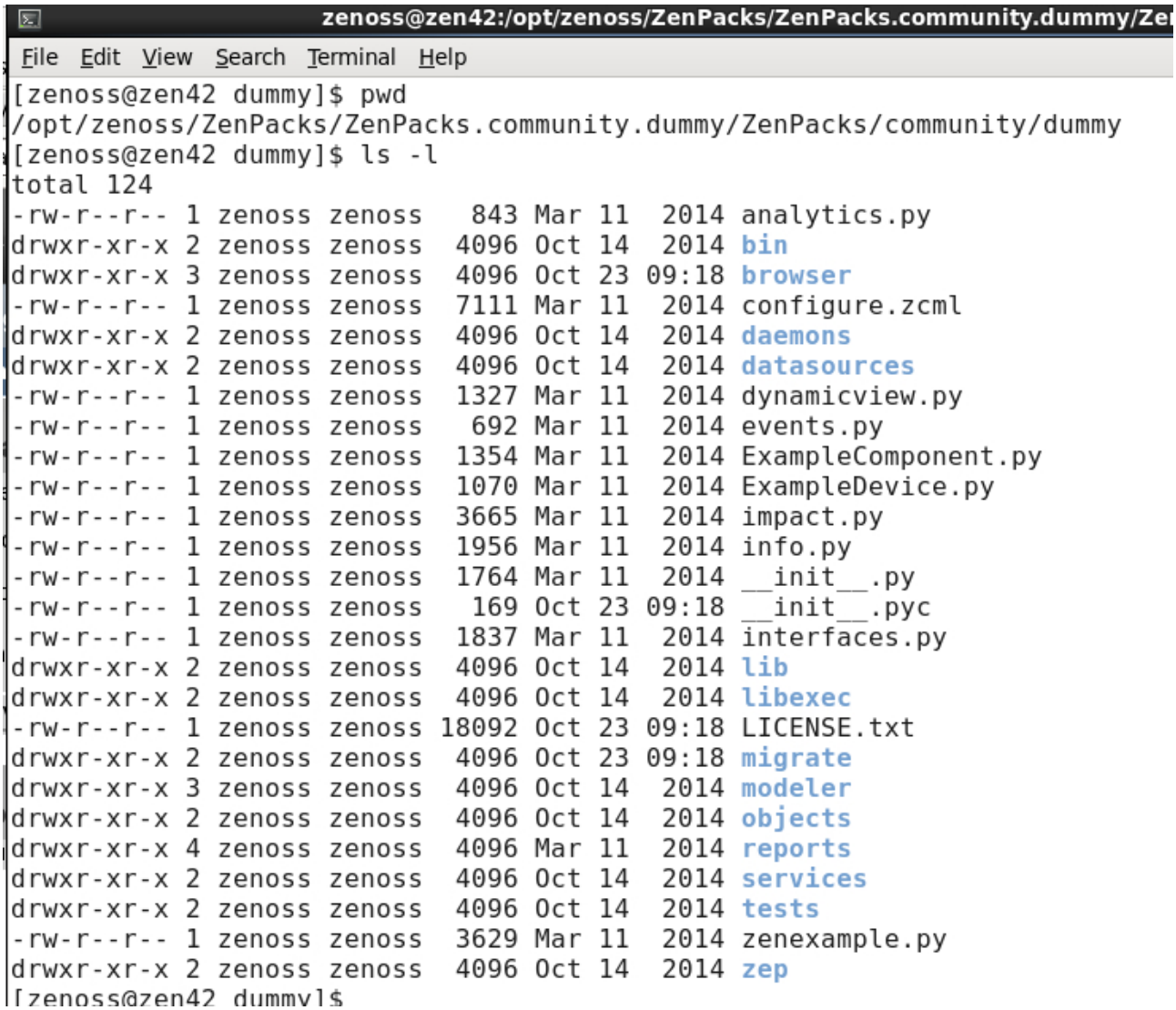


Figure 8: Default files and subdirectories in ZenPack base directory

In the following list, the color-highlighted items are directories.

Name	***Description***
analytics.py	For integration with Service Dynamics Analytics
bin	Any binaries the ZenPack creates
browser	Code for driving the GUI

Name	*Description*
configure.zcml	“Glue” code to link info & interface information to GUI code
daemons	If the ZenPack creates a new daemon, code goes here
datasources	For new datasource code
dynamicview.py	For integration with Service Dynamics dynamic views
events.py	For integration wit Service Dynamics Impact
ExampleComponent.py	Sample code for a new component
ExampleDevice.py	Sample code for a new device object class
info.py	Defines mapping between object attributes and interface classes for display in the GUI
__init__.py	Mandatory. Can modify almost anything - or nothing!
interfaces.py	Defines what attributes should be displayed in GUI & how
lib	For new libraries required by the ZenPack
libexec	For scripts delivered and used by the ZenPack
LICENSE.txt	The text of the selected license
migrate	Code to help migration between Zenoss versions
modeler	Directory hierarchy for modeler plugins
objects	Contains objects.xml - objects added to ZenPack from the GUI and possibly other objects
reports	For reports created by the ZenPack
services	Provides configuration services for custom daemons
tests	test scripts
zenexample.py	Sample daemon code
zep	To provide custom triggers, notifications &event fields

Zenoss 5 also has a ***service_definition*** directory which holds files to register new ZenPack services with Control Center.

When the ZenPack is created through the GUI or command line, it is in **development mode,** also known as **link-installed mode**. This means that objects can be added to it from the GUI. In contrast, installing a ZenPack from a pre-packaged egg file will result in a read-only ZenPack. No changes through the GUI are possible, although it **is** possible to add / modify files and directories in the code.

3.2.3 ZenPack creation for Zenoss 4 and earlier

Versions of Zenoss prior to 5 do not provide a *zenpack* command option to create a new ZenPack; a new ZenPack is generally created through the GUI.

As a Zenoss user with at least the *ZenManager* role, use the top-level *ADVANCED -> Settings* option and select *ZenPacks* from the left-hand menu.

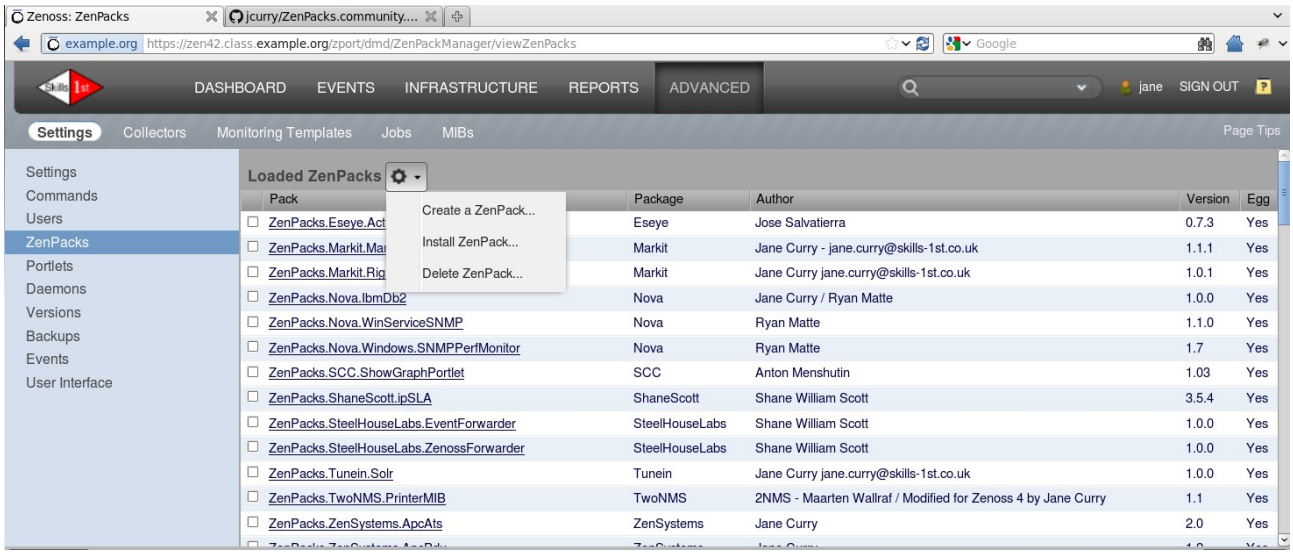


Figure 9: ZenPacks option from the ADVANCED -> Settings menu

The *Action* icon (the "gear" icon at the top of the main panel) then offers the following options:

- Create a ZenPack
- Install ZenPack
- Delete ZenPack

3.2.4 Zenoss 5 ZenPack creation

Zenoss 5 has removed the ability to create a ZenPack through the GUI. The command line is mandatory. The rest of this document assumes that the Zenoss 5 development environment documented in section 3.1.2, has been implemented.

TODO: More information required on filesystems (base host, container, DFS) - probably in architecture section. **<<StevePC:This development environment also assumes that NFS is working to sync the DFS with all v5 pool hosts, including remote collectors, which is not often the case >>**

In Zenoss 5.0.x, logon to the base host as the *zenoss* user and create the *ZenPacks.community.dummy* ZenPack with:

```
zenpack --create ZenPacks.community.dummy
```

The directory hierarchy is created, including the extra ***service_definition*** directory which holds files to register new ZenPack services with Control Center. No containers or daemons need to be started.

The ZenPack directory hierarchy can be inspected by accessing the *zope* container:

```
zope
cd /opt/zenoss/ZenPacks
ls -l
```

In Zenoss 5.1, the ***zenpack*** command is partially replaced by ***zenpack-manager***.

TODO: More info required on zenpack-manager - see Chet foils from GalaxZ.

3.2.5 ZenPack creation using zenpacklib

An alternative method for creating ZenPacks, common to Zenoss 4 and 5, is to use the zenpacklib command.

```
./zenpacklib.py create ZenPacks.community.dummy
```

See the Zenoss documentation at http://zenpacklib.zenoss.com/en/latest/getting-started-5.html and sections 8.3 and 8.4 for more information.

This method results in a minimal (basic) directory hierarchy, only created down to the base directory, */opt/zenoss/ZenPacks/ZenPacks.community.dummy/ZenPacks/community/dummy*. No subdirectories or sample files are created. There are good reasons to start with a minimal structure:

- Provides a reference structure for the new developer
- Makes it unnecessary to manually remove example code created by the UI
- It is easier to add in new functionality as required

3.3 Exporting ZenPacks

When a ZenPack is ready to be tested on a different system or to be packaged for inclusion on the Zenoss Wiki site, it needs to be converted into a Python egg file.

3.3.1 Exporting xml data

The GUI export process creates the *objects/objects.xml* file from the objects associated with the ZenPack in the GUI.

NOTE: This step is required if you've made changes to the associated objects in the GUI and need to update the *objects.xml* file before creating a new egg file or before creating a tar bundle of your ZenPack as backup or to ship elsewhere.

In Zenoss 5, any templates in the ZenPack have their own xml file under *objects/templates*, where the filename is:

<device class path>_rrdTemplates_<template name>.xml

eg. *Device_Server_Linux_UserGroup_rrdTemplates_test.xml*

Other ZenPack elements go in *objects/objects.xml*.

The export process also creates a .egg file, and, if you do not want to use the command-line, all that is required is to export your ZenPack for installation on another system.

From the detailed page of the ZenPack, use the *Action* icon at the bottom of the left-hand menu to *Export ZenPack*.

The options presented are:

- Export to $ZENHOME/exports
- Export to $ZENHOME/exports and download

Typically you leave the top radio button selected to just create the ZenPack egg file in *$ZENHOME/exports*. The file is first created in the ZenPack's dist directory then copied to the $ZENHOME/exports directory.

The exported .egg file can now be moved to a different Zenoss server and installed like any other ZenPack.

3.3.2 Creating the .egg from the command line

If you do not need to update the *objects.xml* data in your ZenPack and only want to create a .egg file from the command-line:

1) *cd* to the top-level of your ZenPack (the directory containing *setup.py*)
2) Issue the following command:
```
python setup.py bdist_egg
```

This creates the .egg file in the *dist* subdirectory, from where you can copy it to another Zenoss system for installation.

3.4 Installing ZenPacks

Always ensure that ZenPack work is done as the *zenoss* user.

It is good practice to install egg versions of ZenPacks on production systems so that they are not inadvertently changed through the GUI.

It is also good practice to install a ZenPack using the command line as it is much easier to see if there are issues.

When developing ZenPacks and making frequent changes it can be advantageous to understand the minimum requirements for recycling daemons in order to speed the development process. The safe recommendation is **always to restart/recycle all Zenoss daemons after ZenPack installation**. Experimentation and experience can guide a more limited recycle.

- The safe (but slow) option is to recycle all daemons.
- With Zenoss Service Dynamics 4.x you may have to recycle all daemons. *zenhub* and *zenwebserver* are an absolute minimum.
- With Zenoss 5 (Core and SD) you have to restart the *Zenoss.core* or *Zenoss.resmgr* application and all its child services.
- As a minimum on Zenoss Core prior to version 5, you need to recycle *zenhub* and *zopectl*. Adding or changing any object (device, component) will necessitate a full recycle. **Adding** a new element (datasource, parser, report, modeler) will necessitate a full recycle. **Changing** an existing datasource, modeler or report you will probably get away with just restarting *zenhub* and *zopectl*. If you have changed a datasource then you will also need to recycle the daemon that runs the datasource eg. *zenpython*.
- The documentation provided with the ZenPack should state what needs recycling.

During the development phase, the sample ZenPack created earlier should immediately be moved out of the *$ZENHOME/ZenPacks* directory. It is good practice for an enterprise to document a known directory for ZenPack development. The directory must have permissions that provide full access for the *zenoss* user. If the code is to be shared outside the organization, ZenPack directories may well be developed under **git**. ***/code/ZenPacks/DevGuide*** will be used throughout this paper as the top-level directory for ZenPack development under Zenoss 4. ***/z/zenpacks*** will be the equivalent on Zenoss 5.

```
cp -r $ZENHOME/ZenPacks/ZenPacks.community.dummy /code/ZenPacks/DevGuide
```

The *--link* parameter to the *zenpack* command should be used to reinstall the ZenPack from the development directory.

```
zenpack --link --install /code/ZenPacks/DevGuide/ZenPacks.community.dummy
```

The result of the *--link* parameter actually removes the *ZenPacks.community.dummy* directory hierarchy from *$ZENHOME/ZenPacks* and the *ZenPacks.community.dummy.egg-link* is modified to point to the new top-level directory */code/ZenPacks/DevGuide/ZenPacks.community.dummy*. Now, if anyone deletes this ZenPack from he *Delete ZenPack* menu, the only thing that is deleted from *$ZENHOME/ZenPacks* is the link file, not all the ZenPack code.

From this point, development of the ZenPack can continue, adding items using the GUI and by writing code in appropriate directories; all changes will follow this link to actually update code in the private directory.

It is perfectly acceptable to reinstall a ZenPack that already exists – it will simply give a warning message that the ZenPack is already installed, but it will do the install. Remember to restart at least *zenhub* and *zopectl*.

If the ZenPack does already exist, in practice the ZenPack's **remove** method is executed with l*eaveObjects=True*, followed by the **install** method. This means that if the ZenPack introduced a new device object class, *dummyDevice*, and several device **instances** had already been discovered with this class, for which data may have already been gathered for weeks, these devices would **not** be deleted by the reinstall. Conversely, if an explicit *zenpack --remove* was executed followed by a *zenpack --install* then such devices, including their events, configurations and performance data, **would be lost**.

When a ZenPack is exported, it automatically creates an egg file whose name includes the Python version, where **2.4** represents Zenoss 2.x, **2.6** represents Zenoss 3.x (for example *ZenPacks.skills1st.bridge-1.0.4-py2.6.egg)* and **2.7** represents Zenoss 4 and 5 (for example *ZenPacks.community.dummy-1.0.0-py2.7.egg*). Attempting to install a 2.6 egg file in a Zenoss 2 environment and vice versa, will often fail with a message including "****BLOCKED**** *by –allow-hosts"*.

A ZenPack compiled for an earlier version of Python will need recompiling under the later version and a new egg file created. Simply install the code in development mode with the *--link* parameter and restart all the Zenoss daemons. All .py files will be recompiled to .pyc files to be incorporated in the new egg when the ZenPack is exported.

Note that some developers deliberately remove the Python version nomenclature from their egg file, especially if it is not Python-version-specific. Great care must be taken if files are

renamed as this can cause major issues if done injudiciously, including making the ZODB inconsistent and rendering the ZenPack neither installable nor uninstallable.

3.4.1 Installing ZenPacks on Zenoss 4

Either use the GUI with *ADVANCED* -> *Settings* -> *ZenPacks* menus and then the *Action* icon option to *Install ZenPack*; or you can use the command line:

```
zenpack --install ZenPacks.community.dummy-1.0.0-py2.7.egg
```

If a link-install is performed, the *.egg-link* file resides in *$ZENHOME/ZenPacks* and points to the top-level directory of the ZenPack.

It is good practice to use the command line for installation as the GUI can sometimes hide error messages.

3.4.2 Installing ZenPacks on Zenoss 5

Before installing any ZenPack it is prudent to take a Control Center snapshot as a backup.

The *.bashrc* file for the *zenoss* user sets up a common directory, */z*, that is shared between the base host and the various containers and creates command aliases that set the current directory to */z*. As a local standard, this document maintains ZenPack code in */z/zenpacks*.

Note that */z/zenpacks* must be local (not mounted) and must be readable, writeable, and executable by all users. *.bashrc* defines that both *zenpack* and *zenpacklib* commands will be executed in the context of the current directory being */z/zenpacks*.

To install a ZenPack, either an egg or a development directory hierarchy, use the *zenpack* command as the *zenoss* user on the base host:

```
zenpack --install ZenPacks.zenoss.PythonCollector-1.7.3-py2.7.egg
zenpack --link --install ZenPacks.community.dummy
```

The Zenoss service must be restarted. “Core” will be used throughout to indicate commands for those using the free version of Zenoss; “Enterprise” will be the annotation for those using the chargeable Service Dynamics product:

```
serviced service restart Zenoss.core          (Core users)
serviced service restart Zenoss.resmgr        (Enterprise customers)
```

The daemons a ZenPack provides (if any) are packaged in Docker containers, and installed as child services of the current instance of Zenoss Core.

If a link-install is performed, the *.egg-link* file resides in /var/zenoss/ZenPacks and points to the top-level directory of the ZenPack.

3.5 Removing ZenPacks

In Zenoss 4, ZenPacks can be removed either from the GUI or using the CLI. It is good practice to use the CLI as it is easier to see if there are any issues.

```
zenpack --remove ZenPack.community.dummy
```

Zenoss should be completely restarted after ZenPack removal.

Zenoss 5 only provides the command-line option.

Remember that removing a ZenPack explicitly removes all objects in the ZenPack. If this includes device classes then all **instances** of such devices will also be lost.

Another serious consequence may be if a ZenPack includes SNMP MIBs and event transforms have been written to check the decoded OID parts of an event, then removing the ZenPack (and hence the MIB) will result in the failure of those transforms.

4.0 Simple ZenPacks

Some ZenPacks can simply be created using the Zenoss GUI; this is especially useful for moving standard configurations from one Zenoss server to another but may also be appropriate when creating ZenPacks to share with other people.

The ZenPack is created exactly as described in chapter 3.2 above. To add “things” to the ZenPack, simply use the *Add to ZenPack* option that is available on many of the dropdown menus:

- Device Classes
- MIBs
- Event Classes
- Event Mappings
- User Commands
- Event Commands
- Service Classes
- Process Classes
- Performance Templates

You will be prompted as to which ZenPack you wish to add the item to; only ZenPacks in development mode will be in the dropdown selections. Devices themselves are the conspicuous omission from this list. Any individual device is usually specific to a particular site and therefore not likely to be useful to other Zenoss users. To port device instances, use Zenoss backups or the zenbatchdump / zenbatchload utilities.

To see what a ZenPack contains, simply use the ZenPacks option from the ADVANCED -> Settings menu and choose the appropriate ZenPack.

Objects can be removed from the ZenPack by selecting the checkboxes next to them under the ZenPack Provides heading, and using the Delete from ZenPack menu item.

When a ZenPack is exported (using the Action icon from the Detail page of the ZenPack), not only is the egg file created but it is at this time that all the objects under the ZenPack Provides list, are written to the objects.xml file under the objects directory of the ZenPack. This file can be inspected with an editor – as the name suggests, it is in xml format.

4.1 Adding performance templates to a simple ZenPack

The most common use for a simple ZenPack is to move locally-created performance templates from one Zenoss installation to another, or to provide such templates to the wider Zenoss community.

Start by creating a new ZenPack from the GUI - *ZenPacks.community.simple1*. It will automatically be created in development mode. For a simple ZenPack, there is no reason to copy the ZenPack to a local directory and link-install it.

4.1.1 Adding SNMP performance templates to a ZenPack

This example uses an SNMP performance template, *SnmpPacketsInOut*, that polls the standard SNMP MIB-2 MIB for snmpInPkts and snmpOutPkts. The template has been created so that it can be applied to all devices under the top-level /.

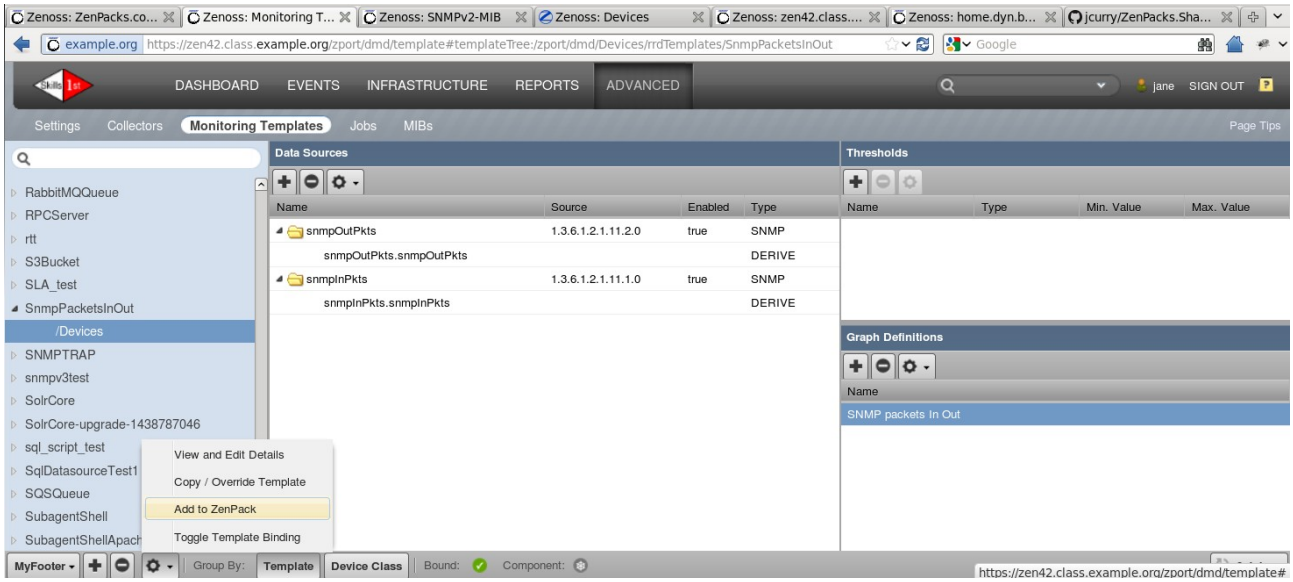


Figure 10: SnmpPacketsInOut template applicable to all devices

To add a template to a ZenPack, simply use the *Add to ZenPack* menu option. If the ZenPack's details page is now inspected, it will show the template; however if the directory hierarchy of the ZenPack is inspected, the *objects/objects.xml* file will be insignificant. The ZenPack must be exported to write all objects to *objects.xml*.

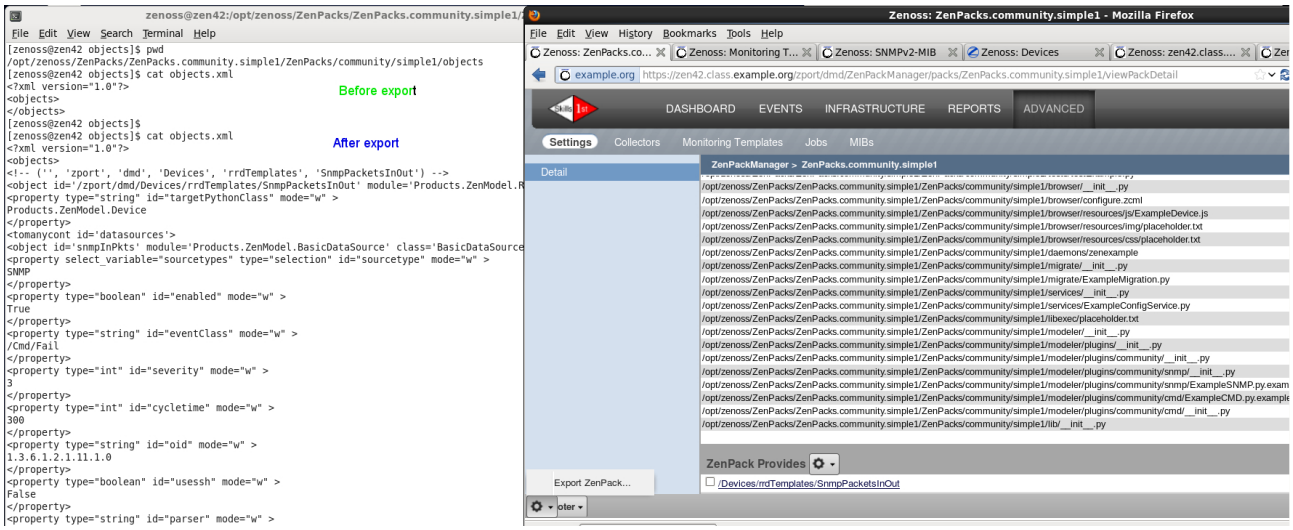


Figure 11: Export the ZenPack to update the ZenPack's objects.xml file

Note in Figure 11 the CLI window showing *objects.xml* before and after the export. Although the GUI window for the ZenPack is only showing the template under *ZenPack Provides*, *objects.xml* has all the definitions for the template, datasource, datapoints and graph.

With Zenoss 5, the templates will go under the *objects/templates* directory with one xml file per template.

The export will also create an egg file in *$ZENHOME/export*.

It is bad practice to add standard objects to a ZenPack that may ever be removed because the ZenPack removal, by default, removes all the objects in that ZenPack. For example, if the standard *ethernetCsmacd_64* template was added to a ZenPack that was then removed, all SNMP interface traffic data collection and graphs would be **lost**!

4.1.2 Adding zencommand performance templates to a ZenPack

A quick and easy way to create performance graphs for local monitoring is to use a performance template with a *COMMAND* data source which will be run by the **zencommand** daemon. Fundamentally a command data source runs a script under a bash shell. The script doesn't have to be a shellscript; using a technique like *!/usr/bin/env python* as the first line of a program, any executable program can be run - but it is still within a bash shell with all the quoting and escaping complexities involved with passing parameters. A COMMAND data source is also rather inefficient and does not scale well.

COMMAND data sources either run a command on the Zenoss server (strictly on the Zenoss collector if there is a distributed architecture), or, if the *Use SSH* box is ticked, then the command is run on the target device which requires *zCommand* properties to be configured for remote user and password.

If *Use SSH* is **not** ticked, and the template is to be packaged as part of a ZenPack, the executable script can also be included in the ZenPack.

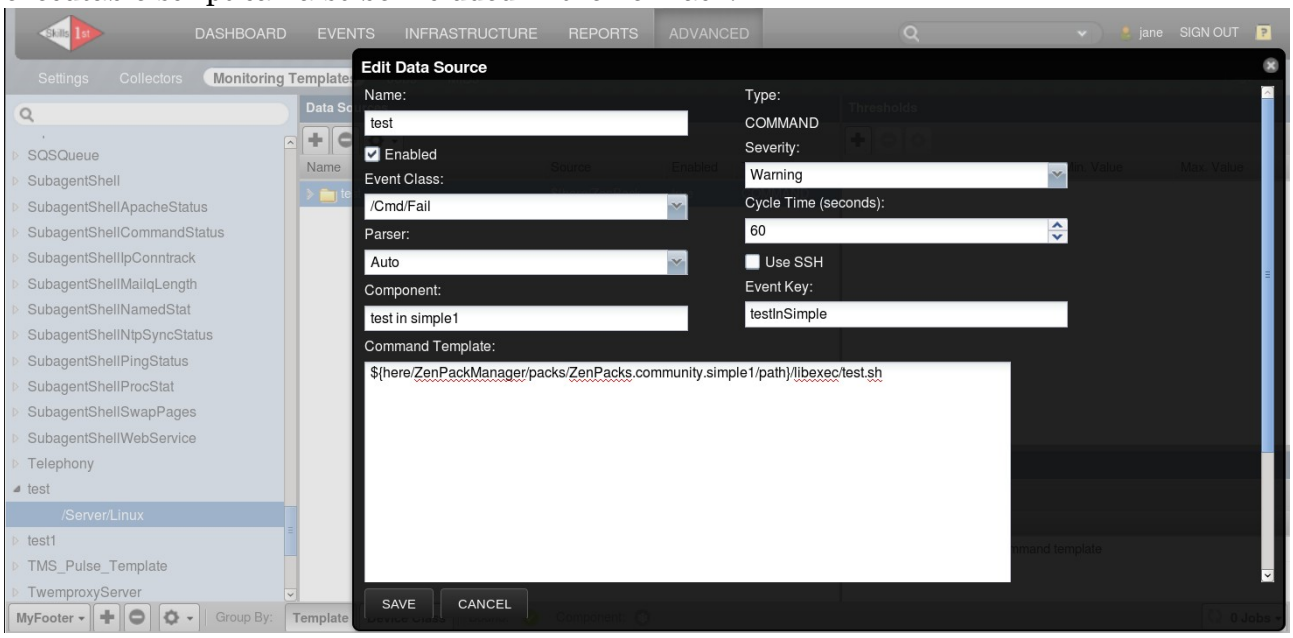


Figure 12: test COMMAND data source in test template

Scripts should be added to the *libexec* directory of a ZenPack; ensure that the script has execute permission.

The script, *test.sh*, can then be referenced in the data source as:

${here/ZenPackManager/packs/ZenPacks.community.simple1/path}/libexec/test.sh

The template is added to the ZenPack using the *Add to ZenPack* menu and the ZenPack exported.

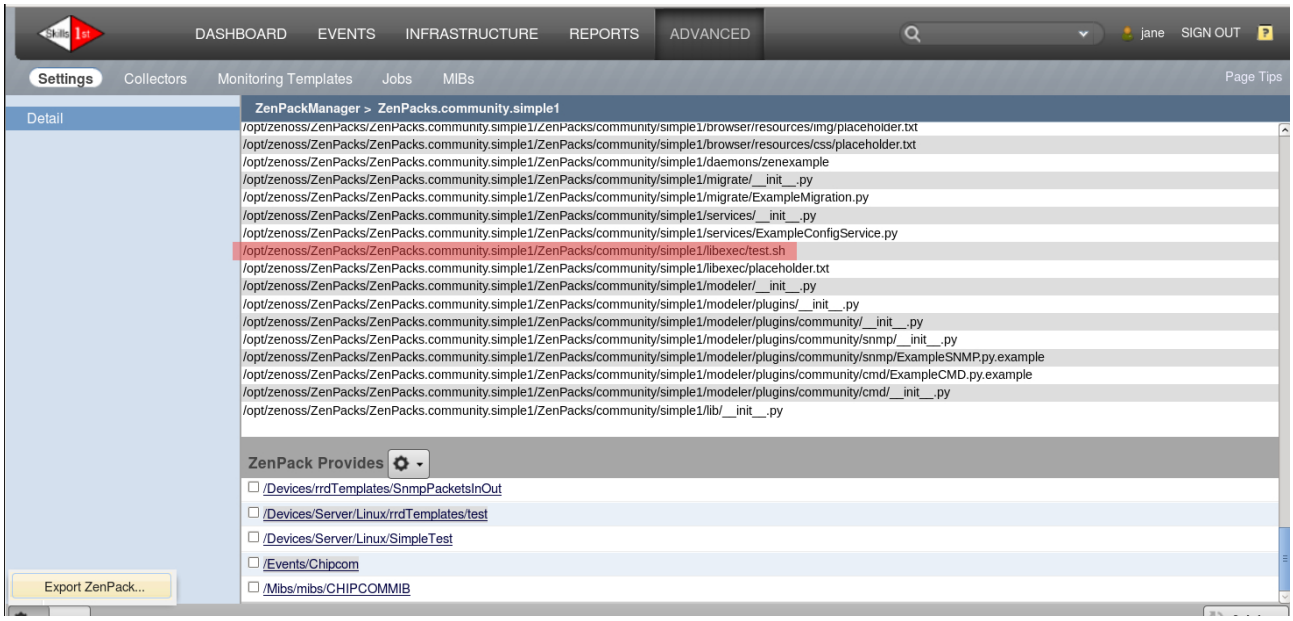


Figure 13: Exporting ZenPacks.community.simple1, including libexec/test.sh and the test template

Note the presence of the *test.sh* script that you copied to the ZenPack's *libexec* directory. If the script needs to be changed then the ZenPack does **not** need reinstalling and **no daemons need to be recycled**.

Without using any Python coding, this is an example of adding script functionality to a ZenPack by leveraging *zencommand* and adding locally-developed scripts to the ZenPack's *libexec* directory.

4.2 Adding SNMP MIBs and event classes to a simple ZenPack

MIBs can be imported into the Zenoss ZODB object database where they are used to help decode SNMP TRAPs (or SNMP V2 Notifications). Importing using the *zenmib* command can be fraught as there may be a sequence of prerequisite and co-requisite MIBs that also need importing. Once the MIB is in the ZODB database, it is largely irrelevant whether the prerequisite/co-requisites are also in the ZODB.

The Chipcom MIB is a simple standalone MIB which includes both MIB OIDs and TRAPs so will be used here as an example. It can be imported into a ZenPack using the *Add to ZenPack* menu.

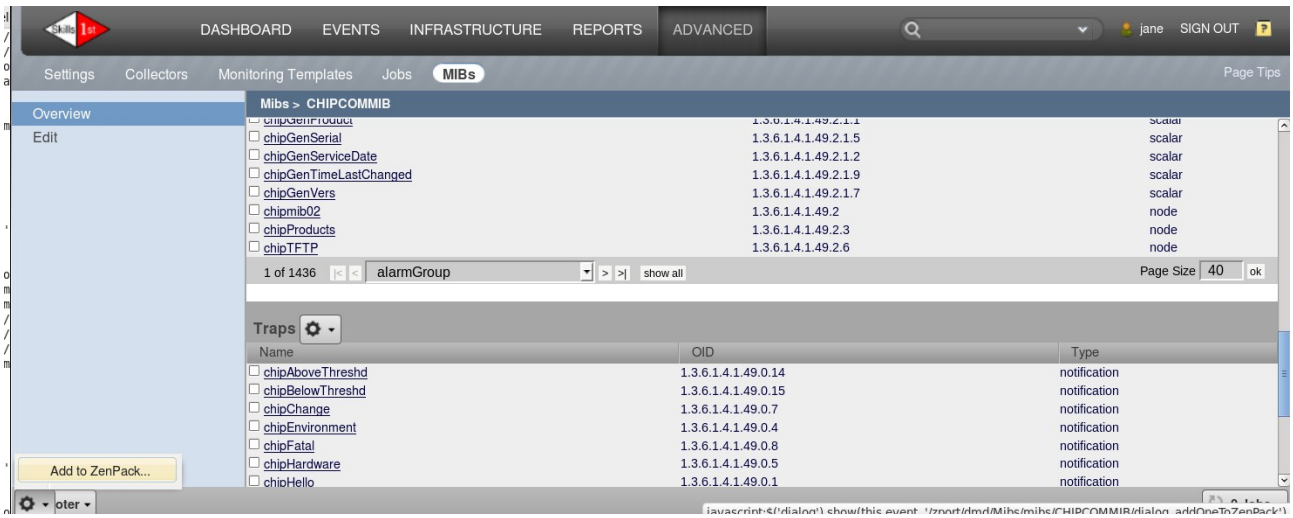


Figure 14: Adding the Chipcom MIB to a ZenPack

Note from Figure 14 that Chipcom OIDs start with 1.3.6.1.4.1.49. If the SNMPV2-SMI MIB has also been imported then the 1.3.6.1.4.1 will be translated in a TRAP to:

```
iso.org.dod.internet.private.enterprises
```

Figure 14 shows that the Chipcom TRAP 0.8 translates to chipFatal. Before the Chipcom MIB was imported, a TRAP 0.8 from a Chipcom device would have “enterprises.49.0.8” as part of the event summary; after the MIB is imported, the 49.0.8 is translated to chipFatal. This is standard Zenoss functionality.

Different SNMP agents sometimes insert or do not insert a 0 between the enterprise part of the OID (1.3.6.1.4.1.49) and the TRAP number (8). The Zenoss code that processes incoming SNMP TRAPs can accommodate either version. This is why the screenshot in Figure 15 has a summary of snmp trap enterprises.49.8 (no zero).

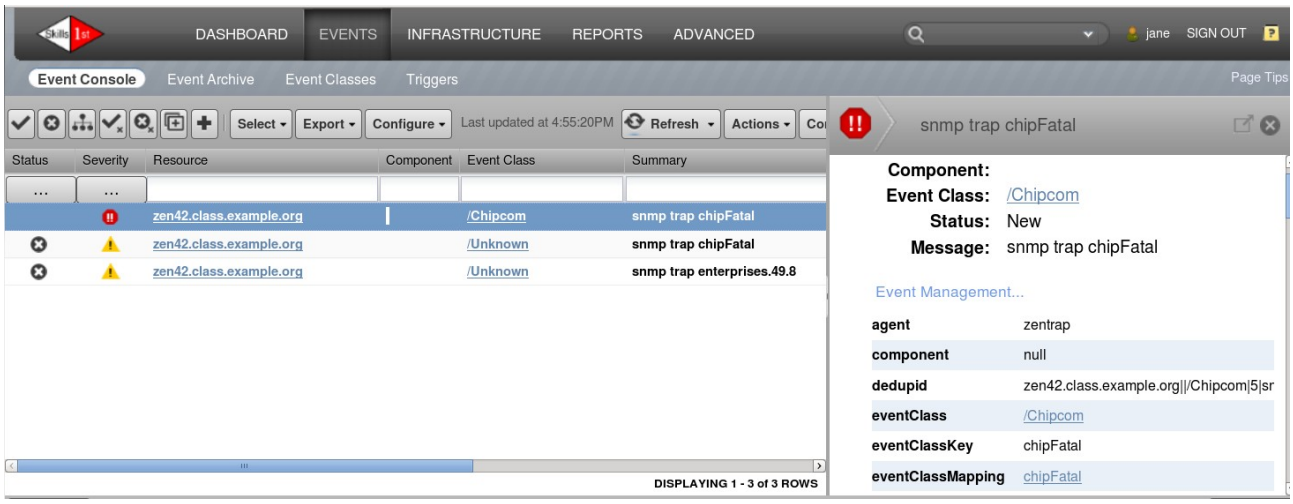


Figure 15: Chipcom events from zentrap before importing the Chipcom MIB (bottom event) and after the MIB import (top two events)

In Figure 15 the bottom event is before the Chipcom MIB is imported; the middle event is after importing the MIB but with no further customisation. Often, a TRAP will need further

customisation through event mapping. A /Chipcom event class may be created with a chipFatal event class mapping; the mapping may have a transform that modifies the event based on various tests.

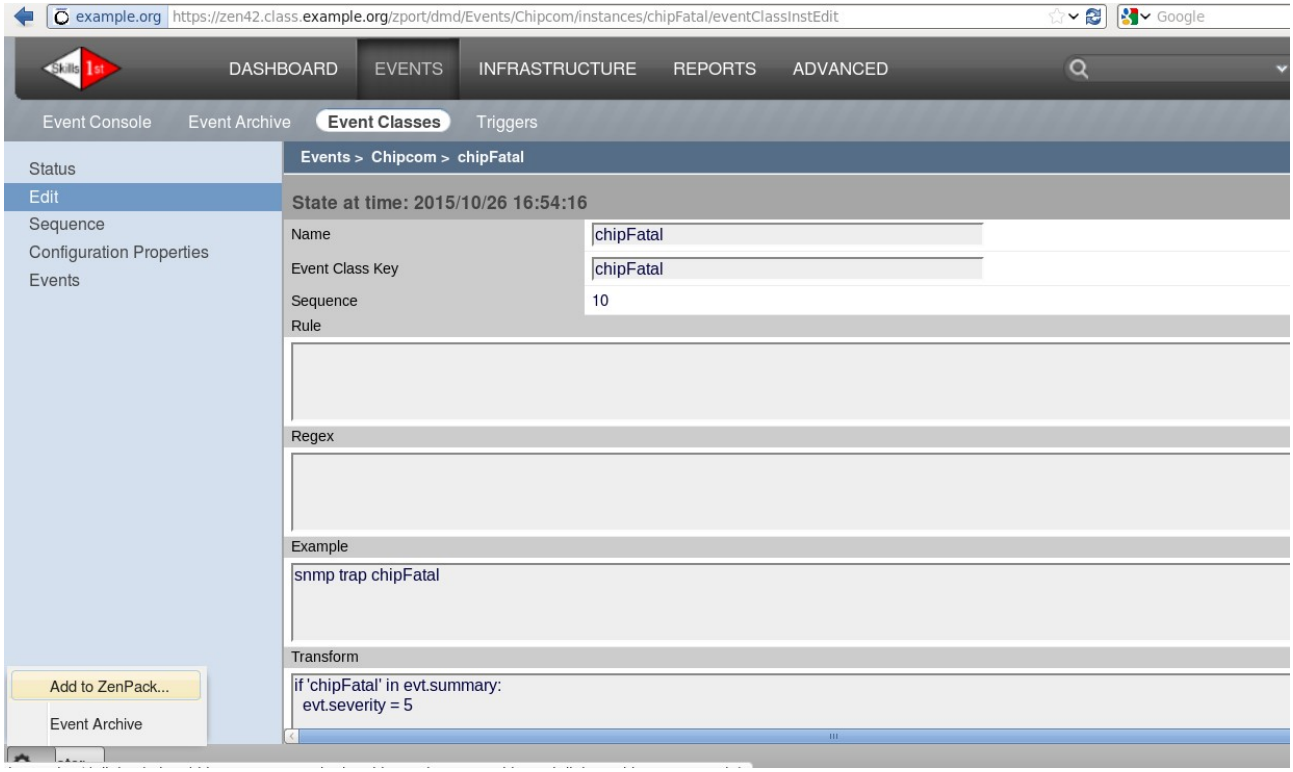


Figure 16: chipFatal event class mapping for event class /Chipcom

Figure 16 shows the mapping, including a transform that tests the event summary field for the presence of “chipFatal” and changes the severity if found.

Of course, the /Chipcom event class, including any associated event class mappings, can also be added to a ZenPack with the Add to ZenPack menu.

Figure 17: objects.xml snippet with the end of a template, the Chipcom event and the start of the Chipcom MIB

Note in objects.xml that an event class mapping (chipFatal) is referred to as an instance.

The subtle point that this whole sub-chapter is trying to make is that, if this ZenPack is removed, the MIB will be removed. Any event classes and event class mappings that are based on the translated TRAP, will now fail to process events as expected. The summary field of the event will revert to containing “enterprises.49.8”, not “chipFatal” so the transform will not be applied. The event mapping will not apply because it is based on the event class key matching “chipFatal” and that will also revert to “enterprises.49.8”.

4.3 Adding device classes to a simple ZenPack

Any device class can be added to a simple ZenPack simply by using the *Add to ZenPack* menu.

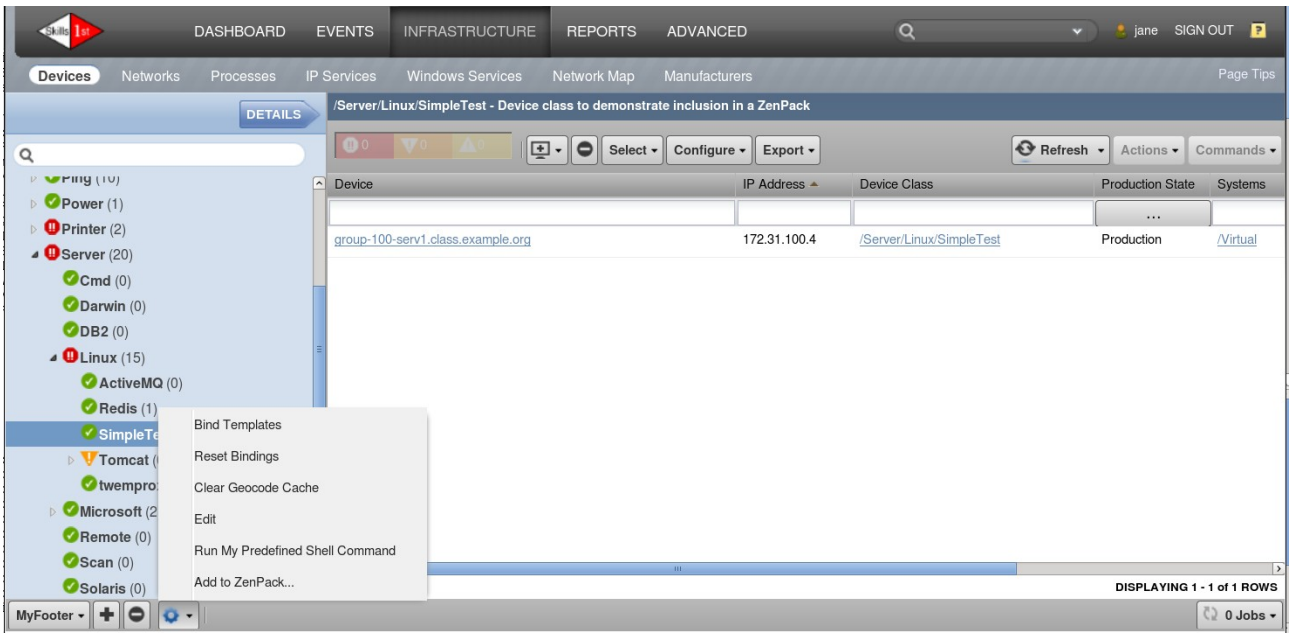


Figure 18: Adding a device class to a ZenPack

Note that, even though the device class contains a device instance, *group-100-serv1.class.example.org*, the device is **not** added to the ZenPack.

It is usually bad practice to add standard device classes to a ZenPack that may be removed at any time in the future as this results in the removal of the DeviceClass, **and any children of the DeviceClass, including device**s.

If the ZenPack is reinstalled, then devices are **not** removed.

4.4 * Adding services and processes to simple ZenPacks

Zenoss provides the ability to monitor IP services, Windows services, and processes.

It is possible to add definitions for each of these to a ZenPack but special care is required. Good practice would be to have separate ZenPacks for relevant IpServices, WinServices and Processes.

4.4.1 Adding IP services to a ZenPack

The usual menu is available for services to *Add Service to ZenPack*. It is also possible to add a Service Organizer if services have been grouped into an Organizer.

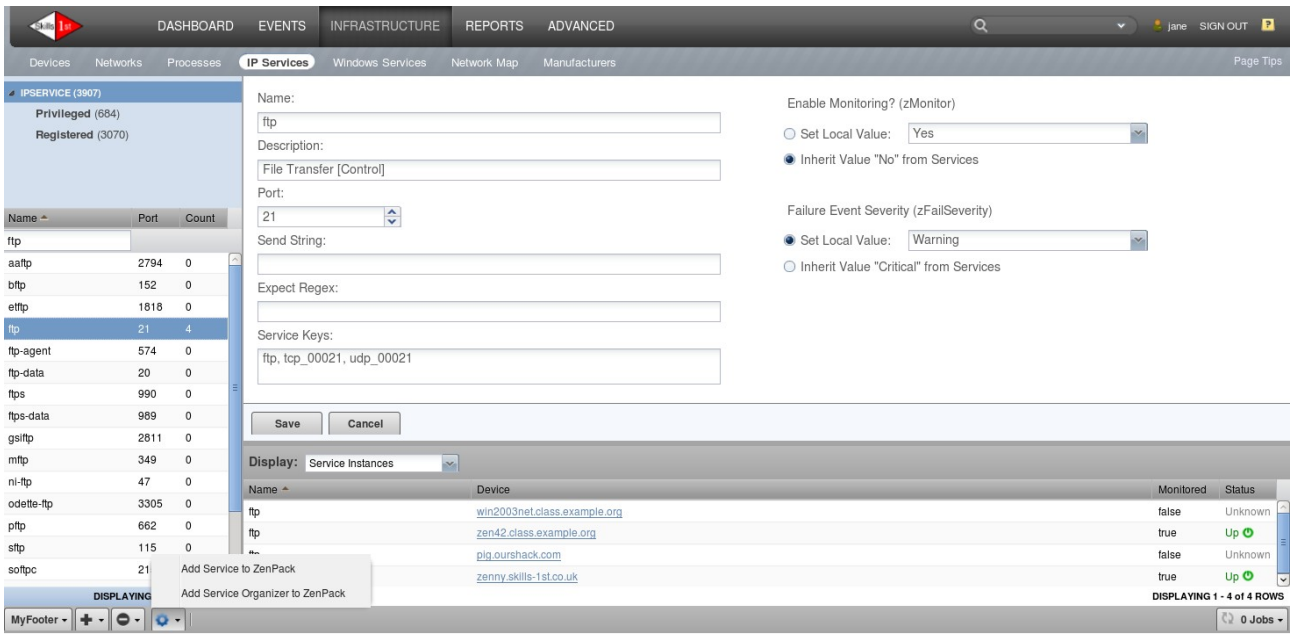


Figure 19: Adding a service to a ZenPack

The ftp service is added to a new ZenPack, *ZenPacks.community.IpService* and the ZenPack is exported. Examining the *objects.xml* shows not only the definition of the service but also **instances**.

```
zenoss@zen42:/opt/zenoss/ZenPacks/ZenPacks.community.IpServices/ZenPacks/community/IpServices/objects
File  Edit  View  Search  Terminal  Help
<?xml version="1.0"?>
<objects>
<!-- ('', 'zport', 'dmd', 'Services', 'IpService', 'Privileged', 'serviceclasses', 'ftp') -->
<object id='/zport/dmd/Services/IpService/Privileged/serviceclasses/ftp' module='Products.ZenModel.IpServiceClass' c
lass='IpServiceClass'>
<property id='zendoc' type='string'>
File Transfer [Control]
</property>
<property type="string" id="name" mode="w" >
ftp
</property>
<property type="lines" id="serviceKeys" mode="w" >
('ftp', 'tcp_00021', 'udp_00021')
</property>
<property type="text" id="description" mode="w" >
File Transfer [Control]
</property>
<property type="int" id="port" mode="w" >
21
</property>
<property visible="True" type="int" id="zFailSeverity" >
3
</property>
<tomany id='instances'>
<link objid='/zport/dmd/Devices/Server/Linux/devices/pig.ourshack.com/os/ipservices/tcp_00021'/>
<link objid='/zport/dmd/Devices/Server/Windows/Snmp/2 Processors/devices/win2003net.class.example.org/os/ipservices/
tcp_00021'/>
<link objid='/zport/dmd/Devices/Server/Linux/devices/zen42.class.example.org/os/ipservices/tcp_00021'/>
<link objid='/zport/dmd/Devices/Server/Linux/devices/zenny.skills-1st.co.uk/os/ipservices/ftp'/>
</tomany>
</object>
</objects>
~
~
~
~
~
~
"objects.xml" [readonly] 30L, 1192C
```

Figure 20: objects.xml for ZenPacks.community.IpServices - note entry for instances

Examine the entries between <*tomany id='instances'*> and </*tomany*> . There is an entry for each device where the service has been seen (compare with the *Service Instances* panel in Figure 19).

This is not helpful. If the ZenPack is installed on a different system, it is very possible that these device instances will not exist.

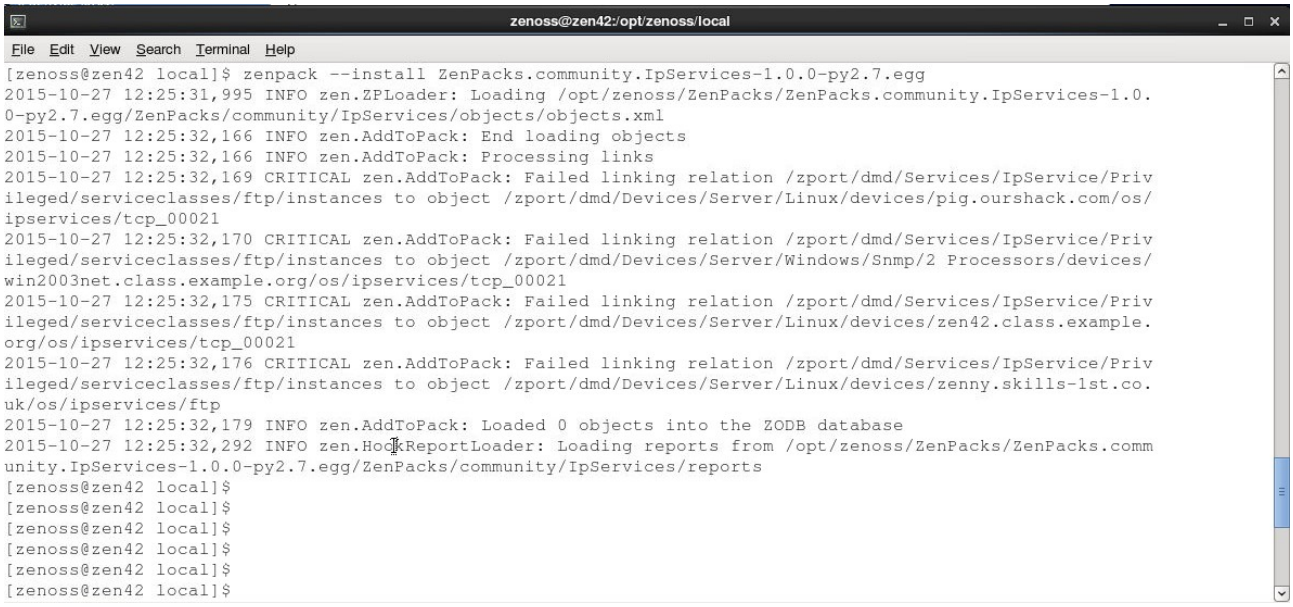


Figure 21: Installing ZenPacks.zenoss.IpServices on a different system

Error messages will be produced on ZenPack installation for each of the service instances. Although the dialogue says “Loaded 0 objects into the ZODB database”, this is not true. Restart Zenoss and check the ftp service to see that it **has** been updated (the *Failure Event Severity* in the source Zenoss was carefully changed from *Error* to *Warning* so that changes in the destination Zenoss could be seen). There are no Service Instances added by the ZenPack. Although the process is unconvincing, it does actually achieve the desired effect.

A cleaner alternative would be to modify the *objects.xml* file to comment out the service instances. This is possible but must be done very carefully. XML comment tags are <*!--* to start and *-->* to finish.

Figure 22: objects.xml with service instances commented out

If the service instances must be commented out then you cannot use a GUI exported egg for installation. If the creation of the egg file happens when the ZenPack is exported, this also writes to *objects.xml* and the egg file contains the **new** *objects.xml*. Either edit the objects.xml file after export and then create the egg using the command-line, or the ZenPack must be moved in source form to the destination Zenoss installation and then link installed (which **does** install cleanly.).

It is important to remember that removal of a ZenPack containing IP Services will remove the service definitions from the Zenoss installation.

The conclusion is that, although the mechanism for adding IP Services to a ZenPack is error -prone, it is not impossible.

4.4.2 Adding Windows Services to a ZenPack

Windows Services work in exactly the same way as IP Services with respect to ZenPacks. There is an *Add Service to ZenPack* option which again adds both the service **definition** and any discovered **instances** of those services, which can be seen in the *objects.xml*.

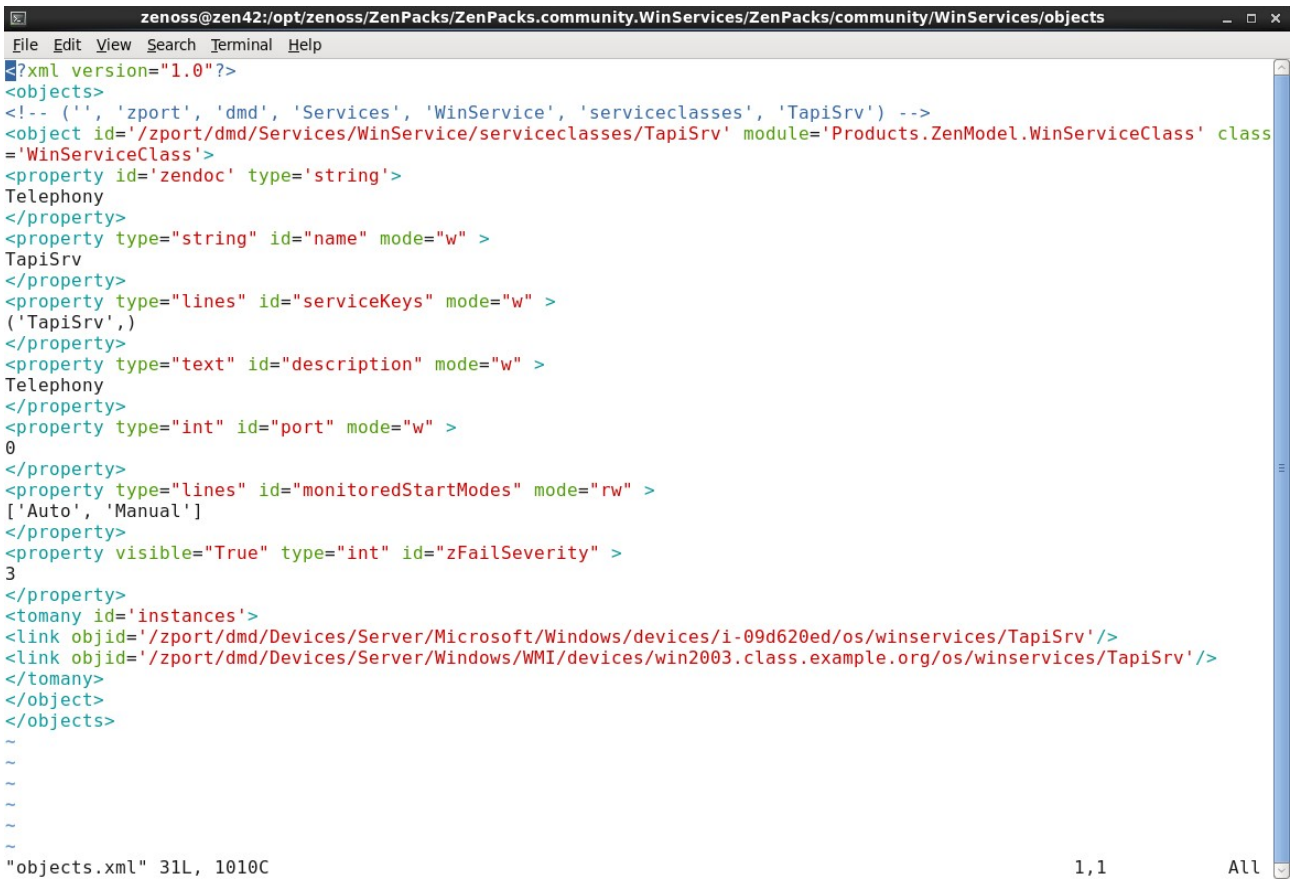


Figure 23: objects.xml for ZenPacks.community.WinServices - note the two instances entries

The same type of error messages are seen when installing the egg version of the ZenPack and, again, the installation is, in fact, successful.

4.4.3 Adding Processes to a ZenPack

In many ways, it is more useful to be able to add processes to a ZenPack than services. IP Services tend to be standard with a huge number of definitions shipped as part of the core product. Windows Services come with a similar huge library and are added to when the modeler finds new services.

Processes, however, are a feature used by many organizations to monitor their specific environment. The only out-of-the-box process definitions are those that manage Zenoss processes.

The implementation with regard to ZenPacks is exactly the same as for services. Process or Process Organizers can be added to a ZenPack and, unfortunately, the process **instances** are also added.

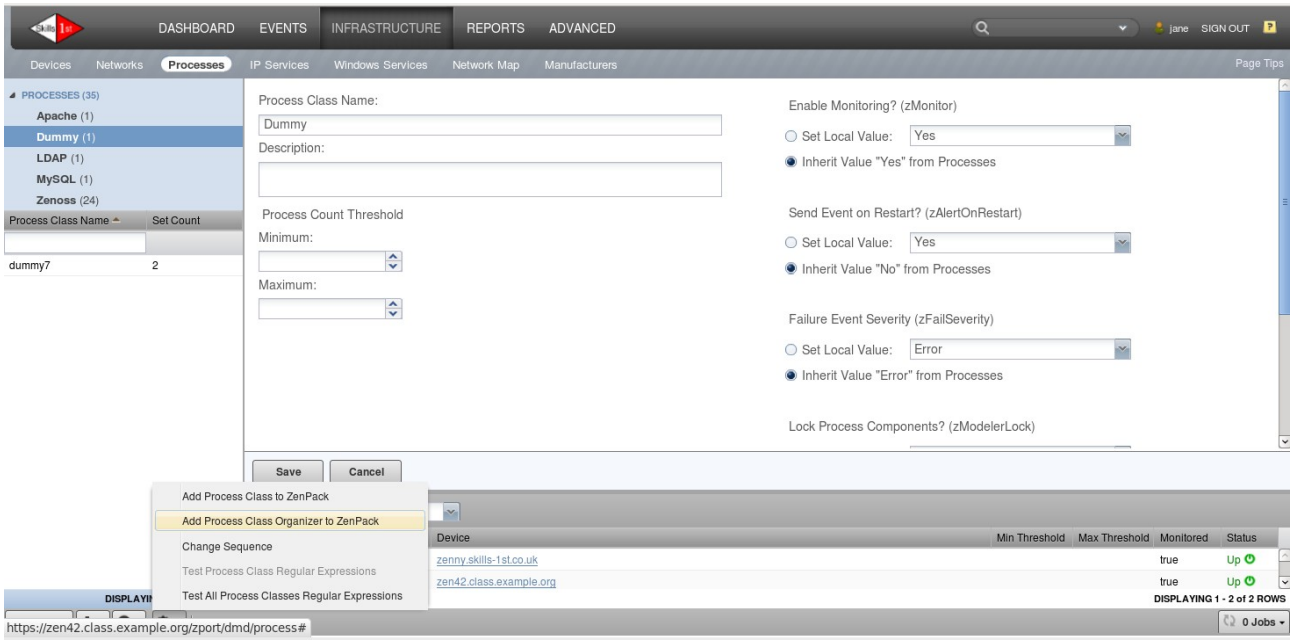


Figure 24: Adding a Process Organizer to a ZenPack

In Figure 24 an organizer is added, rather than a simple process (albeit an organizer containing one process).

zenoss@zen42:/opt/zenoss/ZenPacks/ZenPacks.community.Processes/ZenPacks/community/Processes/objects

File Edit View Search Terminal Help

```
<?xml version="1.0"?>
<objects>
<!-- ('', 'zport', 'dmd', 'Processes', 'Dummy') -->
<object id='/zport/dmd/Processes/Dummy' module='Products.ZenModel.OSProcessOrganizer' class='OSProcessOrganizer'>
<tomanycont id='osProcessClasses'>
<object id='dummy7' module='Products.ZenModel.OSProcessClass' class='OSProcessClass'>
<property type="string" id="name" mode="w" >
dummy7
</property>
<property type="string" id="regex" mode="w" >
.*bash.*\./dummy7
</property>
<property type="boolean" id="ignoreParametersWhenModeling" mode="w" >
False
</property>
<property type="boolean" id="ignoreParameters" mode="w" >
False
</property>
<property type="int" id="sequence" mode="w" >
0
</property>
<property visible="True" type="int" id="zFailSeverity" >
2
</property>
<tomany id='instances'>
<link objid='/zport/dmd/Devices/Server/Linux/devices/zenny.skills-1st.co.uk/os/processes/zport_dmd_Processes_Dummy_
osProcessClasses_dummy7_956e0951ee477dde5d8dd917e813baa8'/>
<link objid='/zport/dmd/Devices/Server/Linux/devices/zen42.class.example.org/os/processes/zport_dmd_Processes_Dummy
_osProcessClasses_dummy7_956e0951ee477dde5d8dd917e813baa8'/>
</tomany>
</object>
</tomanycont>
</object>
</objects>
~
~
~
~
"objects.xml" [readonly] 32L, 1184C                                    1,1           All
```

Figure 25: objects.xml for Zenoss.community.Processes with Process Organizer

Note in Figure 25 the object *Dummy* representing the organizer and the object *dummy7* representing a process inside that organizer. The process object has two instances.

The same caveats apply on installing the ZenPack in a different environment; either live with the error messages when installing the egg or edit the o*bjects.xml* to comment out all process instances and then perform a link install of the source ZenPack.

5.0 Understanding core Zenoss objects

In order to extend Zenoss it is necessary to understand what is provided as standard. “core” in the chapter heading here refers to out-of-the-box functionality and is applicable to both Zenoss Core and Zenoss Service Dynamics products.

Zenoss is written largely in Python (the *zeneventserver* daemon is written in Java for performance reasons). It embraces an object-oriented paradigm for defining “things” - devices, components, MIBs, services, events, reports, locations, users are all defined as object **classes**.

By default, the directory structure of a Zenoss installation sets the environment variable ***$ZENHOME*** to */opt/zenoss*. The ***Products*** directory under $ZENHOME is a hierarchy of directories holding object class definitions.

```
zenoss@zen42:/opt/zenoss/Products
File  Edit  View  Search  Terminal  Help
[zenoss@zen42 Products]$ pwd
/opt/zenoss/Products
[zenoss@zen42 Products]$ ls -l
total 104
drwxr-xr-x  7 zenoss zenoss  4096 Jul  2 17:12 DataCollector
-rw-r--r--  1 zenoss zenoss   244 Mar 11  2014 __init__.py
drwxr-xr-x  4 zenoss zenoss  4096 May 19 19:46 Jobber
drwxr-xr-x  5 zenoss zenoss  4096 May 19 19:46 LDAPMultiPlugins
drwxr-xr-x  8 zenoss zenoss  4096 May 19 19:46 LDAPUserFolder
-rw-r--r--  1 zenoss zenoss   189 Aug 28  2012 README.txt
drwxr-xr-x  3 zenoss zenoss  4096 May 19 19:46 ZenCallHome
drwxr-xr-x  5 zenoss zenoss  4096 May 19 19:50 ZenCollector
drwxr-xr-x  8 zenoss zenoss  4096 Oct 29 11:17 ZenEvents
drwxr-xr-x  4 zenoss zenoss  4096 Jul  1 19:17 ZenHub
drwxr-xr-x  6 zenoss zenoss  4096 May 19 18:38 ZenMessaging
drwxr-xr-x 12 zenoss zenoss 20480 Oct 29 11:17 ZenModel
drwxr-xr-x  2 zenoss zenoss  4096 May 19 19:46 ZenossStartup
lrwxrwxrwx  1 zenoss zenoss    43 Jul  1 16:16 ZenPacks.ShaneScott.ipSLA -> /opt/zenoss/local/ZenPacks.ShaneScott.ipSLA
drwxr-xr-x  5 zenoss zenoss  4096 May 19 19:46 ZenRelations
drwxr-xr-x  5 zenoss zenoss  4096 Jul  3 11:12 ZenReports
drwxr-xr-x  6 zenoss zenoss  4096 Aug  5 19:25 ZenRRD
drwxr-xr-x  5 zenoss zenoss  4096 Jul  2 17:12 ZenStatus
drwxr-xr-x  2 zenoss zenoss  4096 May 19 19:46 ZenTestCase
drwxr-xr-x  8 zenoss zenoss  4096 May 19 19:46 ZenUI3
drwxr-xr-x 11 zenoss zenoss  4096 Jun 17 22:49 ZenUtils
drwxr-xr-x  6 zenoss zenoss  4096 May 19 19:46 ZenWidgets
drwxr-xr-x 10 zenoss zenoss  4096 May 19 19:46 Zuul
[zenoss@zen42 Products]$
```

Figure 26: $ZENHOME/Products directory listing

$ZENHOME/Products/ZenModel holds definitions for most object classes, with the notable exception of event definitions which are held under the *ZenEvents* directory.

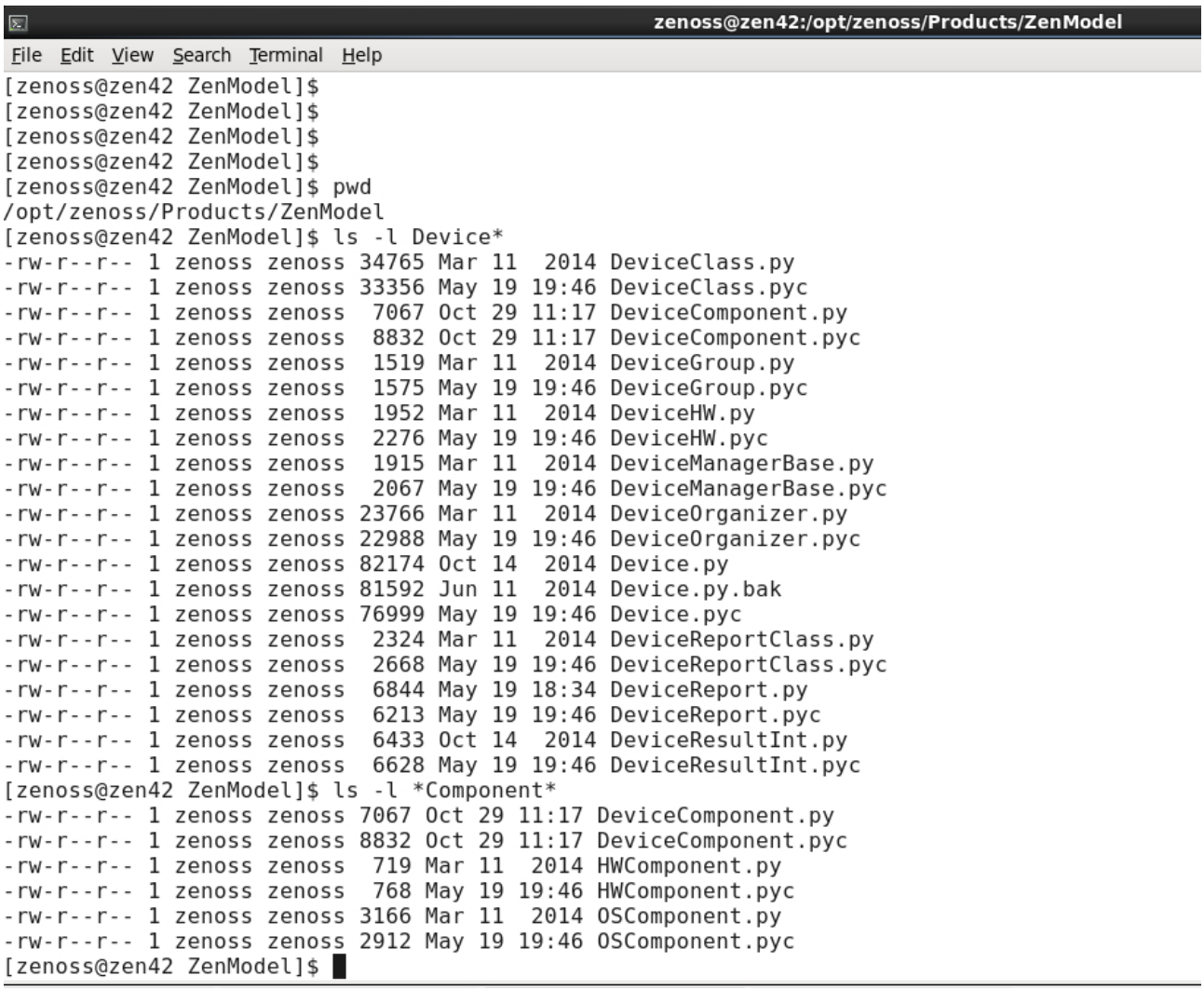


Figure 27: Device and component object class definition files in $ZENHOME/Products/ZenModel

Zenoss provide a base *Device* object class and a base *DeviceComponent* object class; the *DeviceComponent* class has two specialisations, *HWComponent* and *OSComponent*.

Time should be spent understanding at least some of the contents of *Device.py*.

5.1 Device.py

The object class that represents any “device” in Zenoss is defined in *$ZENHOME/Products/ZenModel/Device.py*.

Devices are what you see on the Zenoss Infrastructure view in the web interface. If you see it in the Infrastructure view, it's a device. If you don't, it's not. A device has an **id** attribute that makes it unique in the system. It will have configuration properties associated with it either directly, or acquired from the device class within which it is contained. It will also have a **manageIp** attribute that Zenoss uses for modeling and monitoring. See http://wiki.zenoss.org/ZenPack_Development_Guide/Background_Information for some good basic information.

```
File  Edit  View  Search  Terminal  Help
class Device(ManagedEntity, Commandable, Lockable, MaintenanceWindowable,
             AdministrativeRoleable, ZenMenuable):
    """
    Device is a base class that represents the idea of a single computer system
    that is made up of software running on hardware. It currently must be IP
    enabled but maybe this will change.
    """

    implements(IEventView, IIndexed, IGloballyIdentifiable)

    event_key = portal_type = meta_type = 'Device'

    default_catalog = "deviceSearch" #device ZCatalog

    relationshipManagerPathRestriction = '/Devices'
    title = ""
    manageIp = ""
    productionState = 1000
    preMWProductionState = productionState
    snmpAgent = ""
    snmpDescr = ""
    snmpOid = ""
    snmpContact = ""
    snmpSysName = ""
    snmpLocation = ""
    rackSlot = ""
    comments = ""
    sysedgeLicenseMode = ""
    priority = 3
    macaddresses = None

    # Flag indicating whether device is in process of creation
    _temp_device = False

"Device.py" [readonly] 2298 lines --9%--                                    226,0-1
```

Figure 28: Start of definition of Device class in $ZENHOME/Products/ZenModel/Device.py

The first line of the class object defines other class objects that *Device* inherits attributes from:

```
class Device(ManagedEntity, Commandable, Lockable, MaintenanceWindowable,
       AdministrativeRoleable, ZenMenuable):
```

Either these classes will be defined in this file or they will be referenced in an import statement which by convention, will be at the top of the file; for example, near the top of *Device.py* is a line which imports the *ManagedEntity* **class** from the **file** *ManagedEntity.py* (strictly the class is imported from the *ManagedEntity* Python **module** but the file name and the module name are synonymous):

```
from ManagedEntity import ManagedEntity
```

5.1.1 Object attributes

Referring to Figure 28, several **attributes** are defined for the *Device* class. These attributes are specific to a device and any new class that inherits from *Device*, will inherit these attributes - title, manageIp, productionState and so on.

To see the attributes and methods that *Device* has already inherited from *ManagedEntity*, inspect the *ManagedEntity.py* file:

- snmpindex
- monitor
- productionState
- preMWProductionState

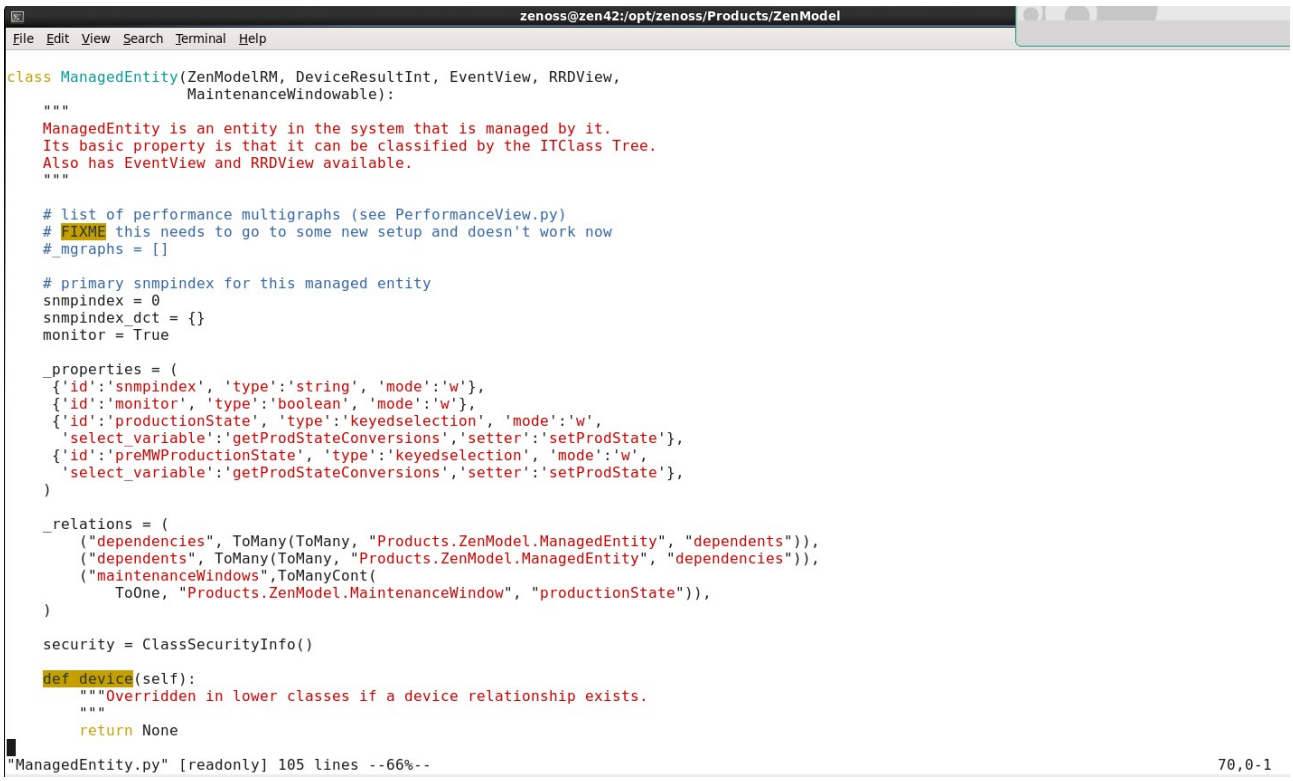


Figure 29: ManagedEntity class definition in $ZENHOME/Products/ZenModel/ManagedEntity.py

Note also the ***device* method** which is effectively null code here but the method can be overridden (ie, redefined) in classes that inherit from this *ManagedEntity* class.

The attributes for any object is the union of all attributes both defined on the object and inherited from parents. For a Zenoss Device, this gives:

- From **Device**
  - id
  - title = ""
  - manageIp = ""
  - productionState = 1000
  - preMWProductionState = productionState
  - snmpAgent = ""
  - snmpDescr = ""
  - snmpOid = ""
  - snmpContact = ""
  - snmpSysName = ""
  - snmpLocation = ""
  - rackSlot = ""
  - comments = ""
  - sysedgeLicenseMode = ""
  - priority = 3

    - macaddresses = None
- From **ManagedEntity**
    - snmpindex = 0
    - monitor = True
    - productionState
    - preMWProductionState

5.1.2 Object relationships

The definition of the *Device* class also specifies **relationships**. Again, http://wiki.zenoss.org/ZenPack_Development_Guide/Background_Information has a good descriptions of relationships.

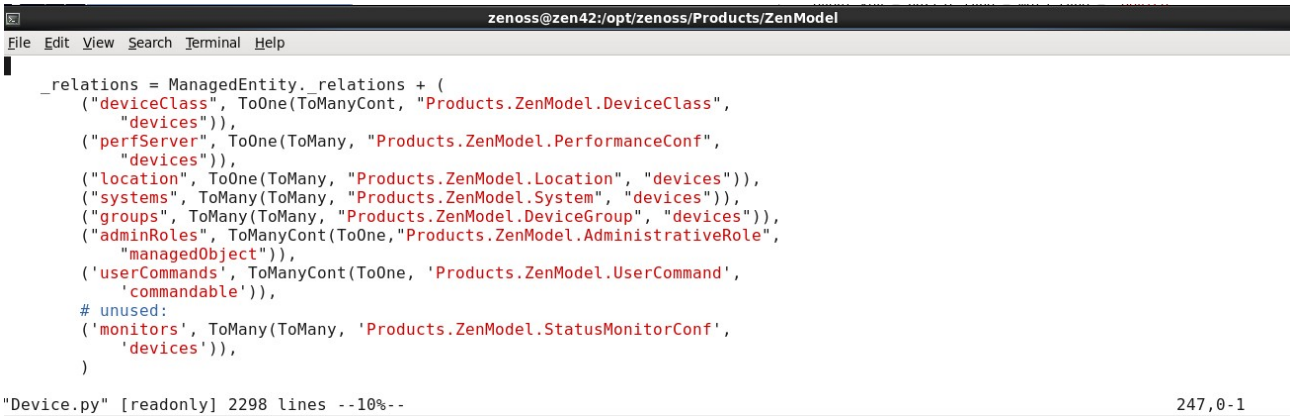


Figure 30: relations defined for the Device class

Within the context of Zenoss, a relationship establishes a connection between Zope objects. All relationships **must** be bi-directional and explicitly defined on both sides of the relationship. However, if one side is **containing**, then the other side is **contained**.

Currently Zenoss supports four types of relationships:

- ToMany
- ToOne
- ToManyCont
- ToOneCont

Figure 30 shows that a *Device* may have:

- **One** *deviceClass* and a *deviceClass* may **contain many** *Devices*
- **One** *perfServer* (collector) and a *perfServer* may have **many** *Devices*
- **One** *location* and a *location* may have **many** *Devices*
- **Many** *systems* and a *system* may have **many** *Devices*
- **Many** *groups* and a *group* may have **many** *Devices*

- **Many** *adminRoles* (containing) and *adminRoles* **is contained by one** *Device*
- **Many** *userCommands* (containing) and *userCommands* **is contained by one** *Device*

There are **containing** and **non-containing** relationships.

The idea of containment comes from Zenoss' ZODB database being a tree of objects. Every object in the database must be attached in some way to another persistent object, and **only one** other persistent object. An object can only be on the right-hand side of **one** ToManyCont relationship.

In the case of a **containing** relationship you're saying that this is the relationship that attaches the member objects to the object model and therefore is how they get persisted. Removing an object from its containing relationship is the same thing as deleting the object from the database entirely. Any non-containing relationships it might also have, will be cleaned up.

On the other hand, a **non-containing** relationship has nothing to do with persistence. Non-containing relationships are used to create logical linkages from one object to another.

In the relations for *Device* above:

- If an instance of a *Device* (say, zen42.class.example.org) is deleted then it's *adminRoles* and *userCommands* objects will also be deleted; however, any related *systems* or *groups* objects will not be deleted.
- If the *deviceClass* containing *zen42.class.example.org* is deleted, then all its contained devices will also be deleted; however, deleting *zen42.class.example.org* will not affect its *deviceClass*.

Remember from Figure 29 that *ManagedEntity* also has relationships so the full set of relationships for **Device** is:

- From **Device**:

```
_relations = ManagedEntity._relations + (
  ("deviceClass", ToOne(ToManyCont, "Products.ZenModel.DeviceClass", "devices")),
  ("perfServer", ToOne(ToMany, "Products.ZenModel.PerformanceConf", "devices")),
  ("location", ToOne(ToMany, "Products.ZenModel.Location", "devices")),
  ("systems", ToMany(ToMany, "Products.ZenModel.System", "devices")),
  ("groups", ToMany(ToMany, "Products.ZenModel.DeviceGroup", "devices")),
  ("adminRoles", ToManyCont(ToOne,"Products.ZenModel.AdministrativeRole",  "managedObject")),
  ('userCommands', ToManyCont(ToOne, 'Products.ZenModel.UserCommand', 'commandable')),
  # unused:
  ('monitors', ToMany(ToMany, 'Products.ZenModel.StatusMonitorConf',  'devices')),
  )
```

- From **ManagedEntity**:

```
_relations = (
```

("*dependencies*", ToMany (ToMany, *"Products.ZenModel.ManagedEntity", "dependents*")),

("*dependents*", ToMany(ToMany, "Products.ZenModel.ManagedEntity", "*dependencies*")),

("*maintenanceWindows*",ToMany(ToOne, "Products.ZenModel.MaintenanceWindow", "*productionState*")),

)

Note that *Device* does **not** have either an *os* or an *hw* relationship.

5.1.3 Object methods

If an object attribute is a feature or field of an object, then a **method** is something that can be done to an object. A method is code, that may well take parameters, to adapt **self** (the object).

One of the simplest examples of a method for a *Device* is *getManageIp* which merely delivers the *manageIp* attribute:

```
security.declareProtected(ZEN_VIEW, 'getManageIp')
def getManageIp(self):
  """
  Return the management ip for this device.

  @rtype: string
  @permission: ZEN_VIEW
  """
  return self.manageIp
```

Another simple example is the *getPerformanceServer* method to get the *performanceServer* (what we know as the collector), for a device:

```
security.declareProtected(ZEN_VIEW, 'getPerformanceServer')
def getPerformanceServer(self):
  """
  Return the device performance server

  @rtype: PerformanceMonitor
  @permission: ZEN_VIEW
  """
  return self.perfServer()
```

Most of the method is comment, enclosed between the pairs of triple quotes.

Remember that *perfServer* was a ToOne relationship on the *Device* object so this method delivers the **object** that represents the device's collector.

One way to distinguish between a device's attributes and relationships in code is that using a relationship will always have () after the relationship name; if it is a simple attribute, there will be no () .

To get the **name** of the device's *perfServer*, there is the *getPerformanceServerName* method which again gets the *perfServer* relationship but then uses the *getId()* method to get the *id* attribute of the collector, provided the relationship exists; otherwise the null string is returned:

```
security.declareProtected(ZEN_VIEW, 'getPerformanceServerName')
  def getPerformanceServerName(self):
    """
    Return the device performance server name

    @rtype: string
    @permission: ZEN_VIEW
    """
    cr = self.perfServer()
    if cr: return cr.getId()
    return ''
```

Note the *security.declareProtected* lines, immediately before the definition of the methods. This is not mandatory but it imposes a permission (*ZEN_VIEW*) that a Zenoss role must have in order to execute the method.

The *Device* class has an *__init__* method which is called to **instantiate an instance** of a device class.

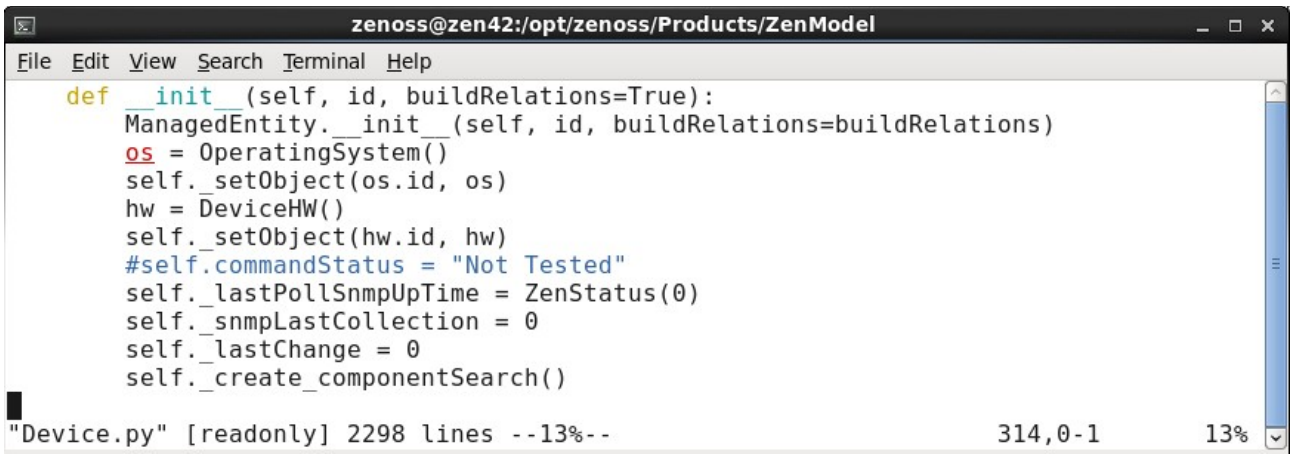


Figure 31: The __init__ method for Device class

Note that the "*os*" line instantiates an object of class *OperatingSystem* and the "*hw*" line instantiates an object of class *DeviceHW*; in other words, the *Device* object contains an *OperatingSystem* object and a *DeviceHW* object.

Most of the files under *$ZENHOME/Products/ZenModel* define a large number of methods, too numerous to mention all of them here. Some common methods for the *Device* object (including some inherited from other classes), are:

- titleOrId(self)
- name(self)
- getRRDTemplates(self)
- sysUpTime(self)
- getDeviceComponents(self, monitored=None, collector=None, type=None)
- getDeviceGroupNames(self)
- setManageIp(self, ip="", REQUEST=None)
- getMacAddresses(self)

- getPingStatusString(self)
- getDeviceClassPath(self)
- getDeviceUrl(self)
- getPingStatus(self)
- getProperty(self, Property)
- setZenProperty(self, zProperty, value)

Note that if zProperties are changed programmatically, whether through code or zendmd, then **always** ensure that the *setZenProperty* method is used. Simply using a construct such as *device.zCommandUsername='fred'* will result in corruptions in the ZODB database for the device where the GUI cannot show the new value of the property.

5.2 DeviceComponent.py

Many devices have components; interfaces, filesystems and OS Processes are examples. Often the reason for writing a ZenPack is to support new types of components. The **DeviceComponent** class is defined in *DeviceComponent.py*.

zenoss@zen42:/opt/zenoss/Products/ZenModel

File Edit View Search Terminal Help

```
class DeviceComponent(Lockable):
    """
    DeviceComponent is a mix-in class for all components of a device.
    These include LogicalComponent, Software, and Hardware.
    """
    zope.interface.implements(IIndexed, IGloballyIdentifiable)
    __pychecker__='no-override'
    event_key = "Component"

    default_catalog = "componentSearch"

    collectors = ('zenperfsnmp', 'zencommand', 'zenwinperf',
                  'zenping')

    security = ClassSecurityInfo()

    perfmonInstance = None

    def getParentDeviceName(self):
        """
        Return the name of this component's device
        """
        name = ""
        dev = self.device()
        if dev: name = dev.getDeviceName()
        return name
    hostname = getParentDeviceName

    def getParentDeviceTitle(self):
        """
        Return the title of this component's device
        """
        title = ""
        dev = self.device()
        if dev: title = dev.titleOrId()
        return title
"DeviceComponent.py" [readonly] 235 lines --28%--                    67,0-1        15%
```

Figure 32: DeviceComponent object class in $ZENHOME/Products/ZenModel/DeviceComponent.py

Note that *DeviceComponent* inherits just from the **Lockable** class.

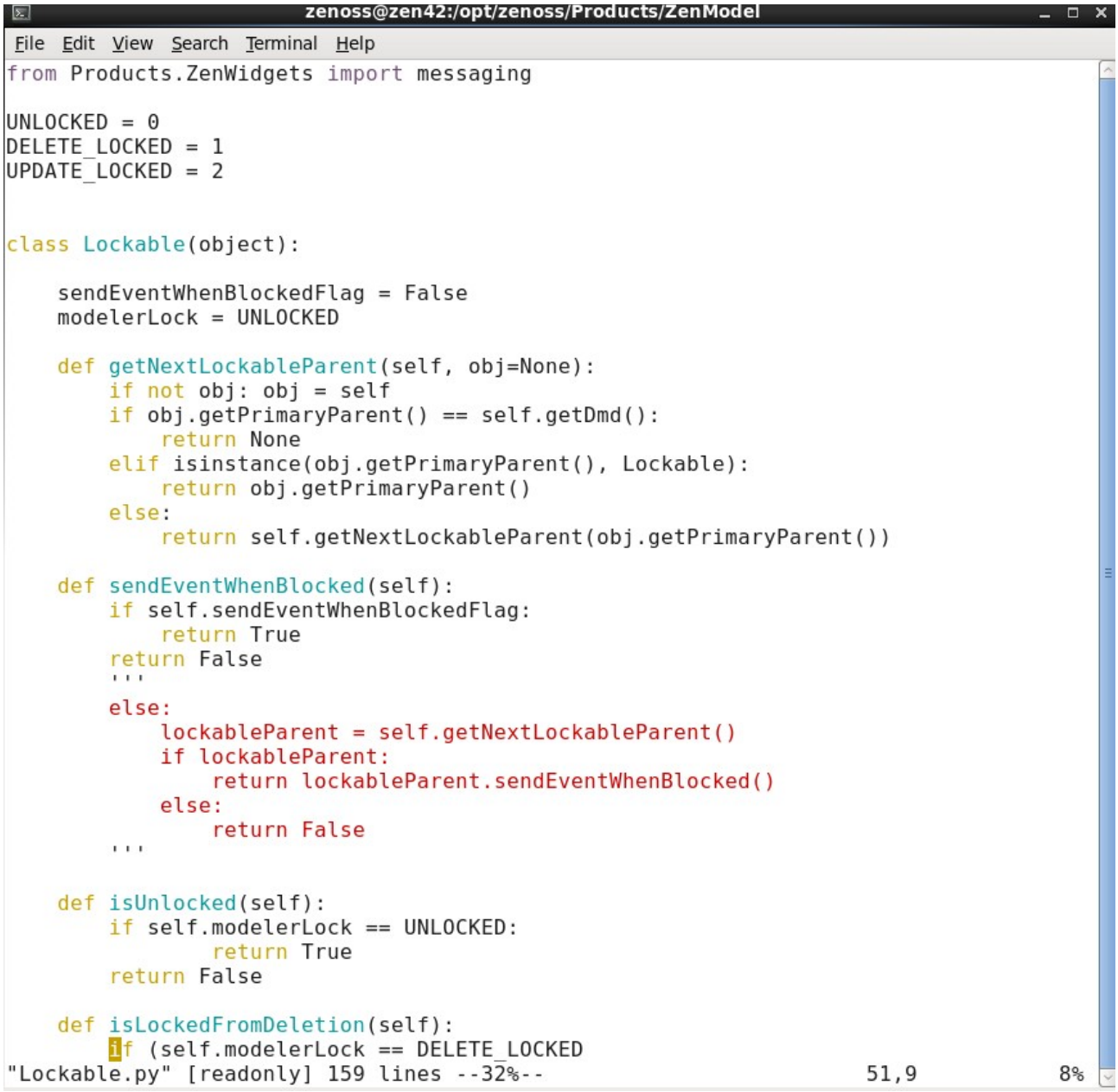


Figure 33: Lockable class defined in $ZENHOME/Products/ZenModel/Lockable.py

The *Lockable* class is about as fundamental as it gets! It inherits from the **object** class, has no attributes and no relationships. It simply has a few methods.

Figure 32 shows that the base *DeviceComponent* class has no attributes or relationships but does have a number of methods.

5.3 * Example object class hierarchy for Fan DeviceComponent

It is possible to build object hierarchies of *DeviceComponents*. The core code provides some.

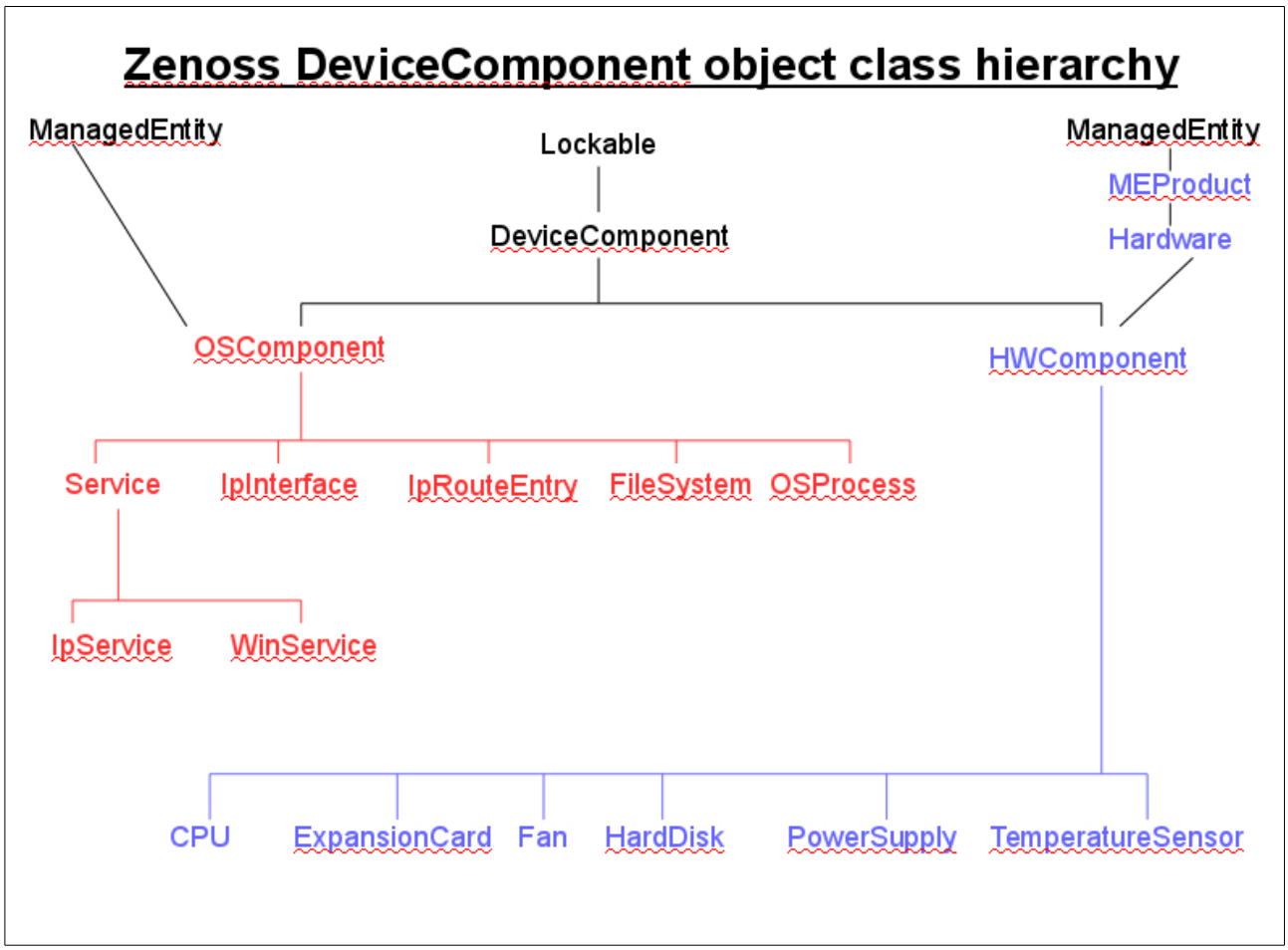


Figure 34: DeviceComponent object class hierarchy

Taking *Fan* as an example of a hardware component, it is possible to trace the attributes, relationships and methods through the class hierarchy.

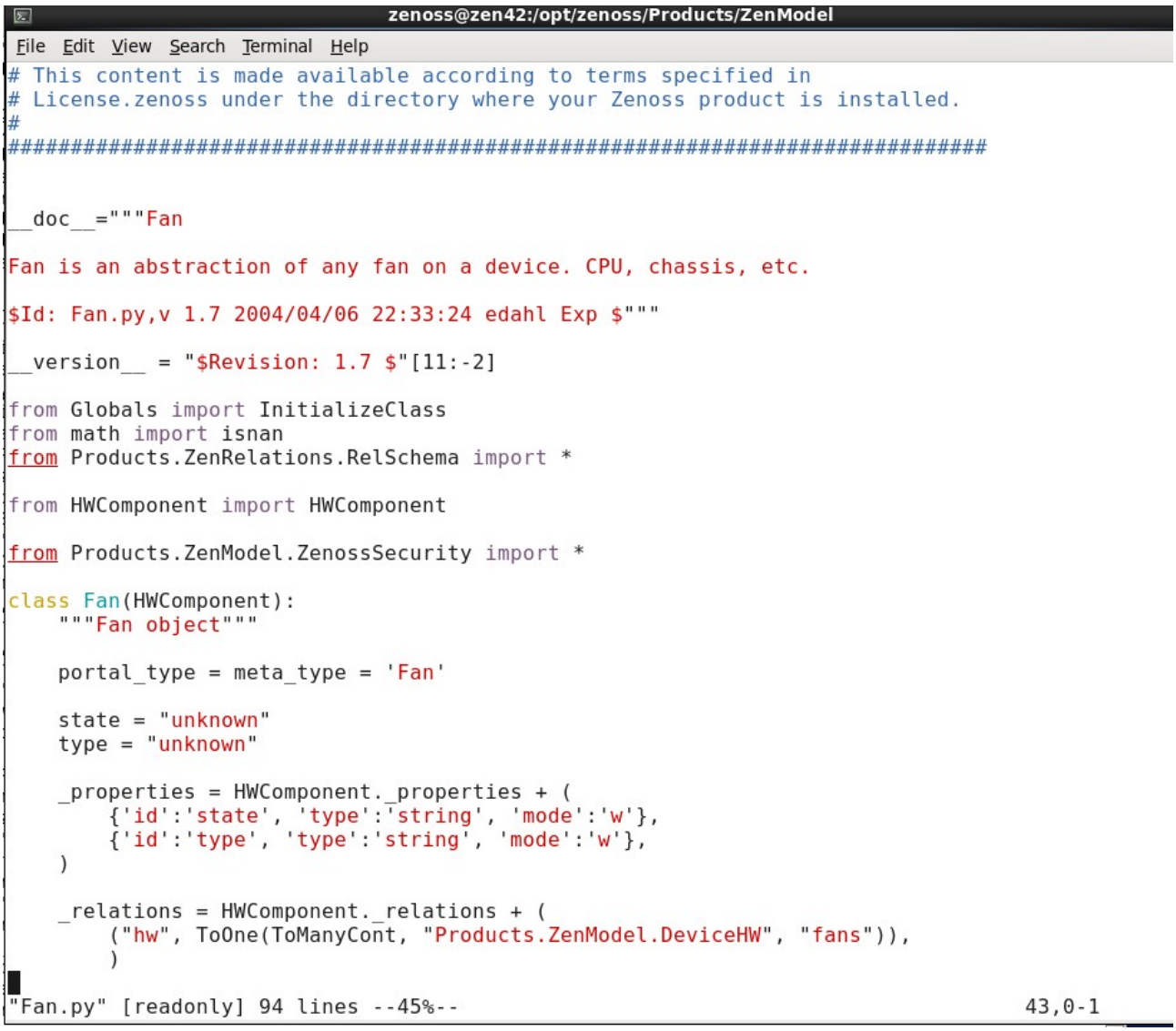


Figure 35: $ZENHOME/Products/ZenModel/Fan.py defining a hardware component

The *Fan* class inherits from the *HWComponent* class.

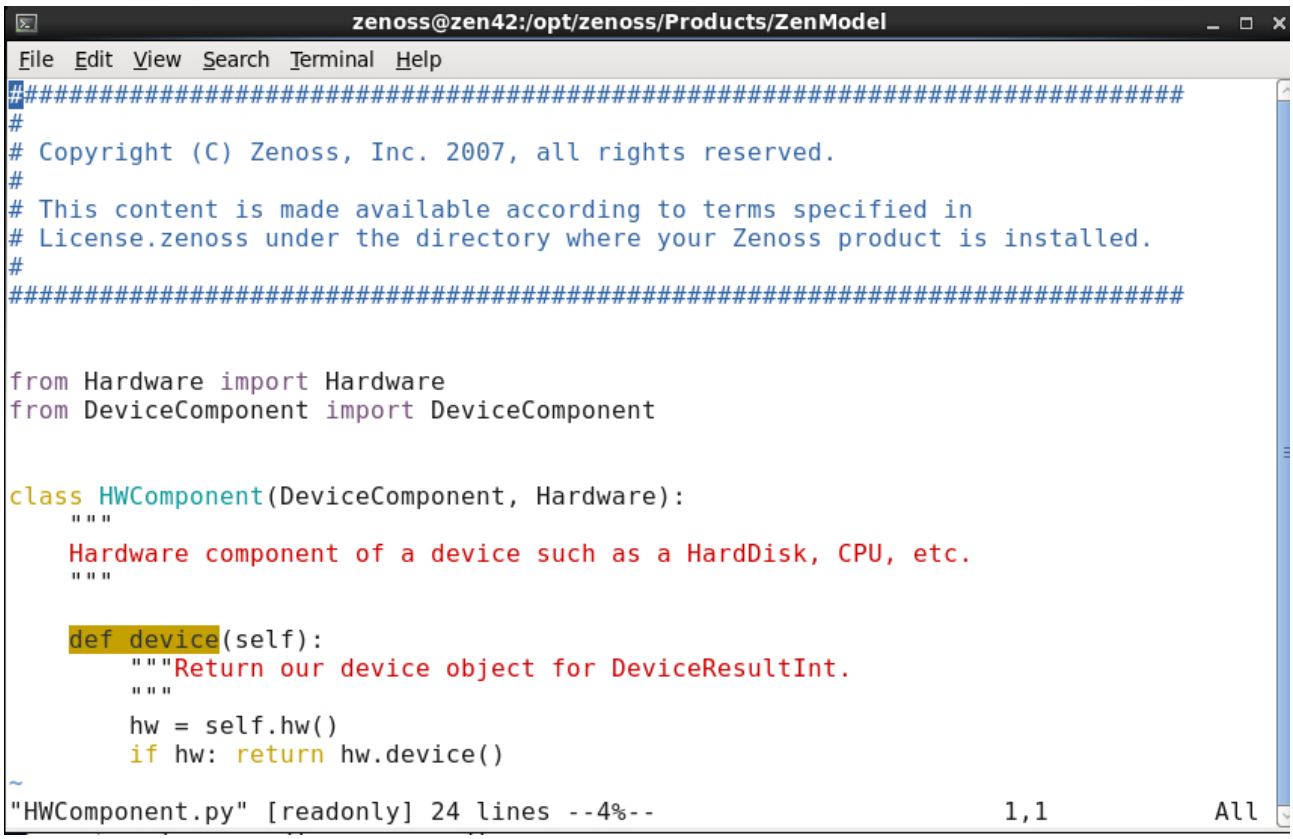


Figure 36: $ZENHOME/Products/ZenModel/HWComponent.py only has a single method

HWComponent only has a single method which redefines the *device* method, inherited from the *ManagedEntity* class. *HWComponent* inherits from the *Hardware* class as well as the *DeviceComponent* class.

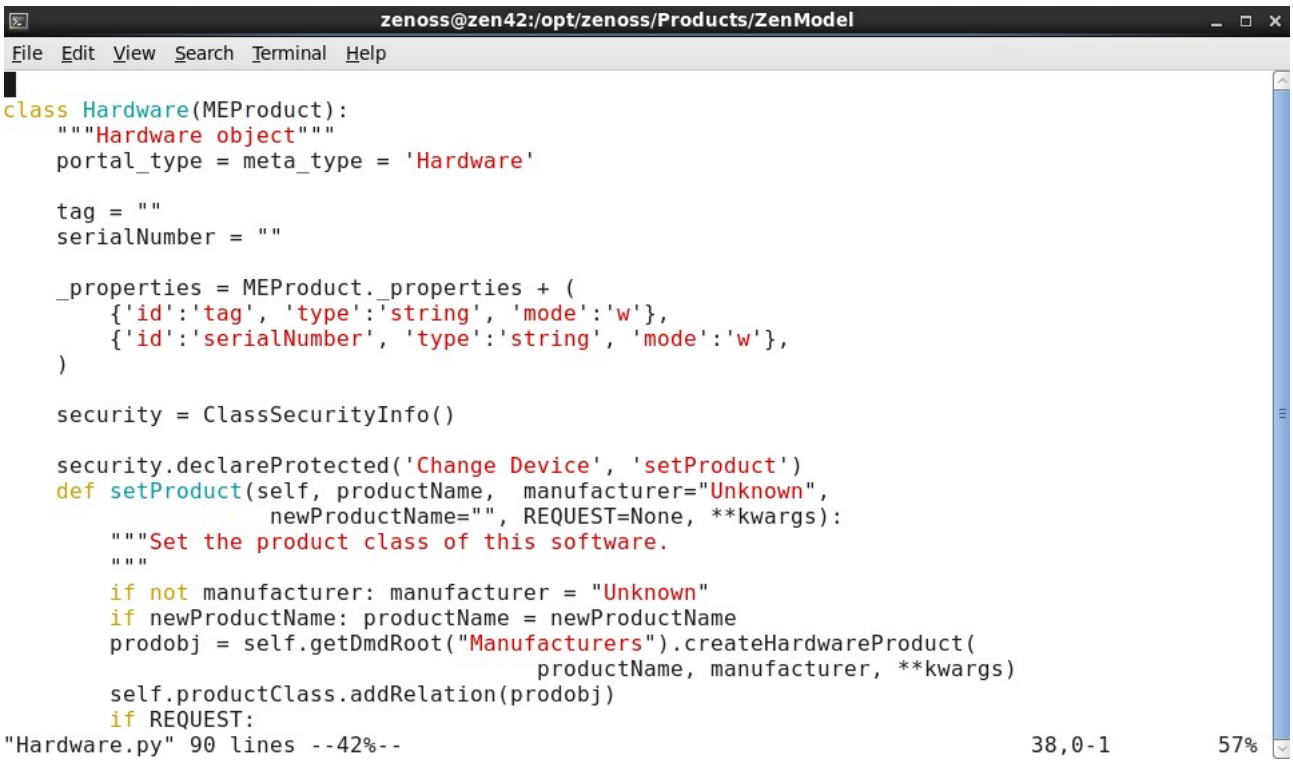


Figure 37: $ZENHOME/Products/ZenModel/Hardware.py

Hardware has no relationships but does have the *tag* and *serialNumber* attributes. The *Hardware* class inherits from *MEProduct*.

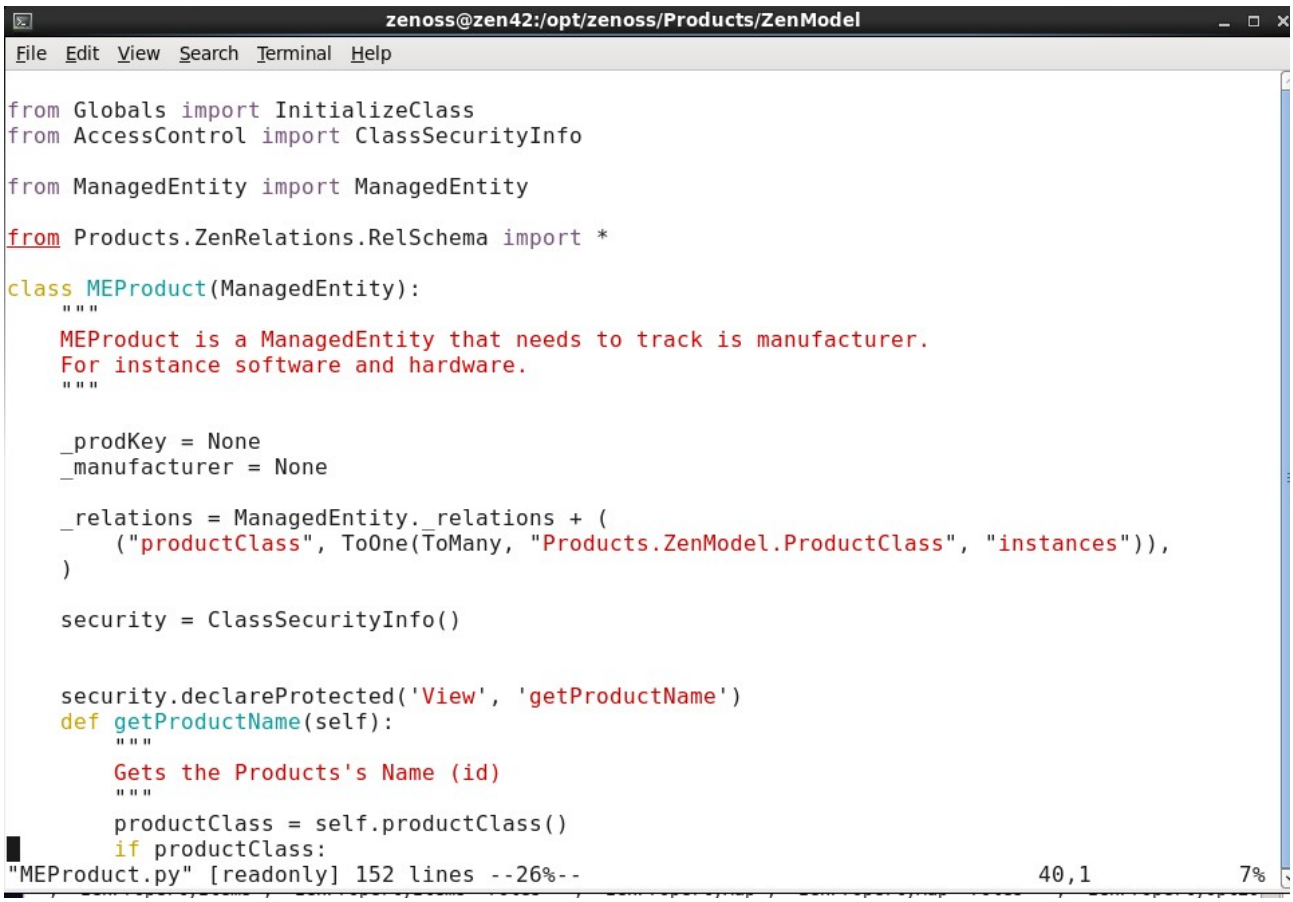


Figure 38: $ZENHOME/Products/ZenModel/MEProduct.py

The *MEProduct* class adds a relationship, *productClass*. This is used to hold manufacturer information. *MEProduct* inherits from *ManagedEntity* which we saw previously in Figure 29 but is repeated here for simplicity. It defines four attributes, three relations and a number of methods, including the *device* method, which delivers the object device that contains this object.

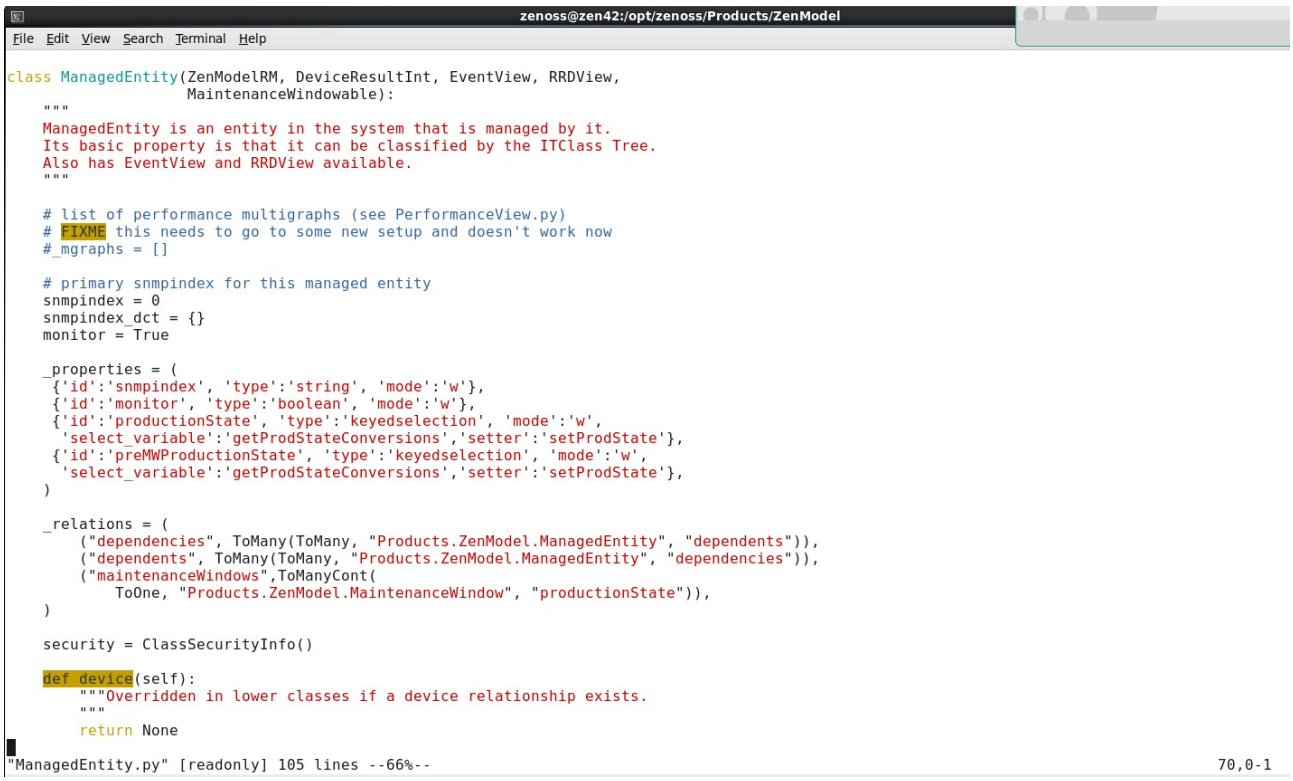


Figure 39: $ZENHOME/Products/ZenModel/ManagedEntity.py

This gives the combined class hierarchy for Fan as:

- ManagedEntity

  Attributes: {'id':'*snmpindex*', 'type':'string', 'mode':'w'},

  {'id':'*monitor*', 'type':'boolean', 'mode':'w'},

  {'id':'*productionState*', 'type':'keyedselection', 'mode':'w', 'select_variable':'getProdStateConversions','setter':'setProdState'},

  {'id':'*preMWProductionState*', 'type':'keyedselection', 'mode':'w','select_variable':'getProdStateConversions','setter':'setProdState'},

  Relations: ("*dependencies*", ToMany(ToMany, "Products.ZenModel.ManagedEntity", "*dependents*")),

  ("*dependents*", ToMany(ToMany, "Products.ZenModel.ManagedEntity", "*dependencies*")),

  ("*maintenanceWindows*",ToManyCont(ToOne, "Products.ZenModel.MaintenanceWindow", "*productionState*")),

  - MEProduct

    Attributes: None

    Relations: ("*productClass*", ToOne(ToMany, " Products.ZenModel.ProductClass", "*instances*")),

    - Hardware

      Attributes: {'id':'*tag*', 'type':'string', 'mode':'w'},

      {'id':'*serialNumber*', 'type':'string', 'mode':'w'},

Relations: None

- DeviceComponent — **Attributes**: None
  Relations: None
  - HWComponent — **Attributes**: None
    Relations: None
    - Fan — **Attributes**: {'id':'*state*', 'type':'string', 'mode':'w'},
      {'id':'*type*', 'type':'string', 'mode':'w'},
      Relations: ("*hw*", ToOne(ToManyCont,
      "Products.ZenModel.DeviceHW", "*fans*")),

Both the *HWComponent* and the *ManagedEntity* classes define a *device* method; the “lowest” definition in the hierarchy will win so, for a *Fan* device component, the *device* method definition from *HWComponent* will be used.

5.4 * Example component class relationships for IpInterface

$ZENHOME/Products/ZenModel has files that define most of the objects available in Zenoss and many of these objects define relationships.

Relationships are bi-directional links between objects. They may be one-to-many or many-to-many. They may be a containment relationship specifying how the object attaches to the object model and therefore how they get persisted; or it may be a non-containing relationship that has nothing to do with persistence and simply creates a logical linkage from one object to another.

In order to extend the overall object model, it is useful to have an appreciation of what relationships exist in the core product and how they work.

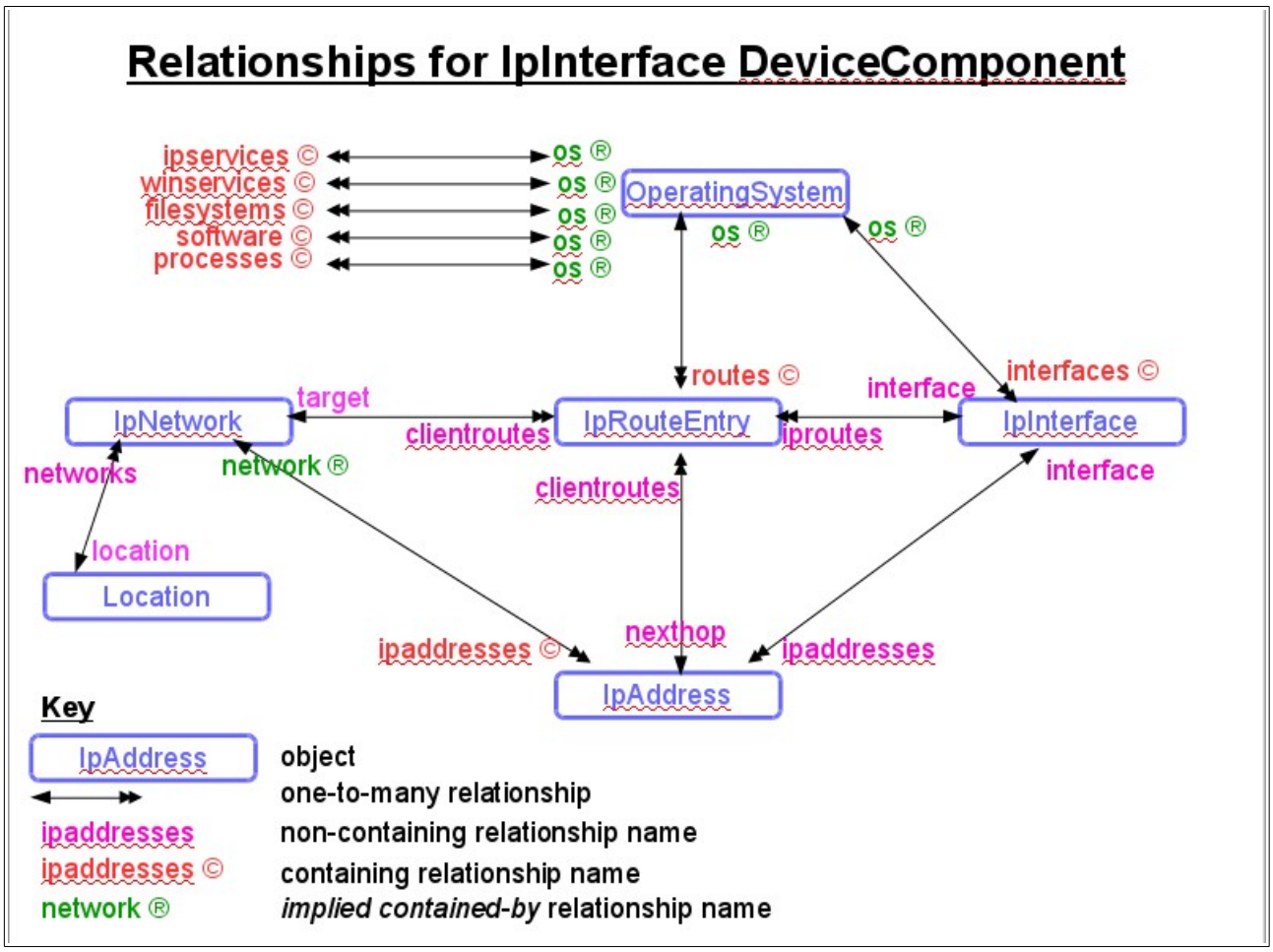


Figure 40: Relationships for the IpInterface DeviceComponent

Figure 40 attempts to depict the various relationships around the *IpInterface* component:

- From *OperatingSystem.py*:

```
_relations = Software._relations + (
  ("interfaces", ToManyCont(ToOne,
    "Products.ZenModel.IpInterface", "os")),
  ("routes", ToManyCont(ToOne, "Products.ZenModel.IpRouteEntry", "os")),
  ("ipservices", ToManyCont(ToOne, "Products.ZenModel.IpService", "os")),
  ("winservices", ToManyCont(ToOne,
    "Products.ZenModel.WinService", "os")),
  ("processes", ToManyCont(ToOne, "Products.ZenModel.OSProcess", "os")),
  ("filesystems", ToManyCont(ToOne,
    "Products.ZenModel.FileSystem", "os")),
  ("software", ToManyCont(ToOne, "Products.ZenModel.Software", "os")),
)
```

  - An *OperatingSystem* object has a **contains** relationship to **many** *IpInterface*, *IpRouteEntry*, *IpService*, *WinService*, *FileSystem*, *Software* and *OSProcess* **objects**. Note the capitalisation carefully. By convention, an object name starts with a capital letter.
  - Relationships must always be bi-directional, thus one expects to find a **ToOne** *os* relationship in the .py files for *IpInterface*, *IpRouteEntry*, *IpService*, *WinService*, *FileSystem*, *Software* and *OSProcess*.
  - Relationship names, by convention, start with a lower case letter.

- Relationship names, by convention, use a singular name for the “one” end of a one-to-many relationship and use a plural name for the “many” end.
- There is no “contained by” relationship key word but, by implication, the “other” end of a “Cont” relationship is “contained by”.

- From I*pInterface.py*:

```
_relations = OSComponent._relations(
("os",ToOne(ToManyCont,"Products.ZenModel.OperatingSystem","interfaces")),
("ipaddresses", ToMany(ToOne,"Products.ZenModel.IpAddress","interface")),
("iproutes", ToMany(ToOne,"Products.ZenModel.IpRouteEntry","interface")),
)
```

  - Note the matching *os* relationship for the *interfaces* relationship defined in *Operating System.py*. The os relationship is singular - an interface is contained by one operating system.
  - Note that the **location** of the matching relationship, strictly, is given as a Python **module path**; for example *Products.ZenModel.OperatingSystem*. This is why there are periods between the elements not file system slashes. Since module names map to file system directory hierarchies, the module name generally does give the path to the containing file.
  - *IpInterface* objects have ToMany relationships with both *IpAddress* objects and *IpRouteEntry* objects; neither is a containing relationship. Note both relationships, *ipaddresses* and *iproutes* are plural as an interface may have many addresses and many routes.
  - The matching relationships expected in *IpAddress.py* and *IpRouteEntry.py*, both called *interface*, are singular names for ToOne relationships.

- From *IpAddress.py*:

```
 _relations = ManagedEntity._relations + (
("network", ToOne(ToManyCont,"Products.ZenModel.IpNetwork","ipaddresses")),
("interface", ToOne(ToMany,"Products.ZenModel.IpInterface","ipaddresses")),
("clientroutes", ToMany(ToOne,"Products.ZenModel.IpRouteEntry","nexthop")),
)
```

  - Note the matching *interface* relationship for the *IpInterface* object.
  - An *IpAddress* also has a ToOne relationship (*network*) to an *IpNetwork* (which contains multiple *IpAddress* objects)
  - An *IpAddress* has a ToMany relationship (*clientroutes*) with *IpRouteEntryObjects* whereas an *IpRouteEntry* has a ToOne relationship (*nexthop*) back to the *IpAddress*.

- From *IpRouteEntry.py*:

```
_relations = OSComponent._relations + (
("os", ToOne(ToManyCont,"Products.ZenModel.OperatingSystem","routes")),
("interface", ToOne(ToMany,"Products.ZenModel.IpInterface","iproutes")),
("nexthop", ToOne(ToMany,"Products.ZenModel.IpAddress","clientroutes")),
("target", ToOne(ToMany,"Products.ZenModel.IpNetwork","clientroutes")),
)
```

  - Note that the only contains relationship is *os*, back to the *OperatingSystem*.

- From *IpNetwork.py*:

```
_relations = DeviceOrganizer._relations + (
("ipaddresses",ToManyCont(ToOne,"Products.ZenModel.IpAddress", "network")),
("clientroutes", ToMany(ToOne,"Products.ZenModel.IpRouteEntry","target")),
("location", ToOne(ToMany, "Products.ZenModel.Location", "networks")),
)
```

- Note that the *IpNetwork* object has the contains relationship for the *IpAddress*
- An *IpNetwork* can also be related to a single Zenoss *Location* - which can be related to many *IpNetworks*.

Please note that these relationships are not exhaustive for the objects described. Other relationships exist that were inherited from various parent object classes.

5.5 zendmd and the ZMI as tools to understand objects

5.5.1 The Zope Management Interface (ZMI)

The Zope Management Interface (ZMI) strictly is part of the Zope web application environment, rather than Zenoss; for this reason, it requires the *Manager* role for a Zenoss GUI user; *ZenManager* is not sufficient. The corollary to this is that, potentially, a user of the ZMI has the power to completely wreck the installation!

Access the ZMI from the web interface; the URL starts the same as for your Zenoss GUI, so if you access the main *INFRASTRUCTURE* menu with:

https://zen42.class.example.org/zport/dmd/itinfrastructure#devices:.zport.dmd.Devices

then the ZMI for all the Zenoss objects can be found at:

https://zen42.class.example.org/zport/dmd/manage

In fact, the ZMI can be accessed at many levels:

https://zen42.class.example.org/zport/dmd/Devices/Server/Linux/manage

only shows the object model in the ZODB from */Devices/Server/Linux*.

https://zen42.class.example.org/zport/manage

shows objects outside the Zenoss *dmd* environment in the containing Zope web application environment.

The ZMI is good for showing the structure and relationship of objects. To some extent, it mirrors the device class hierarchy seen through the *INFRASTRUCTURE* menu but, as you drill down, it also shows relationships, contained object classes, properties and roles required to execute methods.

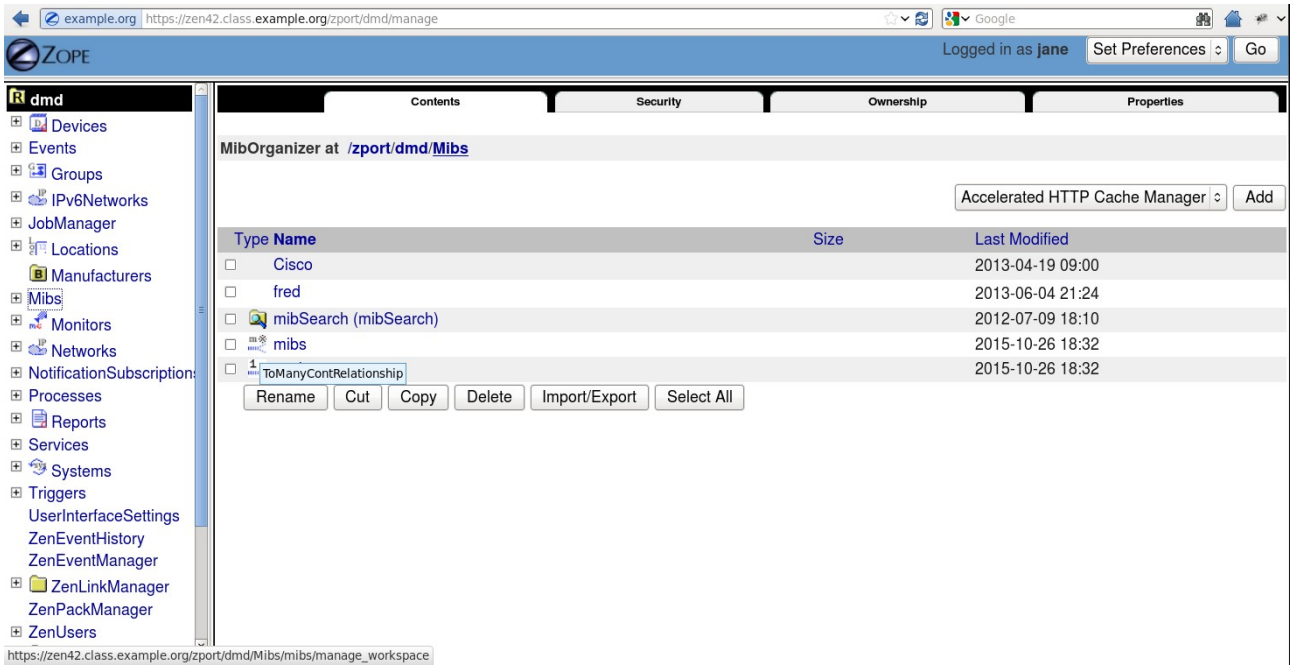


Figure 41: Top-level ZMI showing objects within the Zenoss dmd

Note that hovering over icons in the main part of the window, gives translations for the icons so *mibs* is a ToManyCont relationship on the *Mibs* object (note capitalisation).

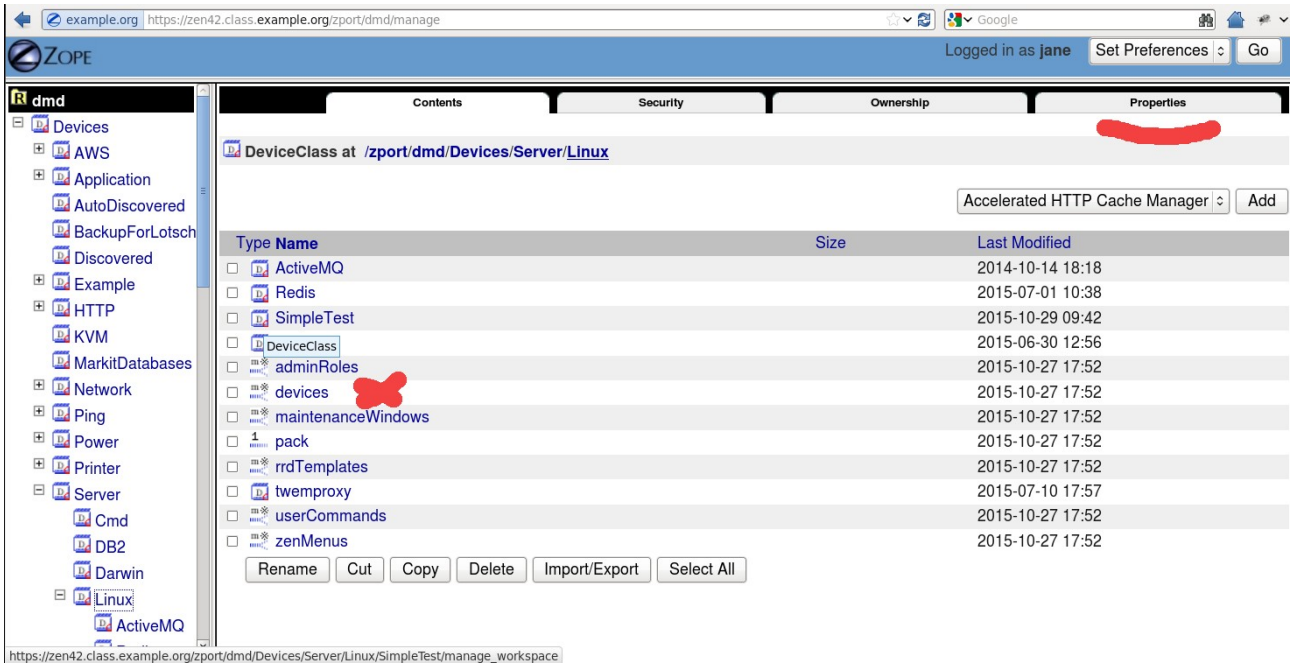


Figure 42: ZMI device class hierarchy

The left-hand menus can be expanded to drill down class hierarchies. In Figure 42, note the ToManyCont relationship, *devices*, between the device class and the devices that are members of that class. Drilling into *devices* shows the device **instances** - the actual discovered devices. Note that the icon for the instance of *group-100-serv1.class.example.org* is a *Device* object.

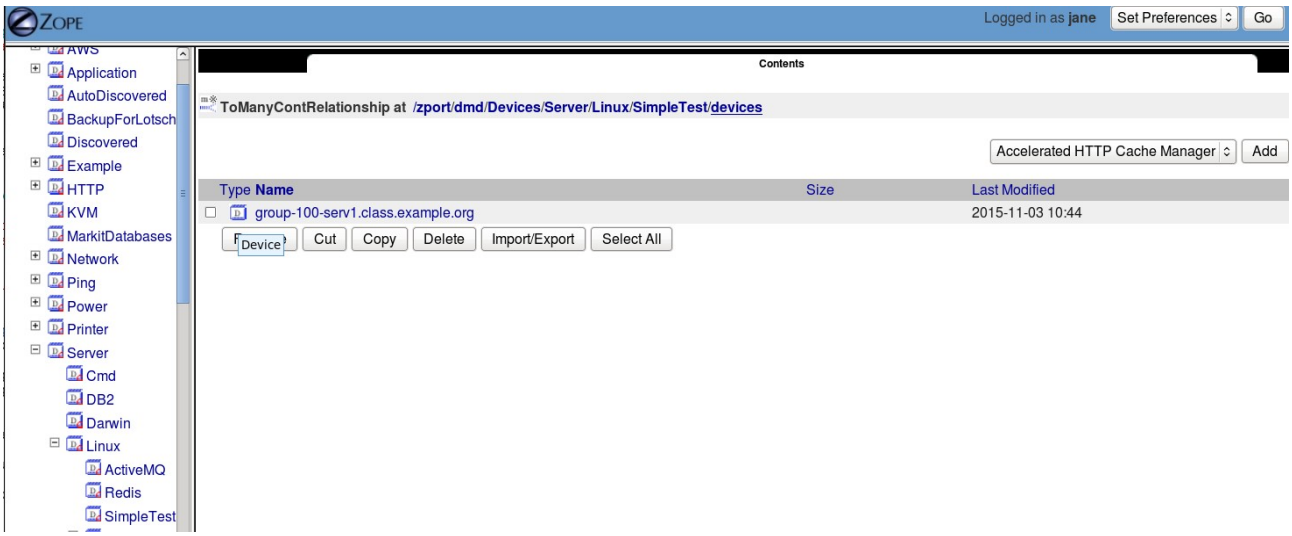


Figure 43: Instances of devices in the /Server/Linux/SimpleTest device class

Also note that many ZMI windows have a *Property* tab which shows attributes of the currently selected object.

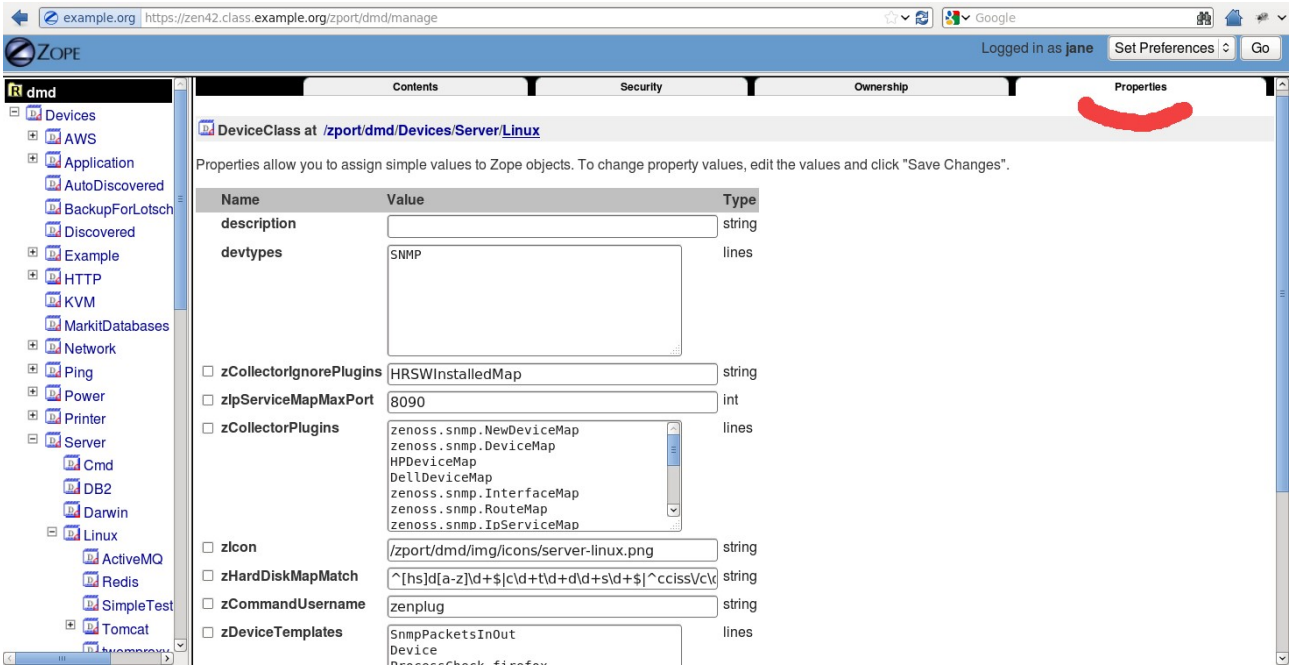


Figure 44: Properties tab for the DeviceClass /Server/Linux

Note that it is (at least in theory) possible to change, add and delete objects using the ZMI; in practice you may be prevented from doing so and there is great potential for causing breakage, so the recommendation would be **not** to change things using the ZMI.

Where the ZMI is particularly useful during ZenPack development is if new objects should have been created but they do not show on the GUI. In other words, a new modeler has apparently run successfully but new components do not appear. The ZMI will show whether the **objects** exist. If the instances can be seen in the ZMI then the modeler has done its job correctly and the problem with the ZenPack is in the JavaScript code that displays those objects.

For more information on Zope and the Zope Management Interface (ZMI), see http://docs.zope.org/zope2/zope2book/index.html , especially Chapter 6, “Using the Zope Management Interface”, http://docs.zope.org/zope2/zope2book/UsingZope.html .

5.5.2 zendmd

zendmd is a command line tool that provides an interpreted Python environment that is already primed with information about the Zenoss dmd (dmd = Device Management Database). The tool should be run as the *zenoss* Operating System user. As with the ZMI, *zendmd* is a powerful tool with possibilities for serious breakage.

The utility has a very helpful tab-completion mechanism when you cannot remember exact method names. It also has command recall available on the up arrow key.

There are lots of *zendmd* tips on the Zenoss wiki - start at http://wiki.zenoss.org/ZenDMD_Tips and check the Tips category - http://wiki.zenoss.org/Category:Tips .

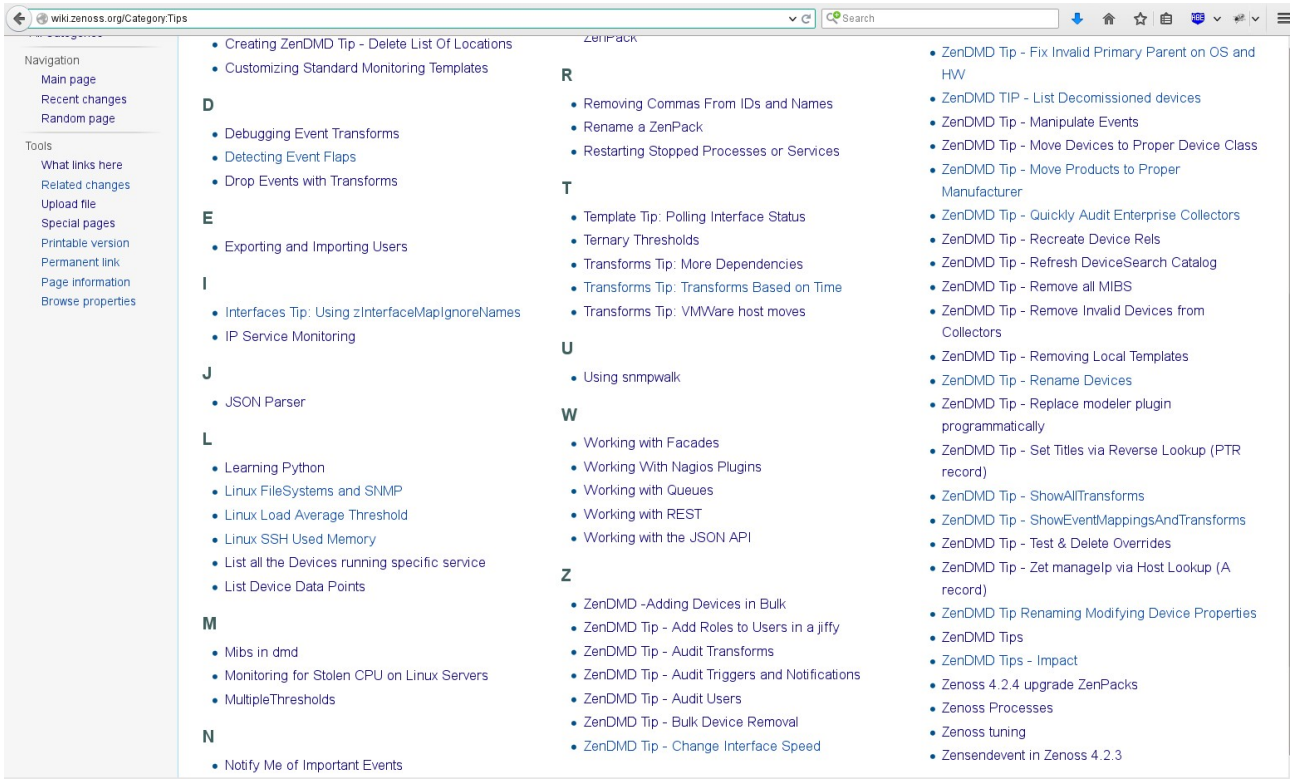


Figure 45: Zenoss wiki - lots of zendmd samples in the Tips category

Use *zendmd* to explore the attributes and methods of a device.

```
[zenoss@zen42 ZenModel]$ zendmd
Welcome to the Zenoss dmd command shell!
'dmd' is bound to the DataRoot. 'zhelp()' to get a list of commands.
Use TAB-TAB to see a list of zendmd related commands.
Tab completion also works for objects -- hit tab after an object name and
'.'
```

```
 (eg dmd. + tab-key).
>>> d=find('zen42.class.example.org')
>>> d
<Device at /zport/dmd/Devices/Server/Linux/devices/zen42.class.example.org>
>>> d.id
'zen42.class.example.org'
>>> d.manageIp
'192.168.10.42'
>>> d.getRRDTemplates()
[<RRDTemplate at /zport/dmd/Devices/Server/Linux/rrdTemplates/test>,
<RRDTemplate at /zport/dmd/Devices/Server/Linux/rrdTemplates/Device>,
<RRDTemplate at /zport/dmd/Devices/Server/rrdTemplates/LDAPMonitor>,
<RRDTemplate at
/zport/dmd/Devices/Server/Linux/rrdTemplates/ProcessCheck_firefox>,
<RRDTemplate at /zport/dmd/Devices/Server/rrdTemplates/PyTest5>,
<RRDTemplate at /zport/dmd/Devices/rrdTemplates/SnmpPacketsInOut>,
<RRDTemplate at
/zport/dmd/Devices/Server/Linux/rrdTemplates/sql_script_test>, <RRDTemplate
at
/zport/dmd/Devices/Server/Linux/devices/zen42.class.example.org/SqlDatasour
ceTest1>, <RRDTemplate at
/zport/dmd/Devices/Server/Linux/devices/zen42.class.example.org/test1>,
<RRDTemplate at /zport/dmd/Devices/Server/Linux/rrdTemplates/UCD_swap>,
<RRDTemplate at
/zport/dmd/Devices/Server/Linux/rrdTemplates/uptimeTwisted>]
>>> d.getDeviceClassPath()
'/Server/Linux'
>>> d.getDeviceUrl()
'/zport/dmd/Devices/Server/Linux/devices/zen42.class.example.org'
>>> d.getProperty('zSnmpCommunity')
'public'
>>>
```

Start by getting a variable that represents the device object; either the id or the IP address can be used as the parameter to *find*:

```
d = find('zen42.class.example.org')
```

Note that *find* is a shortcut for '*dmd.Devices.findDevice()*' or '*app.zport.dmd.Devices.findDevice()*'

Note in the examples above that attributes do not have *()* whereas methods always have *()*, and may have "real" parameters.

It is also possible to explore the contained objects and relationships of a device.

```
>>> for i in d.os.interfaces():
...   print i, i.id
...
<IpInterface at eth0> eth0
<IpInterface at lo> lo
>>>
```

The example above drills into the *os* **object** of the device and then iterates through the *interfaces* **relationship**, printing the interface object instance and the interface object instance id.

To see all the attributes of an object, use the built-in method of *__dict__* to create a dictionary of the object structure, and then iterate through the items of the dictionary. The "f*or k, v in* " trick iterates the pairs of **key** and **value** through the dictionary.

```
>>> for i in d.os.interfaces():
...  for k, v in i.__dict__.items():
...   print i.id, k, v
...
 ...
eth0 adminStatus 1
eth0 links <ToManyRelationship at links>
eth0 _ipAddresses ['fe80::20c:29ff:feb5:8a24/24']
eth0 createdTime 2012/08/28 19:30:39.730741 GMT+1
eth0 dependencies <ToManyRelationship at dependencies>
eth0 iproutes <ToManyRelationship at iproutes>
eth0 speed 1000000000
eth0 id eth0
eth0 macaddress 00:0C:29:B5:8A:24
eth0 interfaceName eth0
eth0 title eth0
eth0 duplex 3
eth0 os <ToOneRelationship at os>
eth0 __primary_parent__ <ToManyContRelationship at interfaces>
eth0 ethernetCsmacd_64 <RRDTemplate at ethernetCsmacd_64>
eth0 _propertyValues {}
eth0 mtu 1500
eth0 operStatus 1
eth0 ipaddresses <ToManyRelationship at ipaddresses>
eth0 _guid 83777b62-e42c-4685-bb5a-18f1619d198e
eth0 ifindex 2
eth0 _objects ({'meta_type': 'ToManyRelationship', 'id': 'dependencies'},
{'meta_type': 'ToManyRelationship', 'id': 'dependents'}, {'meta_type':
'ToManyRelationship', 'id': 'links'}, {'meta_type': 'ToOneRelationship',
'id': 'os'}, {'meta_type': 'ToManyRelationship', 'id': 'ipaddresses'},
{'meta_type': 'ToManyRelationship', 'id': 'iproutes'}, {'meta_type':
'RRDTemplate', 'id': 'ethernetCsmacd_64'})
eth0 dependents <ToManyRelationship at dependents>
eth0 type ethernetCsmacd_64
lo adminStatus 1
lo links <ToManyRelationship at links>
```

Note that relationships are, of course, attributes of the object and this is made clear in the output.

Some attributes may themselves be objects such as:

```
ethernetCsmacd_64 <RRDTemplate at ethernetCsmacd_64>
```

which is a local RRD template object that has been created for this interface on this device.

Where you need to explore an object that has one or more (potentially large) dictionaries, the pretty print, *pprint* class in the standard Python *pprint* module is useful. Since it is not part of the Zenoss Python environment, it needs importing into *zendmd*.

```
>>> from pprint import pprint
>>> for i in d.os.interfaces():
...  print i.id
...  for k,v in i.__dict__.items():
...   if k == '_objects':
...    pprint(v)
```

```
...   else:
...    print k,v
...
eth0
adminStatus 1
links <ToManyRelationship at links>
_ipAddresses ['fe80::20c:29ff:feb5:8a24/24']
createdTime 2012/08/28 19:30:39.730741 GMT+1
dependencies <ToManyRelationship at dependencies>
iproutes <ToManyRelationship at iproutes>
speed 1000000000
id eth0
macaddress 00:0C:29:B5:8A:24
interfaceName eth0
title eth0
duplex 3
os <ToOneRelationship at os>
__primary_parent__ <ToManyContRelationship at interfaces>
ethernetCsmacd_64 <RRDTemplate at ethernetCsmacd_64>
_propertyValues {}
mtu 1500
operStatus 1
ipaddresses <ToManyRelationship at ipaddresses>
_guid 83777b62-e42c-4685-bb5a-18f1619d198e
ifindex 2
({'id': 'dependencies', 'meta_type': 'ToManyRelationship'},
 {'id': 'dependents', 'meta_type': 'ToManyRelationship'},
 {'id': 'links', 'meta_type': 'ToManyRelationship'},
 {'id': 'os', 'meta_type': 'ToOneRelationship'},
 {'id': 'ipaddresses', 'meta_type': 'ToManyRelationship'},
 {'id': 'iproutes', 'meta_type': 'ToManyRelationship'},
 {'id': 'ethernetCsmacd_64', 'meta_type': 'RRDTemplate'})
dependents <ToManyRelationship at dependents>
type ethernetCsmacd_64
```

The example above checks whether the dictionary key is called _objects and, if so, uses *pprint* to output it. This makes dictionaries far more readable.

zendmd commands are working on your live ZODB database and you may occasionally get access clashes with what the rest of the Zenoss code is doing with the database. Generally it is fairly safe to inspect data. If you change data, changes will be reflected within the same *zendmd* environment but will **not** be seen in the GUI. To commit changes made in a *zendmd* session, use:

```
commit()
```

If changes have been made by Zenoss to the ZODB, they can be updated into the running *zendmd* session with:

```
sync()
```

Exit a *zendmd* session with:

```
quit()
```

6.0 Developing complex ZenPacks

A complex ZenPack typically involves writing Python code. It may also require JavaScript, HTML / XML, bash scripts and the pseudo-code used with the zenpacklib utility.

ZenPack development should always be performed as the *zenoss* user.

6.1 Planning considerations

6.1.1 Names and naming convention

It is essential to plan the pieces of code required for a ZenPack and clearly document the names that will be used, as many elements are referenced in other elements. Note that all names are case-sensitive.

- ZenPack name
- Python object classes for devices
- Python object classes for components
- Names of classes representing interfaces and infos
- Attribute names
- Names of relationships
- Methods and their parameters
- Names and type of any zProperties
- Names of modeler plugins
- JavaScript directory hierarchy and file names

By convention, class names start with an upper-case letter and may have mixed case throughout eg. *FileSystem*.

Again by convention, relationship names start with a lower-case letter and may have mixed case eg. *filesystems*, *maintenanceWindows*. Relationship names typically are plural for *ToMany* relationships and singular for *ToOne* relationships:

```
("deviceClass", ToOne(ToManyCont, "Products.ZenModel.DeviceClass","devices"))
("devices", ToManyCont(ToOne,"Products.ZenModel.Device","deviceClass"))
```

Note that, by convention, the relationship name tends to reflect what is being related **to**.

6.1.2 ZenPack prerequisites and other considerations

ZenPacks may require prerequisites and co-requisites, both Zenoss code and external. Typical examples may be:

- Prerequisite ZenPack(s) eg. the PythonCollector ZenPack if you are going to create a Python datasource.
- Specific SNMP MIBs if you are going to manage a new device type with SNMP. Strictly, MIBs only need to be imported into Zenoss in order to decode TRAPs or SNMP V2 NOTIFICATIONS.

- A MIB Browser is extremely useful when building SNMP-based ZenPacks. Consider installing *ZenPacks.community.mib_browser;* it needs some minor code hacks but is invaluable when building ZenPacks that use SNMP. Access is at http://wiki.zenoss.org/ZenPack:MIB_Browser .

- If you are considering a command-based ZenPack, check to see whether there are any existing Nagios plugins that already do what is required. It is generally trivial to drive Nagios plugins from a *COMMAND* datasource.

- If you are contemplating a COMMAND-based ZenPack, consider the scaling implications. A command run against a few devices with a few components will be fine. If there are hundreds of devices each with lots of tens of components, COMMANDS will not work well and Python should be considered. This will depend on the commands being run, how any collectors are balanced, and what the commands are doing (for example, how long do the commands take to run?). Generally one should use COMMAND datasources for running commands on the devices or collectors that are proprietary or pre-compiled and therefore are difficult to use directly from python.

- Are any external packages required? If zenpacklib is to be used then *PyYAML* must be installed (or already available in the Zenoss environment).

- Check http://wiki.zenoss.org/ZenPack_Catalog in case a ZenPack already exists that will do the job or at least be a good starting point.

- Ensure you have a test Zenoss installation. You may wish to run a minimal environment as described in Chapter 3.1.

- Ensure there is access to at least one test device, preferably several with different characteristics.

- Consider firewall implications. If target devices are likely to be behind firewalls then understand what TCP/UDP ports need to be open for the ZenPack to work, and document them.

- Take a backup, preferably a system backup / VMware snapshot **and** a *zenbackup*, before installing any new ZenPack. With Zenoss 5, take a snapshot.

6.2 zenpacklib

In mid-2015, Zenoss delivered zenpacklib which is a package designed to take much of the coding effort out of ZenPacks. The area where it provides most benefit, is in largely eliminating the need for JavaScript, info.py, interfaces.py and configure.zcml. Its documentation pages can be found at http://zenpacklib.zenoss.com/en/latest .

The difficulty with zenpacklib is that it does not do everything; for example, it cannot simplify writing modeler plugins or custom datasources. In order to write such code it is necessary to **really** understand the constructs that zenpacklib simplifies.

zenpacklib.py is simply a Python program that interprets an input file called *zenpack.yaml*. The input file uses pseudo-code to define zProperties, Zenoss device classes, device object classes. component classes, relationships and monitoring templates. Python code is constructed and implemented **in memory** to represent these objects.

The JavaScript code that would normally be necessary to implement the GUI elements of the new objects, along with the associated info and interface classes, are also automatically implemented **in memory**. There are no files to inspect, other than *zenpack.yaml*.

6.3 Developing Python code

Most elements of a ZenPack are written in Python.

There are good Python references around; the O'Reilly books are always a good start:

- “Learning Python” by Mark Lutz
- “Python Pocket Reference” by Mark Lutz
- Twisted - Network Programming Essentials” by Jessica McKellar & Abe Fettig

The online Python reference documentation is extremely useful - start at https://docs.python.org/2/library/index.html .

Zenoss 3 requires Python 2.6; Zenoss 4 and 5 require Python 2.7.

Note that Python 3 is **not** supported by Zenoss (any version).

If Python code is to be written, be aware that Python is very white-space sensitive. Program constructs such as *if-then-else*, *while* loops, *for* loops and many other coding elements depend on white space indentation (and the same number of spaces for the same level of the construct). If testing Python with the *zendmd* utility, the same white-space rules must be obeyed.

6.3.1 pyflakes

pyflakes is a Python library which checks Python source files for errors; more details at https://pypi.python.org/pypi/pyflakes . For those who still use *vi* as their workhorse editor, the *pyflakes-vim* package, obtainable from http://www.vim.org/scripts/script.php?script_id=2441 is very easy to use. Simply:

- Download the zip package
- Create a *.vim* (note the leading dot) directory in your home directory - probably the home directory for the zenoss user
- Unzip the *pyflakes-vim* package into this *.vim* directory
  - A *ftplugin*/*python*/*pyflakes*/*pyflakes* directory hierarchy is created which includes the pyflakes library - no need to install pyflakes separately
- Ensure that the user's *.vimrc* includes

```
filetype plugin indent on
```

- A sample *.vimrc* might be:

```
" Double-quote is comment
" "set bg=dark
" "  :set paste and :set nopaste
" "  vi -R is view but reads rc file
"set tabstop=4

set shiftwidth=4
set expandtab
```

```
set ruler
filetype on
filetype plugin indent on
"highlight SpellBad term=reverse ctermbg=1
highlight clear SpellBad
highlight SpellBad term=standout ctermfg=1 term=underline cterm=underline
```

The result is that syntax checking is performed automatically as you edit a Python document. This saves a huge amount of time, detecting syntax errors without going around a process of reinstalling code and stopping / starting daemons.

In Figure 46 a missing colon at the end of line 22 is detected. Error lines are underlined in red and a message is given at the bottom of the screen.

Figure 46: Example of pyflakes-vim detecting missing colon in Python file

6.3.2 pep8

pep8 is a tool to check your Python code against some of the style conventions in PEP 8. See https://pypi.python.org/pypi/pep8 for more information.

6.4 Developing GUI code

If a ZenPack implements GUI modifications, it may require JavaScript code. In many cases, using the zenpacklib tool avoids this, but it cannot deliver all GUI features.

If there is a real need to modify older ZenPacks that use page templates, then skills related to XML will be required.

6.5 Useful tricks for ZenPack developers

1. When you need to find sample code or find a particular attribute or method in standard Zenoss code, a combination of Unix utilities *find* and *grep* is enormously useful.

```
cd $ZENHOME/Products
grep setHWProductKey `find . -name "*.py" `
```

a. Note the “back ticks” around the *find* to run a command.

b. You need double-quotes around “*.py”.

c. You can often reduce the effort of the *find* if you have a good idea where the code may be; as a general rule-of-thumb:

- ZenModel contains most python classes (Device, components)
- ZenEvents contains most code related to events
- DataCollector contains most modelers
- Zuul contains default interfaces, info, routers and facades
- ZenCollector is the base collector daemon code
- ZenUI3 contains UI code and JavaScript

2. *vi* tricks

a. If you are cutting code from one file and pasting into another ensure you use *:set paste* or the white space will cause you pain.

b. *:se sw=4* sets the shift width to 4 spaces. Any code block in the file, terminated by a blank line, can be moved in by the shift width by positioning on the first line and using *>}* . A code block is moved out by the shift width with *<}*.

3. *zendmd* is an excellent tool for trying out small pieces of Python code and for exploring the attributes and relationships of an object.

7.0 Anatomy of a ZenPack

7.1 Basic principles

Before discussing ZenPack code, let's get some basic principles straight first.

Zenoss documentation is apt to be a little imprecise sometimes in its terminology and uses different words to mean the same thing. There are two very different concepts to do with collecting data.

Configuration data is typically polled for every 12 hours and is held in the Zope Object Database (ZODB).

Performance data is typically polled for every 5 minutes and is held in Round Robin Database (RRD) files (prior to Zenoss 5) from where it can be graphed. With Zenoss 5, performance data is held in the OpenTSDB subsystem.


Configuration data and performance data are very different.

7.1.1 Configuration data, modeler plugins and the zenmodeler daemon

Configuration data is polled for by the **zenmodeler** daemon, using **modeler plugins,** also sometimes called **collector plugins**.

The purpose of a modeler plugin is to **map** collected data into the attributes of Zenoss objects.

Lots of plugins are provided as standard with Zenoss under *$ZENHOME/Products/DataCollector/plugins/zenoss* with separate subdirectories for:

- cmd
- nmap
- portscan
- python
- snmp

Don't be fooled by the directory path containing "*DataCollector*" - these are configuration modeler plugins used by the *zenmodeler* daemon and nothing to do with the collection of performance data that typically is collected by the *zenperfsnmp*, *zencommand* or *zenpython* daemons.

Any device or device class can have several modeler plugins assigned to it. This is configured from the left-hand *Modeler Plugins* menu of a device's *Detail* page or, for a device class, follow the *DETAILS* link at the top of the left-hand menu for the equivalent *Modeler Plugins* option. Available modelers are shown in the left-hand window.

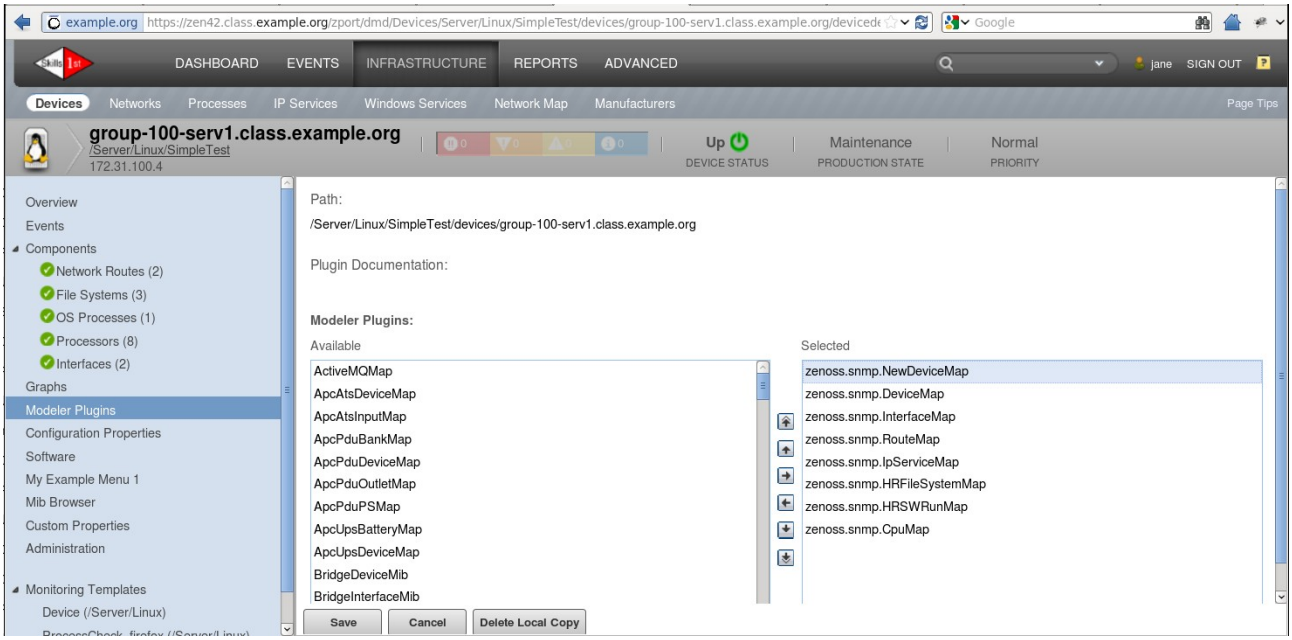


Figure 47: The Modeler Plugins dialogue for a specific device

Note that, through the device class hierarchy, plugins are inherited down the hierarchy but can be overridden at any level. Figure 47 shows the "Delete Local Copy" button as active for the device *group-100-serv1.class.example.org* indicating that the class-inherited plugins **have** been overridden.

Prior to Zenoss 4, another way to achieve exactly the same effect was to go to the device class or individual device's *Configuration Properties* menu and click on the *Edit* button beside *zCollectorPlugins*.

This facility has now been removed from the *Configuration Properties* dialogue but the actual *zCollectorPlugins* property is unchanged and can be seen with *zendmd*:

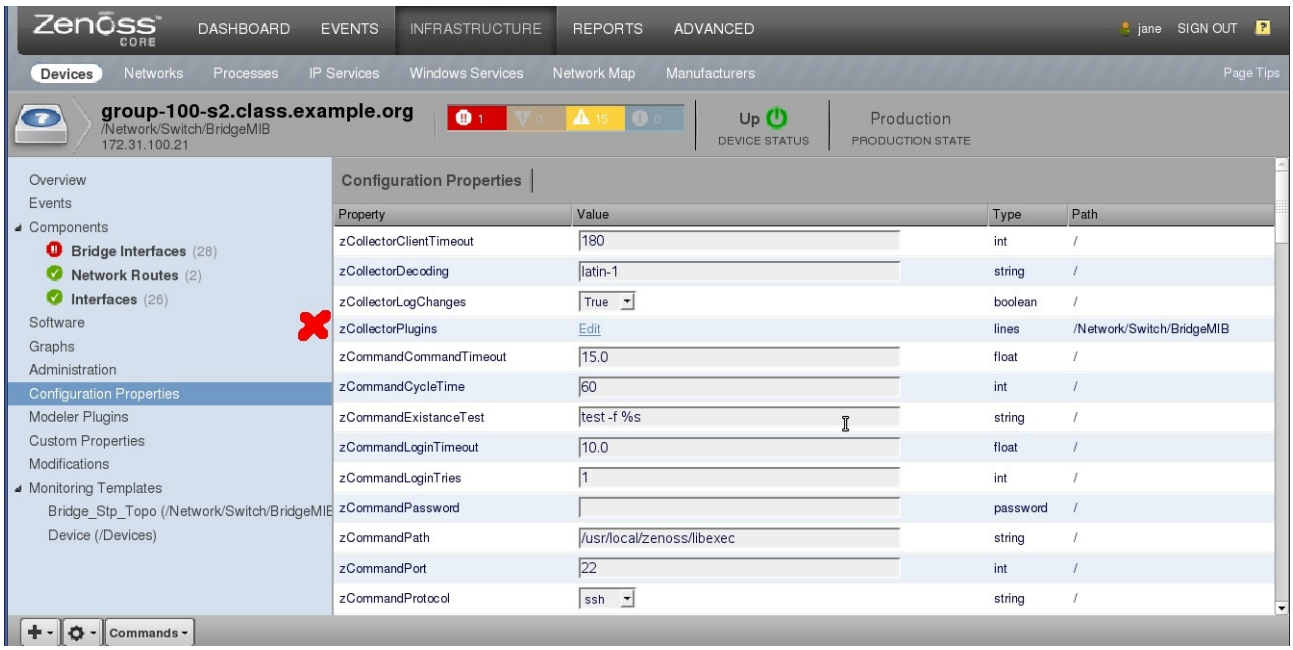


Figure 48: Modify the zCollectorPlugins zProperty to activate modeler plugins - Zenoss 3

```
>>> d=find('group-100-serv1.class.example.org')
>>> d.zCollectorPlugins
['zenoss.snmp.NewDeviceMap', 'zenoss.snmp.DeviceMap',
'zenoss.snmp.InterfaceMap', 'zenoss.snmp.RouteMap',
'zenoss.snmp.IpServiceMap', 'zenoss.snmp.HRFileSystemMap',
'zenoss.snmp.HRSWRunMap', 'zenoss.snmp.CpuMap']
>>>
```

Modelers do **not** do fundamental device discovery, that is the job of the **zendisc** daemon; however, once basic discovery has been performed, usually by ping, and basic data has been added into the ZODB for the device, it is then the job of *zenmodeler* to discover other configuration data. This may be basic SNMP agent information used to lookup Hardware and OS, Manufacturer and Model information, along with the SNMP sysContact and sysLocation data. *zenoss.snmp.NewDeviceMap* and *zenoss.snmp.DeviceMap* are the modelers that achieve this.

A common requirement for the ZenPack developer is to write new modeler plugins to discover new attributes of a device and / or discover new components. Standard modelers discover interfaces, IP routes, IP services, filesystems, processes (*zenoss.snmp.HRSWRunMap*) and cpu components.

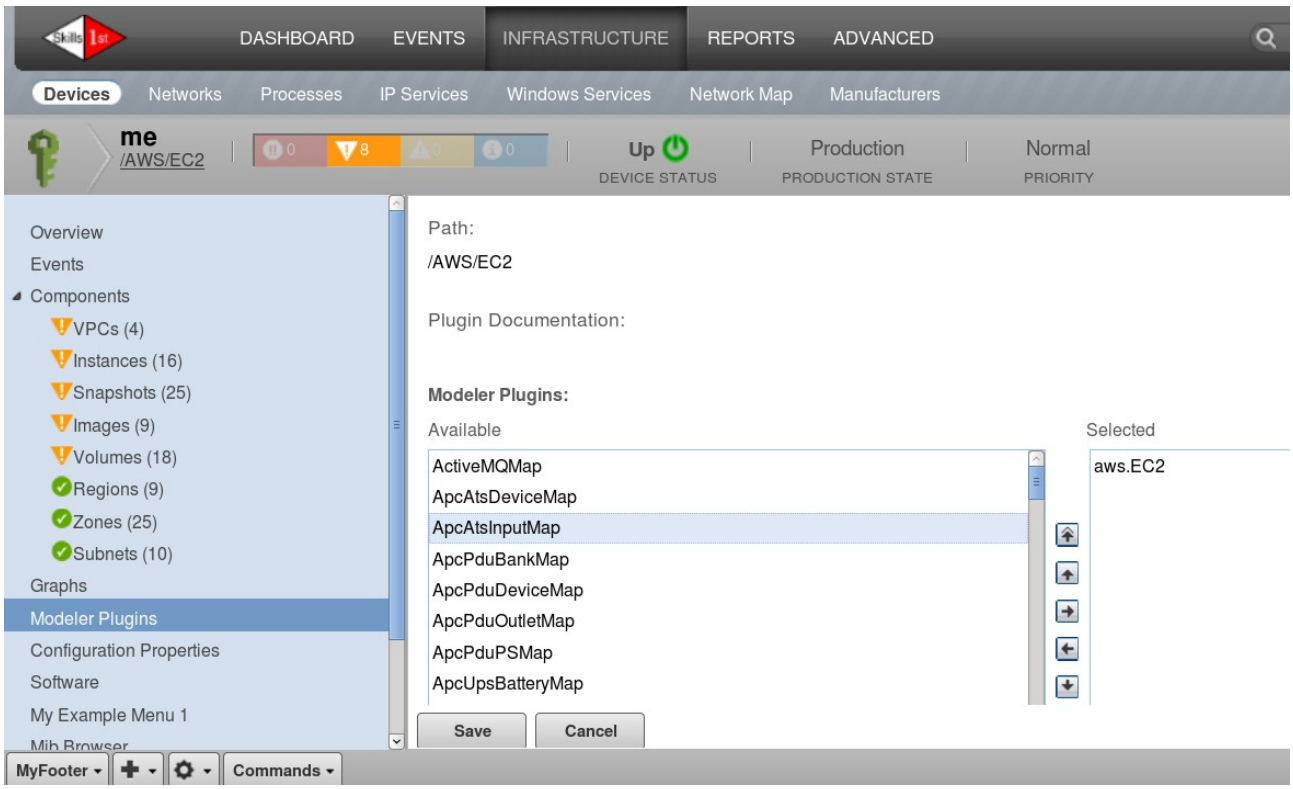


Figure 49: AWS device showing the aws.EC2 modeler and Amazon EWS components

ZenPacks.zenoss.AWS is an example of a free ZenPack available from Zenoss to manage Amazon Cloud devices. It provides a single new modeler plugin, *aws.EC2*, which discovers several new components, seen in the left-hand menu of the *me* device in Figure 49.

When a new ZenPack is created it contains a directory hierarchy for modeler plugins with *ExampleSNMP.py.example* under *modeler/plugins/community/snmp*. It only gathers device configuration data, not component data, but it has a very interesting demonstration of applying scalar data to different, existing components:

- *memTotalSwap* is applied to the *os* component
- *memTotalReal* is applied to the *hw* component

where *TotalSwap* and *TotalMemory* are standard attributes defined on the standard *os* and *hw* components, respectively. The data that is gathered is:

```
snmpGetMap = GetMap({
   '.1.3.6.1.4.1.2021.4.3.0': 'memTotalSwap',
   '.1.3.6.1.4.1.2021.4.5.0': 'memTotalReal',
   })
```

This is applied with:

```
maps = []
maps.append(ObjectMap({
  'totalMemory': getdata['memTotalReal'] * 1024},
  compname='hw'))

maps.append(ObjectMap({
  'totalSwap': getdata['memTotalSwap'] * 1024},
  compname='os'))
```

The standard *NewDeviceMap* modeler in *$ZENHOME/Products/DataCollector/plugins/zenoss/snmp* extends this idea further by assigning data using standard Zenoss setter methods. Thus the data gathered is:

```
snmpGetMap = GetMap({
      '.1.3.6.1.2.1.1.1.0' : 'snmpDescr',
      '.1.3.6.1.2.1.1.2.0' : 'snmpOid',
      })
```

The data is used in the *process* method of the modeler with:

```
om.setHWProductKey = MultiArgs(om.snmpOid, manufacturer)
```

The *setHWProductKey* method can be found as a method for a Device in *$ZENHOME/Products/ZenModel/Device.py* and the *snmpOid* data value gathered is used as a parameter to this method. See the Zenoss Wiki – Diving into the Device Model at http://community.zenoss.org/docs/DOC-2350 for more information on both device setters and properties.

7.1.2 Performance data and monitoring templates

Specify collection of **performance** data using Zenoss **monitoring templates**. As with modeler plugins, templates can be assigned either to a device class hierarchy or to a specific device but the definition of these templates, the RRD databases that contain the data and the daemons that collect the data are entirely separate from the configuration data collection mechanism.

Zenoss provides the **zenperfsnmp** daemon and the **zencommand** daemon, among others. Each works with a Zenoss-supplied datasource, specific to the daemon (*SNMP* and *COMMAND*); look in *$ZENHOME/Products/ /Zuul/infos/template.py* and $*ZENHOME/Products/ /Zuul/interfaces/template.py* for some information on these. The daemons themselves, *zenperfsnmp.py* and *zencommand.py*, are in *$ZENHOME/Products/ZenRRD*.

If you can access performance data using either SNMP or ssh then, typically, there is no need to write new code to collect performance data. You may use a new ZenPack to port new, GUI-built templates from one Zenoss to another, but you don't need to write code. Where code **is** required is if you need a new type of **datasource**.

A datasource defines:

- What performance data to collect, including the GUI dialogue
- How to collect it
- How to convert raw collected data into **datapoints** specified in a performance monitoring template. This may be a separate **parser**.
- How to report errors with the datasource

Standard datasource definitions can be found in *$ZENHOME/Products/ZenModel*:

- BasicDataSource.py
- RRDDataSource.py

- PingDataSource.py

There are several examples of new datasources in the standard, Zenoss-supplied ZenPacks; for example *ZenPacks.zenoss.FtpMonitor* provides the *FtpMonitor* datasource type. If a ZenPack provides a new datasource, it should go in the *datasources* directory.

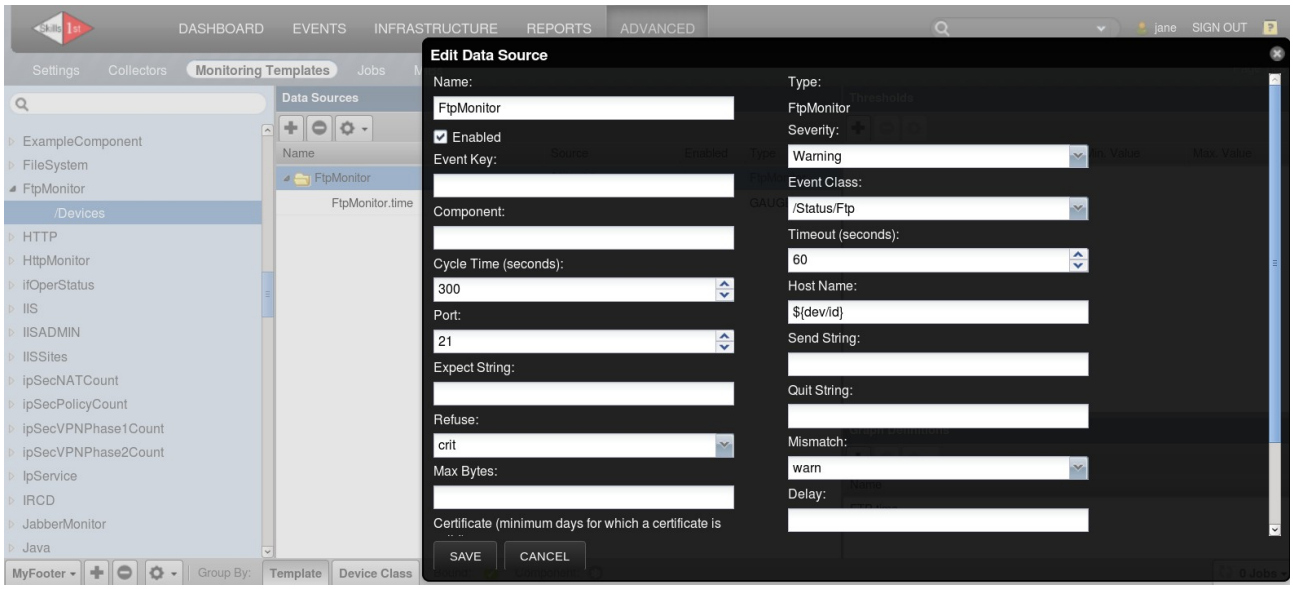


Figure 50: FtpMonitor datasource from ZenPacks.zenoss.FtpMonitor

Note in Figure 50 that there are several fields, specific to FTP communications, that do not appear in *SNMP* or *COMMAND* datasources. These are in addition to standard, inherited fields like Severity, Event Key, Component, Cycle Time and Event Class.

More recently, Zenoss has issued the *ZenPacks.zenoss.PythonCollector* Zenpack which can provide a much more efficient and flexible alternative to collecting data than with *zencommand*. It provides a new ***zenpython*** daemon and a *PythonDataSource.* The base Zenoss code now has a Python modeler plugin.

Many newer, Zenoss-provided ZenPacks demand the PythonCollector ZenPack as a prerequisite.

7.2 New objects in ZenPacks

Probably the most common reason for creating a new, complex ZenPack is to support new device types and components. The AWS ZenPack in Figure 49 is a good example. New attributes are created for the new device type and component object classes are created for Instances, VPCs, Snapshots, Zones, Volumes, Regions and Subnets.

It is important to understand the difference between the Zenoss device class hierarchy seen in the Infrastructure menus - */Server*, */Server/Linux* and so on - and the concept of new Python object classes to represent devices.

ZenPacks.zenoss.AWS defines a new Python object class to represent Amazon AWS accounts.

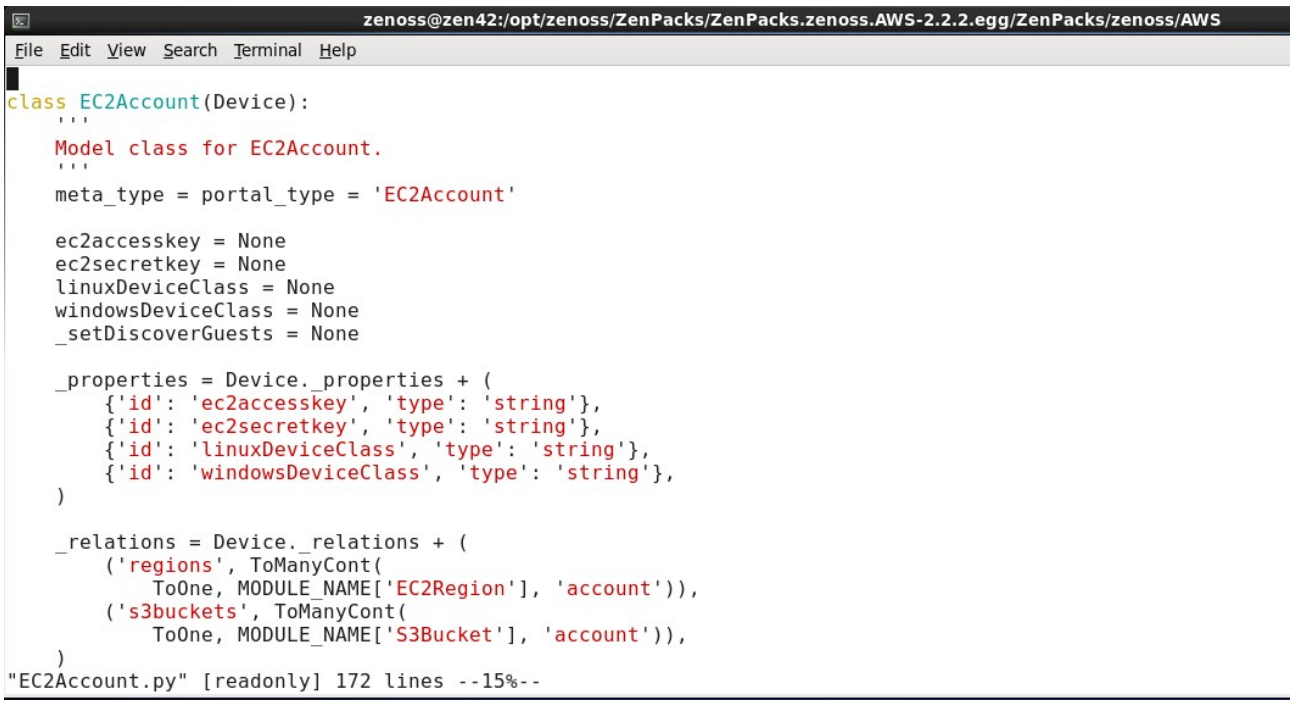


Figure 51: Definition of EC2Account object class to represent AWS devices

Note in Figure 51 that the new *EC2Account* class inherits from ***Device*** and then defines four new properties for the device and two new relationships.

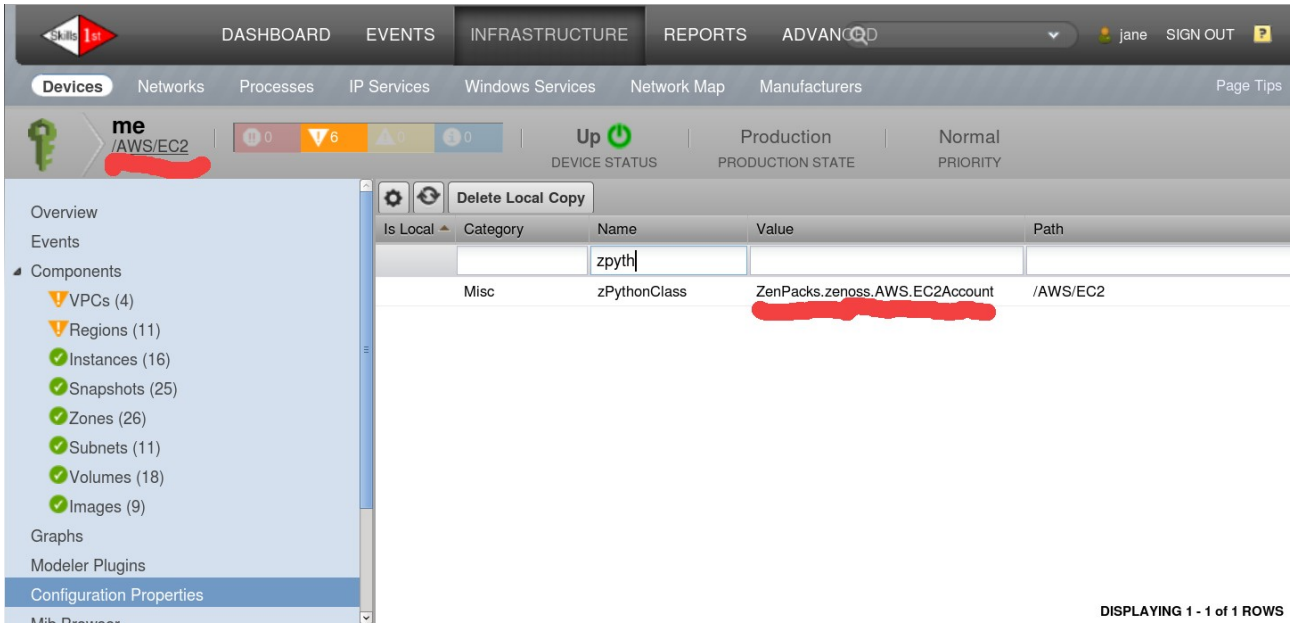


*Figure 52: zPythonClass zProperty for device instance **me** in device class /AWS/EC2*

To ensure that a device **instance** (the device called *me* in the screenshot) has the correct Python object class, the *zPythonClass* **zProperty** must be set at the device class, */AWS/EC2*.

7.3 GUI code

If new components are created, code is needed to display them and their attributes. If zenpacklib is used, this code is created automatically in memory; otherwise the files have to be manually constructed.

7.3.1 Page Template files and skins directories in older Zenoss

Originally in Zenoss 1 and 2, GUI code used a mixture of:

- HyperText Markup Language (HTML)
- Cascading Style Sheets (CSS)
- Zope 2, Zope Page Templates (ZPT) and the Template Attribute Language (TAL)
- ZPT and Macro Expansion for TAL (METAL)
- JavaScript / Asynchronous JavaScript And XML (AJAX))
- Yahoo User Interface (YUI) Library and Mochikit

Code to define the GUI was typically kept under the ZenPack's base directory, under a directory hierarchy of:

```
skins/<ZenPack name>/
```

File extensions were *.pt*, representing **Page Templates**. Gradually this mechanism is being phased out in favour of **JavaScript** but there are still some page template files around in current ZenPacks and in the base Zenoss code - look in */opt/zenoss/Products/ZenModel/skins/zenmodel*. For some examples and explanations of Page Template files, see “Creating Zenoss ZenPacks” at http://www.skills-1st.co.uk/papers/jane/zenpacks/zenpacks.pdf (the original version).

7.3.2 JavaScript code to define GUI elements

The current method of defining GUI code is with JavaScript. Standard Zenoss JavaScript code resides under *$ZENHOME/Products/ZenUI3/browser/resources/js/zenoss*. The Device Detail page is defined in *$ZENHOME/Products/ZenUI3/browser/resources/js/zenoss/devdetail.js* – this presents the overall view for a device. Component detail display is handled by *$ZENHOME/Products/ZenUI3/browser/resources/js/zenoss/ComponentPanel.js* .

A ZenPack should mirror the *browser/resources/js* hierarchy under its base directory. Note that some older ZenPacks skip the *browser* directory and just have a *resources* directory.

TODO: StevePC: should probably mention Sencha’s ExtJS, the JS library used to construct the majority of the v4 and v5 UI

Some examples and explanation is provided in “Creating Zenoss ZenPacks for Zenoss 3” at http://www.skills-1st.co.uk/papers/jane/zenpacks/zenpacks3.pdf .

The usual method for creating these *.js* files was to copy someone else's and modify to suit!

7.3.3 configure.zcml, infos and interfaces

Three files are required in the base directory of a ZenPack to link between the Python objects and the JavaScript GUI code:

- info.py
- interfaces.py
- configure.zcml

The ***info.py*** file abstracts object attribute information saved in the Zope Object Database (ZODB), that will be displayed to the user. It also allows code to be written for display that it is not part of the class definition. Note that the file must have this exact name.

zenoss@zen42:/code/ZenPacks

File Edit View Search Terminal Help

```
##############################################################################

__doc__="""info.py

Representation of Bridge components.

$Id: info.py,v 1.2 2010/12/14 20:45:46 jc Exp $"""

__version__ = "$Revision: 1.4 $"[11:-2]

from zope.interface import implements
from Products.Zuul.infos import ProxyProperty
from Products.Zuul.infos.component import ComponentInfo
from Products.Zuul.decorators import info
#from Products.ZenUtils.Utils import convToUnits
from ZenPacks.skills1st.bridge import interfaces


class BridgeInterfaceInfo(ComponentInfo):
    implements(interfaces.IBridgeInterfaceInfo)

    Port = ProxyProperty("Port")
    RemoteAddress = ProxyProperty("RemoteAddress")
    RemoteInterface = ProxyProperty("RemoteInterface")
    RemoteDevice = ProxyProperty("RemoteDevice")
    PortStatus = ProxyProperty("PortStatus")
    PortComment = ProxyProperty("PortComment")

    @property
    def RemoteInterface(self):
        return self._object.getRemoteInterfaces()

    @property
    def RemoteDevice(self):
        return self._object.getRemoteDevice()
```

"ZenPacks.skills1st.bridge/ZenPacks/skills1st/bridge/info.py" [readonly] 46 lines --97%--

Figure 53: ZenPacks.skills1st.bridge ZenPack - info.py file

In the *ZenPacks.skills1st.bridge* ZenPack, the *info.py* provides a link between the attributes defined on the *BridgeInterface* **object** and the **interface** which defines how it is displayed. The **ProxyProperty** method shuttles data - the attributes *Port, RemoteAddress, RemoteInterface, RemoteDevice, PortStatus* and *PortComment* - from the ZODB to the info object.

Note that it is the names in red in Figure 53 that need to match the object attributes defined in a *BridgeInterface.py* file.

Note that there is nothing in *info.py* describing **how** the data is displayed, just **what** is displayed.

In addition to the defined attributes, there is a requirement for other fields to be displayed in the GUI. These are the *RemoteInterface* and *RemoteDevice* methods that will actually be treated as properties, thanks to the *@property* Python decorator. The *getRemoteInterfaces()* and *getRemoteDevice()* methods are defined with the *BridgeInterface* object and attributes in *BridgeInterface.py*.

interfaces.py describes elements of how the data is displayed (and again, this filename is prescribed).

zenoss@zen42:/code/ZenPacks

File Edit View Search Terminal Help

```
##############################################################################
#
# This program is part of the Bridge Zenpack for Zenoss.
# Copyright (C) 2010 Jane Curry
#
# This program can be used under the GNU General Public License version 2
# You can find full information here: http://www.zenoss.com/oss
#
##############################################################################

__doc__="""interfaces

describes the form field to the user interface.

$Id: interfaces.py,v 1.2 2010/12/14 20:46:34 jc Exp $"""

__version__ = "$Revision: 1.4 $"[11:-2]

from Products.Zuul.interfaces import IComponentInfo
from Products.Zuul.form import schema
from Products.Zuul.utils import ZuulMessageFactory as _t


class IBridgeInterfaceInfo(IComponentInfo):
    """
Info adapter for Bridge Interface component
"""
    Port = schema.Text(title=u"Port", readonly=True, group='Details')
    RemoteAddress = schema.Text(title=u"Remote MAC", readonly=True, group='Details')
    RemoteInterface = schema.Text(title=u"Remote Interface", readonly=True,group='Details')
    RemoteDevice = schema.Text(title=u"Remote Device", readonly=True, group='Details')
    PortStatus = schema.Text(title=u"Port Status", readonly=True, group='Details')
    PortComment = schema.Text(title=u"Port Comment", group='Details')

~
~
~
"ZenPacks.skills1st.bridge/ZenPacks/skills1st/bridge/interfaces.py" [readonly] 34L, 1304C
```

Figure 54: ZenPacks.skills1st.bridge ZenPack - interfaces.py file

The elements on the left-hand side in Figure 54 (*Port, RemoteAddress* etc) **must** match the names on the left-hand side of the corresponding statements in the *info.py* file.

The interfaces file should have basic Zope schema information - formatting details describing the fields of a form for the attributes to populate. *interfaces.py* controls the fields seen in a

component's *Details* dropdown menu. Any field defined in *interfaces.py* will require a matching entry in *info.py* to tell the GUI what element to display.

For more information on Zope interfaces, see
http://docs.zope.org/zope2/zdgbook/ComponentsAndInterfaces.html

configure.zcml provides the “glue” between interfaces and JavaScript display code and this **exact** name will be searched for by the Zope mechanisms. Zope Configuration Markup Language (ZCML) is Zope 3's XML-based component configuration language for “wiring” together application policy and component registrations. It is documented at the Zope site at http://docs.zope.org/zopetoolkit/codingstyle/zcml-style.html .

The actual detailed code for displaying the details for a bridge interface, comes from a JavaScript file *bridge.js* which should be under *browser/resources/js*.

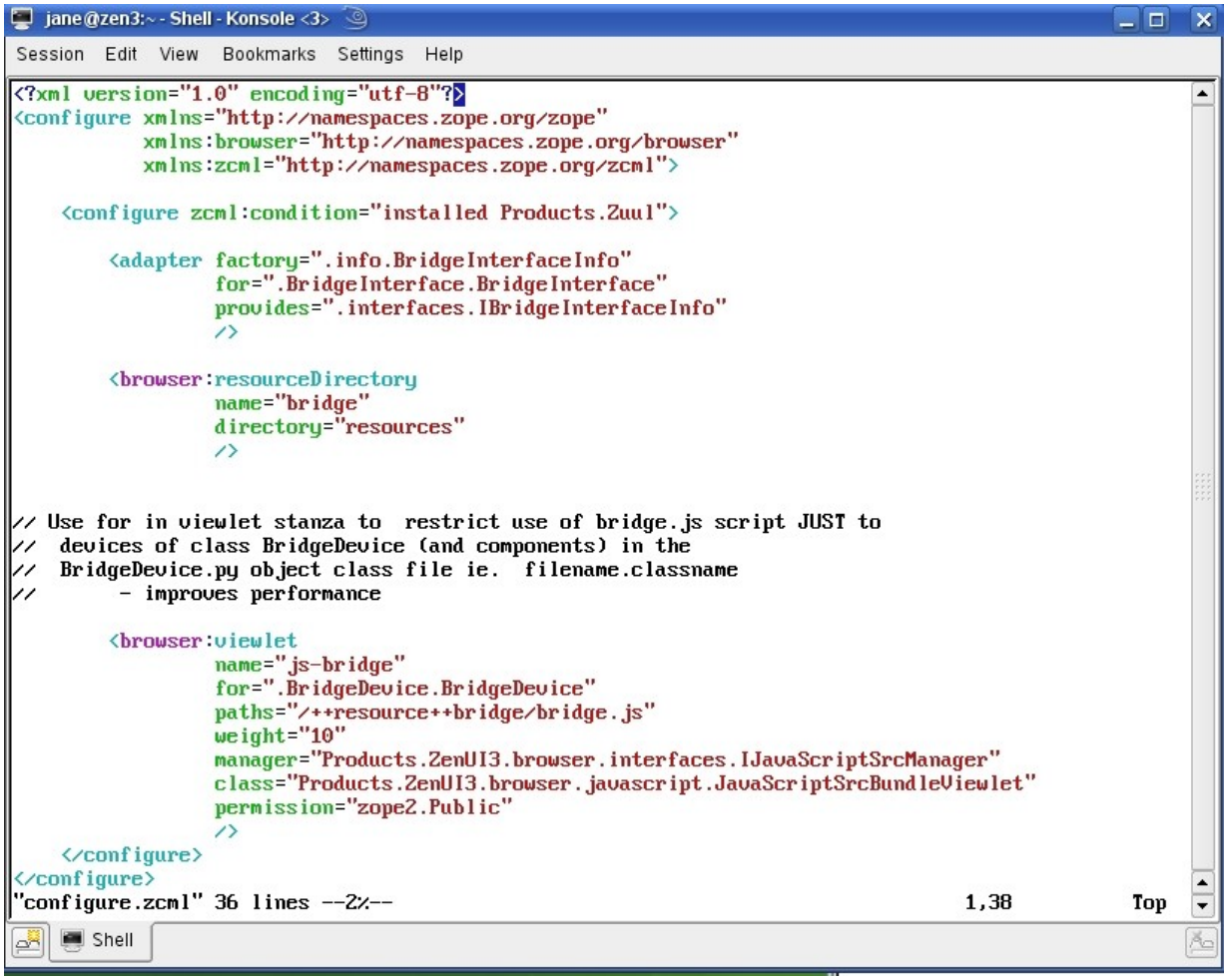


Figure 55: configure.zcml

The **adapter** stanza in *configure.zcml* links the info file and interfaces file with the device component class.

- The **factory** field must match the class defined in *info.py*
- The **provides** field must match the class defined in *interfaces.py*

- The **for** field must match the device component class, *BridgeInterface*, defined in the file *BridgeInterface.py* – hence *BridgeInterface.BridgeInterface* .

The **browser:resourceDirectory** stanza indicates where to find JavaScript files. This ZenPack is an example of earlier coding practice where there was no *browser* directory and JavaScript files were directly under a *resources* directory.

- The **name** (namespace) field can be any unique name
- The **directory** field is the subdirectory from where this *configure.zcml* resides - *resources*

The **browser:viewlet** stanza is the link to the correct JavaScript file to display elements for a particular device.

- The **name** (namespace) for this viewlet can be any unique name
- The **for** field restricts the use of this JavaScript file to the context of devices of class *BridgeDevice* in the file B*ridgeDevice.py* – the syntax is <filename without the .py>.<class name> - don't forget the leading dot! It will be available for all such devices and their components but will **not** be loaded for other device types. This can become a huge performance benefit.
- In the **paths** field

```
paths="/++resource++bridge/bridge.js"
```

  - **bridge** must match the name given to the name in the *browser:resourceDirectory* stanza
  - Thus the JavaScript file that define the page layout, *bridge.js* is under the named resources directory - *resources* - under the ZenPack's base directory; that is, .../*ZenPacks.skills1st.bridge*/*ZenPacks*/*skills1st*/*bridge*/*resources*
- The **class** field should be *Products.ZenUI3.browser.javascript.JavaScriptSrcBundleViewlet* if the paths field has one or more files listed. It must be this where the paths field has **multiple** files, space-separated; it could be *Products.ZenUI3.browser.javascript.JavaScriptSrcViewlet* for a single path file.
- The **weight** field indicates the order of multiple viewlets where 1 would be at the top and 100 would be at the bottom.
- The **permission** field is mandatory

7.4 Other elements of a ZenPack

So far, a ZenPack may have:

- New device classes
- New component classes
- Modeler plugins
- New datasources

- GUI code

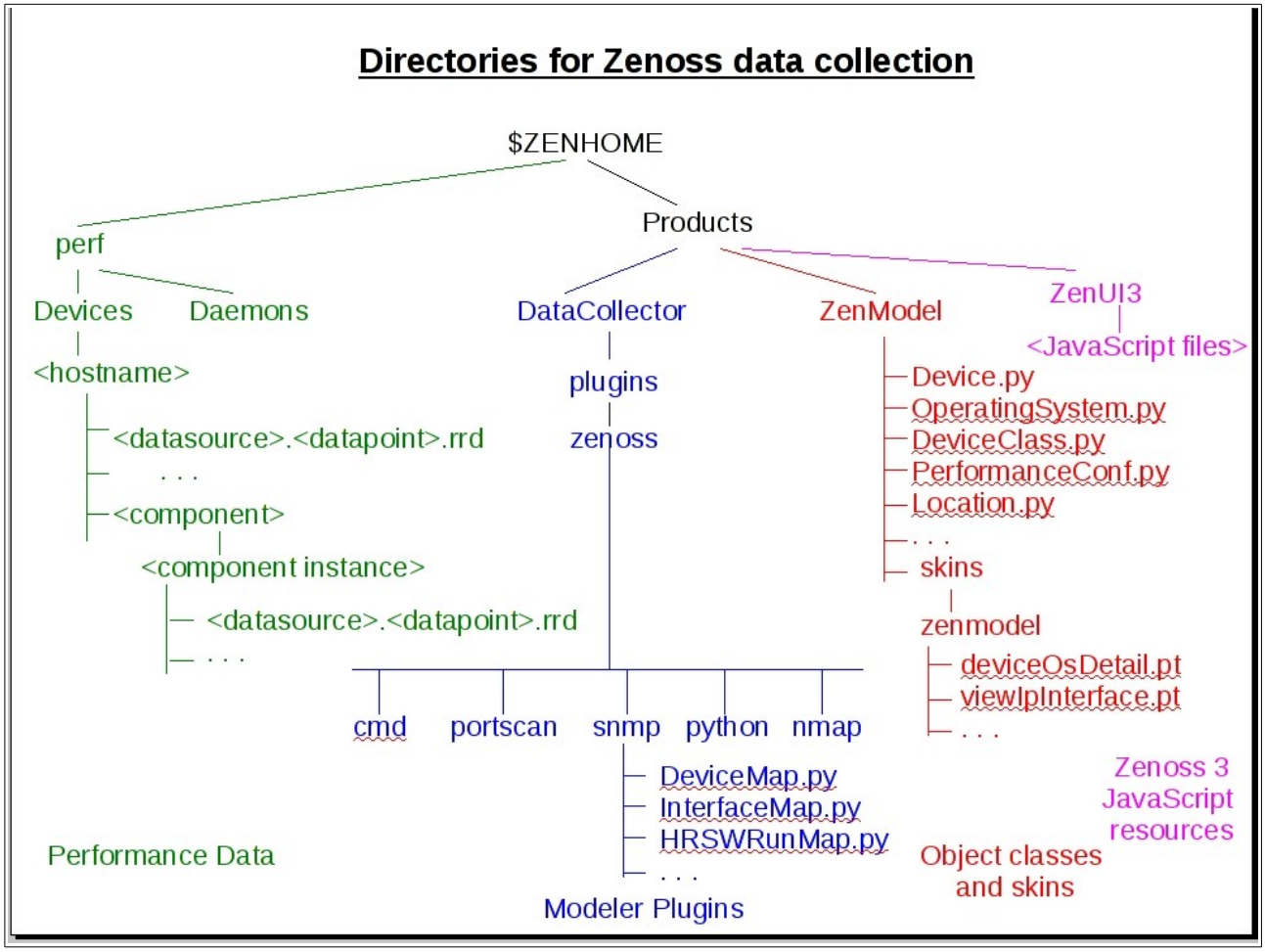


Figure 56: Directory hierarchy for Zenoss data collection

Figure 56 shows the directory structure for many of the standard elements of Zenoss. A ZenPack may extend all these areas but will do so under its own base directory.

Obviously, a ZenPack may also have objects discussed in the “Simple ZenPacks” chapter - monitoring templates, event classes, Mibs, etc.

Other elements that might be included in a ZenPack are:

- Parsers for datasources
- Reports
- New zProperties
- Event triggers and notifications
- Menus

8.0 zenpacklib UserGroup sample ZenPack

This first ZenPack sample will demonstrate using the zenpacklib utility to create a new zProperty, a new Zenoss device class, a new device object class and two new hierarchical components.

A modeler plugin will then be coded in Python to populate the new device and component classes. Once components exist, a component performance template will also be created, ultimately using zenpacklib.

Documentation for zenpacklib is at http://zenpacklib.zenoss.com/en/latest/index.html .

8.1 Requirements specification

The ZenPack will gather user and group information from Linux servers, using bash commands over the ssh protocol. The relationship between a user and their primary user group (also known as the **effective** user group) will be part of the object model so that easy navigation is possible from a group to the users contained within that group.

There will be a configurable option for the minimum UID user to be discovered so that low-numbered system UIDs can be omitted, if required.

A graph will be available for each user with the number of groups to which it is a member.

As extended examples, a colored icon will be used to denote whether a user group has secondary users. There will also be a method for a user to collate its secondary groups.

8.1.1 bash commands to access user and group information

User and group information can be gathered using the *getent* bash command to access data in */etc/passwd* and */etc/group*. Data in each file has several fields, colon-separated.

/etc/passwd has data in the format:

<username>:<password>:<UID>:<GID>:<user comment>:<home directory>:<command/shell>

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
...
zenplug:x:1001:1002::/home/zenplug:/bin/bash
snmp:x:107:110::/var/lib/snmp:/bin/false
mollie:x:1002:1003:Mol:/home/mollie:/bin/bash
```

/etc/group has:

<groupname>:<password>:<GID>:<list of users in group, comma-separated>

```
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:pi
audio:x:29:pi,mollie
...
indiecity:x:1001:root,pi
zenplug:x:1002:
snmp:x:110:
```

```
mlocate:x:111:
mollie:x:1003:
```

Unix users may exist in multiple groups but have the concept of a primary or **effective** group; this is the fourth field from */etc/passwd*. For a given user, *mollie*, this is determined with:

```
id -g -n mollie
mollie
```

The *-n* parameter specifies to deliver the group name rather than the GID.

The list of **all** user groups (sometimes know as **secondary** groups) for the *mollie* user is given by:

```
id -G -n mollie
mollie audio
```

Note that the primary group is also given in this list of "secondary" groups.

A user can have their primary group changed with:

```
usermod -g audio mollie
```

Note that this will require sudo or root privilege.

A user can have an additional secondary group added with the following where the *-a* parameter specifies to **add**:

```
usermod -a -G ntp mollie
id -G -n mollie

mollie audio ntp
```

To remove a user from a secondary group is slightly more intricate. *usermod -G* is used where any group that is **not** specified that the user is currently a member of, is removed from their list of secondary groups:

```
usermod -G audio mollie
id -G -n mollie

mollie audio
```

ntp has been removed. Note that this does **not** remove the user from the primary group, *mollie*.

8.2 ZenPack specification

The new ZenPack will be called ***ZenPacks.community.UserGroup***.

A new zProperty, ***zMinUID*** will be created. Note that any new zProperty is **global** and cannot be restricted to a subset of devices.

A new device object class called ***UserGroupDevice*** will be created. The device will have no extra attributes, beyond the standard *Device* object.

The ZenPack will create new component objects, ***User*** and ***UserGroup*** with attributes:

- **User**
  - userName
  - UID
  - primaryGID
  - primaryGroupName
  - userComment
  - homeDir
  - commandShell
- **UserGroup**
  - groupName
  - GID
  - secondaryUsers

Relationships will be required such that:

- A *UserGroupDevice* will have a contains many relationship (***userGroups***) to *UserGroup* components. The implied corresponding ToOne relationship will be ***userGroupDevice*** (note the capitalisation carefully).
- A *UserGroup* component will have a contains many relationship (***users***) to *User* sub-components. The implied corresponding ToOne relationship will be ***primaryUserGroup***.

A modeler plugin will be required to gather user and group configuration data - ***UserGroupMap***.

A **component** template, ***User***, will be created to deliver a count of the groups that a user belongs to.

8.3 Installing zenpacklib

8.3.1 PyYAML

zenpacklib requires the Python YAML library, **PyYAML** as a pre-requisite. **YAML** stands for "YAML Ain't Markup Language" - see http://yaml.org/.

PyYAML is installed as standard with Zenoss 4 SUP 457 and above and is standard with Zenoss 5; versions prior to this require PyYAML to to be installed explicitly.

To test whether PyYAML is installed, as the *zenoss* user, enter the python environment and import *yaml*:

```
python
import yaml
yaml
quit()
```

If it is installed, you will see something like:

```
 <module 'yaml' from '/opt/zenoss/lib/python2.7/site-packages/PyYAML-3.11-py2.7-
linux-x86_64.egg/yaml/__init__.py'>
```

If PyYAML is not installed, install it, as the *zenoss* user, with:

```
easy_install PyYAML
```

and then rerun the test above. You may see warning messages referring to the absence of *libyaml* - you appear to be able to ignore these.

8.3.2 Installing zenpacklib

Fundamentally, zenpacklib is just another Python program. Get zenpacklib into your nominated ZenPack development directory, */code/ZenPacks/DevGuide* for Zenoss 4 and */z/zenpacks* for Zenoss 5.

To obtain the package, on the base host, as the *zenoss* user:

```
wget http://zenpacklib.zenoss.com/zenpacklib.py
chmod 755 zenpacklib.py
```

Note that you may need to add the *--no-check-certificate* flag to *wget*.

After downloading you can check the version by running the following command (on Zenoss 4) as the *zenoss* user from the directory where the zenpacklib file was placed:

```
./zenpacklib.py version
```

or on Zenoss 5 from the base host:

```
zenpacklib version
```

Note that throughout this document, provided a Zenoss 5 environment has been setup according to section 3.1.2, zenpacklib should be called as follows:

- Zenoss 4
  - Change directory to the base directory of the ZenPack
  - *./zenpacklib.py <parameter>*
- Zenoss 5
  - *zenpacklib <parameter>*
  - The *.bashrc* in the *zenoss* user's home directory will ensure the working directory is */z/zenpacks*

To get some general help on usage of zenpacklib, use:

```
./zenpacklib.py help                Zenoss 4
zenpacklib help                     Zenoss 5
```

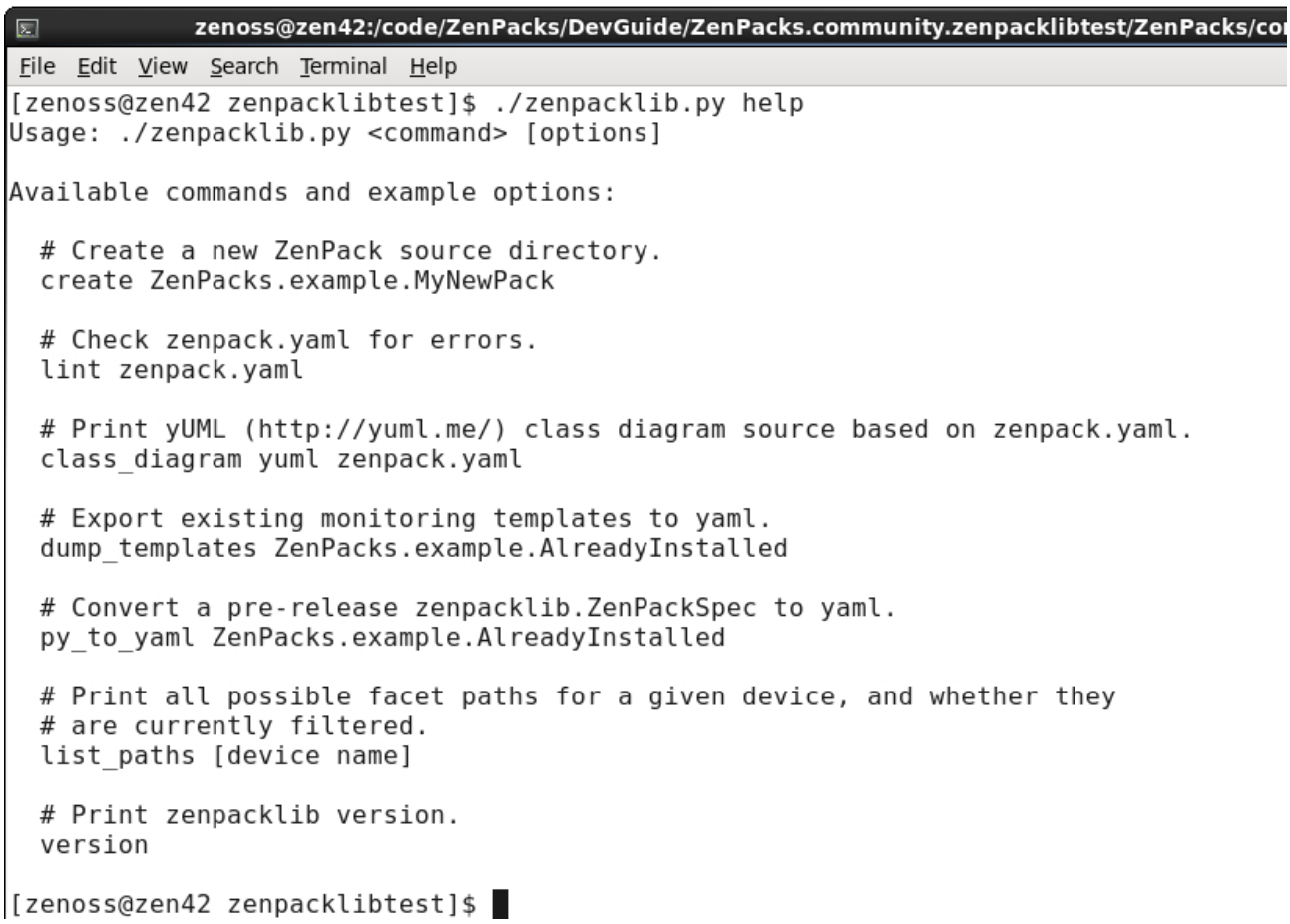


Figure 57: Help for zenpacklib.py

Note the ***lint*** option that performs syntax checking on a *zenpack.yaml* file.

See http://zenpacklib.zenoss.com/en/latest/command-line-reference.html for further documentation.

zenpacklib is developed on GitHub. For current outstanding issues and requests, see https://github.com/zenoss/zenpacklib/issues .

8.4 Creating the ZenPack

This ZenPack will use *zenpack.yaml* to define all the objects and relations. No extra GUI code, beyond that generated automatically to support the objects, is anticipated; neither are customized datasources, reports or events required. Thus the zenpacklib utility will be used to create the ZenPack.

Where several other elements **are** required, with the associated directory hierarchy, it is better practice to create the ZenPack using the command line (Zenoss 5) or the GUI (Zenoss 4), as documented in section 3.2.

With Zenoss 4, to create a ZenPack from the command line, as the *zenoss* user, in the current directory, run the following:

```
cd /code/ZenPacks/DevGuide
./zenpacklib.py create ZenPacks.community.UserGroup
```

or with Zenoss 5:

```
cd /z/zenpacks
```

```
zenpacklib create ZenPacks.community.UserGroup
```

The environment created in the *.bashrc* file ensures that zenpacklib commands are performed in the context of the current directory being */z/zenpacks* so that is where the ZenPack directory hierarchy will be created in Zenoss 5. Note that you **must** change to the directory where *zenpacklib.py* exists as it will be copied from "." (the current directory) into the base directory of the new ZenPack.

Several lines will be printed to document what has been created. Note that the ZenPack's source directory hierarchy has been created, but it has **not** yet been installed.

```
[zenoss@zen50:/z/zenpacks]$ zenpacklib create ZenPacks.community.UserGroup
Creating source directory for ZenPacks.community.UserGroup:
 - making directory: ZenPacks.community.UserGroup/ZenPacks/community/UserGroup
 - creating file: ZenPacks.community.UserGroup/setup.py
 - creating file: ZenPacks.community.UserGroup/MANIFEST.in
 - creating file: ZenPacks.community.UserGroup/ZenPacks/__init__.py
 - creating file: ZenPacks.community.UserGroup/ZenPacks/community/__init__.py
 - creating file: ZenPacks.community.UserGroup/ZenPacks/community/UserGroup/__init__.py
 - creating file: ZenPacks.community.UserGroup/ZenPacks/community/UserGroup/zenpack.yaml
 - copying: ./zenpacklib.py to ZenPacks.community.UserGroup/ZenPacks/community/UserGroup
[zenoss@zen50:/z/zenpacks]$
```

Note that these are the **only** files created. *__init__.py* is created in the *ZenPacks, community* and *UserGroup* directories but no directory hierarchy is built under the *UserGroup* subdirectory.

__init__.py in the ZenPack's base directory, *ZenPacks.community.UserGroup/ZenPacks/community/UserGroup*, will contain:

```
from . import zenpacklib
CFG = zenpacklib.load_yaml()
```

This is with zenpacklib version 1.0.11. Earlier versions of zenpacklib simply had `zenpacklib.load_yaml()` as the second line.

zenpacklib.py is copied to the base directory and a minimal *zenpack.yaml* is created there, containing a single line:

```
[zenoss@zen50:.../community/UserGroup]$ cat zenpack.yaml
name: ZenPacks.community.UserGroup
```

Install the ZenPack **in development mode** (that is, with the *--link* parameter). For Zenoss 4, the current directory should be the nominated ZenPack development directory where the ZenPack hierarchy resides; that is, one level higher than the ZenPack top-level directory - */code/ZenPacks/DevGuide*. For Zenoss 5, the command alias in *.bashrc* enforces that the directory hierarchy must be in */z/zenpacks;* it is irrelevant where the command is executed from.

```
zenpack --link --install ZenPacks.community.UserGroup
```

Restart all the Zenoss daemons

```
zenoss restart                                  for Zenoss 4
serviced service restart Zenoss.core            for Zenoss 5 Core
serviced service restart Zenoss.resmgr          for Zenoss 5 Enterprise
```

Create the *README.rst* skeleton file in the top-level directory of the ZenPack.

Check installed ZenPacks with:

```
zenpack --list
```

From the Zenoss GUI, navigate to *ADVANCED -> Settings -> ZenPacks*, select the new ZenPack and configure the *Author* and *License* fields; also set any Zenoss minimum version and ZenPack co-requisites.

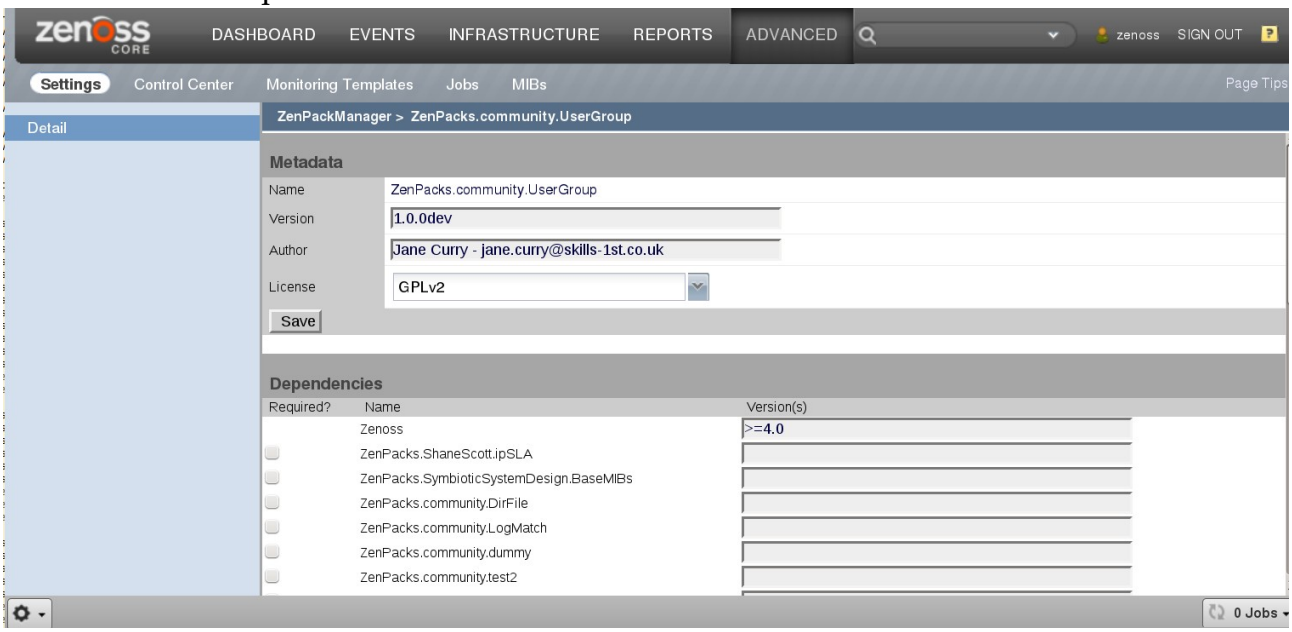


Figure 58: Configuring Author, License, Zenoss minimum version and ZenPack co-requisites

Export the ZenPack from the *Action* icon at the bottom-left of this GUI panel to create the egg file. The base ZenPack directory also gains an *objects* subdirectory and an empty *skins* subdirectory.

8.5 zenpack.yaml

Definitions of new zProperties, object classes, relationships, device classes and templates can be done in a ***zenpack.yaml*** file that must exist in the base directory of the ZenPack. This permits definitions to be done in a kind of pseudo code that is much simpler to write and much less error prone than Python. The other major benefit is that JavaScript is created automatically for these elements.

The YAML reference can be found at http://zenpacklib.zenoss.com/en/latest/yaml-reference.html . This documentation is terse but precise. It includes which keywords are mandatory, what the legal options are, and what the default value will be if omitted. Some keywords are concerned with defining object **attributes** (such as *type*, *properties*, *relationships*); other keywords are concerned with **layout** and appearance (for example, *label_width, icon, order*).

8.5.1 zProperties

The first section of *zenpack.yaml* defines the new zProperty, *zMinUID*:

```
name: ZenPacks.community.UserGroup

zProperties:
 DEFAULT:
```

```
    category: UserGroup
   zMinUID:
    type: string   # note this is string, not int
    default: 0
```

Check the documentation at http://zenpacklib.zenoss.com/en/latest/yaml-zProperties.html, following *YAML Reference -> zProperties*, for permissible fields and values. Note that:

- The *name* **must** start with a lower-case z
- Only the *name* field is mandatory.

8.5.2 Zenoss device classes

The *zenpack.yaml* file is rather focused around Zenoss device classes; the */Server/Linux/UserGroup* device class will be defined with the *zPythonClass* zProperty set to *ZenPacks.community.UserGroup.UserGroupDevice*. *zCollectorPlugins* will also be set.

```
device_classes:
 /Server/Linux/UserGroup:
  remove: False #False=default;ensure no instances left in this class when ZP removed
  zProperties:
   zPythonClass: ZenPacks.community.UserGroup.UserGroupDevice
   zSshConcurrentSessions: 5
   zDeviceTemplates:
    - Device
   zCollectorPlugins: ['zenoss.snmp.NewDeviceMap', 'zenoss.snmp.DeviceMap',
'zenoss.snmp.InterfaceMap', 'zenoss.snmp.RouteMap', 'zenoss.snmp.IpServiceMap',
'zenoss.snmp.HRFileSystemMap', 'zenoss.snmp.HRSWRunMap', 'zenoss.snmp.CpuMap',
'zenoss.snmp.SnmpV3EngineIdMap', 'cmd.UserGroupMap']
```

See http://zenpacklib.zenoss.com/en/latest/yaml-device-classes.html for the Device Classes YAML reference.

Note that:

- Comments can be added to a line, either at the start or anywhere on the line, prefaced by #.
- zProperties for the device class can be set. Do not confuse these with the creation of new global zProperties configured above.
- If *zDeviceTemplates* is set, this must be a **list** and must include **all** required templates. It is a common error to just specify the “extra” templates required.
- A list can either use the hyphen syntax shown for *zDeviceTemplates* or can use the square bracket syntax used for *zCollectorPlugins*. *zCollectorPlugins* must also contain **all** required modelers; *cmd.UserGroupMap* is included in anticipation of the modeler plugin that will be written.
- As with Python, white space indentation is crucial and will cause errors if not honoured.
- Device classes will be created recursively if necessary; ie, should the */Server* or */Server/Linux* class not exist when the ZenPack is installed, they will also be created.

8.5.3 Object classes

Documentation for object classes for device and components can be found at http://zenpacklib.zenoss.com/en/latest/yaml-classes-and-relationships.html .

```
classes:
 DEFAULTS:
  base: [zenpacklib.Component]

 UserGroupDevice:
  base: [zenpacklib.Device]
  meta_type: UserGroupDevice # Will default to this but in for completeness
  label: UserGroup Host
  icon: four-tux-56x56.png

  relationships:
   userGroups:
    label: User Groups
    display: true  # this has no effect as it is on the device class
```

A new device object must inherit from one or more existing classes; *zenpacklib.Device*, which is the standard *Device* object, defined in *$ZENHOME/Products/ZenModel/Device.py* is the default.

Similarly, a new component object inherits from *zenpacklib.Component*.

TODO: How to inherit from some other class??

Note the use of the *DEFAULTS* statement to set the *base* parameter to *[zenpacklib.Component].* Since several components will be defined, this avoids the need to explicitly set *base* for each component. This *DEFAULTS* technique can be used throughout *zenpack.yaml*.

The *label* will default to the *meta_type*, will default to the class *name*.

An icon can be specified for this device object class. Some standard icons are in *$ZENHOME/Products/ZenWidgets/skins/zenui/img/icons*. If no icon is specified then the *noicon.png* file will be used from this directory. The default value for the icon keyword in *zenpack.yaml* is <*device object class name*>*.png*.

To include icons with the ZenPack, appropriate *.png* files should be placed under the *resources/icon* directory hierarchy under the base directory. Icons should be about 56x56 pixels. This hierarchy will not exist if the ZenPack was created with zenpacklib. Create the hierarchy, remembering to create an *__init__.py* in each directory, with `touch __init__.py`.

properties in a class definition are object **attributes** and there is a large reference for property fields in addition to the class fields. A “special case” of an attribute (a characteristic of the object), is a **method** (an action that can be performed on the object) and methods can also be defined in *zenpack.yaml* with the *api_backendtype* and *api_only* keywords.

The *UserGroupDevice* object device class has no attributes beyond those inherited from its parent *Device*.

UserGroupDevice has two components, *UserGroup* and *User*, where the device can contain many user groups and the user group can contain many users.

```
UserGroup:
 label: User Group # NB It is label, with spaces removed, that is used to
                                    match a component template
 meta_type: UserGroup   # Will default to this but in for completeness
 label_width: 50  # This controls the column width for UserGroup in the
                                   Users component display
 order: 20     # before User; lower numbers nearer top / left
 auto_expand_column: secondaryUsers
 monitoring_templates: [UserGroup] # will default to UserGroup but explicit
                                                        for clarity

 properties:
  groupName:
   type: string
   label: Group name
   short_label: Group
   label_width: 150
   order: 3.1

  GID:
   type: int
   label: GID
   short_label: GID
   label_width: 60
   order: 3.2

  secondaryUsers:
   type: string
   label: Secondary Users
   short_label: Secondaries
   label_width: 300
   order: 3.3

 relationships:
  userGroupDevice:      # back to the containing device
   label: userGroupDevice
   display: true
  users:           # down to User sub-component
   label: users
   display: true
```

Note in the component definition:

- The *name* keyword, *UserGroup*, must be a valid Python class name so no white space.
- The *label* keyword is the human friendly label. By default, component performance templates bind **automatically** to a component object whose name exactly matches the template name. **It is the label, with any white space removed, not the component class name, that must match the template name**.
- The *monitoring_templates* keyword may be used to explicitly define performance templates to **automatically** bind to this component class. Note this keyword must be a list. The default is a single-element list with the label name (white space removed). It is possible with this keyword to override the default binding and, indeed, bind several templates to this component type.
- The *order* keyword controls the order that components are shown in the left-hand menu for the device. The lower the number, the nearer it is to the top of the list. Default value is 50.

- The *UserGroup* component object class has three attributes, defined with the *properties* keyword:
  - The *name* (eg. *groupName*) must be a valid Python class
  - The *type* can be one of *string, int, float, boolean, lines, password* or *entity*. TODO: what is an entity? How to use it?
  - *label* is the human friendly label; *short_label* is an alternative, automatically used if space is short.
  - *order* controls the order of properties in the component display, lower numbers nearer the left.

The User component is similar with attributes for *userName, UID, primaryGID, primaryGroupName, userComment, homeDir* and *commandShell*.

```
User:
 label: User    # NB It is label, with spaces removed, that is used to
                              match a component template
 meta_type: User  # Will default to this but in for completeness
 label_width: 50  # This controls the column width for Users in the
                                    UserGroup component display
 order: 30     # after UserGroup
 auto_expand_column: userComment
 monitoring_templates: [User] # will default to User but explicit for clarity

 properties:
  userName:
   type: string
   label: User name
   short_label: User
   label_width: 60
   order: 3.1

  UID:
   type: int
   label: UID
   short_label: UID
   label_width: 30
   order: 3.2

  primaryGID:
   type: int
   label: primaryGID
   short_label: GID
   label_width: 30
   order: 3.3
  ........
  userComment:
   type: string
   label: User Comment
   short_label: Comment
   label_width: 150
   editable: true  # default is false
   order: 3.6
  .........
  commandShell:
   type: string
   label: Command / Shell
   short_label: Shell
```

```
label_width: 150
order: 3.8
display: false  # overridden by grid_display and details_display
grid_display: false
details_display: true
```

Note:

- The *label_width* on the *User* class statement. zenpacklib will **automatically** generate JavaScript code for the *UserGroup* object to represent the number of *User*s in that group. The **class** *label_width* defines the space allocated to this auto-generated *User* display field.
- As the order is higher than that for *UserGroup*, the *User* component will be **lower** down the component list.
- When displaying details for a component, the maximum space available is 750 pixels. The *label_width* keyword (default 80 pixels) can be used to control space for each property; The a*uto_expand_column* keyword on the class defines which property to give any remaining space to. Note that if the combination of all *label_width* keywords exceeds 750 pixels, then none will be honoured.
- There is also a *content_width* keyword which defaults to the same as *label_width*.
- By default, properties in the *Details* dropdown are read-only. This can be changed with the *editable* keyword. Note that when the next modeling cycle is run, any changes will be replaced with the current value from the modeled device.
- There are three keywords to control whether / where a property is displayed. By default, *display* has the value *true* so the property is shown in the component grid and in the component *Details* dropdown display. *commandShell* has the *display* default changed to *false*, *grid_display* set *false* and *details_display* set *true*. The *display* keyword does not enforce display on both if a *grid_display* or *details_display* overrides it.

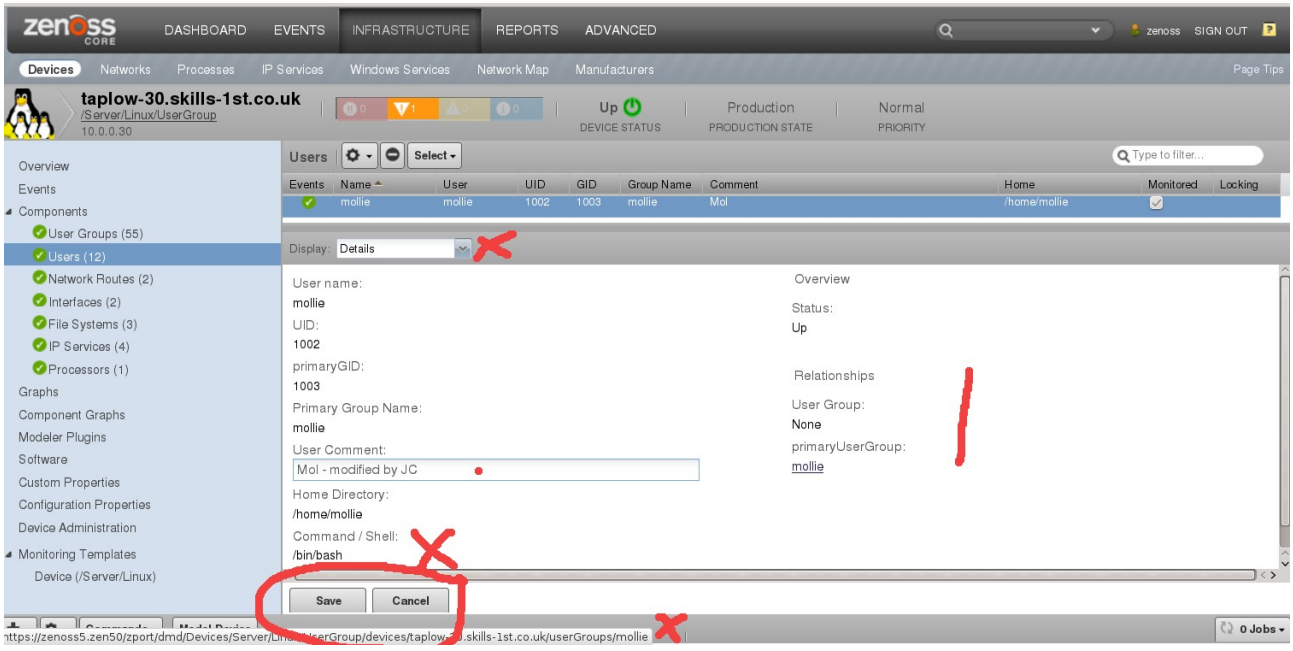


Figure 59: Display of User component

Note in Figure 59:

- *User Groups* are above *Users* in the left-hand menu
- The *short_label* is automatically used for the column headers - GID, rather than primary GID, etc.
- The icon for the device is non-standard.
- The userComment field is editable.
- The *commandShell* attribute is **not** shown in the grid but **is** included in the *Details* dropdown.

8.5.4 Relationships

Relationships are Zenoss' way of saying objects are related to each other. For example in core Zenoss code, the *DeviceHW* class contains many CPUs of the *CPU* class. You must also declare relationships between classes in your ZenPack. If you only declare types based on *zenpacklib.Device* you don't have to do this because they will automatically have a relationship to their containing device class, *Device* by default.. However, you must define at least a containing relationship for every type based on *zenpacklib.Component*. This is because components aren't contained in any relationship by default, and every object in Zenoss **must** be contained somewhere to connect it to the overall object map hierarchy.

zenpacklib supports the following types of relationships.

- One-to-Many Containing (1:MC)
- One-to-Many (1:M)
- Many-to-Many (M:M)
- One-to-One (1:1)

It's important to understand the difference between containing and non-containing relationships. Each component type must be **contained by exactly one** relationship. Beyond that a device or component type may have as many non- containing relationships as required. This is because every object in Zenoss has a single primary path that describes where it is stored in the tree that is the Zenoss object database.

In the UserGroup ZenPack, a *UserGroupDevice* contains many *UserGroup* component objects and a *UserGroup* component object can contain many *User* sub-component objects.

The documentation for relationships is rather scattered at http://zenpacklib.zenoss.com/en/latest/yaml-reference.html . The fundamental *class_relationships* keyword is described at http://zenpacklib.zenoss.com/en/latest/yaml-zenpack.html , as a non-mandatory **list**, whose default is an empty list.

The default way to write the *class_relationships* for the UserGroup ZenPack would be:

```
class_relationships:
 - UserGroupDevice 1:MC UserGroup
 - UserGroup 1:MC User
```

The names here **must** match the object class names. Code will be automatically generated by zenpacklib to implement these relationships. If a modeler plugin is required - and that is necessary to populate the components - then the names of these auto-generated relationships will be needed.

See http://zenpacklib.zenoss.com/en/latest/tutorial-snmp-device/component-modeling-5.html for some explanation of how relationship names are defaulted. The rules are:

- The leading upper-case letter of the class name will be converted to lower-case, i.e. for the ***U**serGroupDevice* object, the relationship name becomes ***u**serGroupDevice* on the ***U**serGroup* object, which is contained by one ***U**serGroupDevice*.

- The letter "s" is added to the end if it is a to-many relationship.
  - In the first *class_relationship* line, the *UserGroup* object is on the "many" side, so the relationship name on the *UserGroupDevice* object becomes *userGroups*.
  - In the second line, the *UserGroup* object is on the "One" side, so the relationship name on the *User* object is *userGroup* and the relationship on the *UserGroup* object will be *users*.

Relationship names can be defined and used **explicitly** and it is good practice to do so.

The *class_relationship* list elements may be augmented by the relationship **name** to be used, in brackets. They may be the default value but do not have to be.

```
class_relationships:
 - UserGroupDevice(userGroups) 1:MC UserGroup(userGroupDevice)
 - UserGroup(users) 1:MC User(primaryUserGroup)
```

The relationship names must then be defined on the appropriate object class - see http://zenpacklib.zenoss.com/en/latest/yaml-classes-and-relationships.html for more information on the *relationships* keyword in *zenpack.yaml*. It is described as being of *type* ***map<name, Relationship Override>*** where the Relationship Override keywords are given later on the same page.

The UserGroup ZenPack has overridden the relationship that would have been *userGroup* (in the second line) to be *primaryUserGroup* (still following the convention for lower-casing the first letter and, in this case, not adding a trailing "s" as the relationship is back to the single *UserGroup* object).

The object class definitions have *relationships* stanzas for:

```
UserGroupDevice:
...
 relationships:
  userGroups:
   label: User Groups
   display: true  # this has no effect as it is on the device class

UserGroup:
...
 relationships:
  userGroupDevice:      # back to the containing device
   label: userGroupDevice
   display: true   # Ensures relationship shown in Details dropdown
  users:           # down to User sub-component
   label: users
   display: true   # Relationship shown on grid and Details

User:
 relationships:
  primaryUserGroup:     # back to the containing primary UserGroup
   label: primaryUserGroup  # label for userGroup in users component
```

```
                                        panel taken from UserGroup label, not
                                              from here
display: true
grid_display: false # this does control whether UserGroup
                                        displayed in users component panel
details_display: true
label_width: 20   # this does NOT control width of UserGroup in
                                        users component panel
order: 3.3
```

Note that the *display* keywords control whether the **relationships** are displayed on grid and *Details* menu, although some are ignored. In Figure 59, both relationships are shown in the *Details* dropdown and the *primaryUserGroup* relationship is presented as a link.

Note that the *content_width* keyword on a Relationship Override Map documents that To-Many relationships are shown simply as a count and will have a shorter width. To-One relationships show a link to the object and will require a width long enough to accommodate the object's title.

One of the huge benefits of the auto-generated JavaScript code, where there is a hierarchical component relationship, is that an extra *Display* dropdown is created for a component which has a sub-component object class.

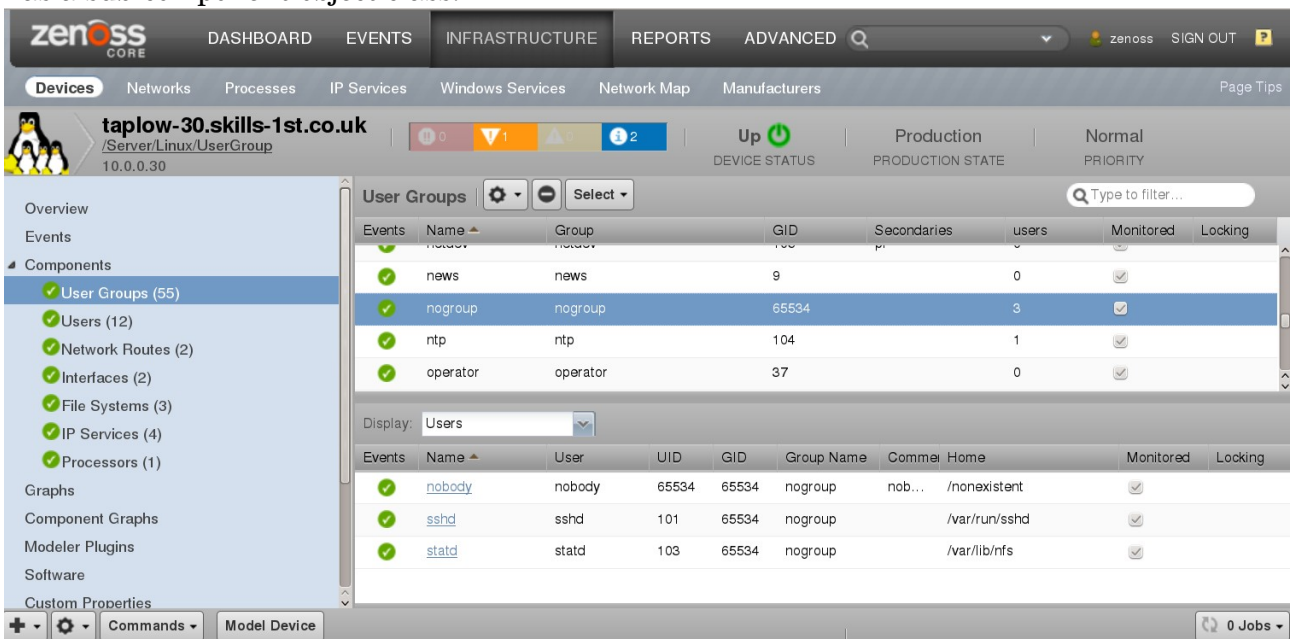


Figure 60: Users dropdown menu for sub-component of User Groups

The dropdown follows the *users* relationship on the *UserGroup* object **instance** from component to sub-component, and displays the *User* component display in the bottom half of the window, for the selected user group.

Note that although the relationship statements define *UserGroupDevice contains many UserGroup components contains many User sub-components,* in the left hand menu for the device, components and sub-components (to any level) are treated in the same way. The left-hand menu cannot show the hierarchy.

8.6 Deploying and testing the ZenPack

Once *zenpack.yaml* is complete, the first step is to check its syntax, with the ***lint*** parameter. With Zenoss 4, run *./zenpacklib.py* in the ZenPack's base directory. The *zenpack.yaml* file is found in the current directory.

```
./zenpacklib.py lint zenpack.yaml
```

With Zenoss 5 and the environment described earlier, the directory is set to */z/zenpacks* automatically. If you have a copy of *zenpacklib.py* in */z/zenpacks* then the following will work:

```
zenpacklib lint ZenPacks.community.UserGroup/ZenPacks/community/UserGroup/zenpack.yaml
```

The path to *zenpack.yaml* must be given, relative to */z/zenpacks*.

Beware that different ZenPacks may have different versions of *zenpacklib.py* so inconsistencies could occur between the file in */z/zenpacks* and that in the base directory of the ZenPack.

Any syntax errors will be reported, usually with helpful pinpointing of line numbers. A simple return prompt indicates a successful check.

To install the ZenPack in development mode, run the following *--link --install* command. For Zenoss 4, the current directory should be one higher than the top-level directory, eg. */code/ZenPacks/DevGuide*; for Zenoss 5, the context will automatically be one higher than the top-level directory - */z/zenpacks*.

```
[zenoss@zen50:.../community/UserGroup]: zenpack --link --install ZenPacks.community.UserGroup


INFO:zen.ZenPackCMD:installing zenpack ZenPacks.community.UserGroup; launching process
INFO:zen.zenpacklib:Creating DeviceClass Server/Linux/UserGroup
INFO:zen.zenpacklib:Setting zProperty zCollectorPlugins on Server/Linux/UserGroup
INFO:zen.zenpacklib:Setting zProperty zDeviceTemplates on Server/Linux/UserGroup
INFO:zen.zenpacklib:Setting zProperty zSshConcurrentSessions on Server/Linux/UserGroup
INFO:zen.zenpacklib:Setting zProperty zPythonClass on Server/Linux/UserGroup
2016-04-14 18:22:15,650 INFO zen.HookReportLoader: Loading reports from
/z/zenpacks/ZenPacks.community.UserGroup/ZenPacks/community/UserGroup/reports
```

All daemons should be restarted:

```
serviced service restart Zenoss.core            Zenoss Core 5
serviced service restart Zenoss.resmgr          Zenoss Enterprise 5
zenoss restart                                  Zenoss 4
```

At this stage, only some functionality will work.

- The new zProperty, *zMinUID*, should exist and be displayed in any device or device class *Configuration Properties* menu.
- The new */Server/Linux/UserGroup* Zenoss device class should appear in the *INFRASTRUCTURE -> Devices* left-hand hierarchy. Check that this class has the zProperties assigned in *zenpack.yaml*.
- The components will not yet work as there is no modeler to populate them

When testing new object class functionality, a device will either need to be newly created into the new Zenoss class, or it is sufficient to move a device from a different class into the new class. The new attributes and relationships will then be created.

A test device is essential. It is good practice to move the test device out of the test class before reinstalling a ZenPack which has had object modifications made; */Ping* is an easy existing class to move it to as this has very basic capability and all existing attributes and relations from the ZenPack, will be removed automatically. It is not necessary to remodel the test device in the */Ping* class.

If the test device is moved into */Server/Linux/UserGroup*, it should inherit the modeler plugins and zProperties from the class, including *zPythonClass* = *ZenPacks.community.UserGroup.UserGroupDevice*. Although no component **instances** can be discovered yet - the *cmd.UserGroupMap* modeler has not yet been written - the *UserGroup* **relationship** should appear when inspected using the ZMI; however, it will have no instances; thus there can be no sub-component *User* relationship to see.

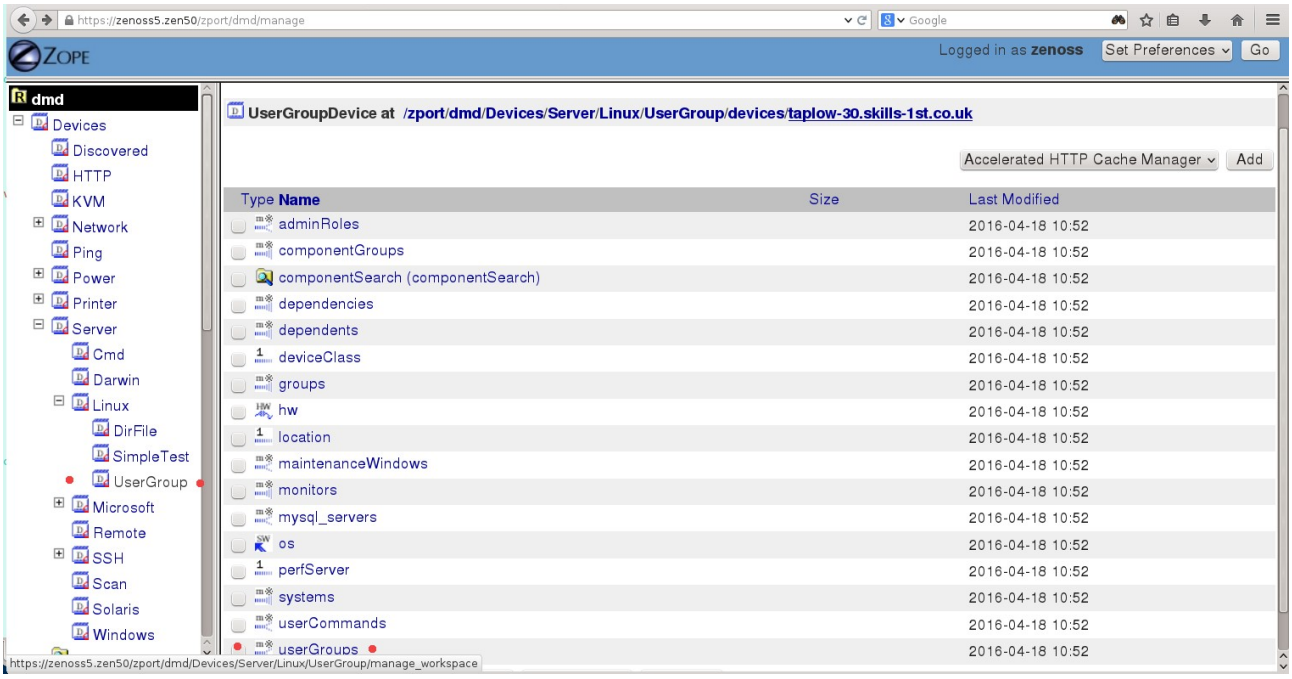


Figure 61: Inspecting test device newly moved to /Server/Linux/UserGroup class; note userGroups relationship

If changes are required, then *zenpack.yaml* must be edited. Some changes do not require a full Zenoss restart. Examples would be:

- Changing the *order* keyword to reorder classes or properties
- Changing the label_*width* keyword for a class or property
- Changing the *label* or *short_label* for a property
- Changing the *auto_expand_column*
- Changing the *renderer* for a property
- Changing *display*, *grid_display* or *details_display*

In this case, only zope needs restarting:

```
zopectl restart                          for Zenoss 4
serviced service restart zope            for Zenoss 5
```

Changes that would mandate a full Zenoss restart are:

- Adding any new zProperty, class, property or relationship
- Changing a relationship
- Changing a class or property *name*
- Changing the *meta_type* of a class
- Changing the *type* of a property
- Changing the *label* for a class (as that changes the default template name bound to it)

As a guideline, keyword changes that only affect the GUI **display** characteristics, can probably just recycle zope; anything that affects the **definition of an object** requires a full Zenoss recycle. If in doubt, recycle Zenoss entirely.

8.7 Modeler plugin

zenpacklib **cannot** provide help with writing a modeler plugin. It is **essential** that the ZenPack writer knows the precise names of objects, properties and relationships defined in *zenpack.yaml*.

8.7.1 Design details

Although zenpacklib avoids the need to write object classes in Python for *UserGroup* and *User*, Python modules **will** be automatically constructed in memory for these two object classes. The module names will be:

```
<ZenPack name>.<object class name>                    That is.......

ZenPacks.community.UserGroup.UserGroup                and
ZenPacks.community.UserGroup.User
```

zenpack.yaml carefully named relationships explicitly and these will also be required in the modeler.

The modeler will use a command run in bash over an ssh session to gather user and user group information using the *getent* Unix utility. The group output will be separated from the user output by a line containing *__SPLIT__*.

The new zProperty, *zMinUID*, will be used to restrict the user sub-components created, to those with a UID greater than or equal to *zMinUID*.

The modeler needs to:

- Create an empty list to hold user group maps
- Create an empty list to hold user maps
- Cycle through each user group:
  - Create the *UserGroup* component instance using the *userGroups* relationship
  - Allocate the attributes found from the *getent group* command.
  - Cycle through the user data looking for the primary GID field equal to this user group.

- If found (and there can only be one primary GID for any user), create a sub-component *User* instance, using the *users* relationship on the *UserGroup* object. A group may, of course, be the primary GID for many users.
- Allocate the user attributes found from the *getent passwd* command
- Return the user relationship map

- Add the user relationship map to the user map list
- Add the user group relationship to the user group map list
- Return all the maps to be processed by *zenhub* into the ZODB database

8.7.2 UserGroupMap modeler plugin code

8.7.2.1 Creating the directory hierarchy

The ZenPack has been created with zenpacklib; thus no directory hierarchy has been created for modeler plugins. As this will be a command plugin, the plugin should go under *modeler/plugins/cmd* under the base directory. Manually create each directory and *touch* an *__init__.py* file in each directory. Create *UserGroupMap.py* under the *cmd* subdirectory. The name of the file **must** match the name of the modeler plugin **class** inside it. The directory path hierarchy will be reflected in the GUI dialogue for choosing plugins so will be shown as *cmd.UserGroupMap*.

Note that any modeler code not being actively used under *modeler/plugins* should not end in ".py". When the zenhub daemon is restarted it will attempt to recompile and add all Python source files under *modeler/plugins* to its list of modelers. Broken modelers, even if never used, will cause error messages in *zenhub.log* and waste resource cycles.

Most plugins are created by starting with an example as close to your target as possible, and then modifying to suit.

8.7.2.2 Imports from other Python modules

Any Python code may use functions from other Python modules. An *import* statement is required to link to any external utilities, before they can be used in the code.

```
# CommandPlugin is the base class that provides lots of help in modeling data
# that's available by connecting to a remote machine, running command line
# tools, and parsing their results.
from Products.DataCollector.plugins.CollectorPlugin import CommandPlugin

# Classes we'll need for returning proper results from our modeler plugin's process method.
from Products.DataCollector.plugins.DataMaps import ObjectMap, RelationshipMap
from Products.ZenUtils.Utils import prepId
import collections
from itertools import chain
```

Note that Zenoss imports specify a path relative to *$ZENHOME*, using a "dot" between elements (because, strictly, this is a Python **module** path, rather than a file directory path, though the two are identical when the slash / dot substitution is made); thus most Zenoss imports will start with *Products*. Other imports can be made from standard Python modules.

If an entire module is required, use the "*import collections*" syntax and all functions in that module are available; if only one or two functions are required from a large module, use the "*from Products.ZenUtils.Utils import prepId*" format. This is better practice as it will reduce the overall footprint required to run this modeler plugin.

8.7.2.3 Base class for the UserGroupMap modeler plugin

The UserGroupMap plugin will inherit from the ***CommandPlugin*** object class defined in *$ZENHOME/Products/DataCollector/plugins/CollectorPlugin.py*.

Fundamentally, a modeler runs against a **device** (not a component). The component to be created on the *UserGroupDevice* will be of object class *UserGroup*, following the device's defined relationship of *userGroups*. Hence, the modeler code specifies:

```
relname = 'userGroups'
modname = 'ZenPacks.community.UserGroup.UserGroup'
```

where the *modname* is the module name that has been auto-created in memory to define the *UserGroup* object class (<ZenPack name>.<object class name>).

UserGroup : vim - Konsole

File Edit View Bookmarks Settings Help

```
# Module-level documentation will automatically be shown as additional
# information for the modeler plugin in the web interface.
"""
UserGroupMap
SSH plugin to gather user group and user information
"""

# When configuring modeler plugins for a device or device class, this plugin's
# name would be cmd.UserGroupMap because its filesystem path within
# the ZenPack is modeler/plugins/cmd/UserGroupMap.py. The name of the
# class within this file must match the filename.

# CommandPlugin is the base class that provides lots of help in modeling data
# that's available by connecting to a remote machine, running command line
# tools, and parsing their results.
from Products.DataCollector.plugins.CollectorPlugin import CommandPlugin

# Classes we'll need for returning proper results from our modeler plugin's process method.
from Products.DataCollector.plugins.DataMaps import ObjectMap, RelationshipMap
from Products.ZenUtils.Utils import prepId
import collections
from itertools import chain

class UserGroupMap(CommandPlugin):
    # relname and modname for the CommandPlugin will be inherited by any calls to
    #    rm = self.relMap()   or om = self.objectMap()
    # No compname specified here as UserGroup is a component directly on the device (defaults to null string)
    # classname not required as largely deprecated. classname is the same as the module name here
    relname = 'userGroups'
    modname = 'ZenPacks.community.UserGroup.UserGroup'

    # Make ZenPack zMinUID zProperty available to plugin code
    deviceProperties = CommandPlugin.deviceProperties +(
        'zMinUID',
        )

"modeler/plugins/cmd/UserGroupMap.py" [Modified] 162 lines --19%--                    32,1          Top
```

Figure 62: Start of UserGroupMap modeler plugin with class inheritance and zProperty

8.7.2.4 Using zProperties in the modeler plugin

Any plugin can extend the *deviceProperties* that are available to the modeler:

```
deviceProperties = CommandPlugin.deviceProperties + (
  'zMinUID',
  )
```

This ensures that the new zProperties defined in *zenpack.yaml* are available to the modeler. Note that this extends the zProperties from the *CommandPlugin* class, which inherits from the *CollectorPlugin* (see *$ZENHOME/Products/DataCollector/plugins/CollectorPlugin.py*):

- **CommandPlugin zProperties**
  - 'zCommandPort',
  - 'zCommandProtocol',
  - 'zCommandUsername',
  - 'zCommandPassword',
  - 'zCommandLoginTries',
  - 'zCommandLoginTimeout',
  - 'zCommandCommandTimeout',
  - 'zKeyPath',
  - 'zCommandSearchPath',
  - 'zCommandExistanceTest',
  - 'zSshConcurrentSessions',
  - 'zTelnetLoginRegex',
  - 'zTelnetPasswordRegex',
  - 'zTelnetSuccessRegexList',
  - 'zTelnetTermLength',
  - 'zTelnetEnable',
  - 'zTelnetEnableRegex',
  - 'zEnablePassword',
- **CollectorPlugin properties and zProperties**
  - 'id',
  - 'manageIp',
  - '_snmpLastCollection',
  - '_snmpStatus',
  - 'zCollectorClientTimeout',

Within the ZenPack plugin code, any of these can simply be referred to as attributes of ***device***; for example, *device.zMinUID*.

8.7.2.5 CommandPlugin command

The essential variable that a *CommandPlugin* must provide is the **command**. This can be anything that will run in a bash shell, so includes other language scripts if they have an appropriate **shebang.** A shebang tells the shell what program to interpret the script with, when executed; for example, *#!/usr/bin/env python*. If the command for the plugin is **not** built-in shellscript code then the command must exist on every target and the correct path to the script must be known. This may be a major undertaking for some organizations.

The command for this ZenPack plugin needs to get the user group information and then the user information; a line with *__SPLIT__* will separate group information from user information. The output will be delivered into a ***results*** variable, to be processed by the ***process*** method.

In this case, it is not worth the overhead of creating a custom shellscript. Multiple shellscript commands can be accommodated by separating lines with semicolon.

```
# The command to run.
# Get user groups (one per line) then a line with __SPLIT__ then users (one per line)
# Beware this has potential to return LOTS of data
command = (
    'getent group ;'
    'echo __SPLIT__; '
    'getent passwd'
    )
```

Each line needs to be single-quoted when using this construct.

Remember that the command will run, taking account of all the zCommand zProperties of the device so if ssh to target devices uses usernames and passwords, then the correct values need to be configured for *zCommandUsername* and *zCommandPassword*. If public keys are used for ssh, the *zCommandUsername,* potentially the *zCommandPassword,* and the *zKeyPath* must be correct and the public key for the *zenoss* user on the Zenoss server needs to have been copied to the *.ssh/authorized_keys* file for the correct user on the target systems. Note that *zCommandPassword* is used to hold the **passphrase** for the key if one has been set; otherwise *zCommandPassword* is not used with public key ssh.

Other zProperties to note that affect running commands over ssh are:

- zCommandLoginTries default 1
- zCommandLoginTimeout default 10s
- zCommandCommandTimeout default 15s
- zCommandSearchPath default is unset
- zSshConcurrentSessions default is 10

Note particularly *zCommandCommandTimeout* if you have a long-running command; extending the time limit potentially just slows the whole modeling process, especially if some targets are not responding at all.

The *zCommandSearchPath* is a good way of defining a standard for where local scripts should be held, with the possibility of device-level override if necessary. If the command provided is not a fully-qualified pathname then the script will be sought for in *zCommandSearchPath*.

If timeouts occur on the ssh sessions, sometimes an event is generated suggesting that *zSshConcurrentSessions* be lowered. *zenpack.yaml* configured *zSshConcurrentSessions* for the */Server/Linux/UserGroup* class to be ***5***. Many newer versions of unix systems have their ssh daemons configured by default to only allow 1 session per connection, so in these cases *zSshConcurrentSessions* will need to be ***1***; that can significantly reduce the performance of COMMAND datasources.

8.7.2.6 The process method of the modeler plugin

The *process* function is passed the command output in the *results* parameter; the *device* object is also passed as is the *log* parameter.

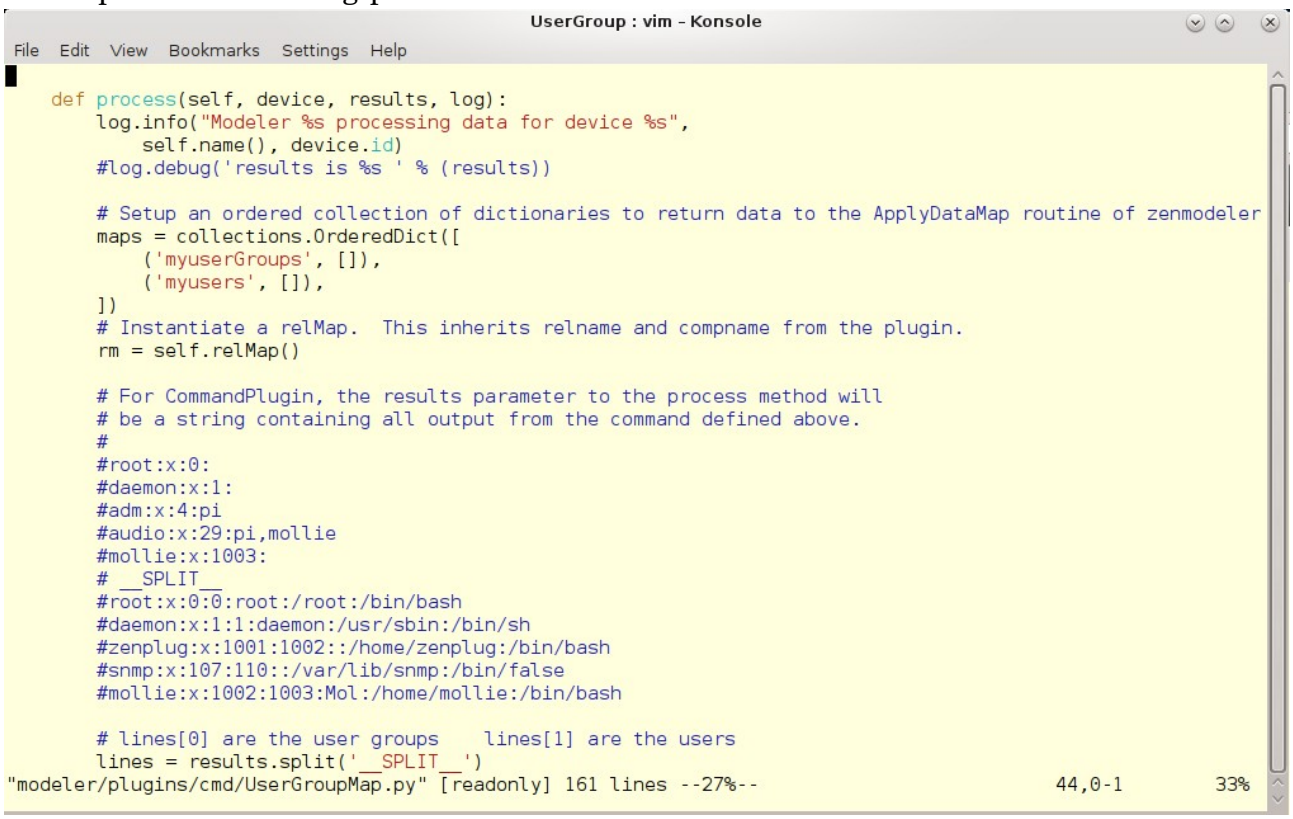


Figure 63: Start of process function for UserGroupMap modeler

Typically the first line of the method provides some logging and may often include debug logging to show the raw results:

```
def process(self, device, results, log):
  log.info("Modeler %s processing data for device %s",
    self.name(), device.id)
  #log.debug('results is %s ' % (results))
```

"self" in this case is the modeler plugin so the *log.info* line would provide output in *$ZENHOME/log/zenmodeler.log* like:

```
2016-04-19 08:49:44,100 INFO zen.ZenModeler: Modeler cmd.UserGroupMap
processing data for device taplow-30.skills-1st.co.uk
```

Ultimately, the modeler will return a collection of **maps** to *zenhub*, with a list of maps for user groups and a list of maps for users. *maps* is initialised with:

```
# Setup an ordered collection of dictionaries to return data to the
```

```
#    ApplyDataMap routine of zenmodeler
maps = collections.OrderedDict([
  ('myuserGroups', []),
  ('myusers', []),
])
```

A relationship map is created to hold the **list** of *UserGroup* object maps that will be added to the device's *userGroups* relationship. The relationship is defined by the *relname* statement at the top of the class definition.

```
rm = self.relMap()
```

The main body of the *process* method builds *ObjectMaps* for components and sub-components and delivers the *RelationshipMaps* that link them together.

Where a modeler - any modeler - has to populate a “component that contains a sub-component” set of relationships, typically there is a *for* loop to process the component and then an internal loop, often handled as a separate function, to process sub-components of the component. The trick is to pass the component relationship and instance as parameters to the inner loop. An algorithmic outline would be:

```
initialise RelationshipMap for component maps
for component in list_of_components
  get relevant data for component
  modify any raw data, as required
  create an ObjectMap for the component
  add ObjectMap to component RelationshipMap
  for sub-component in list of sub-components
    initialise a list for sub-component maps
    get relevant data for sub-component
    modify any raw data, as required
    create an ObjectMap for the sub-component
    add ObjectMap to sub-component map list
    return a RelationshipMap with correct compname, relname, modname
         and the sub-component map list
return the RelationshipMap for the device with correct component relname,
    modname and the component map list
```

There are some really helpful comments from “cluther” at the end of the modeler in the ZenPacks.zenoss.OpenVZ ZenPack (see https://github.com/zenoss/ZenPacks.zenoss.OpenVZ/blob/develop/ZenPacks/zenoss/OpenVZ/modeler/plugins/zenoss/cmd/linux/OpenVZ.py):

```
# a relMap() is just a container to store objectMaps.
# in relmap and objectmap, there is a compname and modname
# any objectmaps and relmaps are temporary objects that the modeler plugin
# sends to zenhub, which then determines if the model needs to be updated.
#
# device/containers/106
# om ^  relmap ^  om^
# we are allowed to return:
  # a relmap - will be filled with object maps that are related to "device"
  # an objectmap -
  # a list of relmaps, objectmaps
```

A *process* method acts upon *results* and must deliver one of:

- None - changes nothing. Good in error cases.

- A *RelationshipMap* for the device - component information
- An *ObjectMap* for the device - device information
- A **list** of *RelationshipMaps* and *ObjectMaps* – both

See http://www.zenoss.org/forum/137406 for an excellent explanation of these options.

A very common and frustrating issue with modelers handling components and sub-components, is that you tend to build lists of maps but when *zenhub* actually tries to apply these *DataMaps* to the Zope Database (ZODB), component relationships may not exist before a sub-component relationship is applied, resulting in an error. Symptoms of this can be seen in *zenhub.log* with lines like:

```
2015-11-26 09:42:20,643 INFO zen.ZenHub: Worker (20122) reports 2015-11-26
09:42:20,642 WARNING zen.ApplyDataMap: Unable to find compname
'userGroups/audio'
```

The neatest sample solution around is in the *ZenPacks.zenoss.AWS ZenPack*, in the *EC2.py* modeler, found at https://github.com/zenoss/ZenPacks.zenoss.AWS/blob/develop/ZenPacks/zenoss/AWS/modeler/plugins/aws/EC2.py .

The method is:

- Setup an ordered collection of dictionaries to return data to the *ApplyDataMap* routine called by *zenhub*:

```
maps = collections.OrderedDict([
  ('myuserGroups', []),
  ('myusers', []),
])
```

- The outer loop (in the algorithmic outline above) creates the *UserGroup* component object map (*om*) and builds the *rm* list of relationship map of components:

```
rm.append(om)
```

- The outer loop calls the function *getUserMap* to accomplish the inner loop, to deliver the related sub-components, appending to the sub-component tuple:

```
um=(self.getUserMap( device,lines[1],int(ugList[2]), ugList[0], 'userGroups/%s' % ug_id, log))
maps['myusers'].append(um)
```

- The outer loop delivers the component *RelationshipMap* to the component tuple:

```
maps['myuserGroups'].append(rm)
```

- The return delivered by the *process* method ensures the maps are delivered in the correct order for application by *zenhub*:

```
return list(chain.from_iterable(maps.itervalues()))
```

In the main body of the *process* function, the *results* command output is split into two lists using the *__SPLIT__* line as the split parameter. This results in:

- lines[0] are the user groups, one per line, newline terminated
- lines[1] are the users, one per line, newline terminated

It is good practice and enormously helpful, for modeler code to include sample output as comments.

Each user group line can then be processed in the body of the outer loop.

UserGroup : vim - Konsole

File Edit View Bookmarks Settings Help

```
        # Instantiate a relMap.  This inherits relname and compname from the plugin.
        rm = self.relMap()

        # For CommandPlugin, the results parameter to the process method will
        # be a string containing all output from the command defined above.
        #
        #root:x:0:
        #daemon:x:1:
        #adm:x:4:pi
        #audio:x:29:pi,mollie
        #mollie:x:1003:
        # __SPLIT__
        #root:x:0:0:root:/root:/bin/bash
        #daemon:x:1:1:daemon:/usr/sbin:/bin/sh
        #zenplug:x:1001:1002::/home/zenplug:/bin/bash
        #snmp:x:107:110::/var/lib/snmp:/bin/false
        #mollie:x:1002:1003:Mol:/home/mollie:/bin/bash

        # lines[0] are the user groups    lines[1] are the users
        lines = results.split('__SPLIT__')
        for ug in lines[0].split('\n'):
"modeler/plugins/cmd/UserGroupMap.py" [readonly] 161 lines --34%--                    55,21         38%
```

Figure 64: UserGroupMap modeler - sample command output is split and user group loop initiated

The main loop of the *process* function splits each user group line using the colon as the separator, to derive the individual attributes for a user group.

UserGroup : vim - Konsole

File Edit View Bookmarks Settings Help

```
        for ug in lines[0].split('\n'):
            #log.debug(' group is %s' % (ug))
            if ug:
                try:
                    # split each line on ':'
                    ugList = ug.split(':')
                    # ugList[0] = groupName, uglist[2] = GID, ugList[3] = secondaryUsers
                    ug_id = prepId(ugList[0])           # Ensure no dodgy characters in id
                    # Add an Object Map for this user group
                    # Use prepId to ensure id is unique and doesn't include any dodgy characters like /
                    # om = self.objectMap() inherits modname and compname (null) from plugin
                    om = self.objectMap()
                    om.id = ug_id
                    om.groupName = ugList[0]
                    om.GID = int(ugList[2])    # GID defined as integer so need to ensure this
                    om.secondaryUsers = ugList[3]
                    rm.append(om)
                    # For this user group, create a map for associated users, passing this ug_id as part of compname
                    log.debug('GID is %s ' % (om.GID))
                    um = (self.getUserMap( device, lines[1], int(ugList[2]), ugList[0], 'userGroups/%s' % ug_id, log))
                    #log.debug('ug %s has um  %s \n um relname is %s and um compname is %s ' % (om.id, um, um.relname, um.compname))
                    maps['myusers'].append(um)
                except Exception as e:
                    log.info('Exception in group processing - %s' % (e))
                    continue

        if len(rm.maps) == 0:
            log.info('No user group data found on %s ' % (device.id))
            return None

        # Add the rm relationships to maps['myuserGroups']
        maps['myuserGroups'].append(rm)

        # Need this complicated setup with maps = collections.OrderedDict and the chain return
        #   to ensure that relationship maps are applied in the correct order.  Otherwise there tend
        #   to be issues trying to create relationships on objects that don't yet exist
        return list(chain.from_iterable(maps.itervalues()))
"modeler/plugins/cmd/UserGroupMap.py" [Modified][readonly] 154 lines --72%--                    112,0-1       63%
```

Figure 65: UserGroupMap modeler - main body of process function

Note in Figure 65:

- It is good practice to code for possible errors using the Python *try..except* construct. This allows a modeler to fail nicely and log a warning. Typically, execution of the loop will continue.

```
except Exception as e:
  log.info('Exception in group processing - %s' % (e))
  continue
```

- The *ug* variable holds a single line of user group data. This is split into a **list** of fields, in the *ugList* variable.. Note that a Python list indexes from 0; hence the list elements are:
  - 0 user group name
  - 1 password field - not used
  - 2 GID
  - 3 secondary users - string of users, comma separated

- Each *UserGroup* object has an *id* field (which is inherited as part of the *Device* class). It is good practice to ensure that *id* fields are unique using the Zenoss utility, *prepId*, which can be used to check that any "unsafe" characters are replaced with an underscore. "Safe" characters are defined in *prepId* as:

```
a-z  A-Z  0-9  - _ , . $ ( )
```

  prepId is imported at the top of the plugin file.

- *om=self.objectMap()* is called to instantiate an **instance** of a *UserGroup* object. The *modname* line at the top of the modeler plugin class, defines the class of the object to be instantiated. ***o****bjectMap* (lower-case "o") is a method on *self* (the modeler plugin class)

- The **attributes** of the object (defined in the *properties* statement in *zenpack.yaml)*, can be assigned to the object instance, using the property ***name***:

```
om.groupName = ugList[0]
om.GID = int(ugList[2])
```

- Note that type definitions must match up between *zenpack.yaml* and the plugin code; otherwise errors will occur when the modeler executes. The *GID* attribute was defined as an *int* so the Python built-in function of *int* is used to convert the string value of the second element of the list, into an integer.

- When all the attribute assignments are complete, the object map, *om*, is appended to the relationship map list, *rm*:

```
rm.append(om)
```

- The *getUserMap* function is called to determine users that have this group as their primary user group. *getUserMap* is passed several parameters:
  - The *device* object
  - *lines[1]* which contains all the output from the *getent passwd* command
  - The GID of the current user group being processed, as *int(ugList[2])*
  - The user group name, passed as *ugList[0]*

- The **component relationship** name in the format *userGroups/<user group id>*, eg. *userGroups/audio*. This provides the hierarchical link from the *UserGroup* object, following the *userGroups* relationship, specifying the instance of the relationship through the user group *id*. Relationship names must match those defined in *zenpack.yaml*.
- *log* is also passed as a parameter to enable logging to continue to take place.

```
um = (self.getUserMap( device, lines[1], int(ugList[2]), ugList[0],
       'userGroups/%s' % ug_id, log))
```

- The *getUserMap* function returns a relationship map of *User*s for this *UserGroup*, which is appended to the list of user maps:

```
maps['myusers'].append(um)
```

- Having cycled through each user group entry, check whether the length of the user group relationships list of maps is zero; if so, no users or groups have been found so report this into the log file and return a *None* value from the modeler plugin:

```
if len(rm.maps) == 0:
 log.info('No user group data found on %s ' % (device.id))
 return None
```

- Otherwise, add the user groups relationship map list to *maps['myuserGroups']*

```
maps['myuserGroups'].append(rm)
```

- Use the imported chain function to return the list of maps

```
# Need this complicated setup with maps = collections.OrderedDict and the
# chain return to ensure that relationship maps are applied in the correct
# order. Otherwise there tend to be issues trying to create relationships
# on objects that don't yet exist
return list(chain.from_iterable(maps.itervalues()))
```

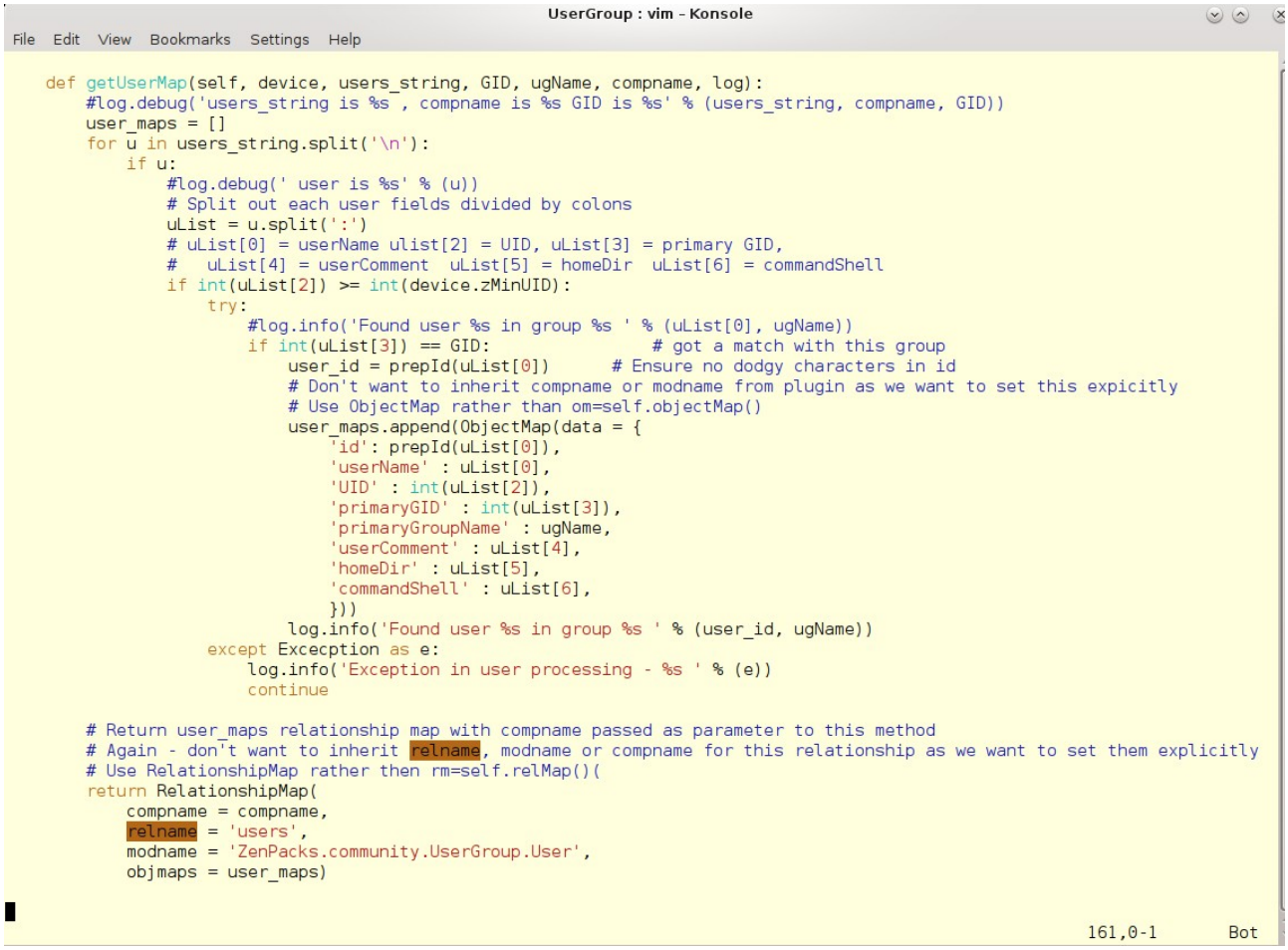


Figure 66: UserGroupMap plugin - getUserMap function

The *getUserMap* function cycles through each user record, to determine whether a user's primary GID matches the user group GID passed as a parameter.

In Figure 66 note:

- Split the users string on newline

  `for u in users_string.split('\n'):`

- For each user:
  - Split each user on colon, to a list for attribute processing:

    `uList = u.split(':')` producing:

    - 0 user name
    - 1 password (not used)
    - 2 UID (as a string)
    - 3 primary GID
    - 4 user comment
    - 5 home directory
    - 6 command or shell

- Test that the *UID* is not less than the *zMinUID* zProperty for the device; both *zMinUID* and the third element of the *uList* are actually string types.

```
if int(uList[2]) >= int(device.zMinUID):
```

- Note the *try..except* construct around the main body of the loop
- If the third element of the *uList* (primary GID) is the same as the GID parameter passed to this function, then:

♦ Use *prepId* to ensure a "safe" string for the *id* attribute

♦ Use ***ObjectMap*** (not ***objectMap***) to create a *User* object instance, with attributes (properties defined in *zenpack.yaml*) set from elements of *uList*. *ObjectMap* (upper-case "O") is a **protobuf;** a dictionary of raw data and has **no** concept of inheriting *relname, modname, compname* or *classname*.

♦ Note that the command output for user data does **not** include the user group name, only the primary user group GID. The group name **does** appear in the group raw data so is passed to this function as a parameter to populate the *primaryGroupName* attribute defined in *zenpack.yaml*.

♦ Append the ObjectMap to a *user_maps* list.

- Finally, return the users sub-component relationship map for the user group **component instance** (passed in as a parameter eg. *userGroups/audio*), for the relation *users* using the object class definition found in *ZenPacks.community.UserGroup.User* (constructed automatically by zenpacklib).

```
# Return user_maps relationship map with compname passed as parameter to
#  this method. Again - don't want to inherit relname, modname or compname
#  for this relationship as we want to set them explicitly
#  Use RelationshipMap rather then rm=self.relMap()(
return RelationshipMap(
  compname = compname,
  relname = 'users',
  modname = 'ZenPacks.community.UserGroup.User',
  objmaps = user_maps)
```

8.7.3 Testing the modeler

If a test device already exists in */Server/Linux/UserGroup* then move it to a different class such as */Ping*.

If a new modeler has been created then the ZenPack should be reinstalled and Zenoss completely recycled. Note that this is a **reinstall** of the ZenPack, which will be reported. Under the covers, a ZenPack remove, followed by a ZenPack install will be executed but objects will **not** be removed from the ZODB.

If the */Server/Linux/UserGroup* Zenoss device class has been **manually** removed, then you will see the message below as the ZenPack remove method attempts to remove the class defined in *zenpack.yaml* but finds it missing. The message is benign.

```
zenpack --link --install ZenPacks.community.UserGroup

INFO:zen.ZenPackCMD:Previous ZenPack exists with same name ZenPacks.community.UserGroup
```

```
WARNING:zen.zenpacklib:DeviceClass /Server/Linux/UserGroup has been removed at some point
after the ZenPacks.community.UserGroup ZenPack was installed. It will be reinstated if
this ZenPack is upgraded or reinstalled
INFO:zen.ZenPackCMD:installing zenpack ZenPacks.community.UserGroup; launching process
INFO:zen.zenpacklib:Creating DeviceClass Server/Linux/UserGroup
INFO:zen.zenpacklib:Setting zProperty zCollectorPlugins on Server/Linux/UserGroup
INFO:zen.zenpacklib:Setting zProperty zDeviceTemplates on Server/Linux/UserGroup
INFO:zen.zenpacklib:Setting zProperty zSshConcurrentSessions on Server/Linux/UserGroup
INFO:zen.zenpacklib:Setting zProperty zPythonClass on Server/Linux/UserGroup
2016-04-14 18:22:15,650 INFO zen.HookReportLoader: Loading reports from
/z/zenpacks/ZenPacks.community.UserGroup/ZenPacks/community/UserGroup/reports
```

For subsequent “tweaks” to an existing modeler, some changes can be accommodated simply by rerunning *zenmodeler* (and a new *.pyc* compiled Python file will be generated). Reinstallation of the ZenPack is **not** required.

cmd.UserGroupMap was added to the *zCollectorPlugins* zProperty for the class */Server/Linux/UserGroup*, in *zenpack.yaml*.

The first check towards success is if the modeler does appear in the Available list of plugins for the class. If it doesn't there is probably a syntax error in the modeler code.

Once the plugin is applied to the Class, move the test device to the class and run *zenmodeler* from the command line, just specifying the new plugin for collection. So for a test device, *taplow-30.skills-1st.co.uk:*

```
zenmodeler run -v 10 -d taplow-30.skills-1st.co.uk --collect cmd.UserGroupMap
```

To redirect output to a file, append:

```
> /tmp/fred 2>&1
```

Note that you need to redirect stderr to stdout (*2>&1*) or you won't see what you need; then inspect */tmp/fred*.

If there are communication errors in the logfile, are *zCommandUsername*, *zCommandPassword* and *zKeyPath* correctly configured for the test device?

Success is when the test device has User Groups and Users in the left-hand menu, each populated with the correct instances and the instances have the correct attributes.

Typically, a certain amount of “tweaking” is required to get attribute fields to an optimum length. Fortunately only the zope daemon needs to be recycled to achieve this. Don't forget to also refresh the browser page.

8.7.4 Where do things go wrong with modelers?

1. Modeler plugin does not appear in GUI list of Available modelers
   a. This is probably a syntax error in the plugin code. If using pyflakes with *vi*, check carefully for errors.
   b. Check particularly for unmatched / missing quotes
   c. Check for lack of ending colon (:)
   d. Check for white-space indentation errors

e. Try importing the modeler into *zendmd*; this will show syntax errors. Note that you need to specify an **object** path to the Python source file, not a **file** path.

```
import ZenPacks.community.UserGroup.modeler.plugins.cmd.UserGroupMap
```

f. Watch for yellow highlighted messages when using a *Modeler Plugin* menu

g. Check *$ZENHOME/log/event.log* for error messages. For example:

```
  File
"/z/zenpacks/ZenPacks.community.UserGroup/ZenPacks/community/UserGroup/mode
ler/plugins/cmd/UserGroupMap.py", line 45', ' def process(self, device,
results, log)', ' ^', 'SyntaxError: invalid syntax')
```

h.

2. Modeler fails

a. Check *zenmodeler* output carefully. From the GUI *Model Device* menu, check whether the modeler appears in the list of modelers to be applied. If it is missing, suspect syntax errors. For example:

```
2016-04-19 17:36:38,150 INFO zen.ZenModeler: Using SSH collection method
for device taplow-30.skills-1st.co.uk
2016-04-19 17:36:38,151 INFO zen.ZenModeler: plugins: cmd.UserGroupMap
```

b. If it appears, check the subsequent output for messages.

c. Run *zenmodeler* standalone in debug mode, optionally sending output to a file:

```
zenmodeler run -v 10 -d taplow-30.skills-1st.co.uk --collect cmd.UserGroupMap

zenmodeler run -v 10 -d taplow-30.skills-1st.co.uk --collect cmd.UserGroupMap \
             > /tmp/UserGroupMap.out 2>&1
```

d. Check that the command in the modeler can be run by the *zCommandUser* over ssh, from a command line interface. If *zCommandUsername* is *zenplug*, and the *zenoss* user's public key on the Zenoss server has been copied to the target machine's *authorized_keys* file in *zenplug*'s *.ssh* directory and there is no passphrase generated for the public key so *zCommandPassword* is null, the following test should work:

```
ssh zenplug@taplow-30.skills-1st.co.uk getent group
```

e. Note that it is important to test ssh to each target **directly** as the first communication will generate the host fingerprint entry and ask whether to add it to the *known_hosts* file in the *zenoss* user's *.ssh* directory. Zenoss modelers will probably fail if asked this question.

i. Note on Zenoss 5 the test must be performed from the *zencommand* **container** as the *.ssh/known_hosts* inside the container is **not** the same as that for the *zenoss* user on the base host.

```
serviced service attach zencommand su zenoss -l
ssh -l zenplug zenny2.class.example.org
cat .ssh/known_hosts
```

f. If a relationship is not created then check relationship names in the plugin and *zenpack.yaml*, for both device and component.

g. If relationship i**nstance(s)** are not created, check:

i. *relname* and *modname* statements in modeler plugin exists

ii. *relname* and *modname* are correct (especially case-sensitivity)

h. If one or more attributes do not have values:

i. Check spelling of attributes in plugin and *zenpack.yaml*.

ii. Check in the modeler log that data is collected

iii. Check type of attributes eg. *string* data assigned to *int* defined attribute

3. Components do not appear or existing components are removed

a. With RelMaps, the RelMap returned for any given relation needs to contain an ObjectMap for each component that exists in that relation, **whether or not that component has changed**, because applyDataMaps will remove any components not represented in the RelMap. This means returning an empty RelMap (which some people do in an error case) will have the likely undesired side-effect of removing all components from that relation.

4. Insert extra *log.debug* statements in the modeler code and rerun the *zenmodeler* command with the *-v 10* flag. No reinstall of the ZenPack is necessary. For example:

```
log.debug('GID is %s ' % (om.GID))
```

a. *log* has a number of logging levels which should be used sensibly. Everything other than *debug* will typically be shown in the GUI modeler output.

i. log.debug

ii. log.info

iii. log.warning

iv. log.error

v. log.critical

b. *log.debug* lines should largely be commented out once testing is complete.

8.8 * Renderers

The class property and the Relationship Override fields may use a ***renderer*** keyword; the default is *None*. This presents an opportunity to control how an item is displayed. renderer code is defined in a JavaScript file, either in the Zenoss core code or supplied with a ZenPack.

As an example, a new attribute will be added to the *UserGroup* object, ***hasSecondaries***. It would normally make sense to define this as a boolean in *zenpack.yaml*, but it will actually be a string.

```
hasSecondaries:
 type: string
 label: Has Secondaries?
 short_label: Has Sec?
 label_width: 40
 renderer: Zenoss.render.severity  # Use event severity renderer
 order: 3.4
```

Note the ***renderer*** line specifying *Zenoss.render.severity*.

Zenoss supplies lots of renderers with the core code; inspect *$ZENHOME/Products/ZenUI3/browser/resources/js/Renderers.js*. The ***severity*** renderer produces the different colored icons in the Event Console display, denoting different severities of events; critical =red, through to clear=green. The renderer simply takes the value of the "severity" string and produces a small icon of the correctly mapped color.

```
severity: function(sev) {
  return '<div class="severity-icon-small '+
    Zenoss.util.convertSeverity(sev) +
    '"'+'><'+'/div>';
}
```

The *convertSeverity* function is in *$ZENHOME/Products/ /ZenUI3/browser/resources/js/zenoss/zenoss.js* and returns the severity string in lower case.

```
Zenoss.util.convertSeverity = function(severity){
  if (Ext.isString(severity)) return severity;
  var sevs = ['clear', 'debug', 'info', 'warning', 'error', 'critical'];
  return sevs[severity];
};
```

It is not necessarily important to understand these details; the crucial point is that a different colored icon can be produced for an object attribute, whose value is coded to be one of 'clear', 'debug', 'info', 'warning', 'error' or 'critical'. (If the attribute does not match any of these exact strings then no icon is rendered).

The modeler plugin must also be updated to populate the *hasSecondaries* attribute.

```
om.secondaryUsers = ugList[3]
# hasSecondaries takes string value that matches an event status so that
#  we can cheat and use Zenoss.render.severity to give icons for this value
#  If secondaries exist then we get the green 'clear' icon. Otherwise red.
if ugList[3]:
  om.hasSecondaries = 'clear'
else:
  om.hasSecondaries = 'critical'
```

Any standard Zenoss renderer can be coded in a *renderer* keyword in *zenpack.yaml*.

Having added a new object attribute in a *properties* statement in *zenpack.yaml*, the ZenPack must be reinstalled and Zenoss restarted entirely.

Remodel the test device to populate and display the new *hasSecondaries* attribute.

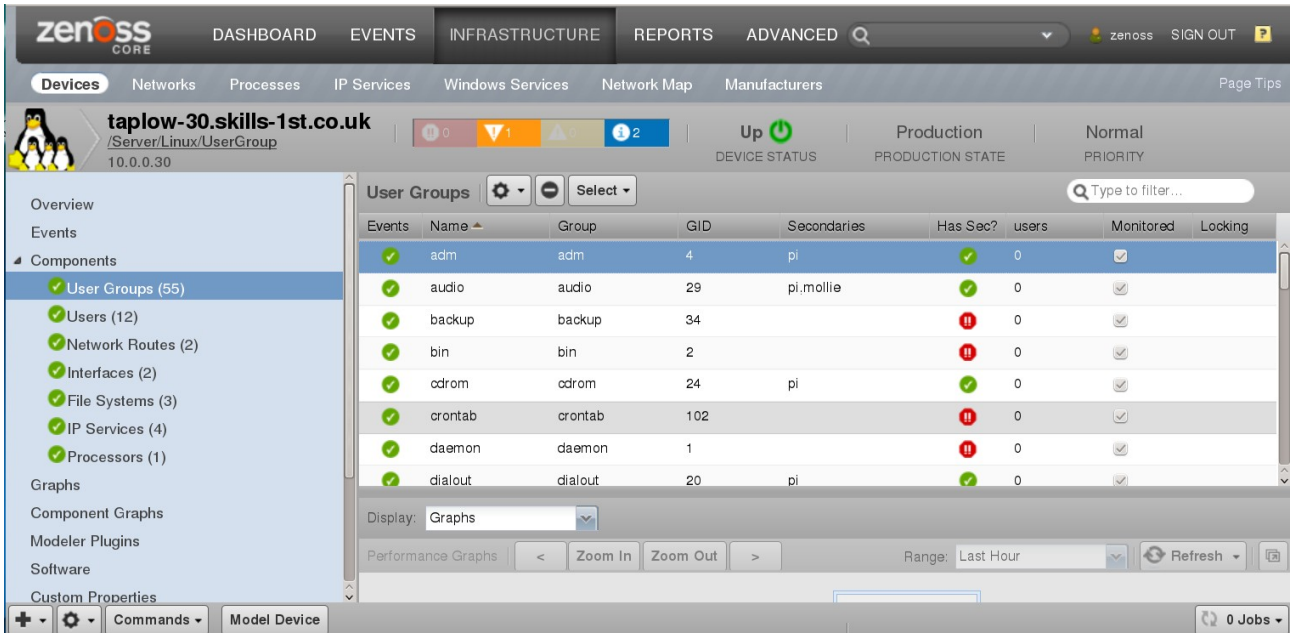


Figure 67: UserGroup component displaying colored icon for hasSecondaries attribute

The red icon denotes no secondaries; green represents that secondaries do exist.

8.9 Templates and zenpacklib

Pre-zenpacklib, performance templates were created using the GUI and then added to a ZenPack using the *Add to ZenPack* menu. When the ZenPack is exported, the template, including any datasources, datapoints and graphs, are written to *objects.xml* under the *objects* directory for Zenoss 4, or under *objects/templates* for Zenoss 5, with one xml file per template.

zenpacklib offers an alternative method to ship performance templates; they can be defined as part of the definition of a Zenoss device class, including datasources, datapoints, thresholds and graphs. The limitation is that templates **must** be defined as part of a Zenoss device class, whether they are device-level templates or component templates; see http://zenpacklib.zenoss.com/en/latest/yaml-monitoring-templates.html in the zenpacklib documentation for more detail. This documentation highlights the difference between template location and binding.:

- **Location** is the device class in which a monitoring template is contained (and nothing whatsoever to do with the Zenoss Location groupings that can be used in the GUI). This is determined when the template is created by associating it with the highest point in the Zenoss device class hierarchy tree where this template **may** be bound.
- **Binding** is the device class, device or component to which a monitoring template is bound.
  - Device-level templates are bound using the standard *zDeviceTemplates* zProperty (which is a list), to bind to either a Zenoss device class or, in the GUI, to a specific device. Note that this property must contain **all** device-level templates to be bound; specifying a single new template in the *zDeviceTemplates* field of a *zenpack.yaml* definition, **will remove any existing templates**.
  - Component templates must **not** be specified in *zDeviceTemplates*. Component templates are **automatically** associated with Python component objects. The best way to understand what template applies to a component is to navigate to a device

detail page, select a component and change the dropdown menu from *Graphs* to *Templates*.

- Pre-zenpacklib, component templates are generally associated with a component Python object class of **exactly** the same name, though that can be overridden in code.

- With zenpacklib, the automatic association is with the **label** field of the component class (**not the object class name,** if different).

- Alternatively, zenpacklib offers the *monitoring_templates* keyword that can be used with component templates, to specify one or more templates to be automatically associated. *monitoring_templates* expects a value of type **list**. This way, any appropriate template can be associated with a component, regardless of the label name.

To define templates associated with a Zenoss device class in *zenpack.yaml*. use the ***templates*** keyword. Note that this keyword is **only** valid on Zenoss device classes, not on object class stanzas. The zenpacklib documentation only lists *SNMP*, *COMMAND* and *PING* as valid datasource types for templates but any valid datasource type can be specified.

There are advantages and disadvantages with defining templates in a *zenpack.yaml* file. The definitions are fairly straightforward but tedious to create; however they can easily be modified. Often, the easiest way to start building a template is to build it through the GUI and then use zenpacklib to dump the correct yaml definitions. Similar templates can be added to *zenpack.yaml* using cut-and-paste editor techniques.

8.9.1 Creating a User component template with the GUI

The ZenPack requirements specify the creation of a graph for each user sub-component, with a count of the number of user groups that the user is a member of.

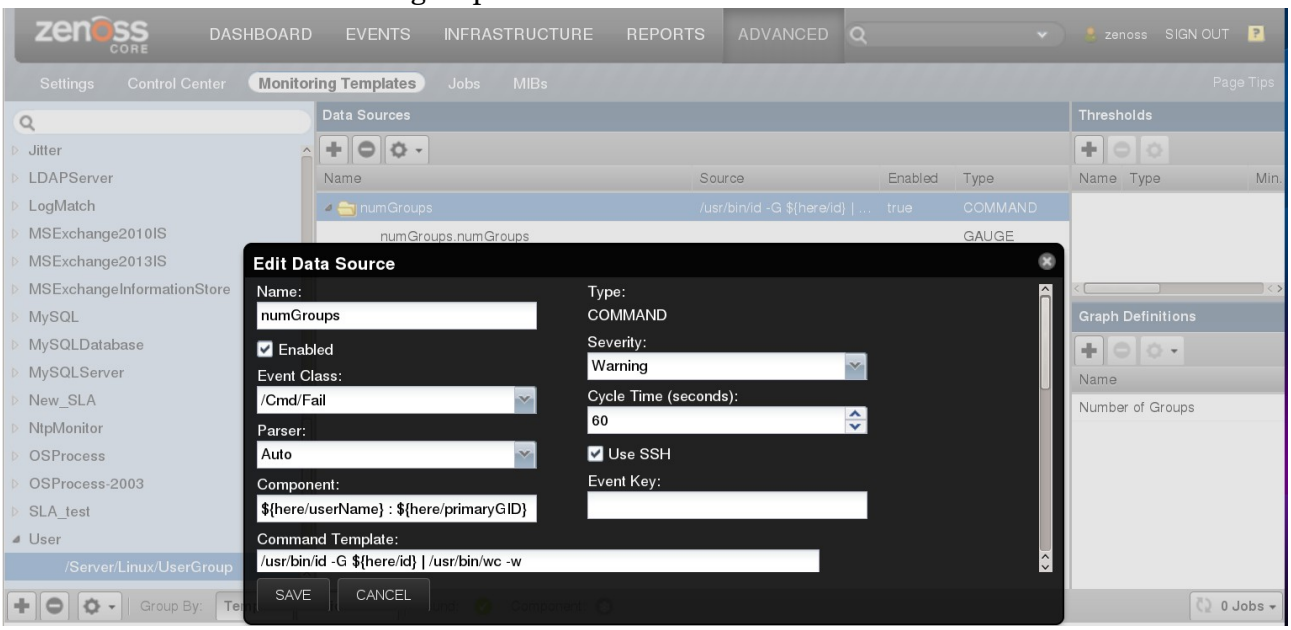


Figure 68: User component template with COMMAND datasource for group membership

A template called *User* should be created whose "location" is */Server/Linux/UserGroup*. The group membership information will be gathered using a command over an ssh session; thus

the template needs a datasource created, *numGroups*, of type *COMMAND* (the datasource name does **not** have to match up with any other definitions).

Ensure that the *Use SSH* box is ticked.

The *Cycle Time* is set to 60 seconds for testing; this should be increased when the ZenPack is in production.

Note that changing the cycle time in Zenoss 4 will prevent any further data from being collected; RRD data files should be deleted (or hidden) from the *$ZENHOME/perf/Devices* directory hierarchy, and allowed to recreate with the new cycle time. With Zenoss 5, changes to the cycle time are accommodated seamlessly at the next cycle time.

The command will use the unix *id* utility with the *-G* parameter to gather all groups for the specified user. The output is piped into the word count (*wc*) utility to get the count.

```
/usr/bin/id -G ${here/id} | /usr/bin/wc -w
```

Note that fully-qualified path names are used for *id* and *wc;* otherwise these utilities will be assumed to be in the *zCommandPath*-specified directory on the target host.

The template will be run against each component object instance, which will be available to the template as the *here* variable; hence the command has *${here/id}* to be substituted into the command for each user component.

It is good practice to specify the *Component* field of the datasource, which will appear in any event that is generated by this template; *${here/id}* is often a good choice, though any legal attribute (or combination of the attributes) of the user object can be used; for example:

```
${here/userName} : ${here/primaryGID}
```

A *COMMAND* datasource has a *Test Against a Device* button; note that this does **not** work for many component templates and definitely does not work when the command is run remotely over ssh.

The datasource will deliver a **single** value for the number of groups. A *GAUGE*-type datapoint. *numGroups*, can be created which will automatically receive this value. Note that if more than one value is delivered by the datasource command then a more complex design is required.

A graph definition, *Number of Groups*, is created to display the single datapoint.

This template should **not** be bound to any device; as a component template it will be **automatically** associated with any component object whose name (label in *zenpack.yaml*) matches the template name.

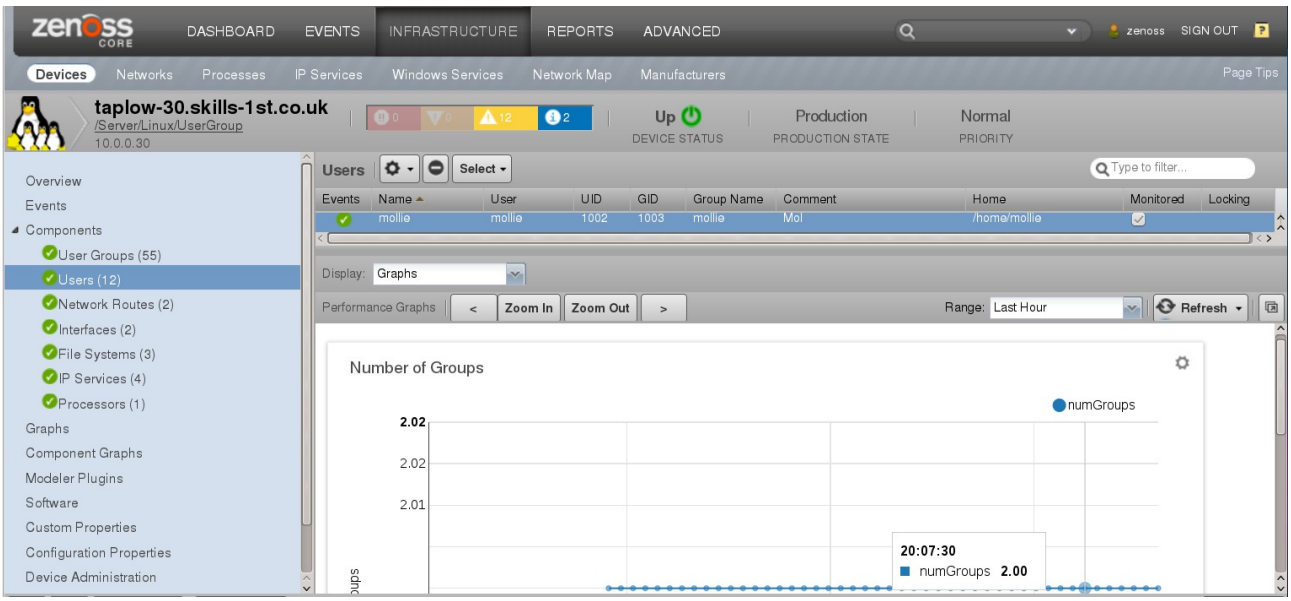


Figure 69: Number of Groups graph for user mollie

8.9.2 Exporting templates with zenpacklib

zenpacklib provides a means of exporting templates from an existing ZenPack, provided such templates are part of the ZenPack and the device class defined as its location is also **explicitly included** in the ZenPack. Use the *Add to ZenPack* menu option for both the template and Zenoss device class.

If the ZenPack does not include the containing Zenoss device class then only the ZenPack name will be output.

Note that it is **not** necessary to export the ZenPack having added the device class and template. Once the template has been output, the device class and template should be removed again from the ZenPack.

The *dump_templates* parameter of zenpacklib is used as follows:

```
zenpacklib dump_templates ZenPacks.community.UserGroup > UGTemps.yaml     (Z5)
zenpacklib.py dump_templates ZenPacks.community.UserGroup > UGTemps.yaml   (Z4)
```

Templates are output to Unix stdout (ie the screen) so redirect the output to a temporary file, **not** your main *zenpack.yaml*; it will go to the current directory. This output then needs incorporating into the main *zenpack.yaml,* under the appropriate class.

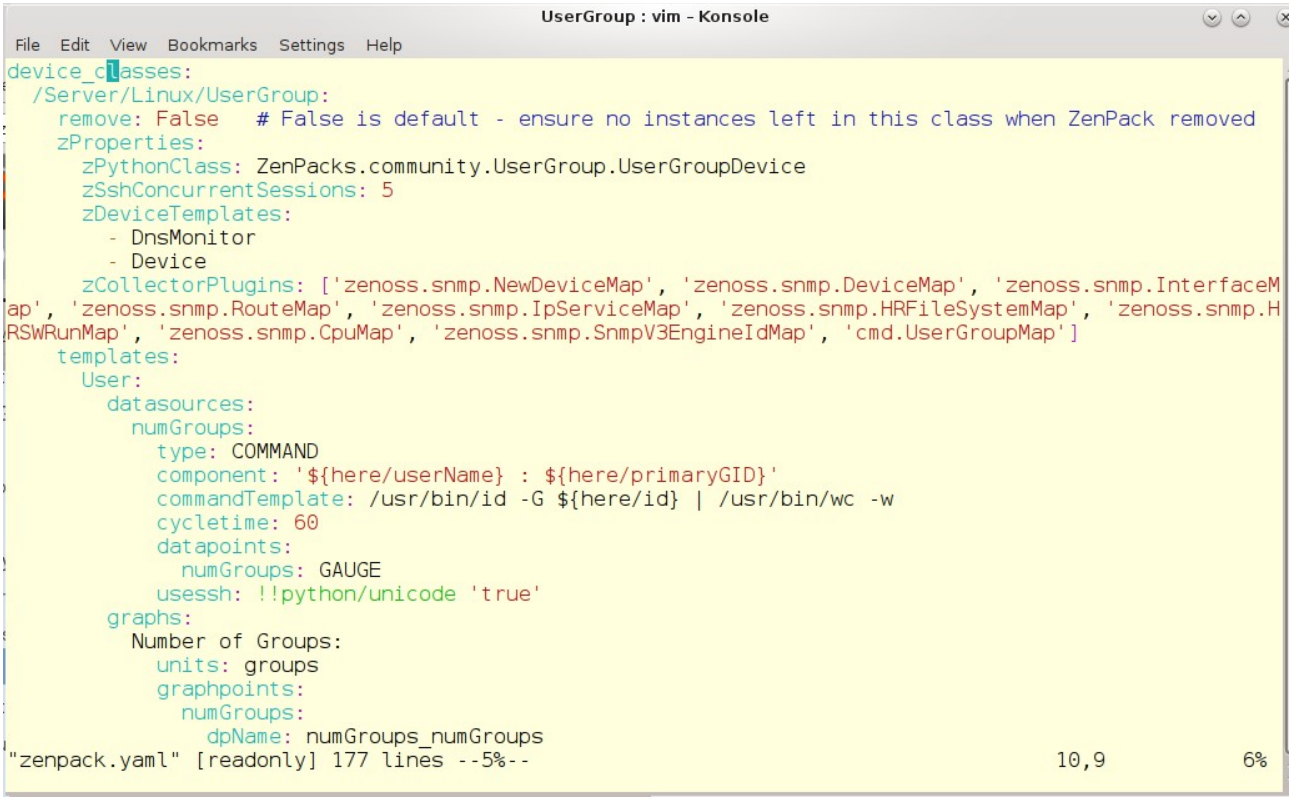


Figure 70: zenpack.yaml - /Server/Linux/UserGroup class with template

Note in Figure 70 that the *DnsMonitor* **device** template has been added to the *zDeviceTemplates* zProperty list, in addition to defining the **component** *User* template. The **order** of the templates in *zDeviceTemplates* determines the order of graphs that will be displayed in the GUI.

When the amalgamated *zenpack.yaml* is complete, the ZenPack should be reinstalled and Zenoss completely restarted.

If there are templates in *zenpack.yaml* that replace existing templates in ZODB, then an *ERROR* message is generated, which seems excessive as the new template **will** be installed and the existing ZODB template will be renamed, as reported

Figure 71: ZenPack installation where yaml template definition differs from ZODB

Differences between the new and existing templates are shown in *diff* format and the old template is saved under a different name. Once the yaml versions of the templates are proven, the old templates, which have a suffix of the epoch time when created, should be deleted manually in the GUI.

In Figure 71 the ZODB template had the *Severity* field changed from *Warning* to *Error*.

Note that changes to **existing** template definitions in *zenpack.yaml* only requires zope to be restarted.

Note one issue with exporting templates is that a template description field tends to have two single quotes representing the empty string; *zenpack.yaml* requires double-quotes, otherwise subsequent lines are all interpreted as comment. The Unix *vi* editor provides automatic color coding for files with a yaml suffix, which helps spot this; red text denotes “quoted”.

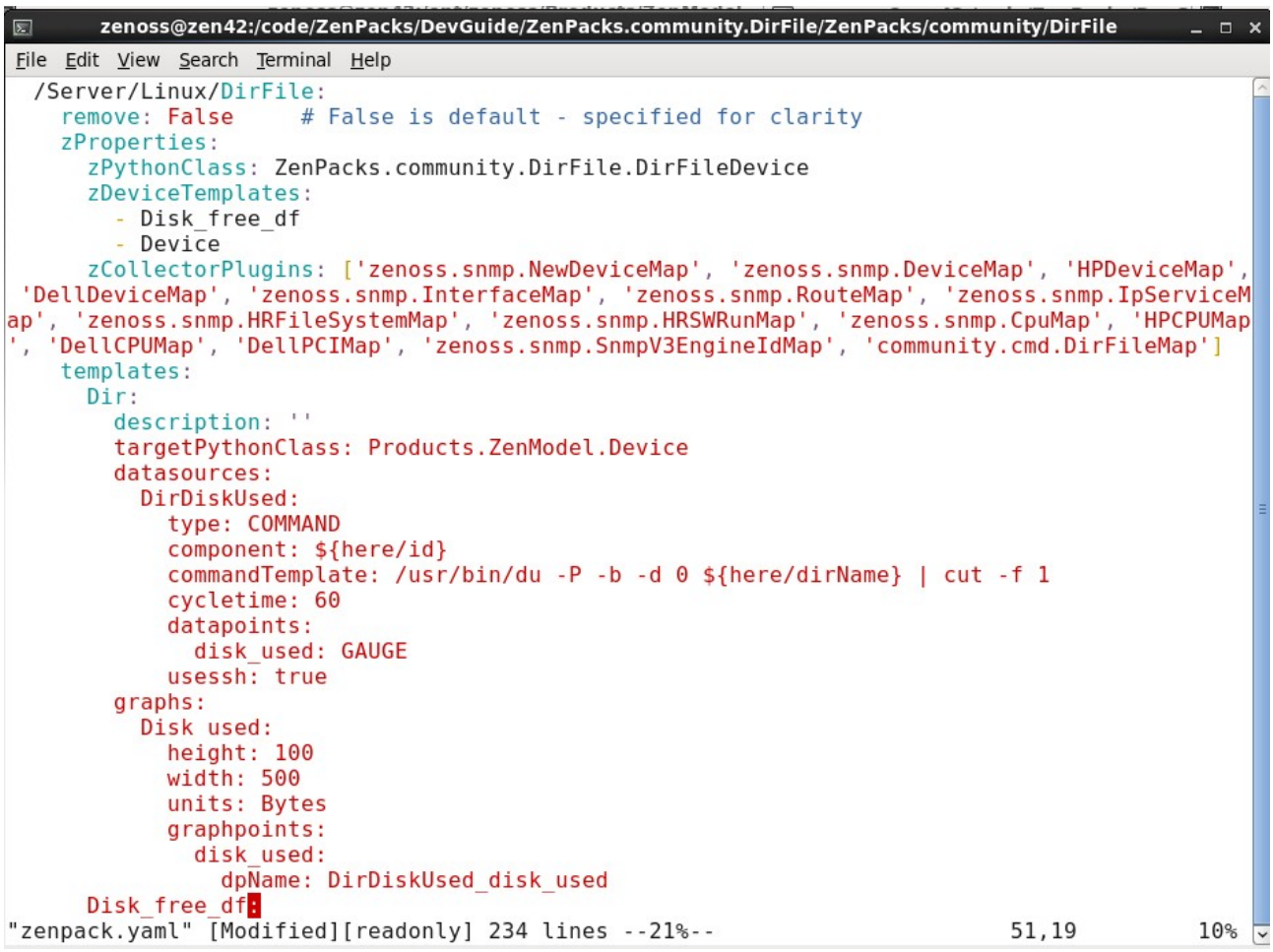


Figure 72: zenpack.yaml with incorporated templates - note red lines denoting comments

Changing the two single quotes for the description field to two double quotes, resolves the issue.

8.10 * Creating object methods with zenpacklib

When creating object classes for devices and components, most properties for these classes are **attributes** - characteristics of the object. It is also possible to work with **methods** - actions to be executed against the object.

The zenpacklib documentation at http://zenpacklib.zenoss.com/en/latest/yaml-classes-and-relationships.html provides two keywords:

- **api_backendtype**
  - Implementation style for the property if *api_only* is true. Must be *property* or *method*. Default is *property*.
- **api_only**
  - Should this property be for the API only? The property or method (according to api_backendtype) must be manually implemented if this is set to true.

Consider the UserGroup ZenPack scenario. The raw data from the Linux *getent passwd* command provides user information, including primary group; it does **not** provide all secondary groups for a user. The *getent group* command provides all user groups, including a field for users for whom this group is a secondary.

The modeler plugin has gathered all the relevant information into the object model in the ZODB; however, no attribute answers the question for a user “What secondary groups am I in?”.

8.10.1 Writing methods for objects

A Python method can be written which, given the user object instance, can determine the secondary groups.

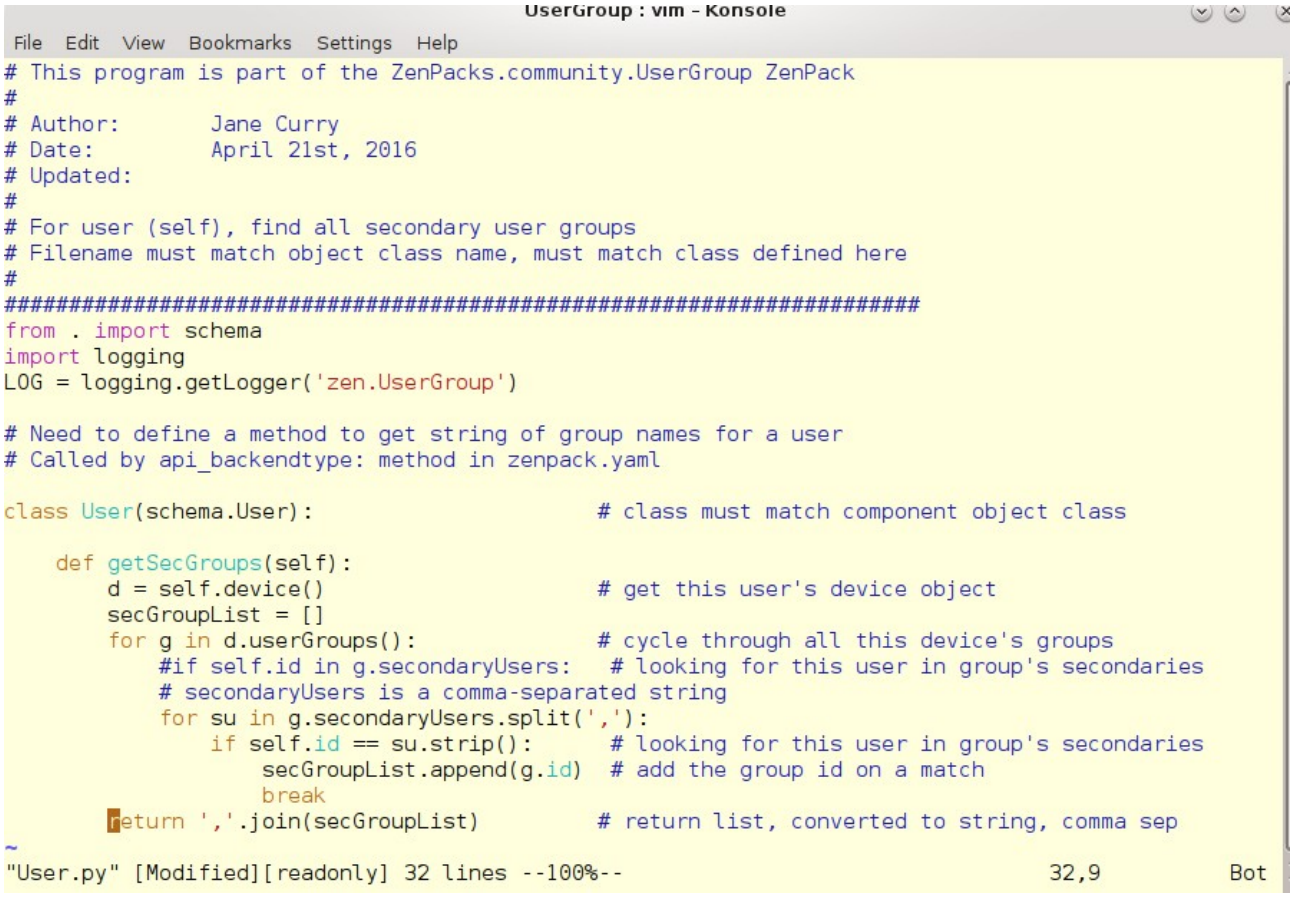


Figure 73: getSecGroup method for User class

In Figure 73:

- The class name (*User*) must match the sub-component object class name defined in *zenpack.yaml*. The filename must also match the classname, with *.py* appended.
- “self” will be an instance of the *User* object class.
- Note the *from . import schema* line. This is mandatory and will import from the code automatically generated by zenpacklib, held in memory.
- The function name, *getSecGroups*, must match with a stanza under *properties*, for the *User* object, whose name matches *getSecGroups* and whose *api_backendtype* is *method*.
- Any component class must have a method called *device()*, that returns the containing device object. If there is a component hierarchy then each sub-component follows the ToOne relationship until the device is reached:

```
For a UserGroup:
       def device():
             return self.userGroupDevice()

For a User sub-component:
       def device():
             return self.userGroup().device()
```

These *device* methods are constructed automatically by zenpacklib, in memory.

- The *getSecGroups* method calls *device()* and then uses the ToMany relationship, *userGroups*, to cycle through all user groups for that device, checking whether any of the group's *secondaryUsers* is the same as the *id* field of the user; if so, the group *id* is added to a list.

- The function returns a string that concatenates the discovered group ids, joined by a comma.

zenpack.yaml needs a matching definition for this method.

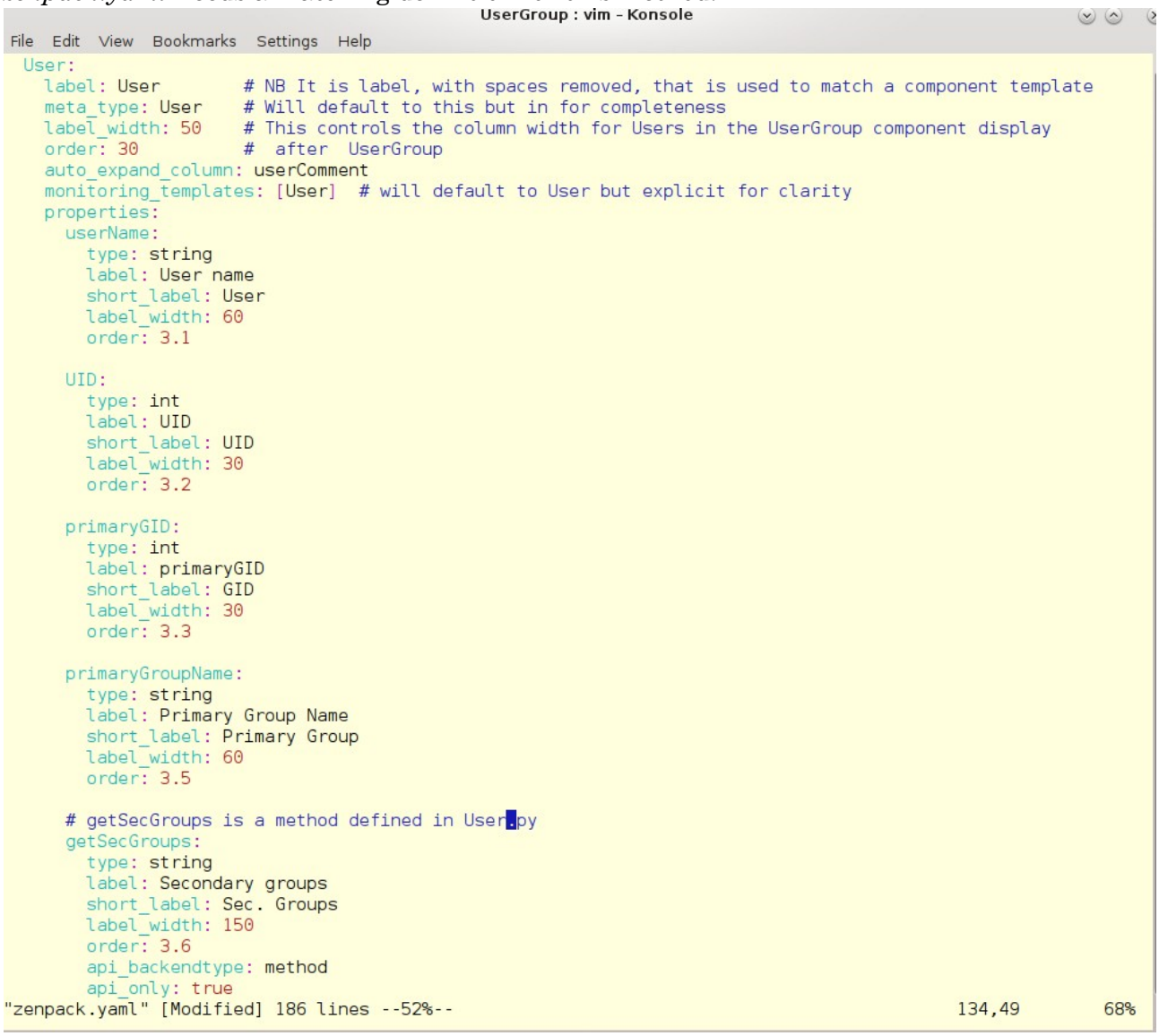


Figure 74: zenpack.yaml with getSecGroups method for User sub-component

Other keywords for the *getSecGroups* method property are defined as for attribute properties.

TODO: What does the *api_backend* keyword really do? Tried with both true and false and zendmd still sees the method, even after deleting the device and rediscovering.

The ZenPack should be reinstalled and Zenoss completely restarted after these modifications. Move the test device out of */Server/Linux/UserGroup* and then back in again to instantiate the new property. Remodel the device to see the extra *Sec Groups* property in both the component grid and the *Details* drop-down.

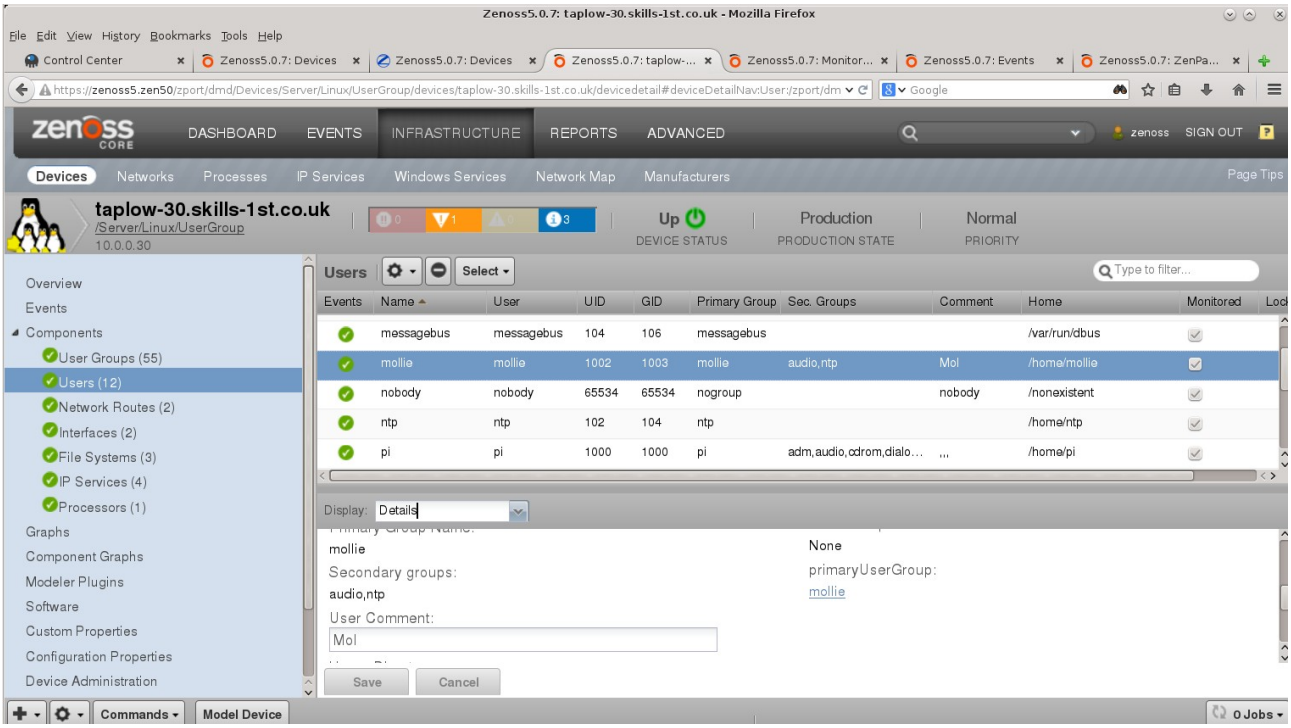


Figure 75: User mollie with Sec. Groups displayed in grid and Details dropdown

8.11 *Creating new components directly on Device object class

So far, this sample has relied on creating a new device object class of *UserGroupDevice* and a new Zenoss device class of */Server/Linux/UserGroup*, setting the *zPythonClass* property of */Server/Linux/UserGroup* to *ZenPacks.community.UserGroup.UserGroupDevice* to ensure that new attributes and components can be populated.

If these new attributes and components are fairly ubiquitous and such devices really should be placed in other specific device classes, perhaps provided by other ZenPacks, how can the new features be added directly to the standard Zenoss-supplied *Device* object class?

8.11.1 * zenpack.yaml modifications

zenpack.yaml needs modifying to eliminate the *UserGroupDevice* device object; simply comment out the entire *UserGroupDevice* definition, including its relationships stanza.

Add a new relationship, *userGroups* to the Zenoss-provided standard *Device* object, commenting out the previous version:

```
class_relationships:
 #- UserGroupDevice(userGroups) 1:MC UserGroup(userGroupDevice)
 - Products.ZenModel.Device.Device(userGroups) 1:MC UserGroup(uGDevice)
 - UserGroup(users) 1:MC User(primaryUserGroup)
```

Note that the relationship from the *UserGroup* object back to the containing device has been changed from *userGroupDevice* to *uGDevice*.

Note that the object path to the standard *Device* object class has to be provided relative to $ZENHOME - *Products.ZenModel.Device.Device*.

The Zenoss device class definition may be retained but the zPythonClass association should be removed:

```
device_classes:
 /Server/Linux/UserGroup:
  remove: False  # False is default - ensure no instances left in this
class when ZenPack removed
  zProperties:
   # Don't set zPythonClass - it should default to Device
   #zPythonClass: ZenPacks.community.UserGroup.UserGroupDevice
   zSshConcurrentSessions: 5
   zDeviceTemplates:
    - DnsMonitor
    - Device
   zCollectorPlugins: ['zenoss.snmp.NewDeviceMap', 'zenoss.snmp.DeviceMap',
'zenoss.snmp.InterfaceMap', 'zenoss.snmp.RouteMap',
'zenoss.snmp.IpServiceMap', 'zenoss.snmp.HRFileSystemMap',
'zenoss.snmp.HRSWRunMap', 'zenoss.snmp.CpuMap',
'zenoss.snmp.SnmpV3EngineIdMap', 'cmd.UserGroupMap']
  templates:
   User:
    datasources:
     numGroups:
```

The *UserGroup* component object class also requires a little modification as its relationship back to its containing device has changed:

```
UserGroup:
  ........
 relationships:
   #userGroupDevice:    # back to the containing device
   uGDevice:      # back to the os relationship on the containing device
    #label: userGroupDevice
    label: uG Device
    display: true     # Ensures relationship shown in Details dropdown
   users:          # down to User sub-component
    label: users
    display: true     # Relationship shown on grid and Details
```

Everything else in *zenpack.yaml* can remain the same; the relationship between UserGroups and Users is unchanged.

8.11.2 * Other modifications

No change is required to the *__init__.py* in the base directory of the ZenPack. In addition to creating the memory-held Python code representing the *UserGroup* and *User* objects, zenpacklib will also monkeypatch the standard *Device* object with the new *userGroups* relationship.

No change is required to the modeler plugin. The *userGroups* relationship that is populated by the modeler is now on on the *Device* object class, rather than the *UserGroupDevice* object class but the same relationship name was carefully retained.

One issue with this change is that device instances will not need to be in the */Server/Linux/UserGroup* Zenoss device class in order to gather User and Group components; however the *User* template defined in *zenpack.yaml* is **located** at */Server/Linux/UserGroup* - a template must be defined against a Zenoss device class, not an object class, and there is no way to add templates to pre-existing classes in *zenpack.yaml*.

zenpack.yaml includes a *monitoring_templates* definition for the *User* object but if the device instance resides in the */Server/Linux* class or other subclasses, then the *User* template will simply be ignored.

```
User:
 label: User    #
 meta_type: User  # Will default to this but in for completeness
 label_width: 50  #
 order: 30     # after UserGroup
 auto_expand_column: userComment
 monitoring_templates: [User] # will default to User but explicit for clarity
```

To retain the template for more universal use, use the GUI to copy the existing *User* template to a higher **location**, eg. */Server/Linux*. Use the *Action* icon at the bottom of the *Monitoring Templates* left-hand menu and choose *Copy / Override Template*. The template does not require changes unless you wish to do so; however, the */Server/Linux* version should be added to the ZenPack, through the GUI, in order to ensure that a version is available for all relevant classes. Don't forget to re-export the ZenPack to save this template under the *objects* file directory hierarchy. The *User* template does not need to be bound as it is a component template.

This version of the ZenPack is included on GitHub under the *device* branch and is Version 1.0.1 of the ZenPack.

8.11.3 * Testing the changes

Significant changes have been made to device objects and relationships. Move any existing devices in */Server/Linux/UserGroup* to a different temporary class such as */Ping*. The ZenPack must be reinstalled and everything restarted.

Move a test device to a new Zenoss device class (**not** */Server/Linux/UserGroup*). The *cmd.UserGroupMap* modeler plugin will need adding either to the device directly or to the new test device class.

Adjust the *zMinUID* property as required.

Remodel the device and check that User and User Group components are created. Check especially that a graph is created for each user, showing the number of groups that it is associated with.

A small detail that is lost with this scenario is that the icon at the top-left of a device panel is associated with a device object class, so the “four-tux” icon is no longer displayed.

8.11.4 * Binding device templates in __init__.py

Remember that *zenpack.yaml* added the *DnsMonitor* device-level template to the */Server/Linux/UserGroup* Zenoss device class, shown in Figure 70 on page 123. By adding the User and Group components directly to the *Device* object and removing the need for the */Server/Linux/UserGroup* device class, this template binding is lost.

The *__init__.py* in the base directory of the ZenPack can be used to perform various customisations, including template binding. For a good example, look at https://gist.github.com/James-Newman/9609c84688a0b9a4fee842878b9a5b00 which adds device classes, templates to device classes and event classes. This gist was designed for a non-zenpacklib ZenPack but it is fairly easily changed to work with zenpacklib.

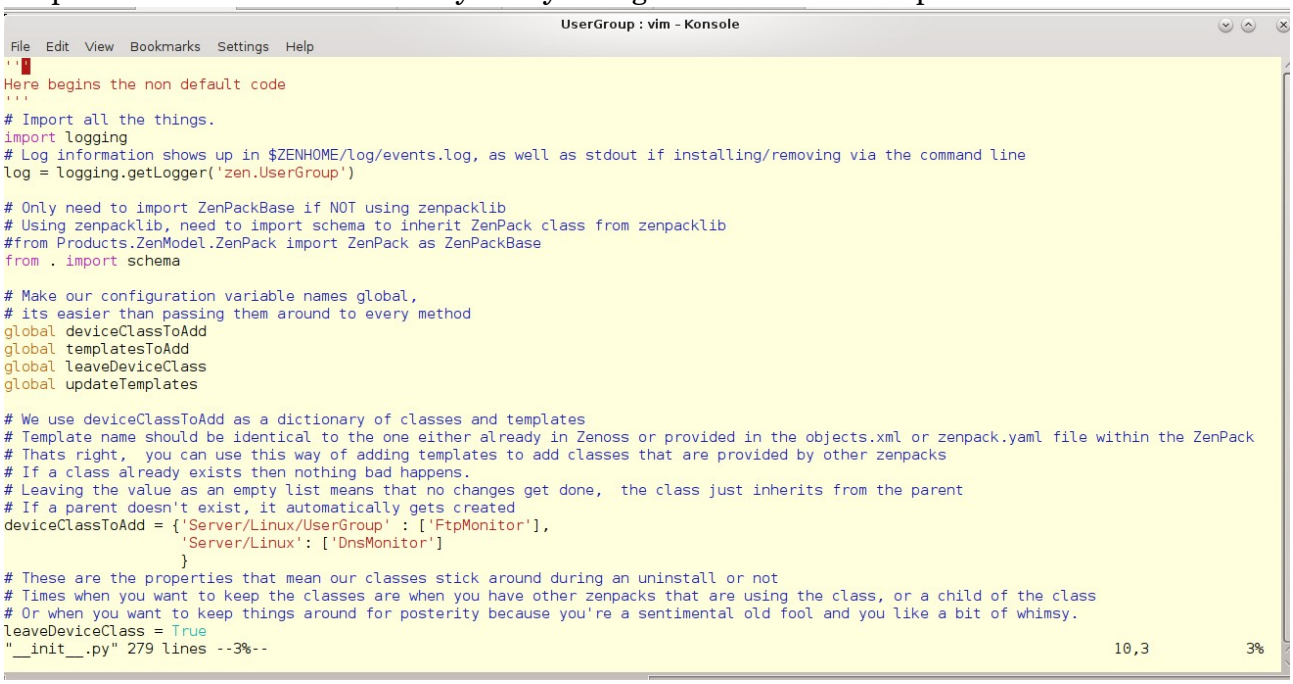


Figure 76: __init__.py to bind templates to devices - part 1

For zenpacklib scenarios, *schema* must be imported from the current directory, ".".

The *deviceClassToAdd* global variable is a dictionary where the keys are the Zenoss device class and the values are a list of templates to be added. As an example, the *FtpMonitor* template is added to */Server/Linux/UserGroup* and the *DnsMonitor* template is added to */Server/Linux*:

```
deviceClassToAdd = {'Server/Linux/UserGroup' : ['FtpMonitor'],
                    'Server/Linux': ['DnsMonitor']
                   }
```

If the device class does not exist (or any of its parents) then they will be added; if they do exist then no action is taken on the class other than adding the templates. Where zenpacklib is used, device classes should be added through *zenpack.yaml*.

The *leaveDeviceClass* global variable should probably be left as *True* to ensure that classes are not accidentally deleted when the ZenPack is removed.

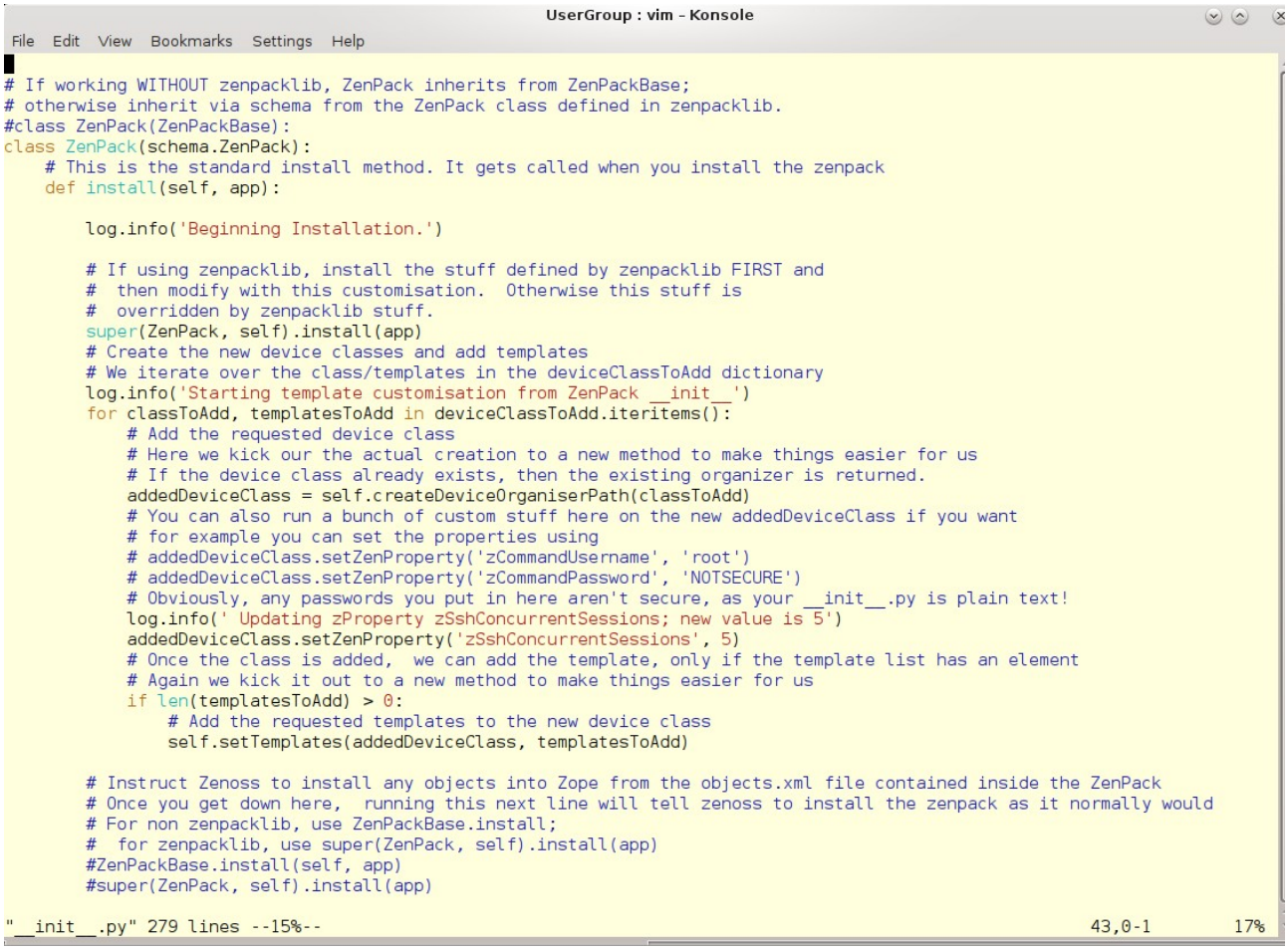


Figure 77: __init__.py to bind templates to devices - part 2, install method

The *__init__.py* needs to monkeypatch the *install* and *remove* methods of the ZenPack class. In a non-zenpacklib ZenPack, the *ZenPack* class inherits from the imported *ZenPackBase*; if zenpacklib is used, the *ZenPack* class must inherit from the *ZenPack* class defined in zenpacklib.

Note that the order of installation can be controlled. In Figure 77 the zenpacklib install method is run first with:

```
super(ZenPack, self).install(app)
```

The configuration in *__init__.py* is then run.

For each specified device class to be added or modified, the local *createDeviceOrganiserPath* method is run, returning the object representing the device class. It is perfectly benign to run this if the class already exists.

```
addedDeviceClass = self.createDeviceOrganiserPath(classToAdd)
```

The object for the device class can be used to update existing zProperties, if required. Ensure that the *setZenProperty* method is used.

```
addedDeviceClass.setZenProperty('zSshConcurrentSessions', 5)
```

Setting a property directly, such as:

```
addedDeviceClass.zSshConcurrentSessions = 5
```

can result in an inconsistent ZODB database.

The templates are updated from the *deviceClassToAdd* dictionary by calling the local *setTemplates* method with:

```
if len(templatesToAdd) > 0:
  # Add the requested templates to the new device class
  self.setTemplates(addedDeviceClass, templatesToAdd)
```

The remove method follows the same principles.

UserGroup : vim - Konsole

File Edit View Bookmarks Settings Help

```
    # This is the standard remove method
    # it gets called when you remove the zenpack
    # If the install method is just to add templates, not classes, ensure
    # that leaveDeviceClass=True so that classes are not removed, only templates
    def remove(self, app, leaveObjects=False):

        log.info('Beginning ZenPack removal.')

        log.info('Starting template customisation from ZenPack __init__')
        # Remove the device class, this ensures that we don't remove devices as well if we don't want to
        # Again we iterate over the device classes and templates that the zenpack adds
        for classToRemove, templatesToRemove in deviceClassToAdd.iteritems():
            deviceClassToRemove =  self.dmd.Devices.getOrganizer(classToRemove)
            if deviceClassToRemove:
                # Only if we're removing templates do we run the new method to remove templates from device classes
                if len(templatesToRemove) > 0:
                    self.removeTemplatesFromDeviceClass(classToRemove, templatesToRemove)
                # This check here is what stops us removing device classes.  If we have leaveDeviceClass set to True,
                # then the device class will be left behind
                if leaveDeviceClass == False:
                    deviceList = self.removeDeviceOrganiser(classToRemove)
                # reset zproperty to default value of 10
                log.info(' Resetting zProperty zSshConcurrentSessions to default; new value is 10')
                deviceClassToRemove.setZenProperty('zSshConcurrentSessions', 10)

        # Instruct Zenoss to remove any objects from Zope from the objects.xml file contained inside the ZenPack
        # Once you get down here,  running this next line will tell zenoss to remove the zenpack as it normally would
        # For non zenpacklib, use ZenPackBase.remove;
        #  for zenpacklib, use super(ZenPack, self).remove(app, leaveObjects)
        #ZenPackBase.remove(self, app)
        super(ZenPack, self).remove(app, leaveObjects)
"__init__.py" 279 lines --30%--                                                    85,0-1        34%
```

Figure 78: __init__.py to bind templates to devices - part 3, remove method

The local *setTemplates* method manipulates the *zDeviceTemplates* property.

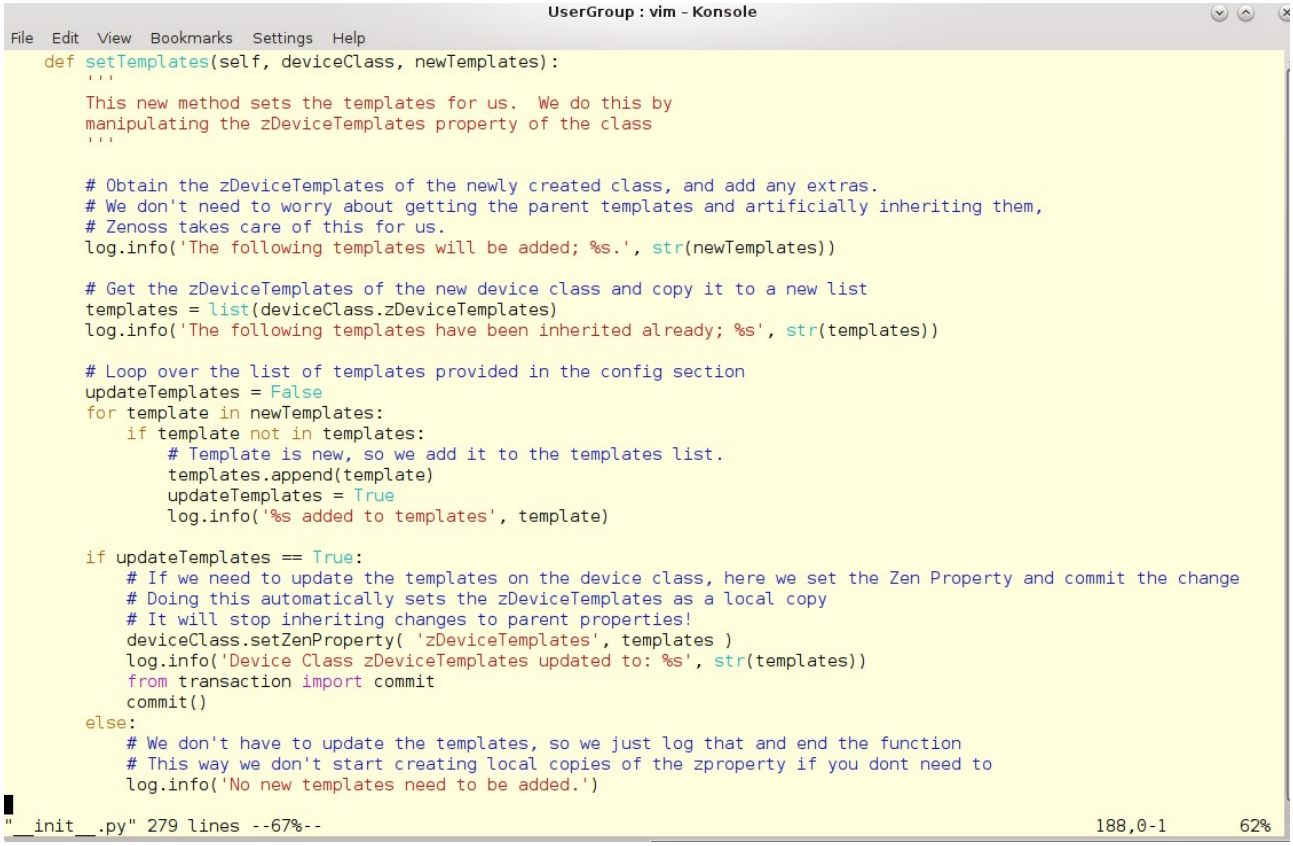


Figure 79: __init__.py to bind templates to devices - part 3, local setTemplates method

The existing templates on the device class are obtained with:

```
templates = list(deviceClass.zDeviceTemplates)
```

This list is then augmented with any new templates specified from the *deviceClassToAdd* dictionary, passed as *newTemplates*.

This method is updating the *zDeviceTemplates* zProperty on a device in the ZODB database so the changes must be committed:

```
from transaction import commit
commit()
```

The ZenPack should be reinstalled and all daemons recycled to test these changes in *__init__.py*.

8.12 *Creating new components inherited from existing components

Section 8.11 demonstrated adding the *UserGroup* component on to the existing Zenoss-supplied *Device* object class; but the standard code also provides an existing *os* relationship and users and groups would perhaps fit better as sub-components of the OperatingSystem object class.

At this time, zenpacklib does not appear to fully support such a mechanism but the example is provided here anyway in the hope that this functionality will soon work. Objects, relationships and modeling work but the JavaScript for the component display does not work.

8.12.1 zenpack.yaml modifications

Starting from the *zenpack.yaml* show in section 8.11, the class_relationships stanza should be modified so that the containing relationship for a *UserGroup* object is *os* on the existing *OperatingSystem* component object.

```
class_relationships:
 #- Products.ZenModel.Device.Device(userGroups) 1:MC UserGroup(uGDevice)
 - Products.ZenModel.OperatingSystem.OperatingSystem(userGroups) 1:MC UserGroup(os)
 - UserGroup(users) 1:MC User(primaryUserGroup)
```

The relationship for *UserGroup* should be changed to match this:

```
    UserGroup:
    .....
     relationships:
      #userGroupDevice:    # back to the containing device
      #uGDevice:         # back to the os relationship on the containing device
      os:             # back to the os relationship on the containing device
        #label: userGroupDevice
        #label: uG Device
        label: OS
        display: true    # Ensures relationship shown in Details dropdown
       users:          # down to User sub-component
        label: users
        display: true    # Relationship shown on grid and Details
```

8.12.2 Other modifications

One issue with this scenario is that zenpacklib code will ensure that when the ZenPack is installed, all **device** relationships will be rebuilt; however it does **not** ensure that device **component** relationships are rebuilt. The *__init__.py* in the base directory of the ZenPack must do this.

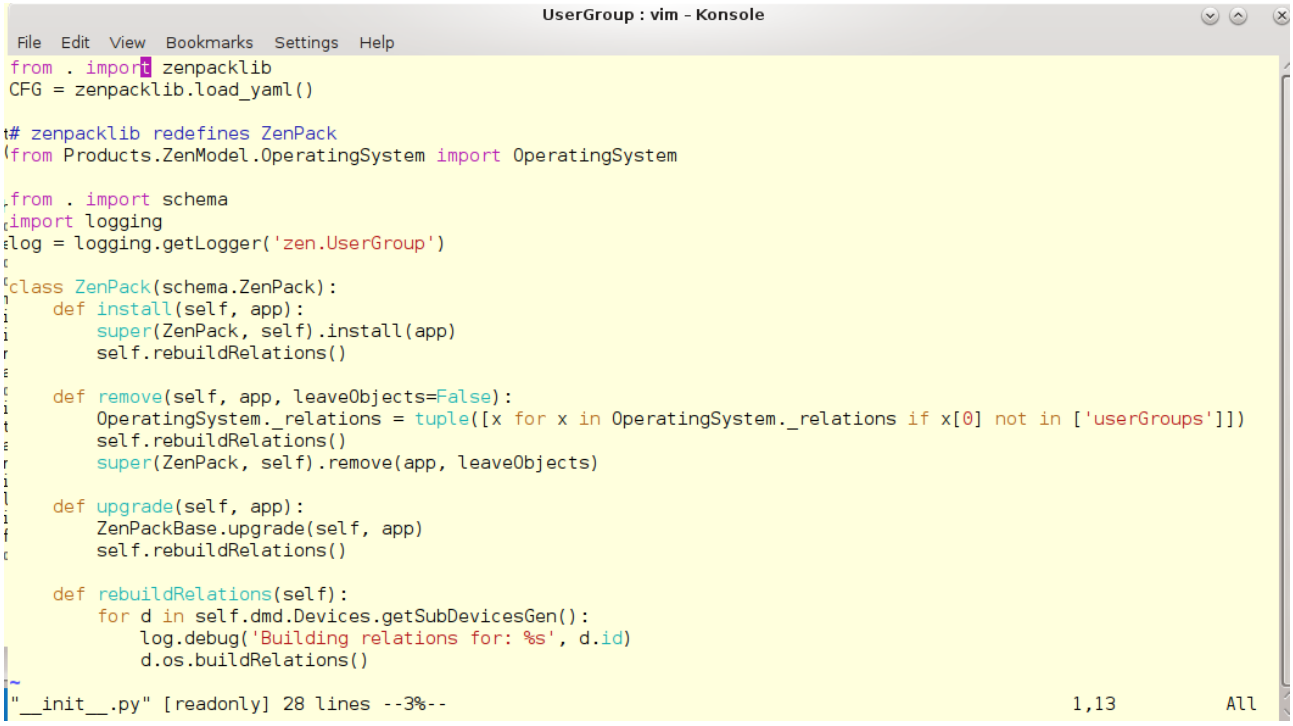


Figure 80: __init__.py to rebuild os component relationships on ZenPack install / remove / upgrade

In Figure 80 note:

- The *ZenPack* class is redefined (monkeypatched). It must inherit from *schema.ZenPack*, which is the *ZenPack* class definition created by zenpacklib.
- An import for *schema* from "." is required.
- A small function is created for the *ZenPack* class to actually perform the relation rebuild.
- The *remove* function ensures that the *userGroups* relation is removed from *OperatingSystem* component objects. This means that an import is also required for *OperatingSystem*.

The modeler plugin also requires small changes to populate the os relationship:

```
class UserGroupMap(CommandPlugin):
  # relname and modname for the CommandPlugin will be inherited by any
  # calls to rm = self.relMap()  or om = self.objectMap()
  # No compname specified here as UserGroup is a component directly on
  #   the device (defaults to null string)
  # classname not required as largely deprecated. classname is the same
  #   as the module name here
  relname = 'userGroups'
  modname = 'ZenPacks.community.UserGroup.UserGroup'
  # Need to add UserGroup objects to the os component relationship
  compname = 'os'
```

The modeler must populate the *UserGroup* component object on to the existing *os* relationship, not directly on to a device class. *compname* will then be used throughout the modeler plugin.

When the *User* sub-component is populated, the component is passed as a parameter from the calling loop to the *getUserMap* function. This needs adjusting to reflect the additional sub-component hierarchy.

```
#um = (self.getUserMap( device, lines[1], int(ugList[2]), ugList[0], 'userGroups/%s' % ug_id, log))
um = (self.getUserMap( device, lines[1], int(ugList[2]),ugList[0], 'os/userGroups/%s' % ug_id, log))
```

8.12.3 Testing the changes

As with the preceding section, test devices should be moved out of their existing test classes, the ZenPack reinstalled and everything must be restarted.

Move a test device to a test device class, ensure that the *cmd.UserGroupMap* modeler is assigned to the device or device class, and re-model.

Users and Groups should be modeled and populate down from the *os* relationship. The ZMI shows that the relationship hierarchy is populated correctly.

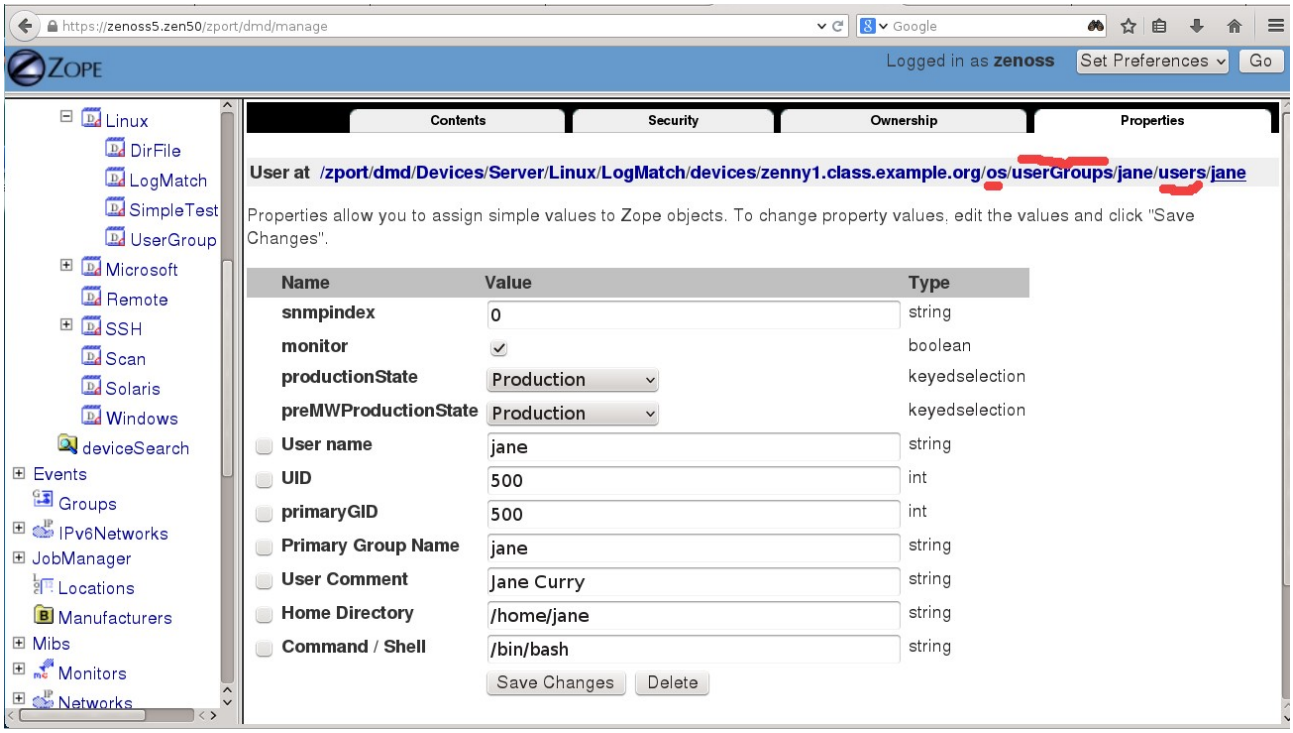


Figure 81: ZMI demonstrating population of userGroups as an os relationship

The *userGroups* relationship is on *os* and has an instance of *jane*, which in turn has a *users* relationship and an instance of the *User* object class, of *jane*.

The part of this solution that does **not** work is the JavaScript to display the component grid panel.

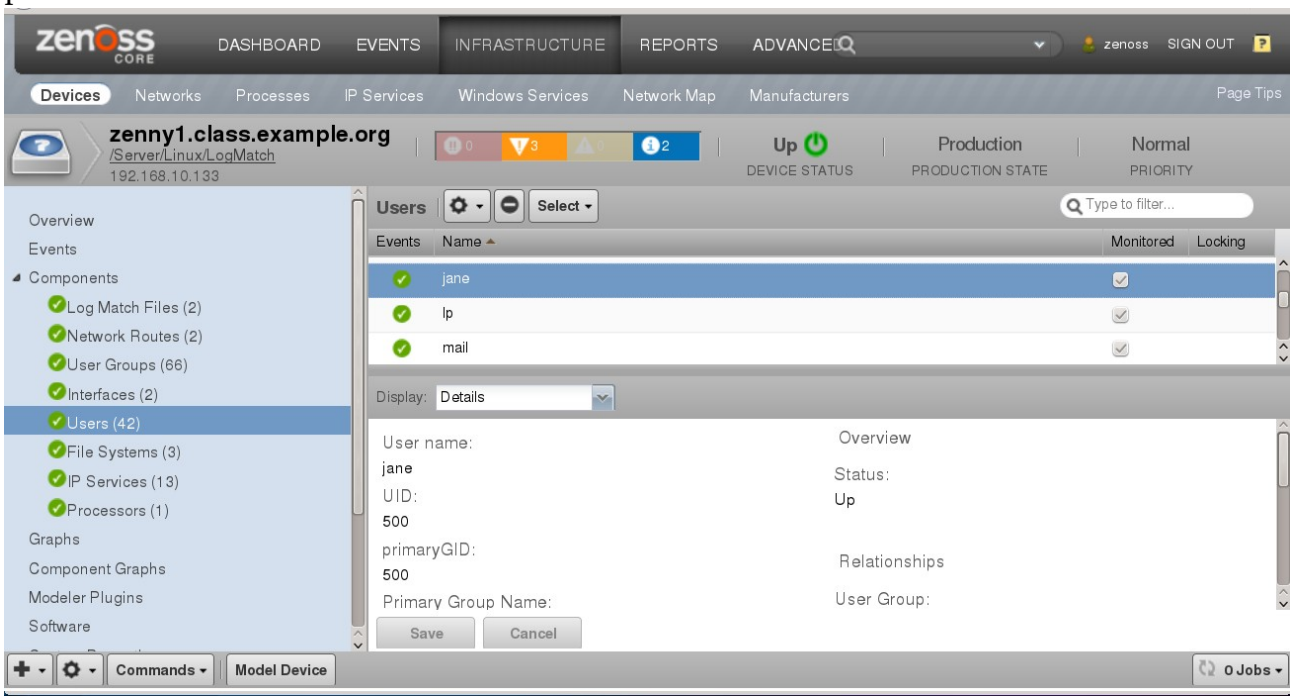


Figure 82: Component panel when userGroups is a relationship on os

The upper part of the main component window only shows the default fields of *Events, Name, Monitored* and *Locking*; all the custom fields such as *UID* and *primaryGID* are not shown although the ZMI demonstrates that these fields have been correctly populated.

The *Details* dropdown does usually work and any graphs for the component **do** work.

Although building deep hierarchies of components and sub-components **is** supported, Zenoss recommends a flatter tree structure because the deeper the relationship hierarchy, the harder it is to make subsequent changes.

9.0 SNMP LogMatch sample ZenPack

This sample ZenPack demonstrates using SNMP as the protocol to gather data.

The sample deliberately does **not** use zenpacklib but explicitly codes all elements with Python or JavaScript. There are a great number of ZenPacks that do not use zenpacklib; the intention here is to demonstrate (in the next section) how to convert a ZenPack to use zenpacklib.

9.1 Using smidump to get MIB information

smidump is a Unix / Linux utility for exploring SNMP Management Information Bases (MIBs). There is also a version shipped as part of the Zenoss core code. Source MIBs can be specified as a file name or a module name. If *UCD-SNMP-MIB.txt* is placed into *$ZENHOME/share/mibs/site*, change to that directory and:

```
smidump -k -f identifiers UCD-SNMP-MIB.txt | grep table
```

will show all the tables in the MIB. The "*-k*" parameter specifies to keep-going if warning errors are found in the source. Output should be similar to:

```
UCD-SNMP-MIB prTable                    table       1.3.6.1.4.1.2021.2
UCD-SNMP-MIB extTable                   table       1.3.6.1.4.1.2021.8
UCD-SNMP-MIB dskTable                   table       1.3.6.1.4.1.2021.9
UCD-SNMP-MIB laTable                    table       1.3.6.1.4.1.2021.10
UCD-SNMP-MIB fileTable                  table       1.3.6.1.4.1.2021.15
UCD-SNMP-MIB logMatchTable              table       1.3.6.1.4.1.2021.16.2
UCD-SNMP-MIB mrTable                    table       1.3.6.1.4.1.2021.102
```

From the output, we can see that there is a logMatchTable. To get more information about this, use:

```
smidump -k -f identifiers UCD-SNMP-MIB | grep logMatch
```

Output will show any line containing logMatch:

```
UCD-SNMP-MIB logMatch                   node        1.3.6.1.4.1.2021.16
UCD-SNMP-MIB logMatchMaxEntries         scalar      1.3.6.1.4.1.2021.16.1
UCD-SNMP-MIB logMatchTable              table       1.3.6.1.4.1.2021.16.2
UCD-SNMP-MIB logMatchEntry              row         1.3.6.1.4.1.2021.16.2.1
UCD-SNMP-MIB logMatchIndex              column      1.3.6.1.4.1.2021.16.2.1.1
UCD-SNMP-MIB logMatchName               column      1.3.6.1.4.1.2021.16.2.1.2
UCD-SNMP-MIB logMatchFilename           column      1.3.6.1.4.1.2021.16.2.1.3
UCD-SNMP-MIB logMatchRegEx              column      1.3.6.1.4.1.2021.16.2.1.4
UCD-SNMP-MIB logMatchGlobalCounter      column      1.3.6.1.4.1.2021.16.2.1.5
UCD-SNMP-MIB logMatchGlobalCount        column      1.3.6.1.4.1.2021.16.2.1.6
UCD-SNMP-MIB logMatchCurrentCounter     column      1.3.6.1.4.1.2021.16.2.1.7
UCD-SNMP-MIB logMatchCurrentCount       column      1.3.6.1.4.1.2021.16.2.1.8
UCD-SNMP-MIB logMatchCounter            column      1.3.6.1.4.1.2021.16.2.1.9
```

```
UCD-SNMP-MIB logMatchCount              column       1.3.6.1.4.1.2021.16.2.1.10
UCD-SNMP-MIB logMatchCycle              column       1.3.6.1.4.1.2021.16.2.1.11
UCD-SNMP-MIB logMatchErrorFlag          column       1.3.6.1.4.1.2021.16.2.1.100
UCD-SNMP-MIB logMatchRegExCompilation   column       1.3.6.1.4.1.2021.16.2.1.101
```

Having found useful MIB variables like *logMatchName* and *logMatchCurrentCounter*, they can be queried using *snmpwalk*. Either use the full OID seen above or, if the MIB module is under */usr/share/snmp/mibs*, then the module name can be used with the OID name. So, for device *zenny1,* which supports SNMP v2, with a community of *fraclmye*, try:

```
[zenoss@zen42 mibs]$ pwd
/usr/share/snmp/mibs
[zenoss@zen42 mibs]$ snmpwalk -v 2c -c fraclmye zenny1 UCD-SNMP-MIB::logMatchName
UCD-SNMP-MIB::logMatchName.1 = STRING: fred1_daily
UCD-SNMP-MIB::logMatchName.2 = STRING: fred2_daily
[zenoss@zen42 mibs]$ snmpwalk -v 2c -c fraclmye zenny1 UCD-SNMP-MIB::logMatchCurrentCounter
UCD-SNMP-MIB::logMatchCurrentCounter.1 = Counter32: 8
UCD-SNMP-MIB::logMatchCurrentCounter.2 = Counter32: 25
[zenoss@zen42 mibs]$
```

9.2 Requirements specification

The first complex, coded sample ZenPack will use the capability of a netSnmp agent to report on logfiles. Strictly, information comes from the UCD (University of California, Davis or ucdavis) MIB which is now being maintained by the netSnmp team.

Details for configuring logfile monitoring are provided in the man pages for *snmpd.conf*.

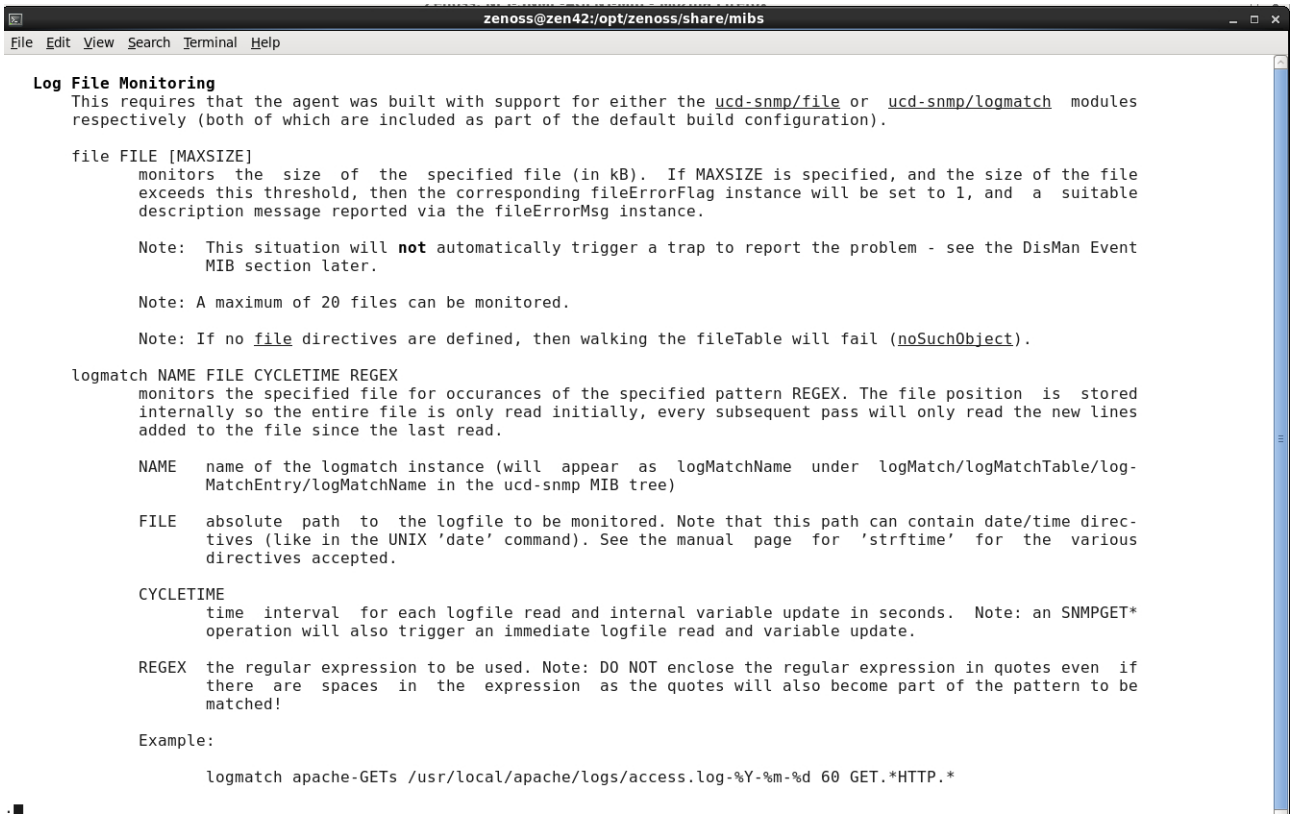


Figure 83: man snmpd.conf provides documentation on configuring for logfile monitoring

To monitor for the string “test”, every 5 minutes, in a file under */opt/zenoss/local/fredtest*, whose name is f*red1.log_YYYYMMDD*, where YYYYMMDD changes each day to represent the current date, put the following entry into the *snmpd.conf* file (typically in */etc/snmp*). You will need root privileges.

```
logmatch fred1_daily /opt/zenoss/local/fredtest/fred1.log_%Y%m%d 300 test
```

The snmpd daemon will need restarting (with root privilege). The command may vary slightly depending on your Operating System but:

```
service snmpd restart           or
/etc/init.d/snmpd restart         should work
```

To get logfile information using SNMP, snmpwalk the logfile table. For the device *zen42.class.example.org*, using SNMP v2c with a community of *public*, use:

```
snmpwalk -v 2c -c public zen42.class.example.org 1.3.6.1.4.1.2021.16
```

where 1.3.6.1.4.1.2021.16 is the Object Id (OID) for the logfile table in the UCD MIB. You should **not** need root privilege for snmpwalk tests.

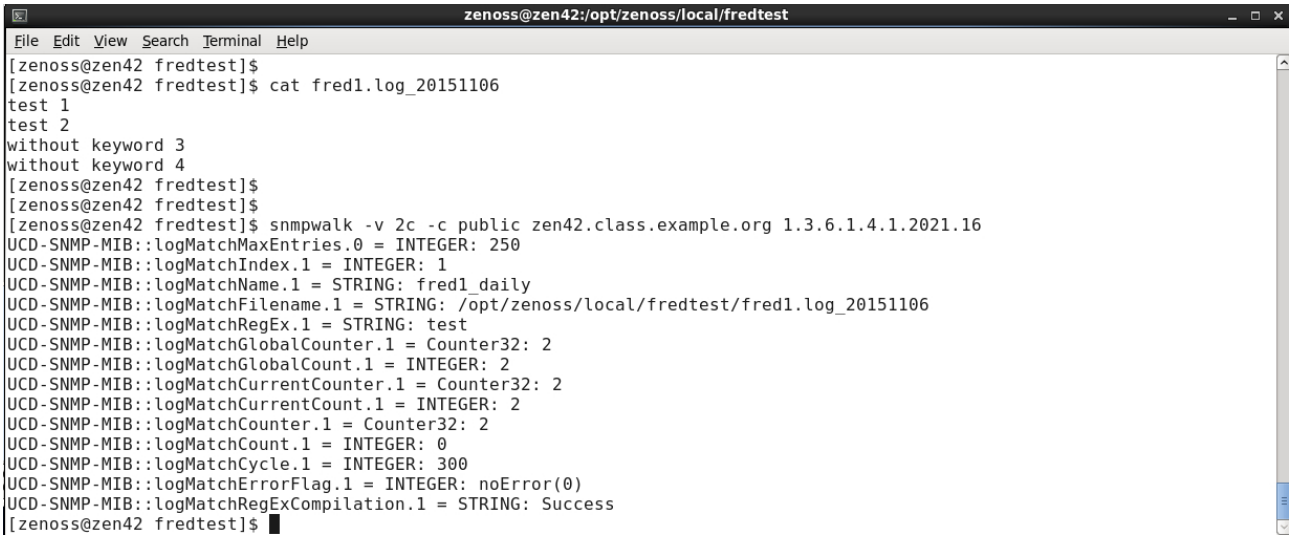


Figure 84: snmpwalk command for UCD logfile information

If you need the numeric OIDs rather than the slightly more friendly OID names, add “-O n” to the snmpwalk command.

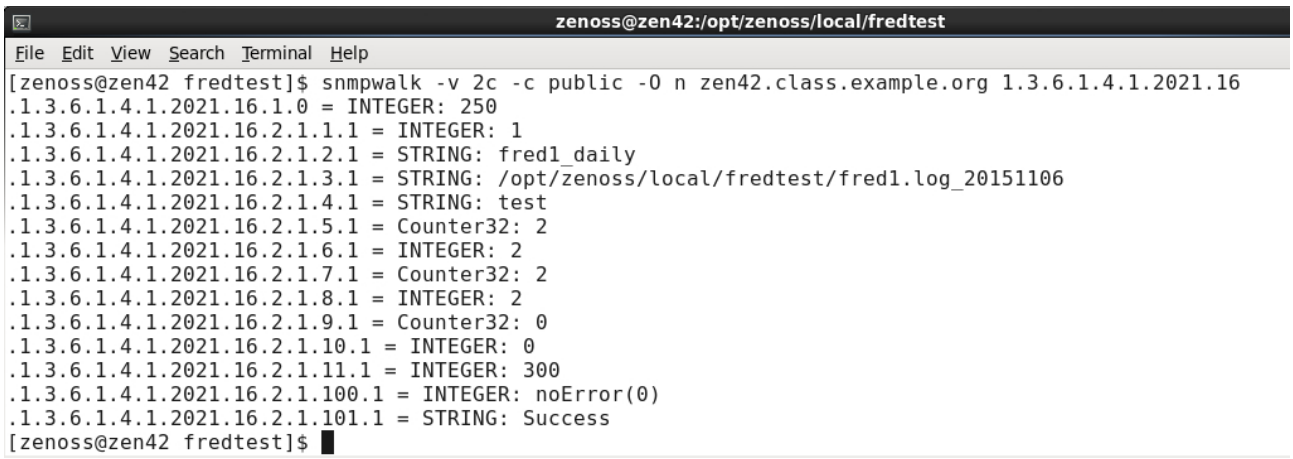


Figure 85: snmpwalk command for UCD logfile information showing full numeric OIDs

If there are several logmatch entries in *snmpd.conf* then there will be several sets of OID values in the MIB table, each with an increasing index number.

The *UCD-SNMP-MIB.txt* MIB file can be obtained from various places on the internet such as http://www.net-snmp.org/docs/mibs/UCD-SNMP-MIB.txt .

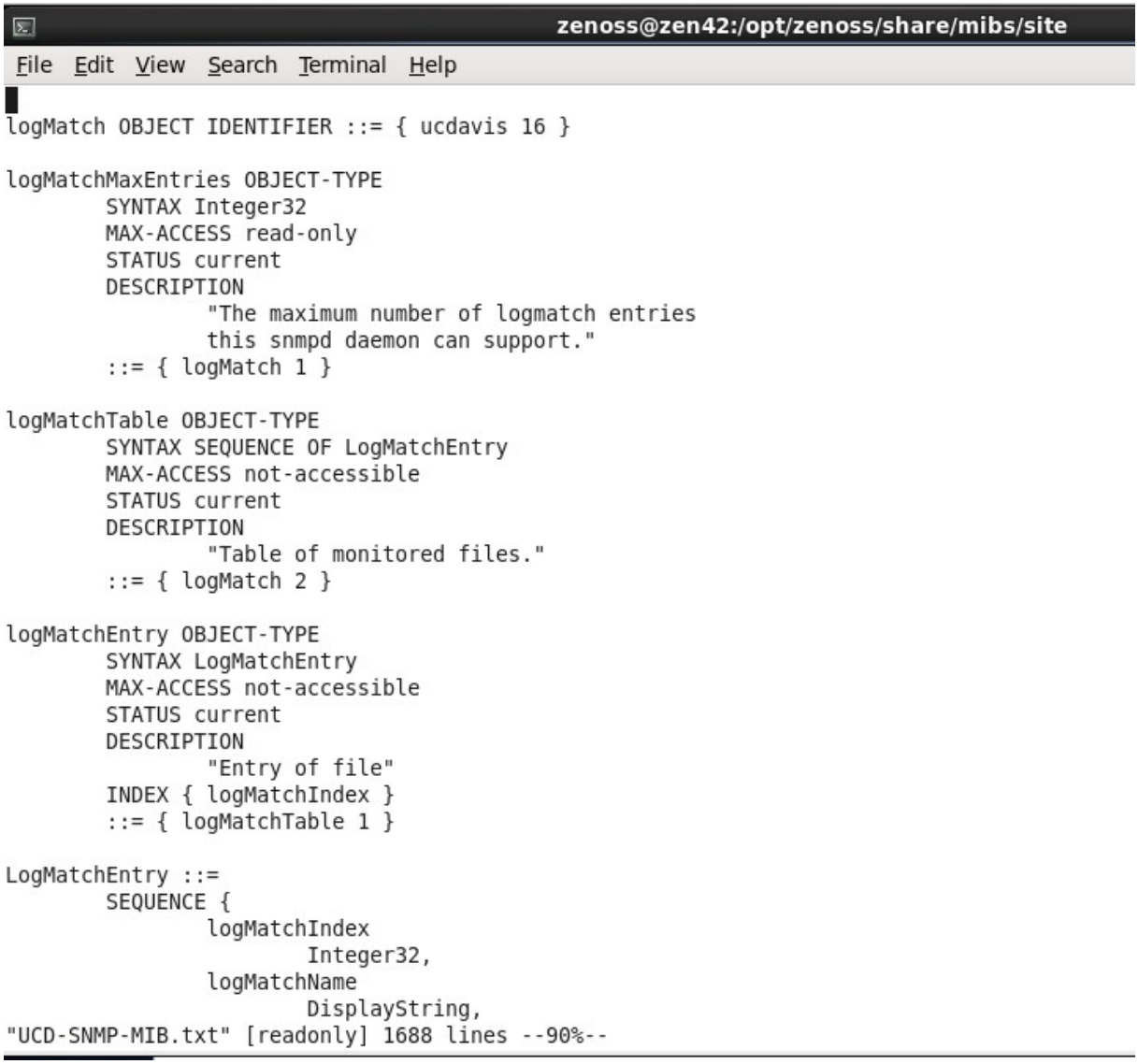


Figure 86: UCD-SNMP-MIB.txt showing start of logMatch definitions

Figure 86 shows that the logMatch entries start with OID { ucdavis 16 } (ucdavis is defined at the top of this file as { enterprises 2021 }, which gives the OID to logMatch as **1.3.6.1.4.1.2021.16**. There is then a table of entries, one entry for each logMatch file configured in the agent. This takes us to OID **1.3.6.1.4.1.2021.16.2.1**.

Some of the LogMatch OIDs are useful for configuration data; others for performance data.

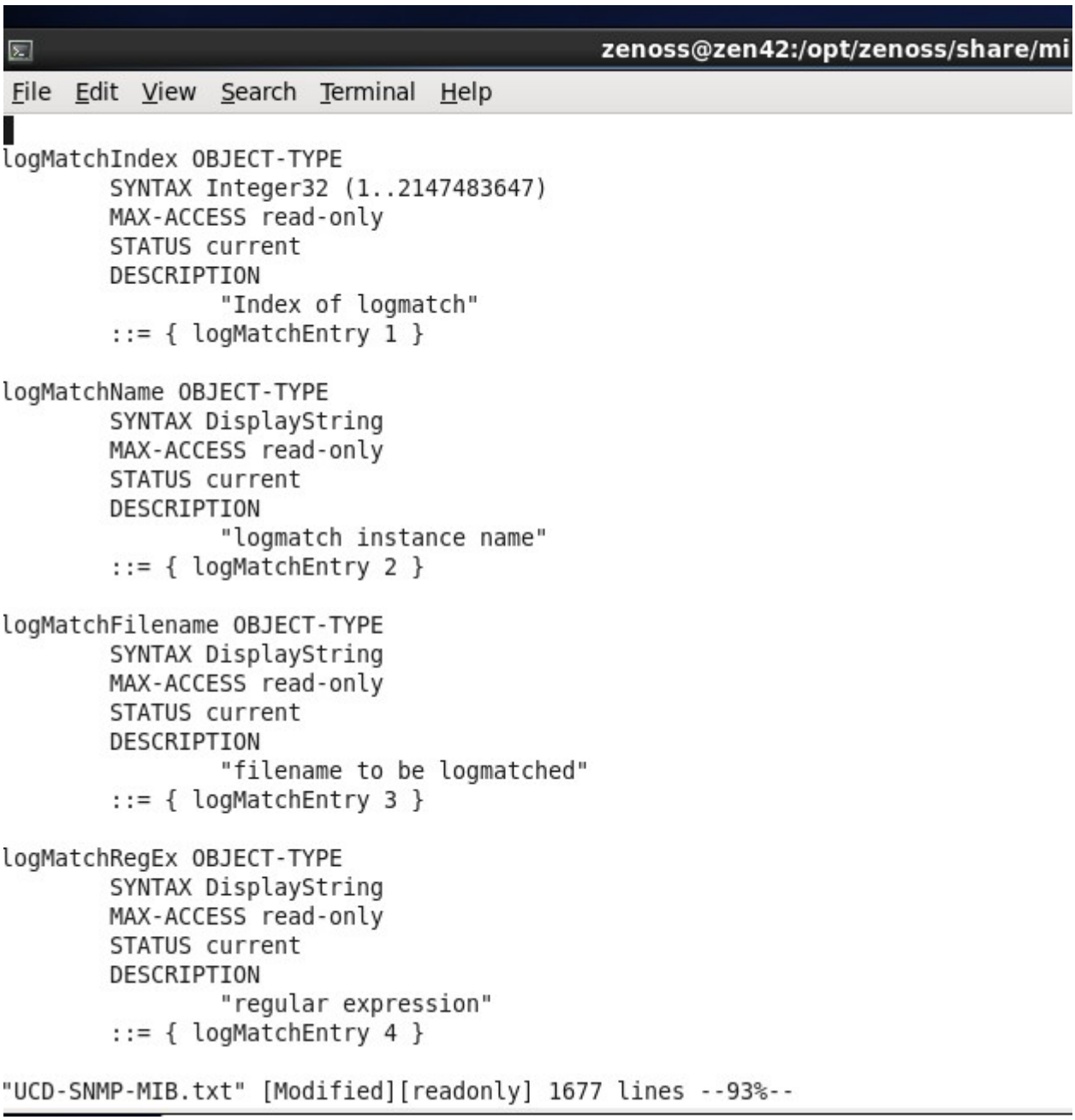


Figure 87: LogMatch OIDs for configuration data

Note the *SYNTAX* statement in a MIB file - it gives the type of data. Note that *DisplayString* is text - but only upto **255** characters.

Thus useful configuration information is:

- 1.3.6.1.4.1.2021.16.2.1.1 index number into the logMatch table
- 1.3.6.1.4.1.2021.16.2.1.2 logMatch instance name
- 1.3.6.1.4.1.2021.16.2.1.3 filename being matched
- 1.3.6.1.4.1.2021.16.2.1.4 regular expression used for matching
- 1.3.6.1.4.1.2021.16.2.1.11 cycle time for this logMatch entry
- 1.3.6.1.4.1.2021.16.2.1.100 error flag for this logMatch entry

- 1.3.6.1.4.1.2021.16.2.1.101 message of regex precompilation for this entry

Examining further down the MIB file, the various relevant performance data counters are:

```
logMatchGlobalCounter OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "global count of matches"
    ::= { logMatchEntry 5 }

logMatchCurrentCounter OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Regex match counter. This counter will
        be reset with each logfile rotation."
    ::= { logMatchEntry 7 }

logMatchCounter OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Regex match counter. This counter will
        be reset with each read"
    ::= { logMatchEntry 9 }
```

Counter data types are often more useful than raw data. For displaying data graphically, the *logMatchCurrentCounter* (1.3.6.1.4.1.2021.16.2.1.7) is probably the most useful as it will deliver the number of matches, taking account of the logfile rotating.

In addition to monitoring attributes of the logMatch files, the ucdavis MIB provides some version information that is pertinent to the overall device.

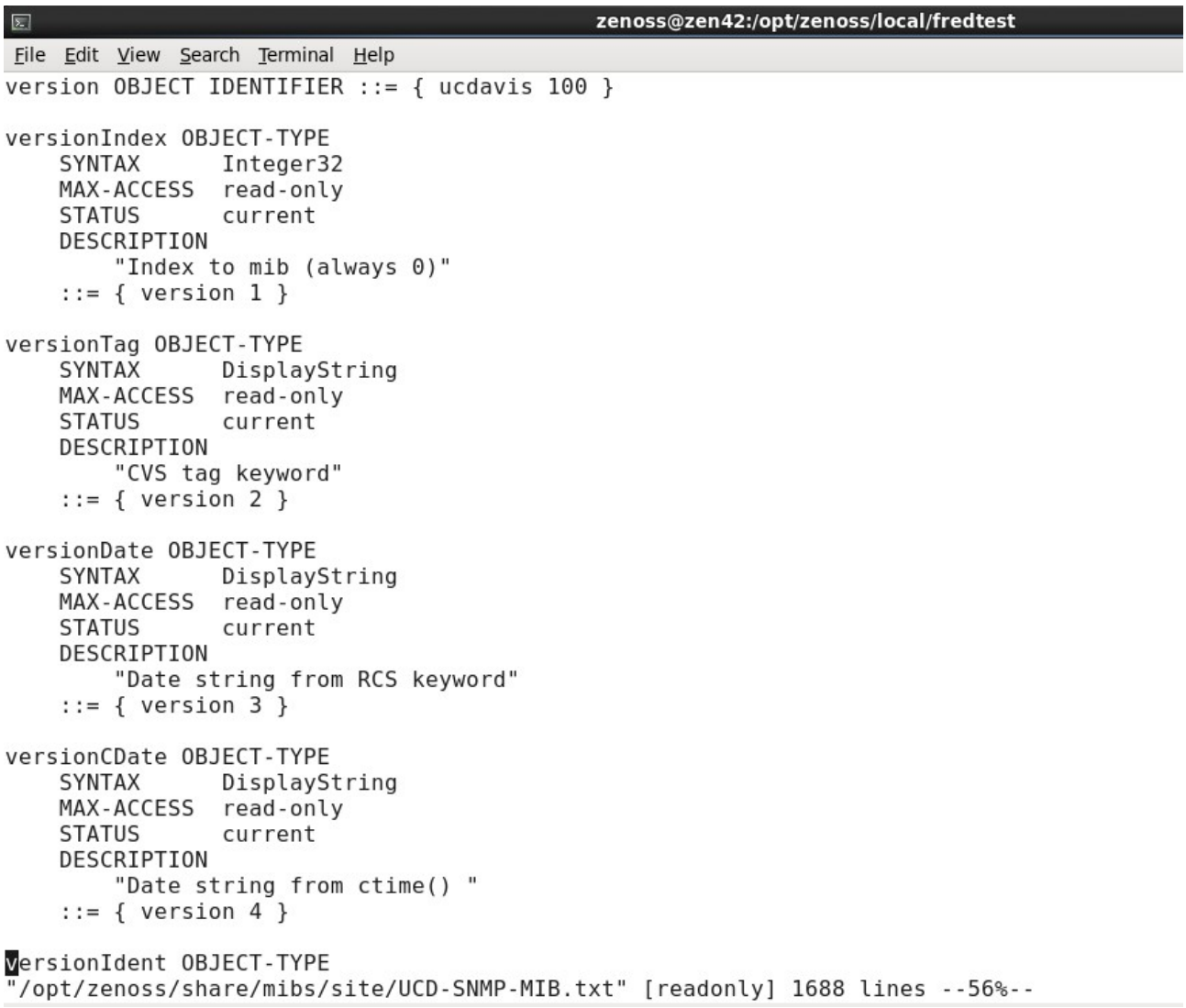


Figure 88: version OIDs in UCD-SNMP-MIB.txt

Note that the version OIDs are all **scalar** values; in other words, there is only one value, not a table of values. There is no Table -> Entry hierarchy. When using scalar values within Zenoss, typically a "**.0**" needs adding to an OID string.

9.3 ZenPack specification

The new ZenPack will be called **ZenPacks.community.LogMatch**.

The ZenPack will create a new component type called **LogMatch** with various attributes:

- logMatch table entry 1.3.6.1.4.1.2021.16.2.1
  - logMatchIndex (integer) .1
  - logMatchName (string) .2
  - logMatchFilename (string) .3
  - logMatchRegex (string) .4
  - logMatchCycletime (integer) .11

- logMatchErrorFlag (integer where 0 = noError and 1 = Error) .100
- logMatchRegExCompilation (string) .101

For now, the ZenPack will also create a new object class for devices that support LogMatch - **LogMatchDevice**. This device will have extra attributes providing version information:

- versionTag 1.3.6.1.4.1.2021.100.2.0
- versionDate 1.3.6.1.4.1.2021.100.3.0

A *LogMatchDevice* will have a ToManyCont relationship with *LogMatch* components called **logMatchs**. The corresponding ToOne relationship from the *LogMatch* component will be **logMatchDevice** (note the capitalisation and presence / lack of a trailing "s", carefully).

A modeler plugin will be required to discover LogMatch components - **LogMatchMap**.

JavaScript will be needed to display the new component. The ***LogMatch.js*** file will be under the ZenPack's ***browser/resources/js*** directory hierarchy.

Info and Interface classes for the LogMatch component will be **LogMatchInfo** and **ILogMatchInfo** respectively.

Templates will be created through the GUI and added to the *objects.xml* of the ZenPack.

9.4 Creating the ZenPack

Unless otherwise noted, all ZenPack instructions from here on will be for Zenoss 4 Core (or earlier). Remember that Service Dynamics and Zenoss 5 have different requirements for restarting daemons. Refer to Chapter 3 for more details.

1. Create the ZenPack, *ZenPacks.community.LogMatch* using the GUI. Add an owner name and a License.
2. Copy the directory hierarchy from *$ZENHOME/ZenPacks* to */code/ZenPacks/DevGuide*

```
cd /code/ZenPacks/DevGuide
cp -R $ZENHOME/ZenPacks/ZenPacks.community.LogMatch .
```

3. Reinstall the ZenPack in development mode with the link parameter

```
zenpack --link --install ZenPacks.community.LogMatch
```

4. Restart the necessary Zenoss daemons

```
zenhub restart
zopectl restart
```

5. Create the *README.rst* skeleton file in the top-level directory of the ZenPack

9.5 Creating device and component object classes

The creation of the ZenPack creates various sample files that are good templates for creating the required files. At first, it is good practice to **copy** the sample files, rather than modify them, so the unchanged samples can be referred back to.

ExampleDevice.py has been copied to *LogMatchDevice.py* and then modified. Note that it is good practice for the filename to match the object class name.

zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.LogMatch/ZenPacks/community/LogMatch

File Edit View Search Terminal Help

```
from Products.ZenModel.Device import Device
from Products.ZenRelations.RelSchema import ToManyCont, ToOne

class LogMatchDevice(Device):
    """
    LogMatch device subclass. In this case the reason for creating a subclass of
    device is to add a new type of relation. We want many "LogMatch"
    components to be associated with each of these devices.

    If you set the zPythonClass of a device class to
    ZenPacks.community.LogMatch.LogMatchDevice, any devices created or moved
    into that device class will become this class and be able to contain
    LogMatch components.
    """
    meta_type = portal_type = 'LogMatchDevice'

    #**************Custom data Variables here from modeling************************
    versionTag = ''
    versionDate = ''
    #**************END CUSTOM VARIABLES ****************************
    #*************  Those should match this list below ******************
    _properties = Device._properties + (
        {'id':'versionTag', 'type':'string', 'mode':''},
        {'id':'versionDate', 'type':'string', 'mode':''},
        )
    #****************

    # This is where we extend the standard relationships of a device to add
    # our "logMatchs" relationship that must be filled with components
    # of our custom "LogMatch" class.
    # ZenPacks.community.LogMatch.LogMatch.LogMatch (starting from the right) is the
    #  LogMatch class in the LogMatch file (strictly module) in the ZenPacks.community.LogMatch ZenPack
    _relations = Device._relations + (
        ('logMatchs', ToManyCont(ToOne,
            'ZenPacks.community.LogMatch.LogMatch.LogMatch',
            'logMatchDevice',
            ),
        ),
    )
"LogMatchDevice.py" [Modified][readonly] 39 lines --41%--
```

Figure 89: LogMatchDevice.py

The **LogMatchDevice** class inherits from the Zenoss-supplied **Device** class.

In the

```
meta-type = portal-type = 'LogMatchDevice'
```

line, the name at the end should match the object class that is being defined.

TODO: What is meta_type=portal_type ?????

The *ExampleDevice.py* template did not provide examples for new attributes so these have been added.

The relationship statement inherits from **Device._relations** and then adds on the new l**ogMatchs** ToManyCont relationship.

Note that we have chosen to call the new component object ***LogMatch***, not *LogMatchComponent*.

The *LogMatch.py* file has been copied from *ExampleComponent.py* and modified.

zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.LogMatch/ZenPacks/community/LogMatch

File Edit View Search Terminal Help

```
from Products.ZenModel.DeviceComponent import DeviceComponent
from Products.ZenModel.ManagedEntity import ManagedEntity
from Products.ZenModel.ZenossSecurity import ZEN_CHANGE_DEVICE
from Products.ZenRelations.RelSchema import ToManyCont, ToOne

class LogMatch(DeviceComponent, ManagedEntity):
    meta_type = portal_type = "LogMatch"

    #**************Custom data Variables here from modeling***********************
    logMatchName = ''
    logMatchFilename = ''
    logMatchRegEx = ''
    logMatchCycle = 300
    logMatchErrorFlag = 0
    logMatchRegExCompilation = ''
    #**************END CUSTOM VARIABLES ****************************
    #*************  Those should match this list below ******************
    _properties = ManagedEntity._properties + (
        {'id': 'logMatchName', 'type': 'string', 'mode': ''},
        {'id': 'logMatchFilename', 'type': 'string', 'mode': ''},
        {'id': 'logMatchRegEx', 'type': 'string', 'mode': ''},
        {'id': 'logMatchCycle', 'type': 'int', 'mode': ''},
        {'id': 'logMatchErrorFlag', 'type': 'int', 'mode': ''},
        {'id': 'logMatchRegExCompilation', 'type': 'string', 'mode': ''},
    )
    #****************

    _relations = ManagedEntity._relations + (
        ('logMatchDevice', ToOne(ToManyCont,
            'ZenPacks.community.LogMatch.LogMatchDevice.LogMatchDevice',
            'logMatchs',
            ),
        ),
    )

    # Custom components must always implement the device method. The method
    # should return the device object that contains the component.
    #    ie. follow the logMatchDevice relationship
    def device(self):
        return self.logMatchDevice()
"LogMatch.py" [Modified] 52 lines --69%--
```

Figure 90: LogMatch.py file defining the LogMatch component object class

Note the third line in the *LogMatch.py* file is underlined in red. This is the pyflakes plugin to the vi editor pointing out an “error”. Moving to that line gives the explanation “ZEN_CHANGE_DEVICE' imported but unused”; in this case, it is informational rather than a real issue.

The *LogMatch* object class inherits from both *DeviceComponent* and *ManagedEntity*.

The properties inherit from *ManagedEntities._properties* and the relations for the object inherits from *ManagedEntity._relations*.

Attributes are defined with default values; note that the default value must be of the correct type to match with the definition in the *_properties* statement.

Note that Python is perfectly happy with either single quotes (') or double quotes (“) around strings but the opening and closing quotes must match.

TODO: What should mode be and what does it do???

It is essential that the ToOne relationship, ***logMatchDevice***, matches exactly with the corresponding definition in the *LogMatchDevice.py* file.

Any component object class must define a method called **device** which delivers the object representing the containing device. In practice, this simply follows the *logMatchDevice* relationship.

Remember from Figure 29 in chapter 5 that the *ManagedEntity* class had a null **device** method. The code in *LogMatch.py* overrides this null method. It is perfectly possible to build hierarchies of components through relationships and the device method will follow these relationships.

The *ExampleComponent.py* template defines f**actory_type_information** with a comment that implies that this stanza is necessary to see the *Graphs* dropdown menu for a component. In Zenoss 4 and later, this is no longer required. (Incidentally, this is why pyflakes was highlighting the import of *ZEN_CHANGE_DEVICE* as it is the *factory_type_information* stanza that uses this).

When the two object class files are complete, reinstall the ZenPack, watching for error messages. When a new object class file has been added, restart Zenoss entirely rather than just *zenhub* and *zopectl*.

9.5.1 Checking the device attributes and relationship

One way to check the new device object class is to set the *zPythonClass* and then check attributes and relationships. A good way to do this is to set the *zPythonClass* on a test Zenoss Device Class and then move a test device into that class.; simply setting the *zPythonClass* on a test device generally does not work as the ***buildRelations()*** method needs to be run.

1. Create a new Zenoss Device Class for testing - */Server/Linux/SimpleTest*
2. From the *DETAILS* link at the top of the left-hand menu, use the *Configuration Properties* menu to set *zPythonClass* to *ZenPacks.community.LogMatch.LogMatchDevice* . This must match the device object class file in the ZenPack (without the “.py”).
3. If the test device is already in that class, move it to a different class - */Ping* is often a good choice.
4. Move the test device into the sample class. This action runs the *buildRelations()* method for any moved devices and creates any new attributes.
5. Examine the test device with the ZMI. You should see a *logMatchs* relationship (which will have no instances if you drill into it).
6. Under the ZMI Properties tab for the test device, you should find the new attributes.

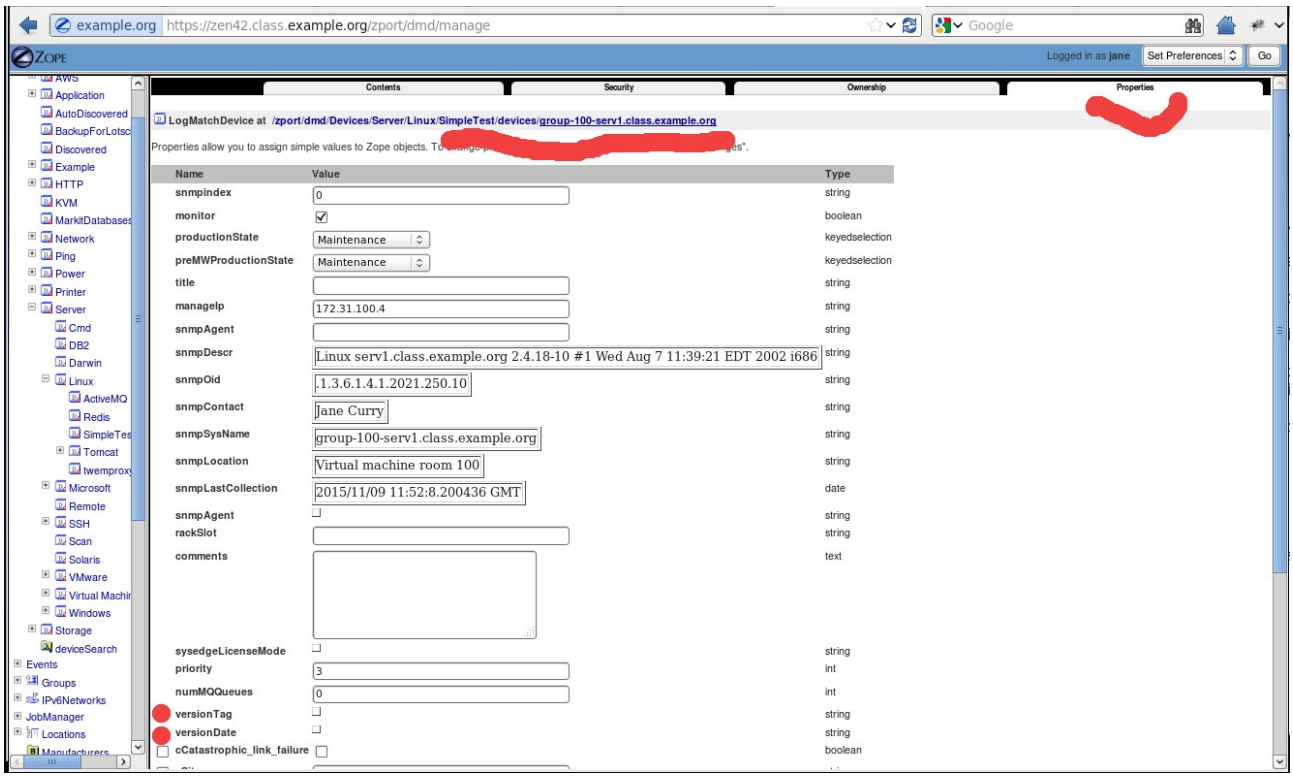


Figure 91: ZMI showing attributes of sample device, group-100-serv1, in the /Server/Linux/SimpleTest device class

9.6 Creating the component modeler

When the ZenPack was created, a directory hierarchy for modelers was created under the base directory:

- modeler/plugins
  - /community
    - /snmp
    - /cmd

It is essential that modelers go under the *modeler/plugins* hierarchy but the directory hierarchy is not mandated beyond that. Particularly if there is only one modeler, it may be clearer to simply place it under *modeler/plugins*. For this sample, we will adhere to the full directory hierarchy built on ZenPack creation so, since, the component is managed using SNMP, the *LogMatchMap.py* modeler file should go under *modeler/plugins/community/snmp*.

Remember that the *modeler/plugins* directory hierarchy must not contain any files ending in *.py*, other than valid, correct modelers; otherwise *zenhub* will report errors on them. This is why the sample files all end in ".example".

The SNMP modeler example provided, *ExampleSNMP.py.example,* only gathers device configuration data, not component data. There is an example component modeler under the *cmd* subdirectory. That said, *ExampleSNMP.py.example* has a very interesting demonstration of applying scalar data to different, existing components (*os* and *hw*) of a device.

So, how to get the OIDs into the relevant standard device attributes? Zenoss provides a number of *setter methods* for standard attributes, including *setHWSerialNumber* and *setHWTag* (see the Zenoss Wiki – Diving into the Device Model at http://monitoringartist.github.io/community.zenoss.org/docs/DOC-2350.html for more information on both device setters and properties) . The really useful feature that this plugin demonstrates is that SNMP data can not only be mapped to object attributes; it can also be mapped to setter methods.

9.6.1 * SNMP modeler code in core Zenoss

The core Zenoss code provides the basic building blocks for modelers. *$ZENHOME/Products/DataCollector/plugins/CollectorPlugin.py* defines a **CollectorPlugin** class and then several more specific plugins that inherit from that class:

```
class CollectorPlugin(object):
  """
  Base class for Collector plugins
  """

  order = 100
  transport = ""
  maptype = ""
  relname = ""
  compname = ""
  modname = ""
  classname = ""
  weight = 1
  deviceProperties = ('id',
            'manageIp',
            '_snmpLastCollection',
            '_snmpStatus',
            'zCollectorClientTimeout',
            )
```

There are also several methods defined for the *CollectorPlugin*, including:

```
def objectMap(self, data={}):
  """Create an object map from the data
  """
  om = ObjectMap(data)
  om.compname = self.compname
  om.modname = self.modname
  om.classname = self.classname
  return om
```

Note that **objectMap** instantiates an **ObjectMap** (note upper-case “O”). There is also a **relMap** method that instantiates a **RelationshipMap**.

```
def relMap(self):
  """Create a relationship map.
  """
  relmap = RelationshipMap()
  relmap.relname = self.relname
  relmap.compname = self.compname
  return relmap
```

Both methods pass the *CollectorPlugin* **compname** (component name) to the object or relationship map. The plugin's **modname** (module name) and **classname** are also passed to the object map and the **relname** is passed to the relationship map.

ObjectMap and *RelationshipMap* can be found in *DataMaps.py* in the same directory; both are, effectively, **protobufs** (raw data).

Three other methods in the *CollectorPlugin* class are:

```
def condition(self, device, log):
  """Test to see if this CollectorPlugin is valid for this device.
  """
  return True

def preprocess(self, results, log):
  """Perform any plugin house keeping before calling user func process.
  """
  return results

def process(self, device, results, log):
  """Process the data this plugin collects.
  """
  raise NotImplementedError
```

These are methods that may be overridden in classes that inherit from *CollectorPlugin*. Note that the *process* method **must** be implemented in subclasses or an error will be raised.

CollectorPlugin.py also defines specializations of the *CollectorPlugin* class:

- class PythonPlugin(CollectorPlugin):
- class CommandPlugin(CollectorPlugin):
- class LinuxCommandPlugin(CommandPlugin):
- class SoftwareCommandPlugin(CommandPlugin):
- class SnmpPlugin(CollectorPlugin):
  - *SnmpPlugin* attributes

```
class SnmpPlugin(CollectorPlugin):
  """
  An SnmpPlugin defines a mapping from SNMP MIB values to a datamap.
  A valid SnmpPlugin must define 'collectoids' (a list of OIDs to be collected)
  and the process() method which converts the OID data to a datamap. It
  can override the condition() method if necessary.
  """

  transport = "snmp"
  conditionOids = []
  snmpGetMap = None
  snmpGetTableMaps = []
  deviceProperties = CollectorPlugin.deviceProperties + ATTRIBUTES + (
    'snmpOid',
    'zMaxOIDPerRequest',
    )
```

  - *SnmpPlugin* **preprocess** method. The *GetMap* and *GetTableMap* methods, each of which has a *mapdata* method, are defined in the same file.

```
def preprocess(self, results, log):
  """Gather raw data for process() to process
```

```
    """
    getdata, tabledatas = results
    if self.snmpGetMap:
      getdata = self.snmpGetMap.mapdata(getdata)
    tdata = {}
    for tmap, tabledata in tabledatas.items():
      tdata[tmap.name] = tmap.mapdata(tabledata)
    return (getdata, tdata)
```

- Note that the *SnmpPlugin* class does **not** have a *process* method defined. It **must** be written as part of the modeler in the ZenPack.

9.6.2 The LogMatchMap modeler plugin for component data

Create *LogMatchMap.py* in the *modeler*/*plugins*/*community*/*snmp* subdirectory of the ZenPack. The name of the file **must** match the name of the modeler plugin class inside it. The directory path hierarchy will be reflected in the GUI dialogue for choosing plugins so will be shown as *community.snmp.LogMatchMap*.

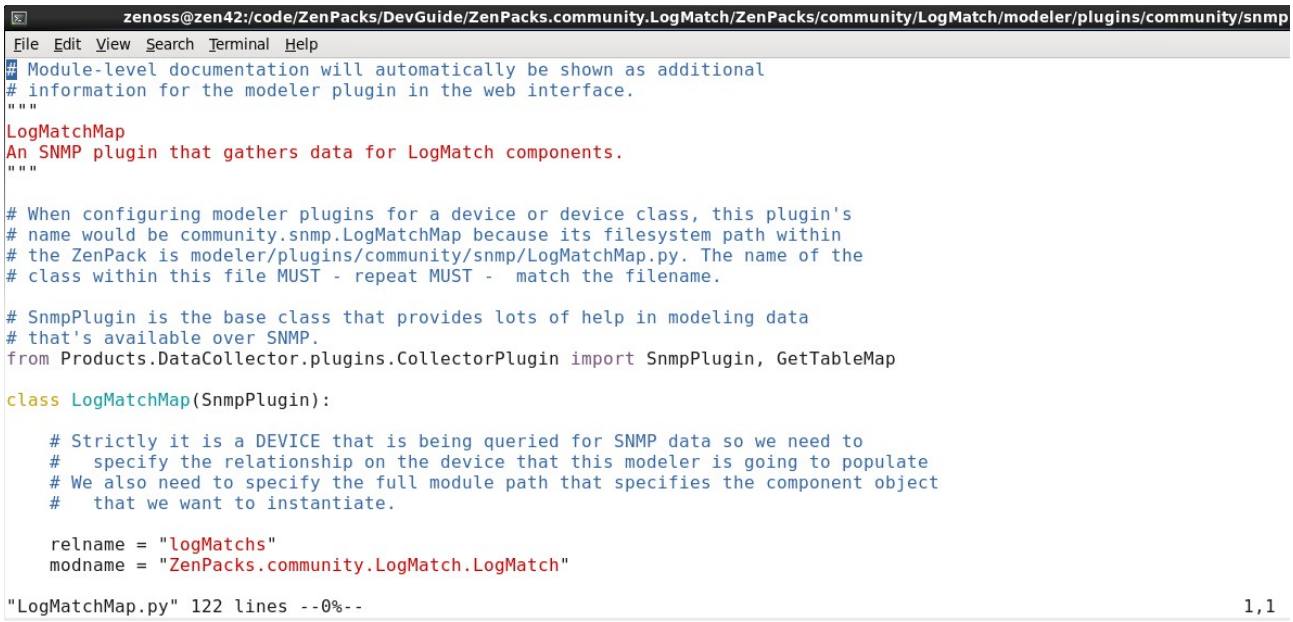


Figure 92: Modeler plugin LogMatchMap.py - initial class definition

The **LogMatchMap** class inherits from the **SnmpPlugin** class provided in *$ZENHOME*/*Products*/*DataCollector*/*plugins*/*CollectorPlugin*.

Note the definition of *relname* and *modname*. It is **essential** that these exist and are correct for the modeler to actually create components.

- *relname* is the name of the relationship **on the device** (since it is the device, not the component that responds to SNMP) - **logMatchs**.
- *modname* is the full path to the module containing the object definition for the **component**. In practice, this is found from the filename in the ZenPack that contains the object definition - **ZenPacks.community.LogMatch.LogMatch**.

An *SnmpPlugin* must specify scalar and / or table OIDs to collect. These are the 'collectoids' referenced in the comments at the top of the *SnmpPlugin* class in *$ZENHOME/Products/DataCollector/plugins/CollectorPlugin.py*. The *LogMatchMap* plugin will only collect tabular data.

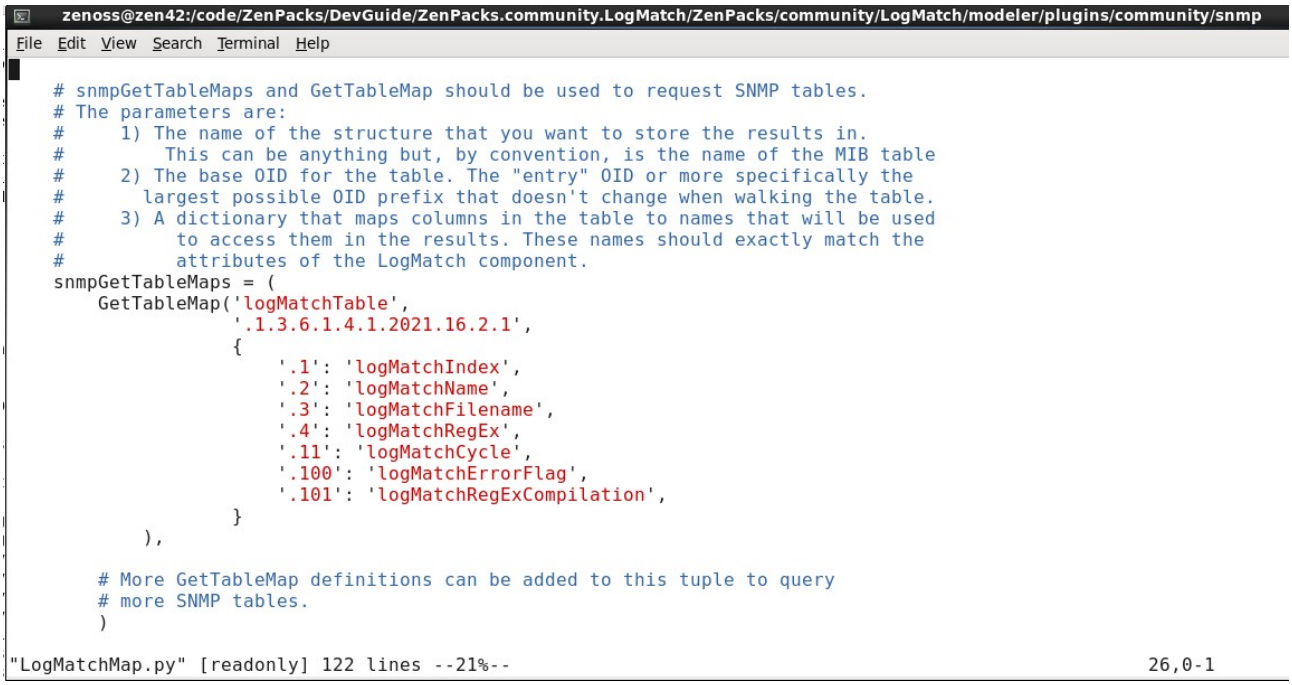


Figure 93: Modeler plugin LogMatchMap.py - snmpGetTableMaps to collect a table of OIDs

snmpGetTableMaps and *GetTableMap* (defined in *$ZENHOME/Products/DataCollector/plugins/CollectorPlugin.py*) should be used to request SNMP tables. The parameters are:

1. The name of the structure that you want to store the results in.
   - This can be anything but, by convention, is the name of the MIB table
2. The base OID for the table.
   - The "entry" OID or more specifically the largest possible OID prefix that doesn't change when walking the table.
   - This is simply good practice, not mandated. It would be possible to have the following but it would simply mean more typing:

```
'.1.3.6.1.4.1.2021.16.2',
{
  '1.1': 'logMatchIndex',
  '1.2': 'logMatchName',
  '1.3': 'logMatchFilename',
  '1.4': 'logMatchRegEx',
  '1.11': 'logMatchCycle',
  '1.100': 'logMatchErrorFlag',
  '1.101': 'logMatchRegExCompilation',
}
```

3. A dictionary that maps columns in the table to names that will be used to access them in the results.

- These names must **exactly** match the attributes of the *LogMatch* component.

The remainder of *LogMatchMap.py* is the mandatory **process** method.

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.LogMatch/ZenPacks/community/LogMatch/modeler/plugins/community/snmp
File  Edit  View  Search  Terminal  Help
    def process(self, device, results, log):
        log.info("Modeler %s processing data for device %s",
            self.name(), device.id)

        # Results is a tuple with two items. The first (0) index contains a
        # dictionary with the results of the "snmpGetMap" queries. The second
        # (1) index contains a dictionary with the results of the
        # "snmpGetTableMaps" queries.
        # NB. For this modeler, getdata is null
        getdata, tabledata = results

        # tabledata contents..
        # {'logMatchTable': {'1':
        #          {'logMatchRegExCompilation': 'Success', 'logMatchRegEx': 'test',
        #          'logMatchCycle': 300, 'logMatchErrorFlag': 0, 'logMatchName': 'fred1_daily',
        #          'logMatchFilename': '/opt/zenoss/local/fredtest/fred1.log_20151110',
        #          'logMatchIndex': 1},
        #      '2':
        #          {'logMatchRegExCompilation': 'Success', 'logMatchRegEx': 'without',
        #          'logMatchCycle': 180, 'logMatchErrorFlag': 0, 'logMatchName': 'fred2_daily',
        #          'logMatchFilename': '/opt/zenoss/local/fredtest/fred2.log_20151110',
        #          'logMatchIndex': 1},
        # } }

        # tabledata may have several dictionaries of tables; we want the logMatchTable dict
        logMatchTable = tabledata.get('logMatchTable')
        log.debug('logMatchTable is %s ' % (logMatchTable))
        # if no tabledata then return logging a warning
        if not logMatchTable:
            log.warn( 'No SNMP response from %s for the %s plugin ', device.id, self.name() )
            log.warn( "Table Data= %s", tabledata )
            return
"LogMatchMap.py" 121 lines --70%--                                                    85,18
```

Figure 94: Modeler plugin LogMatchMap.py - start of process method

The process method is passed the *results* parameter from the **preprocess** method of the *SnmpPlugin* class. This is a Python tuple with two items (note that all Python indexing starts at 0, not 1).

- The first (0) index contains a dictionary with the results of the "snmpGetMap" queries; ie the scalar data. For the *LogMatchMap* modeler, this is null.
- The second (1) index contains a dictionary with the results of the "snmpGetTableMaps" queries; ie the tabular data.
  - Strictly the table data is a dictionary of dictionaries of dictionaries. Once you can run the modeler and get debug output it is useful to provide as comments, sample output for tabledata.
  - This modeler only requests one GetTableMap, *logMatchTable*, but it is perfectly possible to specify several OID tables to collect. Thus the “outer” dictionary only has one element, with the key *logMatchTable*.
  - The value of the “outer” dictionary is a dictionary of table **instances** - whose keys are the index numbers - *1*, *2* and so on, in the case of this data.
  - The value of the “instance” dictionary is also a dictionary where the **keys** are the names provided for the columns in the OID table (which matches the component object attributes) - *logMatchName*, *logMatchFilename*, etc and the **values** are the data values received for those attributes.

Whilst not strictly necessary here where the modeler is only collecting one table of data, it is good practice to assign each separate table in tabledata, to a variable so that it is more easily manipulated in the ensuing code:

```
logMatchTable = tabledata.get('logMatchTable')
```

Data gathering may fail for all sorts of reasons - broken network, incorrect community names, slow response from network or target, target SNMP agent down, MIB value not supported by an agent, etc. It is good practice to always check that there is data for each table requested and provide a warning in the log if data collection failed. Whether the process method continues or halts is up to the plugin developer. In this case, the process method simply returns (with an implied *None* result).

```
if not logMatchTable:
  log.warn( 'No SNMP response from %s for the %s plugin ', device.id, self.name() )
  log.warn( "Table Data= %s", tabledata )
  return
```

The *process* method of the modeler plugin class is expected to return output in one of the following forms.

- A single ObjectMap instance
- A single RelationshipMap instance
- A list of ObjectMap and RelationshipMap instances
- None

If the plugin encounters a bad state and you don't want to affect Zenoss' model of the device you should return ***None***.

The rest of the *process* method is devoted to returning a single RelationshipMap instance.

zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.LogMatch/ZenPacks/community/LogMatch/modeler/plugins/community/snmp

File Edit View Search Terminal Help

```
        # Create a relationship map - relname above specifies the logMatch relationship
        rm = self.relMap()
        # For each entry in the SNMP table, we need to create a LogMatch component
        for oid, data in logMatchTable.items():
            # Use try / except to prevent nasty failures
            try:
                # Next line instantiates a LogMatch component object, populating the
                #   component object's attributes with the matching values from the LogMatchTable
                #   logMatchName, logMatchFilename, etc
                #   modname (specified above) defines the object class for the component
                om = self.objectMap(data)
                # Any attribute can then be overwritten, if required.  id is an inherited
                #   attribute but we want to ensure uniqueness
                om.id = self.prepId(om.logMatchName)
                # snmpindex is also an inherited attribute. Set it to the oid index
                om.snmpindex = oid
                # Append this object instance to the relationship map
                rm.append(om)
            # If something goes wrong, fail nicely with a logged warning and then
            #   continue aroound the loop again
            except (KeyError, IndexError, AttributeError, TypeError), errorInfo:
                log.warn( ' Error in %s modeler plugin %s' % ( self.name(), errorInfo))
                continue
        return rm
"LogMatchMap.py" [Modified] 121 lines --72%--                                    88,26
```

Figure 95: Modeler plugin LogMatchMap.py - main body of process method to create a RelationshipMap

First instantiate a RelationshipMap, *rm*:

```
rm = self.relMap()
```

Bear in mind that the *relMap()* method populates the *RelationshipMap* with the *relname* and *compname* defined in the plugin. *compname* is not specified but *relname* is set at the start of the plugin to "***logMatchs***".

An SNMP plugin typically then has an iterative loop through the instances (or rows in the table):

```
for oid, data in logMatchTable.items():
```

where *oid* will be the instance (or index) number - *1*, *2* in our data here, and *data* will be the dictionary of attributes and values eg. {'logMatchRegEx': 'test', 'logMatchCycle': 300, ... }.

To actually create a component instance, we need a component object. This is achieved with:

```
om = self.objectMap(data)
```

Again, remember that the *objectMap()* method populates the *ObjectMap* with the *compname*, *modname* and *classname* defined in the plugin. *compname* and *classname* are not specified but *modname* is set at the start of the plugin to ***"ZenPacks.community.LogMatch.LogMatch"***. This is the link that tells *objectMap* what sort of object to create, with which attributes and methods.

The *data* parameter passed to *objectMap* is used to populate the *LogMatch* component attributes - **provided the keys of the data dictionary match the attributes of the component object**.

At this stage, assuming the data in the first dictionary instance shown in the comments in Figure 94, we have a *LogMatch* component object, populated with attributes and values:

```
logMatchName = 'fred1_daily'
logMatchFilename = '/opt/zenoss/local/fredtest/fred1.log_20151110'
logMatchRegEx = 'test'
logMatchCycle = 300
logMatchErrorFlag = 0
logMatchRegExCompilation = 'Success'
```

Often, some values need modification. For example, if a MIB data value is presented in bytes and the *size* attribute would rather use KBytes, then that attribute can be changed:

```
om.size = om.size / 1024
```

Remember that there are also inherited attributes for objects, such as *id* and *snmpindex*. It is good practice to ensure that the *id* attribute is unique and the *prepId()* method on the *CollectorPlugin* class provides that function.

```
om.id = self.prepId(om.logMatchName)
```

The *snmpindex* attribute is crucial when gathering performance data with monitoring templates. It is the index used to associate the correct row of performance data with the appropriate component. Typically this is the index or instance of the table, which is being used as the oid key into the MIB table.

```
om.snmpindex = oid
```

Be aware that the index oid is not always a simple, increasing integer. IP interfaces, especially from Windows SNMP agents, can be large random numbers. The *snmpindex* attribute is inherited from *ManagedEntity*, where it is actually defined as a **string**.

Having finished modifying this component object, it is added to the RelationshipMap and the loop continues with the next instance.

```
rm.append(om)
```

It is good practice to code for possible errors using the Python *try..except* construct. This allows a modeler to fail nicely and log a warning. Typically, execution of the loop will continue.

```
except (KeyError, IndexError, AttributeError, TypeError), errorInfo:
  log.warn( ' Error in %s modeler plugin %s' % ( self.name(), errorInfo))
  continue
```

When all instances have been processed and a component created for each, the RelationshipMap is returned from the *process* method. It is then up to the *zenmodeler* daemon to populate the ZODB with the new objects and relationships.

9.6.3 Testing the modeler

If a new modeler has been created then the ZenPack should be reinstalled and Zenoss completely recycled. For subsequent “tweaks” to an existing modeler, restarting *zenhub* and *zopectl* generally suffices.

If you have a test Zenoss Device Class then the easiest way to deploy the modeler is to add it to the Modeler Plugins dialogue for the Device Class.

The first check towards success is if the modeler does appear in the *Available* list. If it doesn't there is probably a syntax error in the modeler code.

Once the plugin is applied to the Class and a device is in that class, run zenmodeler from the command line, just specifying the new plugin for collection. So for a test device, *taplow-11.skills-1st.co.uk*:

```
zenmodeler run -v 10 -d taplow-11.skills-1st.co.uk --collect community.snmp.LogMatchMap
```

To redirect output to a file append:

```
> /tmp/fred 2>&1
```

Note that you need to redirect stderr to stdout (*2>&1*) or you won't see what you need; then inspect */tmp/fred*.

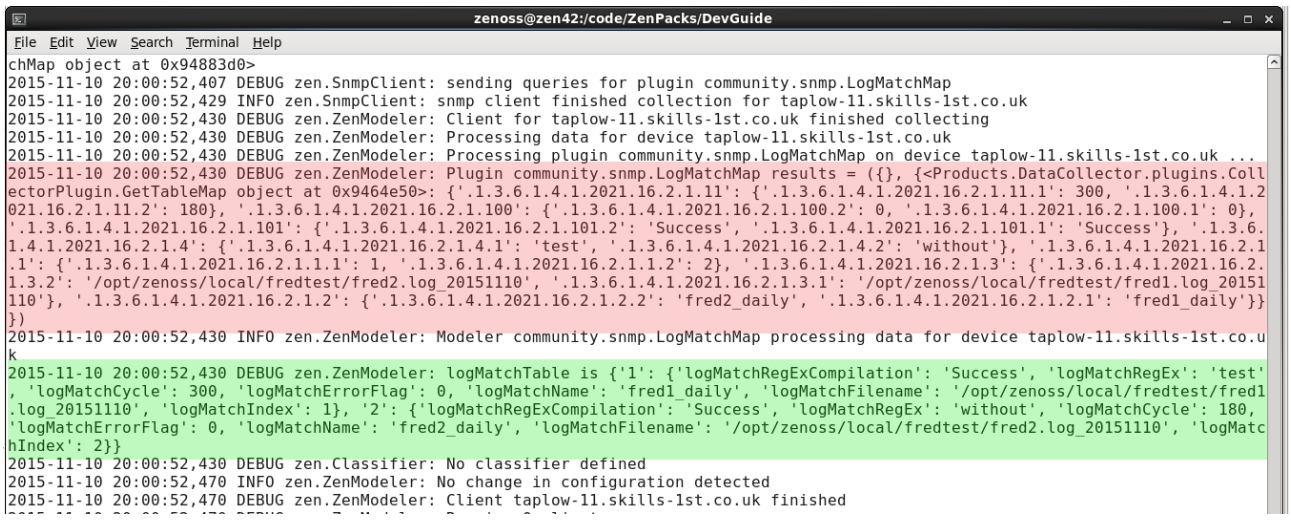


Figure 96: Output from zenmodeler running the LogMatchMap plugin with full debugging

The upper highlighted area in the zenmodeler output in Figure 96 (*Plugin community.snmp.LogMatchMap results* = ...) is the raw results data and often is hard to sort out the various dictionaries.

The lower highlighted area is the result of the *log.debug* line inserted into the plugin code after splitting out the *logMatchTable*;

```
log.debug('logMatchTable is %s ' % (logMatchTable))
```

Use the ZMI to check whether the test device now has instances for the *logMatchs* relationships; if so, are the properties of the components correct?

The ultimate sign of success is to refresh the web browser page for the sample device and have it show *LogMatch* components.

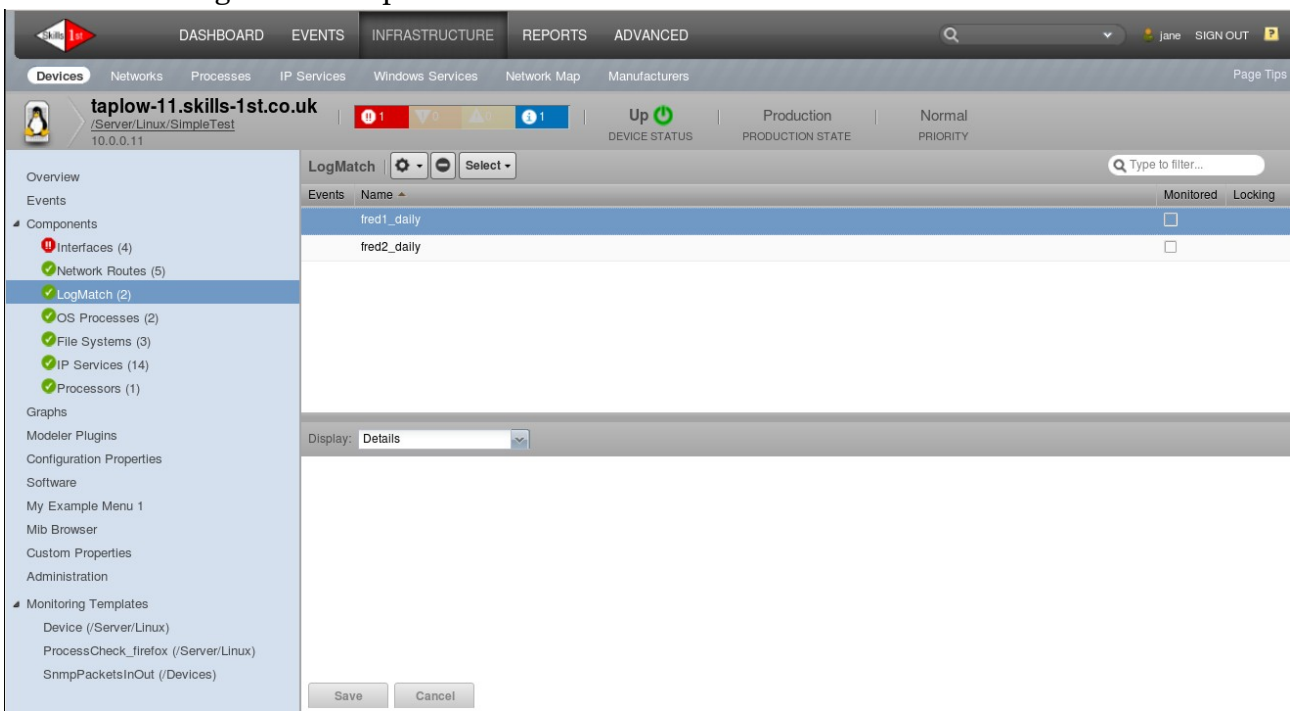


Figure 97: LogMatch components found by the LogMatchMap modeler plugin

Note, at this stage, there will be dropdown menus in the middle of the main panel but none will be active. Similarly, the top half of the component display will show very little detail.

A minimal, default presentation is being used to display the new components and some JavaScript code is required to enhance this.

Note that the modeler collects data for the "*.1*" column representing *logMatchIndex*. This value has not been used in the modeler. Further, referring back to the attributes of a *LogMatch* component in Figure 90, there is no attribute called *logMatchIndex*. Ultimately, the *zenhub* daemon will try to apply the map presented by *zenmodeler*, to the ZODB database and it will have trouble with this undefined attribute, generating a WARNING error message in *zenhub.log*:

```
2015-11-15 17:17:21,973 WARNING zen.ApplyDataMap: The attribute
logMatchIndex was not found on object fred2_daily from device taplow-
11.skills-1st.co.uk
```

If the *logMatchIndex* is really not required then the line should be removed from the modeler's *GetTableMap* entry; however, if it may be used for a later "phase 2" development, a trick is to preface the name with underscore which is a standard Python technique to denote a local variable. This avoids error messages in *zenhub.log*.

```
snmpGetTableMaps = (
    GetTableMap('logMatchTable',
          '.1.3.6.1.4.1.2021.16.2.1',
          {
            '.1': '_logMatchIndex',
            '.2': 'logMatchName',
            '.3': 'logMatchFilename',
            '.4': 'logMatchRegEx',
            '.11': 'logMatchCycle',
            '.100': 'logMatchErrorFlag',
            '.101': 'logMatchRegExCompilation',
          }
      ),
```

9.6.4 The LogMatchDeviceMap modeler for the device

In addition to component configuration, the ZenPack specification requires *versionTag* and *versionDate* attributes for the **device**. A very simple SNMP modeler will suffice for this.

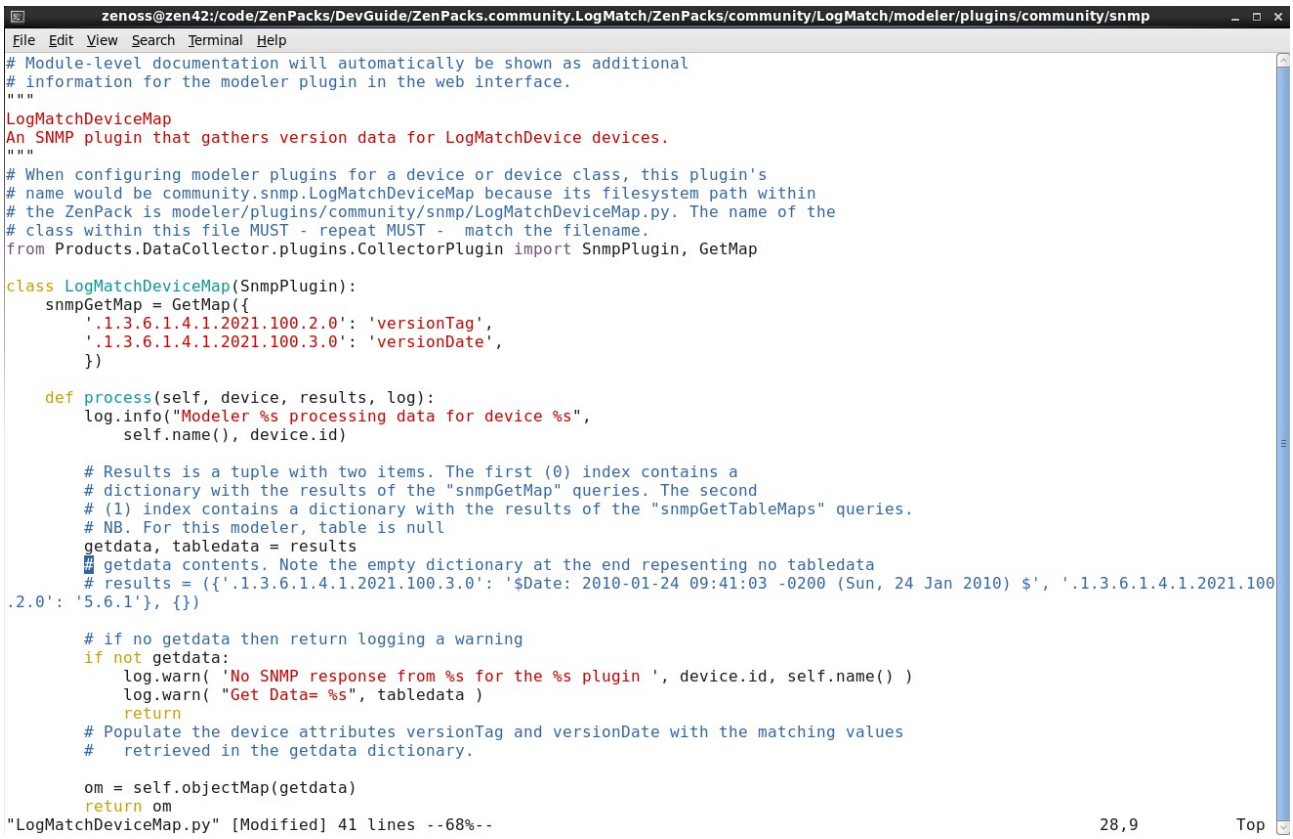


Figure 98: LogMatchDeviceMap modeler to gather versionTag and versionDate attributes for a device

The snmpGetMap 'collectoids' structure is created to define the two OIDs required and the names they will map to (which correspond with the attributes defined in *LogMatchDevice.py*).

Note that the OIDs have "*.0*" on the end in Figure 98- this is **scalar** data.

An *objectMap* is instantiated using the "getdata" data. Since the modeler is run against the **device** (whose *zPythonClass* zProperty is set to *ZenPacks.community.LogMatch.LogMatchDevice*), the *versionTag* and *versionDate* attributes are known and populated. The *objectMap* is returned to the *zenmodeler* daemon which will merge this *objectMap* data into the existing configuration data for the device.

9.6.5 Where do things go wrong with SNMP modelers?

Check section 8.7.4 for general issues with modeler plugins.

1. Modeler fails

   a. Check *zenmodeler* output carefully. From the GUI *Model Device* menu, check whether the modeler appears in the list of modelers to be applied. If it is missing, suspect syntax errors. For example:

```
 2015-11-12 12:52:02,337 INFO zen.ZenModeler: SNMP collection device
taplow-11.skills-1st.co.uk
2015-11-12 12:52:02,337 INFO zen.ZenModeler: plugins:
zenoss.snmp.NewDeviceMap, zenoss.snmp.DeviceMap, HPDeviceMap,
DellDeviceMap, zenoss.snmp.InterfaceMap, zenoss.snmp.RouteMap,
zenoss.snmp.IpServiceMap, zenoss.snmp.HRFileSystemMap,
zenoss.snmp.HRSWRunMap, zenoss.snmp.CpuMap, HPCPUMap, DellCPUMap,
DellPCIMap, community.snmp.LogMatchMap
```

   b. If it appears, check the subsequent output for messages.

c. Run *zenmodeler* standalone in debug mode, optionally sending output to a file:

```
zenmodeler run -v 10 -d taplow-11.skills-1st.co.uk --collect community.snmp.LogMatchMap

zenmodeler run -v 10 -d taplow-11.skills-1st.co.uk --collect community.snmp.LogMatchMap \
               > /tmp/LogMatchMap.out 2>&1
```

d. Check that output is actually received from SNMP and that it matches what you expect. The following message would be an indication that the OID you are requesting is incorrect or not supported on the agent:

```
No decoder for oid 1.3.6.1.4.1.2021.16.100.3.0 type ASN_BIT8 - returning None
```

e. If a relationship is not created then check relationship names and object files for both device and component.

f. If relationship i**nstance(s)** are not created, check:

i. *relname* and *modname* statements in modeler plugin exists

ii. *relname* and *modname* are correct (especially case-sensitivity)

g. If one or more attributes do not have values:

i. Check spelling of attributes in plugin table column names

ii. Check OID is correct and that data is collected

iii. Check attribute names in object class files do match with (i)

iv. Check type of attributes eg. string data received for *int* defined attribute

2. Use the standalone *snmpwalk* utility as a test tool to check SNMP access, authentication parameters and OIDs. For example:

```
snmpwalk -v 2c -c fraclmye taplow-11.skills-1st.co.uk .1.3.6.1.4.1.2021.16.2.1

snmpwalk -v 3 -a MD5 -A fraclmyea -l authNoPriv -u jane zenny .1.3.6.1.4.1.2020.16.2.1
```

a. Check that Zenoss really does have the same parameters configured in the various SNMP zProperties for the device.

b. Note that the leading dot "." on the OID is optional (it confirms a "fully-qualified" OID starting at the root of the MIB tree).

3. Insert extra *log.debug* statements in the modeler code and rerun the *zenmodeler* command in debug. For Zenoss Core 4 and earlier, you only need to recycle *zenhub* and *zopectl* daemons if you insert extra log statements. No reinstall of the ZenPack is necessary. For example:

```
log.debug('logMatchTable is %s ' % (logMatchTable))
```

4.

9.7 GUI display code

JavaScript code is required to determine **what** attributes should be displayed and **how** they should be displayed.

Note that the zenpacklib utility can dramatically reduce the amount of human-created JavaScript but zenpacklib does not do everything and the ZenPack creator needs to

understand what names are related in the different ZenPack elements. For that reason, this sample will hand-code all the JavaScript. Subsequent examples will take advantage of zenpacklib.

A ZenPack that is providing JavaScript must also provide an i*nfo.py,* an *interfaces.py* and at least one *configure.zcml* file.

9.7.1 JavaScript for new components

TODO: Explain why sample doesn't work.

I cannot make the constructs in the sample *ExampleDevice.js* file under the *browser*/*resources*/*js* directory, work. The left-hand component menu sees the correct component configuration and *Details* in the dropdown menu shows correct data but the component configuration panel only shows the default fields; no fields customized in the ZenPack js file.

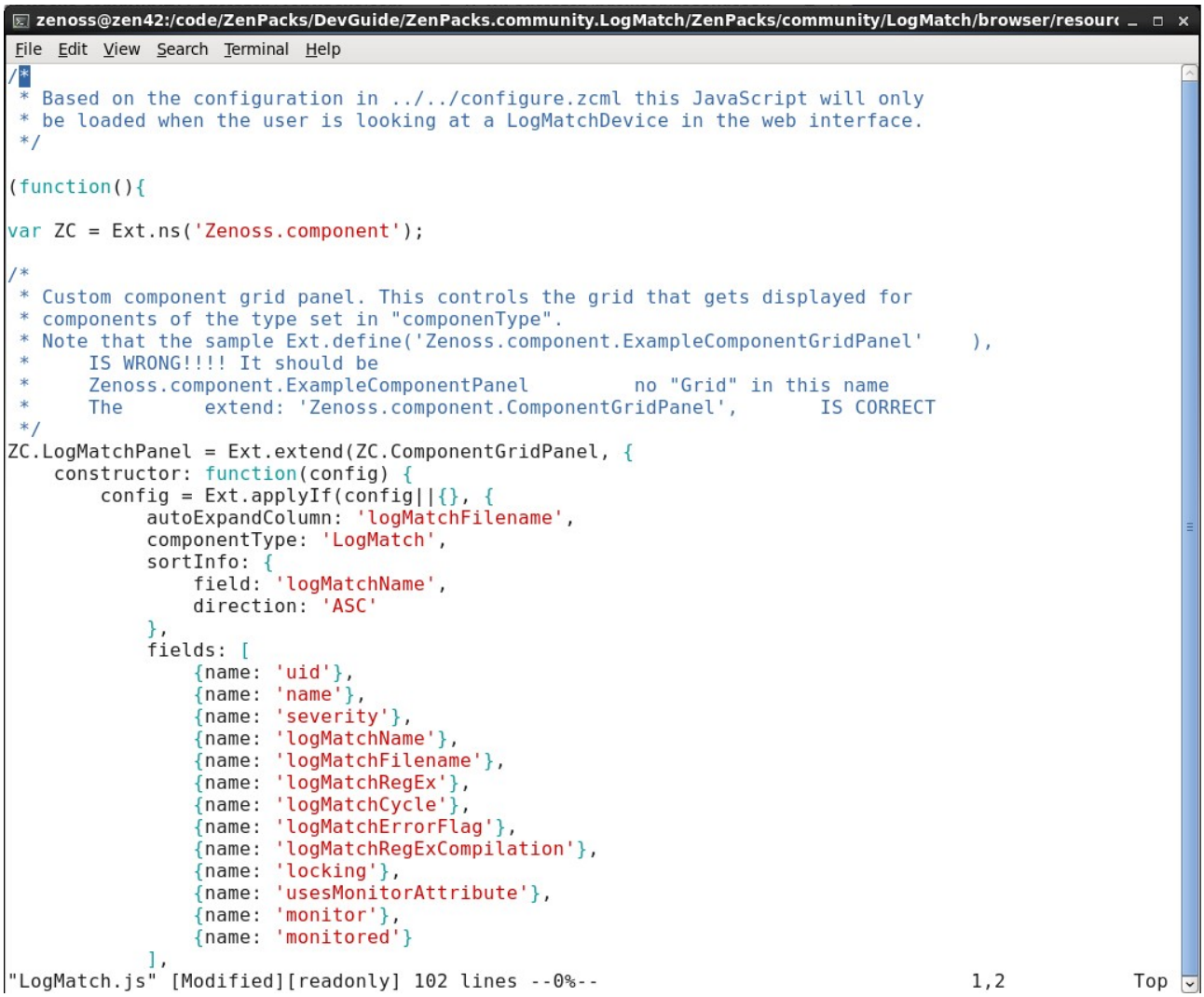


Figure 99: Start of LogMatch.js under browser/resources/js directory of the ZenPack

The code is extending the standard **ComponentGridPanel** that is defined in *$ZENHOME*/*Products*/*ZenUI3*/*browser*/*resources*/*js*/*zenoss*/*ComponentPanel.js*.

Note that the new component panel **must** be called <object component name>Panel eg. *LogMatchPanel*. **There should be no *Grid* in this name**.

At the top of the definition can be placed parameters that apply to the whole panel, rather than an individual field:

- **autoExpandColumn** - the fieldname (which must be defined below) is allowed to take any remaining width in the panel
- **sortInfo** - which field (defined below) will the panel be sorted on by default

The **fields** section defines any field that will be used in the GUI dialogue panel. These names **must** match attribute names in the **corresponding ComponentInfo** class; that is, the *LogMatchInfo* class defined in *info.py* in the base directory of the ZenPack. Although by convention these names will often be the same as component object attributes, it is the **LogMatchInfo** class attributes that are required, not the *LogMatch* class attributes.

Some of the field names may be inherited.

zenoss@zen42:/opt/zenoss/Products/Zuul

File Edit View Search Terminal Help

```
##############################################################################
#
# Copyright (C) Zenoss, Inc. 2010, all rights reserved.
#
# This content is made available according to terms specified in
# License.zenoss under the directory where your Zenoss product is installed.
#
##############################################################################


from zope.interface import implements
from zope.component import adapts

from Products.Zuul.interfaces import IComponentInfo, IComponent
from Products.Zuul.infos import InfoBase, ProxyProperty, HasEventsInfoMixin, LockableMixin
from Products.Zuul.form.builder import FormBuilder
from Products.Zuul.decorators import info
from Products.Zuul.utils import safe_hasattr as hasattr


class ComponentInfo(InfoBase, HasEventsInfoMixin, LockableMixin):
    implements(IComponentInfo)
    adapts(IComponent)

    @property
    @info
    def device(self):
        return self._object.device()

    @property
    def usesMonitorAttribute(self):
        return True

    monitor = ProxyProperty('monitor')

    @property
    def monitored(self):
        return self._object.monitored()
"./infos/component/__init__.py" [readonly] 86 lines --1%--                1,9           Top
```

Figure 100: ComponentInfo attributes of usesMonitorAttribute, monitor and monitored

The *ComponentInfo* class is in *$ZENHOME/Products/Zuul/infos/component/__init__.py* and defines the *usesMonitorAttribute*, *monitor* and *monitored* attributes.

The *locking* attribute is defined in the *LockableMixin* class in *$ZENHOME/Products/Zuul/infos/__init__.py*.

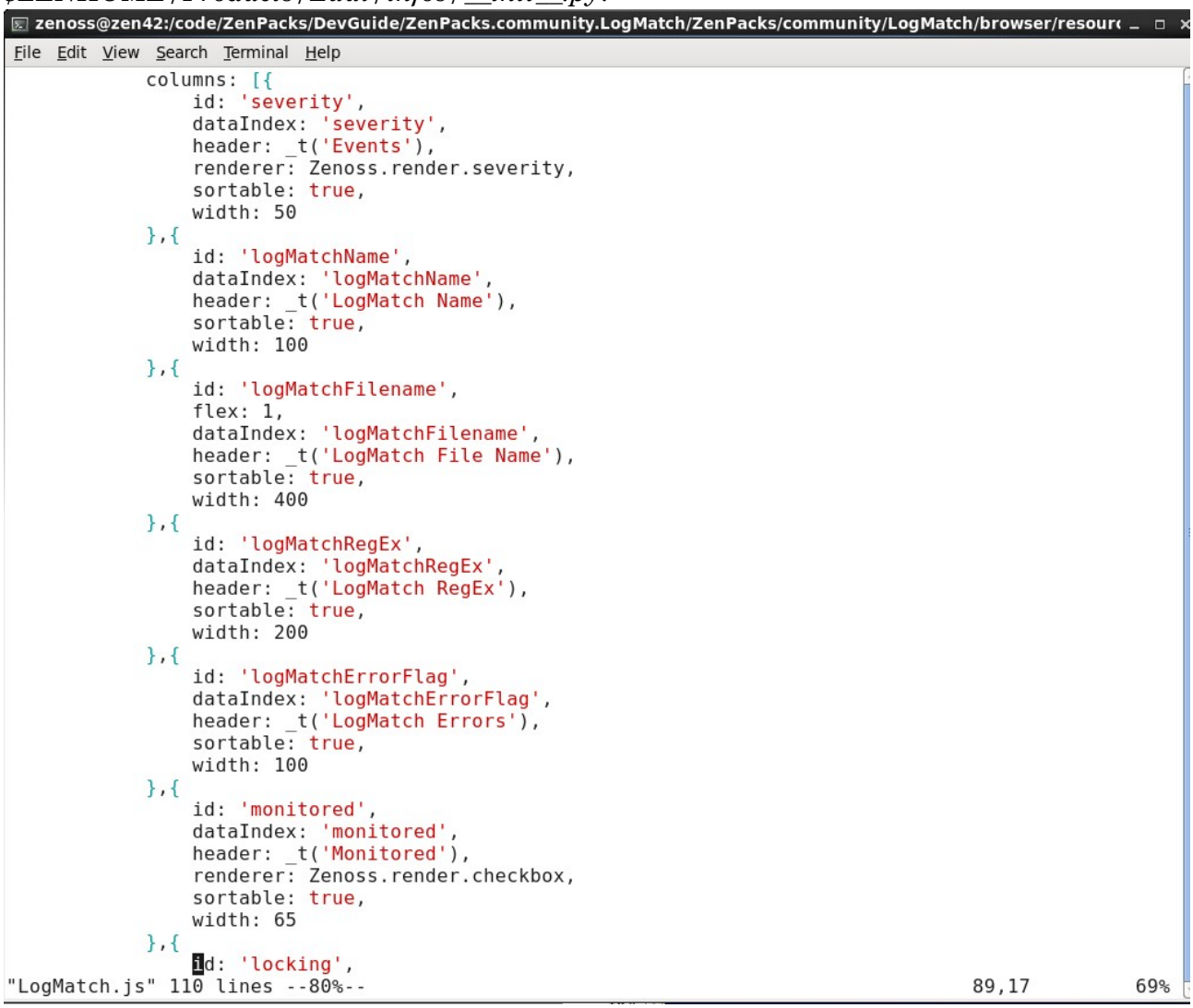


Figure 101: columns definitions for LogMatch.js

The stanza that determines exactly what gets displayed and how, is ***columns***.

In columns:

- The *id* can be any unique name
- The *dataIndex* must match an attribute in the **ComponentInfo** class and **must** have been defined in the fields statement above.
- The *header* field is simply a text string for the column header
- It is good practice to include *sortable: true*
- The *width* is a pixel width and is adjusted after examining the panel display. Remember that the overall *autoExpandColumn* directive, if used, will take any remaining column width.

- By default, a column is simply rendered as text but there are several Zenoss-provided renderers (see *$ZENHOME/Products/ZenUI3/browser/resources/js/zenoss/Renderers.js*) and it is also possible to write your own.
    - The column that displays *Events* severity uses *Zenoss.render.severity* to provide the color-coded icon.
    - The *monitored* column uses *Zenoss.render.checkbox*.
    - The *locking* column uses *Zenoss.render.locking_icons*.

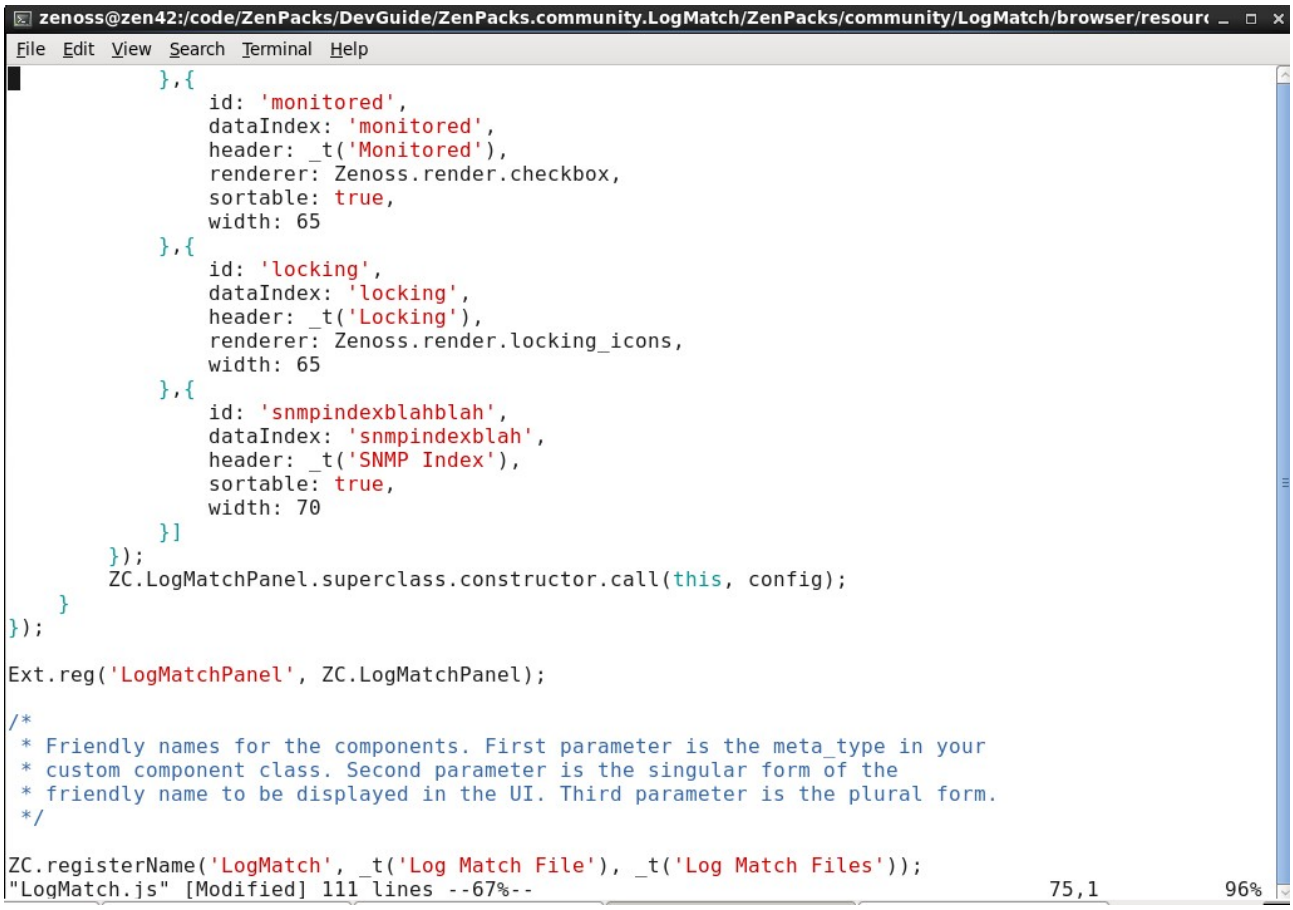


Figure 102: End of LogMatch.js JavaScript file

The rest of the JavaScript file is boilerplate. The line that controls the component name in the left-hand menu is:

```
ZC.registerName('LogMatch', _t('Log Match File'), _t('Log Match Files'));
```

where the first parameter must match the **meta-type** declared at the top of the *LogMatch.py* component object file. By convention this meta-type is the same as the component object name. In *LogMatch.py*:

```
class LogMatch(DeviceComponent, ManagedEntity):
  meta_type = portal_type = "LogMatch"
```

9.7.2 info.py

The ***info.py*** file abstracts object attribute information saved in the Zope Object Database (ZODB), that will be displayed to the user. It also allows code to be written for display that it is not part of the class definition. Note that the file must have this exact name.

It describes **what** will be displayed not how something will be displayed.

zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.LogMatch/ZenPacks/community/LogMatch

File Edit View Search Terminal Help

```
# This file is the conventional place for "Info" adapters. Info adapters are
# a crucial part of the Zenoss API and therefore the web interface for any
# custom classes delivered by your ZenPack. Examples of custom classes that
# will almost certainly need info adapters include datasources, custom device
# classes and custom device component classes.

# Mappings of interfaces (interfaces.py) to concrete classes and the factory
# (these info adapter classes) used to create info objects for them are managed
# in the configure.zcml file.

from zope.component import adapts
from zope.interface import implements

from Products.Zuul.infos import ProxyProperty
from Products.Zuul.infos.component import ComponentInfo

from ZenPacks.community.LogMatch.LogMatch import LogMatch
from ZenPacks.community.LogMatch.interfaces import ILogMatchInfo

class LogMatchInfo(ComponentInfo):
    implements(ILogMatchInfo)
    adapts(LogMatch)

    logMatchName = ProxyProperty("logMatchName")
    logMatchFilename = ProxyProperty("logMatchFilename")
    logMatchRegEx = ProxyProperty("logMatchRegEx")
    logMatchCycle = ProxyProperty("logMatchCycle")
    logMatchErrorFlag = ProxyProperty("logMatchErrorFlag")
    logMatchRegExCompilation = ProxyProperty("logMatchRegExCompilation")
    snmpindexblah  = ProxyProperty("snmpindex")

"info.py" [readonly] 31 lines --3%--                                    1,1
```

Figure 103: info.py in base directory of LogMatch ZenPack

The *LogMatchInfo* class inherits from the standard *ComponentInfo* class defined in *$ZENHOME/Products/Zuul/infos/component/__init__.py*. It shuttles data between the ***LogMatch*** device component class defined in this ZenPack and the ***ILogMatchInfo*** class defined in i*nterfaces.py* in this ZenPack.

In practice, the names on the left-hand side of definitions in *info.p*y need to match names used in *interfaces.py* and in the fields of any JavaScript files. The right-hand side of info.py definitions specify what device / component object attributes are to be displayed.

To demonstrate the point, the last line in Figure 103 links the “real” *snmpindex* attribute found on any component, with a name ***snmpindexblah***, to be used in *interfaces.py* and it can be seen in the JavaScript file in Figure 102 as the *dataIndex* value of the last column definition.

9.7.3 interfaces.py

interfaces.py describes **how** the data is displayed (and again, this filename is prescribed).

zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.LogMatch/ZenPacks/community/LogMatch

File Edit View Search Terminal Help

```
from Products.Zuul.form import schema
from Products.Zuul.interfaces.component import IComponentInfo

# ZuulMessageFactory is the translation layer. You will see strings intended to
# been seen in the web interface wrapped in _t(). This is so that these strings
# can be automatically translated to other languages.
from Products.Zuul.utils import ZuulMessageFactory as _t

# In Zenoss 3 we mistakenly mapped TextLine to Zope's multi-line text
# equivalent and Text to Zope's single-line text equivalent. This was
# backwards so we flipped their meanings in Zenoss 4. The following block of
# code allows the ZenPack to work properly in Zenoss 3 and 4.

# Until backwards compatibility with Zenoss 3 is no longer desired for your
# ZenPack it is recommended that you use "SingleLineText" and "MultiLineText"
# instead of schema.TextLine or schema.Text.
from Products.ZenModel.ZVersion import VERSION as ZENOSS_VERSION
from Products.ZenUtils.Version import Version
if Version.parse('Zenoss %s' % ZENOSS_VERSION) >= Version.parse('Zenoss 4'):
    SingleLineText = schema.TextLine
    MultiLineText = schema.Text
else:
    SingleLineText = schema.Text
    MultiLineText = schema.TextLine


class ILogMatchInfo(IComponentInfo):
    logMatchName = SingleLineText(title=_t(u"LogMatch Name"))
    logMatchFilename = SingleLineText(title=_t(u"LogMatch Filename"))
    logMatchRegEx = SingleLineText(title=_t(u"LogMatch RegEx"))
    logMatchCycle = schema.Int(title=_t(u"LogMatch Cycle"))
    logMatchErrorFlag = schema.Int(title=_t(u"LogMatch ErrorFlag"))
    logMatchRegExCompilation = SingleLineText(title=_t(u"LogMatch RegExCompilation"))
    snmpindexblah = SingleLineText(title=_t(u"SNMP Index for datasource"))
"interfaces.py" 34 lines --2%--                                    1,1           All
```

Figure 104: interfaces.py file for LogMatch ZenPack

The *ILogMatchInfo* class inherits from *IComponentInfo* defined in *$ZENHOME/Products/ /Zuul/interfaces/component.py*. The definitions for this ZenPack are simply string or int data but more complex constructs are possible.

The left-hand side names must match entries defined in *info.py*.

The right-hand side specifies the name that will be used in the GUI for this attribute and its type. Specifically, these names are seen in the *Details* dropdown menu for components; any details to be seen must appear both in *info.py* and i*nterfaces.py*.

When working with SNMP-based ZenPacks, it is good practice to display the *snmpindex* attribute in the *Details* dropdown, as a debugging aid.

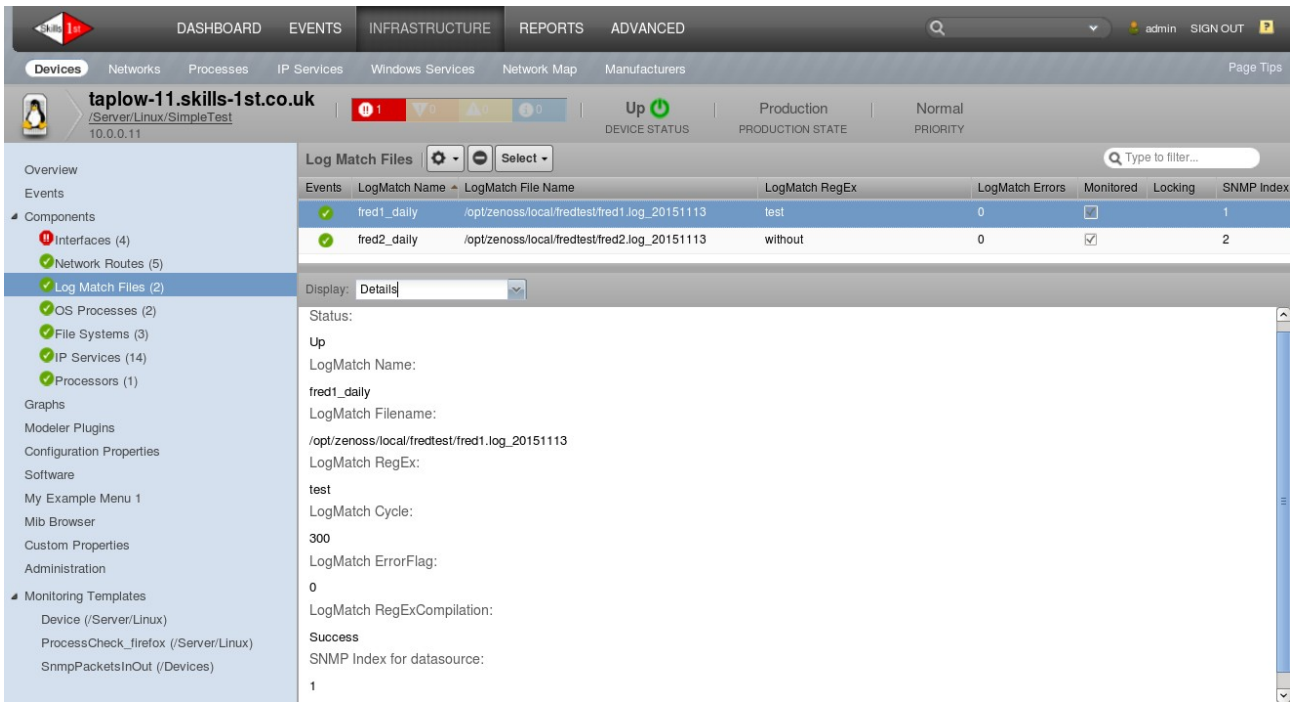


Figure 105: Component panel for LogMatch component with Details dropdown data

9.7.4 configure.zcml

Something needs to ultimately tie together the different display elements. That is the role of ***configure.zcml*** which provides the "glue" between interfaces and JavaScript display code and this exact name will be searched for by the Zope mechanisms (ZCML = Zope Configuration Markup Language, a variant of XML).

Often a ZenPack will use two separate c*onfigure.zcml* files.

- **configure.zcml in the base** directory of the ZenPack
  - Provides ***adapter*** stanzas that link an info with an interface, optionally for an object class. If no object class is defined then the default is for all classes.

```
 <?xml version="1.0" encoding="utf-8"?>
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:zcml="http://namespaces.zope.org/zcml" >

  <!-- Includes: Browser Configuration -->
  <include package=".browser"/>

  <adapter factory=".info.LogMatchInfo"
     for=".LogMatch.LogMatch"
     provides=".interfaces.ILogMatchInfo"
     />

</configure>
```

  - The two namespace definitions at the top of the file are mandatory.
  - It is common to have <*include package=".browser"*/> in this configure.zcml. It is pointing to the *browser* subdirectory where a second *configure.zcml* provides viewlet information. This is not mandatory but is good practice.

- **configure.zcml in the *browser*** subdirectory of the ZenPack:
  - Provides **viewlet** stanzas that link object displays to JavaScript files.
  - The following is required boilerplate:

```
<?xml version="1.0" encoding="utf-8"?>
<configure xmlns="http://namespaces.zope.org/browser">
```

  -

zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.LogMatch/ZenPacks/community/LogMatch

File Edit View Search Terminal Help

```
<?xml version="1.0" encoding="utf-8"?>
<configure xmlns="http://namespaces.zope.org/browser">

    <!-- A resource directory contains static web content. -->
    <!-- name can be anything unique but is used below in the paths statement -->
    <!-- directory is the path from this configure.zcml to where the js directory is -->
    <resourceDirectory
        name="LogMatchJavascript"
        directory="resources"
        />

    <!-- Register custom JavaScript for LogMatch devices. -->
    <!-- name can be anything unique -->
    <!-- In paths, /++resource++LogMatchJavascript substitutes the LogMatchJavascript name
         defined above, into the path to the Javascript file, resulting in
         resources/js/LogMatch.js, relative to where this configure.zcml is -->
    <!-- The weight field indicates the order of multiple viewlets where 1 would be at the top
 -->
    <!-- In for, the path to the LogMatchDevice class in the LogMatchDevice module
         is up one directory from where this configure.zcml sits  -->

    <viewlet
        name="js-LogMatchJavascriptJs"
        paths="/++resource++LogMatchJavascript/js/LogMatch.js"
        weight="10"
        for="..LogMatchDevice.LogMatchDevice"
        manager="Products.ZenUI3.browser.interfaces.IJavaScriptSrcManager"
        class="Products.ZenUI3.browser.javascript.JavaScriptSrcBundleViewlet"
        permission="zope2.Public"
        />

</configure>
"browser/configure.zcml" 31 lines --3%--                    1,1           All
```

Figure 106: configure.zcml in LogMatch ZenPack browser subdirectory

  - In the *resourceDirectory* stanza:
    - *name* can be any unique name
    - *directory* will be substituted for this name in the *paths* statement
  - In the *viewlet* stanza:
    - *name* can be any unique name
    - *paths* gives the path to the JavaScript file, typically including a substitution of the resourceDirectory name; hence *paths* will be in ***resources/js/LogMatch.js***, starting from where this *configure.zcml* resides
    - *weight* indicates the order of multiple viewlets where 1 would be at the top
    - *for* restricts the use of this JavaScript file to the context of **devices** of class *LogMatchDevice* in the module *LogMatchDevice* (ie. in the file *LogMatchDevice.py*)

- The rest is boilerplate.

9.7.5 Where do things go wrong with GUI display code?

1. Field names in JavaScript files must match **ComponentInfo** class attributes, not the *DeviceComponent* class attributes.
2. Any field defined in the *columns* stanza **must** be declared in the *fields* stanza.
3. JavaScript should not have a comma at the end of the last statement in a clause.
4. Brackets mismatch is extremely easy, especially in JavaScript files. The *vi* % command to match brackets is enormously helpful.
5. Ensure that *IComponentInfo* classes in *interfaces.py* specify suitable display types. Using a *schema.Int* definition when the object data is actually a string, will cause issues.
6. Forgetting to write / update *info.py, interfaces.py* and/or *configure.zcml* is common.
7. Specifying a "for" statement in *configure.zcml* that is incorrect will result in no errors but no component display.
8. Errors in a *configure.zcml* are scary as they will prevent *zenhub* from starting, with an error message. Fortunately it is quite good at pinpointing where the error is.

```
  File "/opt/zenoss/lib/python/zope/configuration/fields.py", line 229, in
fromUnicode
  raise InvalidToken("%s in %s" % (v, u))
zope.configuration.xmlconfig.ZopeXMLConfigurationError: File
"/opt/zenoss/etc/site.zcml", line 16.2-16.23
  ZopeXMLConfigurationError: File
"/code/ZenPacks/DevGuide/ZenPacks.community.LogMatch/ZenPacks/community/Log
Match/configure.zcml", line 15.4-18.11
  ConfigurationError: ('Invalid value for', 'for', 'ImportError: Module
ZenPacks.community.LogMatch.LogMatch has no global LogMatchDevice in
.LogMatch.LogMatchDevice')
```

9.

9.7.6 * Architecture of the ComponentPanel

Many ZenPacks extend device component capabilities so understanding the component panel is important. The code that defines it is in *$ZENHOME/Products/ZenUI3/browser/resources/js/zenoss*.

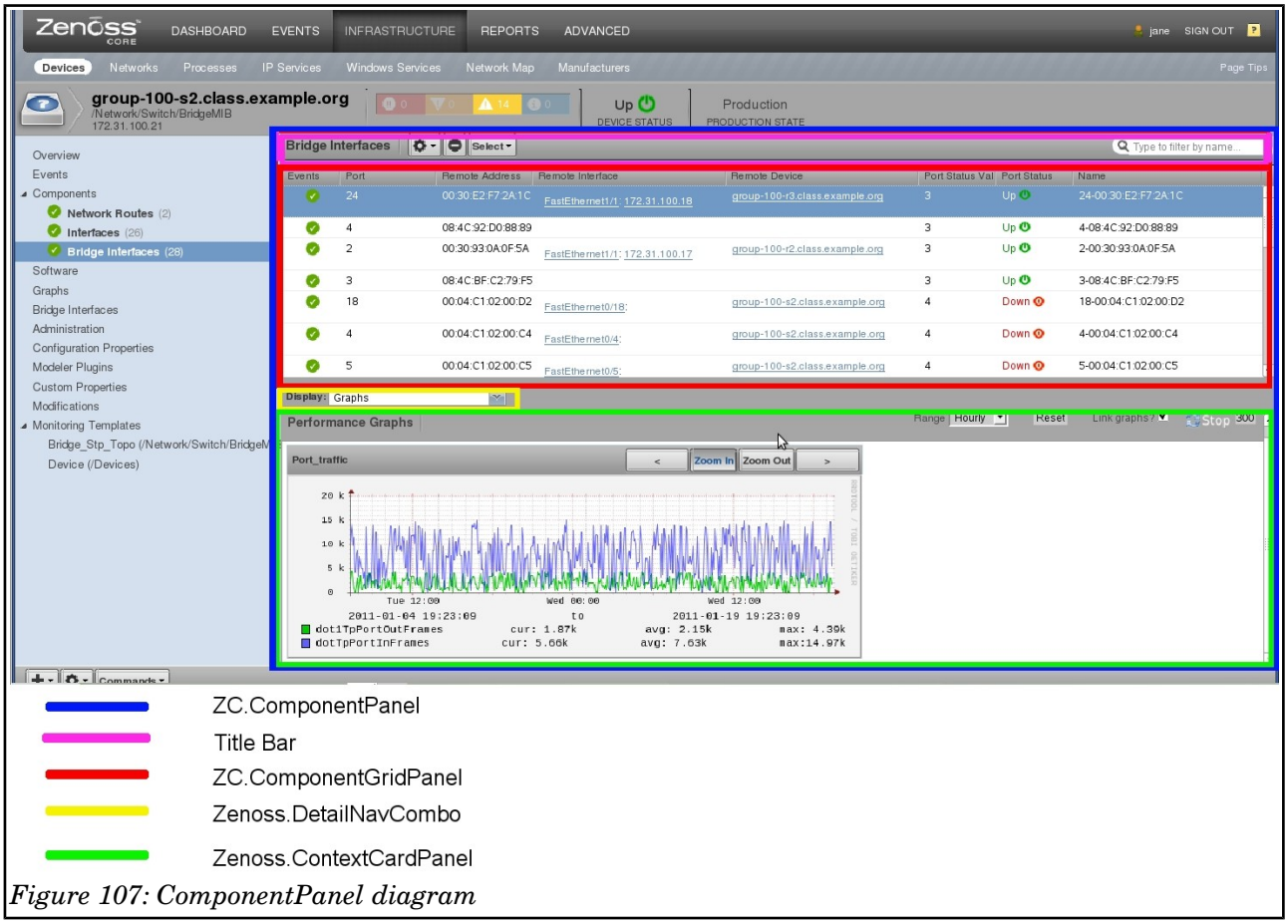


Figure 107: ComponentPanel diagram

Examining *ComponentPanel.js* (line numbers here are given for Zenoss 4.2.5, SUP 457):

Zenoss.nav.register from lines 59 - 260 sets up the default dropdown menus from the Display DetailNavCombo box – Graphs, Events, Details and Templates. This is why the LogMatch ZenPack has these menus without the ZenPack having to define it.

ZC.ComponentDetailNav from lines 262 to 320, is concerned with augmenting the Display dropdown menu and explicitly prohibits menu items with the names status, events, resetcommunity,pushconfig, objtemplates,modeldevice and historyevents.

ZC.ComponentPanel runs from lines 325 to 458. Fundamentally there are four main areas inside the entire Component Panel (outlined in blue in Figure 54):

The Title Bar (tbar) outlined in pink

The Component Grid Panel with attribute values for each instance of a component, outlined in red

The title bar of the bottom half of the window is the text Display and a DetailNavCombo dropdown box to select the data to be seen at the bottom. This is outlined in yellow. This section prohibits the display of the Graphs dropdown menu if the monitor attribute is not set for the object. It also filters out any dropdown menu items that match the list given above under *ZC.ComponentDetailNav*.

The Context Card Panel is the bottom window with graphs, events, details, etc and is outlined in green.

ZC.ComponentGridPanel (lines 461 – 639) defines the container for the top part of the component panel – the Component Grid Panel. It defines the default object attribute fields that will be used to help construct the top panel, unless they are overridden by custom JavaScript via config.fields. Omitting any of these fields in custom JavaScript may lead to unpredictable results.

The remaining definitions **ZC.IPInterfacePanel, ZC.WinServicePanel, ZC.IpRouteEntryPanel, ZC.IpServicePanel, ZC.OSProcessPanel, ZC.FileSystemPanel, ZC.CPUPanel, ZC.ExpansionCardPanel, ZC.PowerSupplyPanel, ZC.TemperatureSensorPanel** and **ZC.FanPanel** each define the specific ComponentGridPanel for the individual, standard component objects. Note that the code here would be good samples from which to start writing custom JavaScript for new component objects.

When the *ZC.ComponentPanel* constructor executes, the upper space of the window (designated by the red border) is left empty. A *ZC.ComponentGridPanel* is loaded into this space by the *ZC.ComponentPanel.setContext* method. The code begins on line 412 of *ComponentPanel.js*. One of the parameters passed to this method is type, which is a string containing the component object class name. The code searches for a registered component with the name type + "Panel". So if your component is named *LogMatch*, the code searches for a registered object named *LogMatchPanel* and uses the code to fill in this upper slot. Note that all the definitions for standard components at the end of *ComponentPanel.js* follow this model – *IpInterfacePanel, FileSystemPanel*, etc.

9.8 Adding component performance templates

The ZenPack has added a component of object class **LogMatch**. The requirements specification in section 9.2 said to gather the **logMatchCurrentCounter** (1.3.6.1.4.1.2021.16.2.1.7) as the OID best representing the number of matches in a logfile. Remember that this OID is tabular data and there will be a further index / instance id on the end of this OID, one for each match file configured.

The tricks when creating **component** performance templates are:

- The name of the template must **exactly** match the name of the component object class.
- A component template is **automatically** bound to all components of the matching class (and indeed should **not** be manually bound).
- A component template does not include the index (or instance) for an OID. That comes from the *snmpindex* attribute of the component object.
- The component template is **not** seen in the left-hand menu under *Monitoring Templates* but **can** be seen by selecting a component and then selecting the *Templates* dropdown from the *Display* menu.

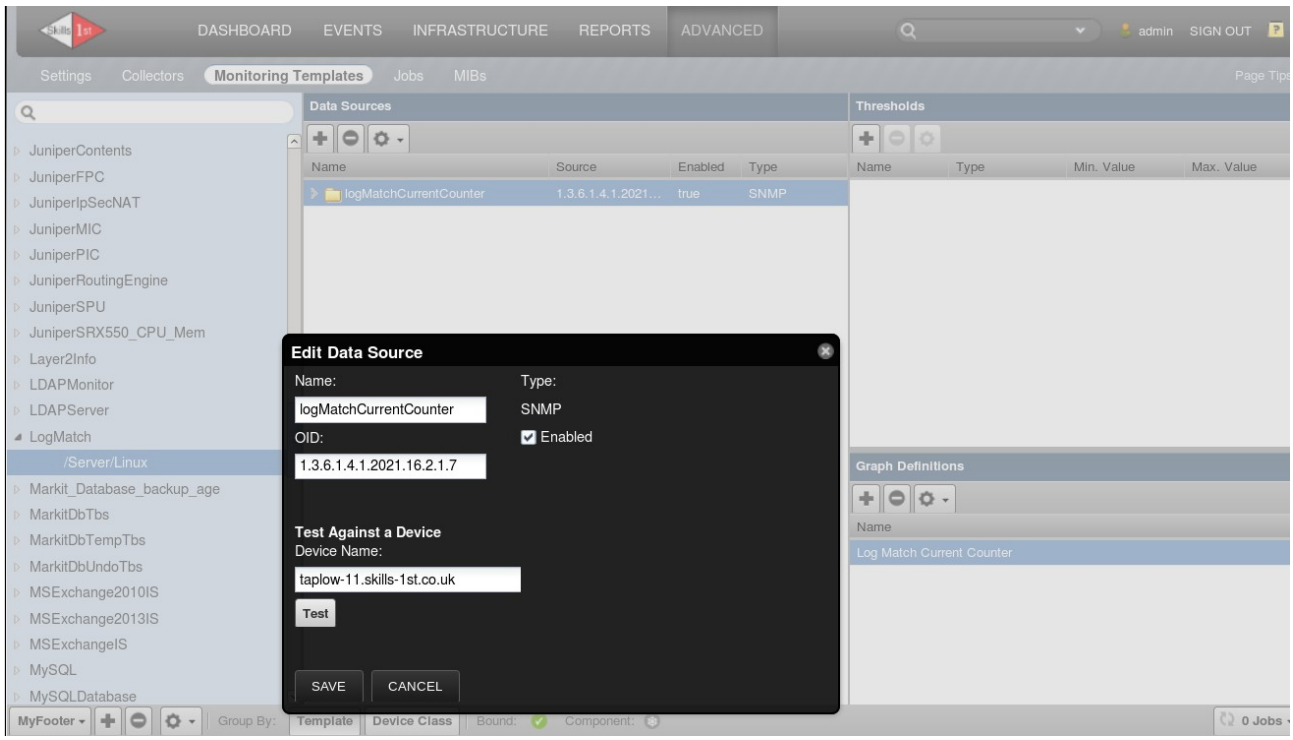


Figure 108: LogMatch component template with logMatchCurrentCounter datasource

Provided a graph is also created based on the datasource / datapoint seen in Figure 108, the *Graphs* dropdown option should start showing data after 2 *zenperfsnmp* cycles (*zenperfsnmp* cycle time is 300s by default).

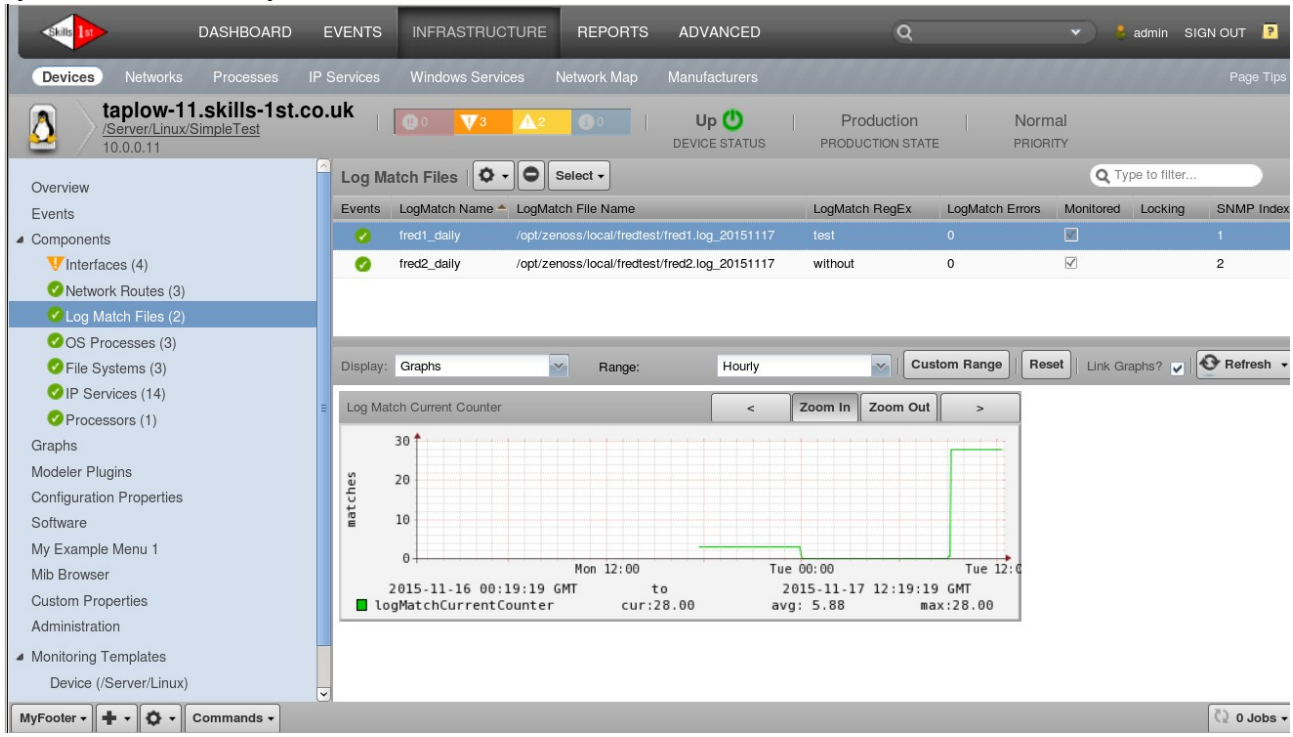


Figure 109: Log Match current Counter graph for LogMatch component

Note that the datapoint is created automatically for an SNMP datasource and, by default, will be of type *GAUGE*. In practice, this means that the *logMatch* value being graphed will be the total number of matches. Although the SNMP agent is providing a *COUNTER* value, Zenoss, by default, is not treating it as such.

When monitoring real files, it may be better to change the datapoint to a *COUNTER* type to get a **rate of change** of matches. If you do this, ensure that you delete any existing *logMatchCurrentCounter_logMatchCurrentCounter.rrd* files under *$ZENHOME/perf/Devices* as they will fail to collect data after the data type change.

Better still, the datapoint mechanism provides for a *DERIVE* type which acts like a *COUNTER* but, if you set the *RRD Minimum* value to zero then it prevents graph spikes if a counter wraps around.

When the template is tested, use the *Action* icon from the bottom of the left-hand menu from *ADVANCED -> Monitoring Templates*, to add the template to the correct ZenPack.

9.9 Adding other ZenPack elements through the GUI

Since the LogMatch ZenPack is working on OIDs in the UCD-SNMP-MIB, it would be useful to add that to the ZenPack. Assuming that this MIB is already loaded into Zenoss, using the GUI, simply navigate to *ADVANCED-> MIBs*, select the tick-box beside the UCD-SNMP-MIB and from the dropdown menu at the top of the list of MIBs, choose *Add to ZenPack*. In the ensuing dropdown box, choose the *ZenPacks.community.LogMatch* ZenPack..

Remember that it will be added to the ZenPack's *objects/object.xml* file, but only when the ZenPack is exported.

9.10 Finalising the ZenPack

When the ZenPack is complete and tested, there are a few things to check:

- Is there a *README.rst* in the top-level directory?
- Has the version, author and license information been completed? This will be Version 1.0 of the LogMatch ZenPack.
- Have any Zenoss or Zenoss ZenPacks dependencies been filled in?
- It is good practice to remove any unused files and directories from the ZenPack directory hierarchy.
- Has the ZenPack been exported?

The export writes the *object/objects.xml* file and creates the ***.egg*** file.

The egg file is created by first copying the entire ZenPack directory hierarchy to its own *build/lib* subdirectory. The egg file is actually constructed from this build subtree. Note that for the .egg file to be completely up-to-date, the ZenPack must be exported after **any** change; even adding a comma to *README.rst* needs an export for that change to be included in the egg file.

The egg file is first created under the ZenPack's ***dist*** directory and it is then copied to ***$ZENHOME/export***.

Note that there is a bug whereby the *build/lib* subdirectory is **not** cleaned out before export. This means that existing files **will** be updated, new files **will** be added but if old files have been deleted then they will **still exist** in the *build/lib* subdirectory and hence, in the new egg file. This is documented in ticket 7324 (http://dev.zenoss.com/trac/ticket/7324) and https://jira.zenoss.com/browse/ZEN-20977 . If files have been deleted from ZenPack sources, ensure that everything under the *build/lib* subdirectory is removed before exporting the ZenPack (this is safe to do).

Once an egg file has been created in the *export* subdirectory, it can be moved to a different system and loaded there.

9.11 Extending the ZenPack to modify the device Overview

The LogMatch ZenPack has created *versionTag* and *versionDate* attributes for a *LogMatch* Device and the *LogMatchDeviceMap* modeler populates these attributes; however the only way to see these is with the ZMI or *zendmd*.

There is an excellent wiki tip at http://wiki.zenoss.org/Device_Overview_Panels which describes how to modify the Overview page for a device. There is also a link to the code on GitHub, https://github.com/cluther/ZenPacks.example.CustomOverview .

JavaScript can be created to modify any of the Overview panels:

- deviceoverviewpanel_summary (top-left)
- deviceoverviewpanel_idsummary (top-middle)
- deviceoverviewpanel_descriptionsummary (top-right)
- deviceoverviewpanel_customsummary (bottom-left)
- deviceoverviewpanel_systemsummary (inner panel in bottom-left)
- deviceoverviewpanel_snmpsummary (bottom-right)

The GitHub example uses *__init__.py* in the base directory of the ZenPack, to create a new attribute on the standard device, called *contact*. To access this through a JavaScript file, an info definition is needed so the standard *DeviceInfo* class is extended with *contact*. Note the imports at the top of the file.

```
from Products.ZenModel.Device import Device
from Products.Zuul.infos import ProxyProperty
from Products.Zuul.infos.device import DeviceInfo


# Set a default value for a device's contact.
Device.contact = ''

# Make a device's contact available through the API.
DeviceInfo.contact = ProxyProperty('contact')
```

Suppose we wish to modify the SNMP Summary panel to:

- Remove the SNMP community name
- Add the *versionTag* and *versionDate* fields to this panel

- This should only happen for *LogMatchDevice* devices

The solution requires:

- Some JavaScript to modify the Overview panel - *custom-overview-device.js*

  - We could use the existing JavaScript file but it is good practice and easier debugging to keep different bits of functionality separate.
  - It also potentially makes the application of the new JavaScript more flexible by using the "*for*" statement in *browser/configure.zcml*
- An entry in *browser/configure.zcml* to point to the new JavaScript file
- An entry in *info.py* to define the *LogMatchDeviceInfo* class with the two version properties
- An entry in *interfaces.py* to define the *ILogMatchDeviceInfo* class with entries for the two new properties
- An entry in the top-level *configure.zcml* with an adapter stanza for *LogMatchDevice*

9.11.1 custom-overview-device.js

The following is in *custom-overview-device.js* under *browser/resources/js*.

```
Ext.onReady(function() {
  var DEVICE_SNMP_ID = 'deviceoverviewpanel_snmpsummary';
  Ext.ComponentMgr.onAvailable(DEVICE_SNMP_ID, function(){
    var overview = Ext.getCmp(DEVICE_SNMP_ID);

    /* overview.addListener("afterrender", function(){ */
      overview.removeField('snmpCommunity');

      overview.addField({
        name: 'versionTag',
        xtype: 'displayfield',
        fieldLabel: _t('Version Tag')
      });
      overview.addField({
        name: 'versionDate',
        xtype: 'displayfield',
        fieldLabel: _t('Version Date')
      });
    /*}); */
  });
});
```

The standard snmp fields for the *DeviceInfo* class are defined in *$ZENHOME/Products/Zuul/infos/device.py*.

Note the two lines highlighted in pink that are commented out with /* ..... */ . This construct is used in the wiki item and in the GitHub sample but I cannot make it work. The device Overview field is completely blank. It works perfectly well without the *overview.addListener* function.

TODO: Why does the sample construct not work ?

9.11.2 browser/configure.zcml

A viewlet entry is required to attach to the new JavaScript.

```
<viewlet
  name="js-custom-overview-device"
  paths="/++resource++LogMatchJavascript/js/custom-overview-device.js"
  for="..LogMatchDevice.LogMatchDevice"
  weight="10"
  manager="Products.ZenUI3.browser.interfaces.IJavaScriptSrcManager"
class="Products.ZenUI3.browser.javascript.JavaScriptSrcBundleViewlet"
  permission="zope2.Public"
  />
```

Note the *for* statement to restrict the application of this JavaScript just to devices with object class *LogMatchDevice*.

9.11.3 info.py

Do not forget the extra import statements required for *DeviceInfo*, *LogMatchDevice* and *ILogMatchDevice*.

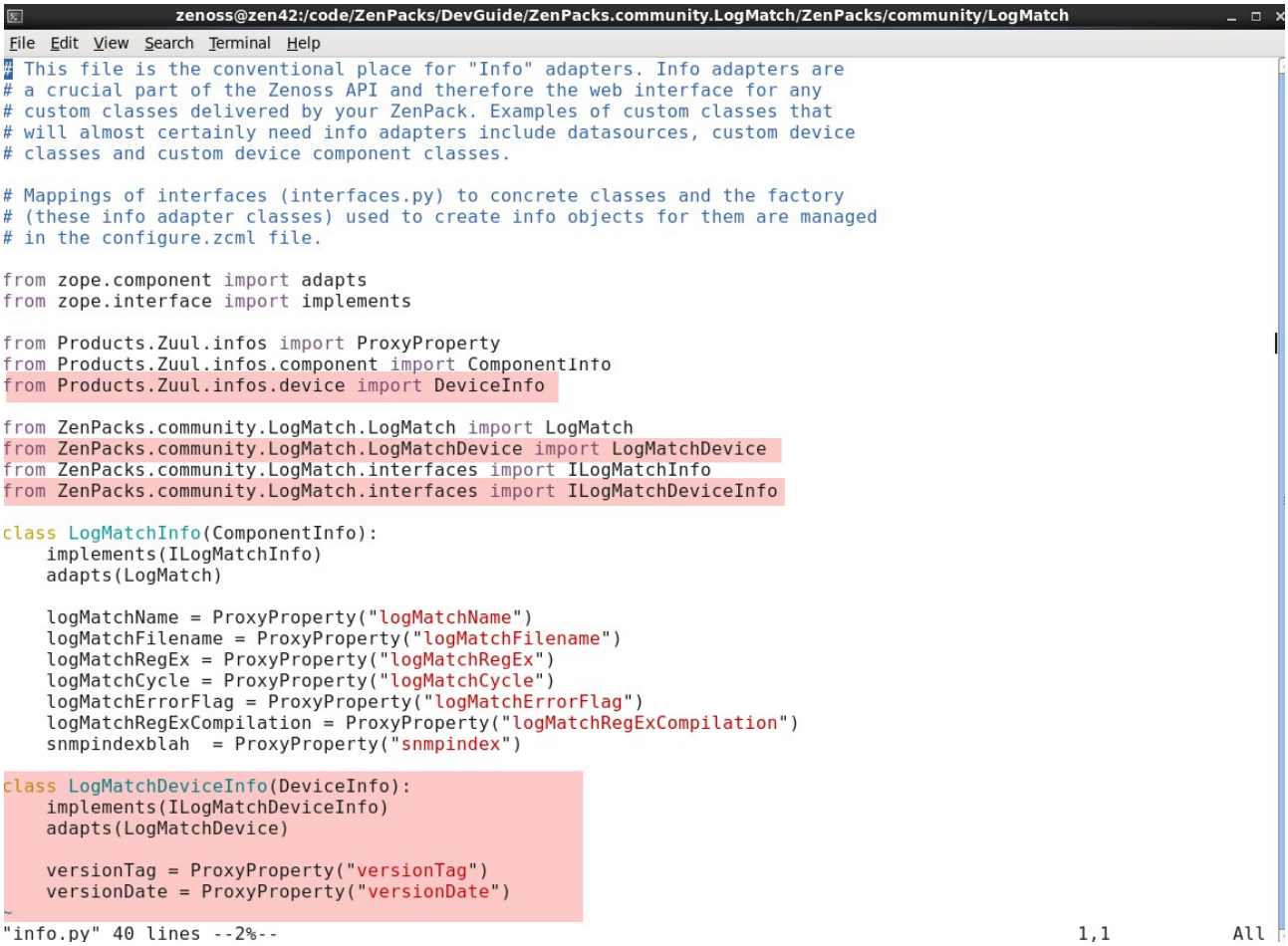


Figure 110: info.py with new lines highlighted for LogMatchDevice

9.11.4 interfaces.py

The extra entries in *interfaces.py* are an import for *IDeviceInfo* and entries for the two version attributes:

```
from Products.Zuul.interfaces.device import IDeviceInfo

class ILogMatchDeviceInfo(IDeviceInfo):
```

```
versionTag = SingleLineText(title=_t(u"Version Tag"))
versionDate = SingleLineText(title=_t(u"Version Date"))
```

9.11.5 Top-level configure.zcml

An adapter stanza is required for the *LogMatchDevice*:

```
<adapter factory=".info.LogMatchDeviceInfo"
    for=".LogMatchDevice.LogMatchDevice"
    provides=".interfaces.ILogMatchDeviceInfo"
    />
```

9.11.6 Testing the new changes

Once all edits are complete, it should be sufficient just to recycle *zenhub* and *zopectl*.

Refresh the browser window showing the Overview for a *LogMatchDevice* and check that the changes are correct.

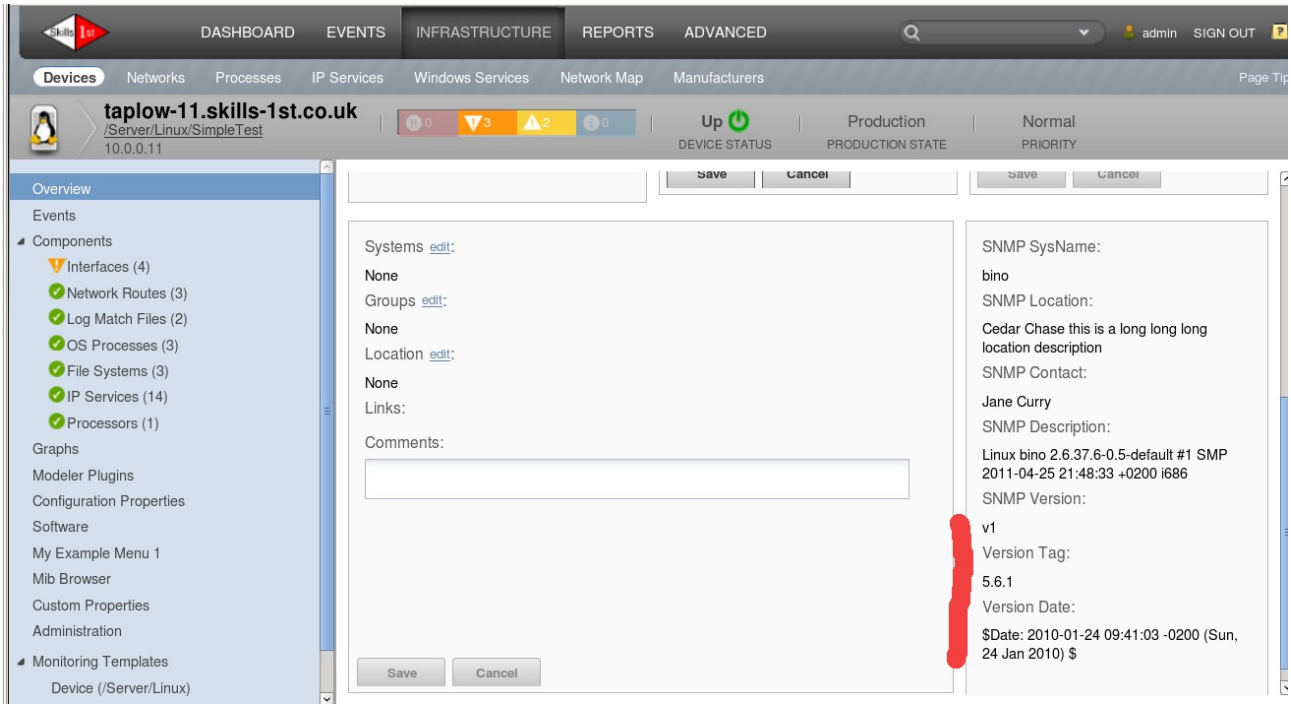


Figure 111: Modified overview panel for a LogMatchDevice instance

Note that the SNMP Community item has disappeared as well as adding the two version items.

Since new functionality has now been added to the ZenPack, the minor version number should be changed to 1.0.1.

Update the R*EADME.rst*, including the “Change History”.

When all is complete, re-export the ZenPack.

9.12 Modifying the ZenPack to remove LogMatchDevice

The ZenPack has created a new device object class of *LogMatchDevice* which can contain many *LogMatch* components. This is only possible if an instance of a device has its *zPythonClass* set to *ZenPacks.community.LogMatch.LogMatchDevice*. Fundamentally, a device instance is associated with one *zPythonClass*. In practice, this also often drives the Zenoss device class hierarchy; a device instance can only be in one Zenoss device class.

What happens if someone else creates a useful ZenPack with a device object class specialisation that we also want to use?

A possible answer is that the *LogMatch* component is so prevalent that it is worth adding to the base, Zenoss-supplied object class of ***Device***. This can be achieved by adding code to the *__init__.py* in the base directory and then adjusting other elements of the ZenPack to match.

Better still, a *Device* has existing *os* and *hw* components; the *LogMatch* component fits better as a relationship on the *os* (Operating System) component.

The solution requires:

- Entries in *__init__.py* to **monkeypatch** the *logMatchs* relationship on to the existing *OperatingSystem* relationship, *os*, and rebuild relations when the ZenPack is installed and removed. The *versionTag* and *versionDate* attributes could also be monkeypatched on to the *Device* object class.
- Modify *LogMatch.py* to change the relationship, *logMatchDevice*, to be *os* on an *OperatingSystem* component. The *device* method will also need to be modified.
- The entries for *LogMatchDevice* in *info.py* and *interfaces.py* become redundant
- *browser/configure.zcml* needs its "for" statement modifying so that a *LogMatch* component can be displayed for a standard *Device* (not a *LogMatchDevice*).
- The *LogMatchMap* modeler plugin will need modifying. The *LogMatchDeviceMap* modeler is fine as-is.
- If the *LogMatchDevice* object class is effectively removed, the *LogMatch* modeler plugins will need manually assigning to relevant devices.

When developing variants of a ZenPack, one way to maintain the original safely is to use **git** and develop the variant solution in a new git **branch**. git will be discussed in more detail in Chapter 17. This ZenPack variant will be developed in the ***device*** branch and will be Version 1.0.2 of the ZenPack.

9.12.1 monkeypatching standard objects in __init__.py

Initially, *__init__.py* was just comments, including some hints for what one might achieve there:

```
# Nothing is required in this __init__.py, but it is an excellent place to do
# many things in a ZenPack.
#
# The example below which is commented out by default creates a custom subclass
# of the ZenPack class. This allows you to define custom installation and
# removal routines for your ZenPack. If you don't need this kind of flexibility
# you should leave the section commented out and let the standard ZenPack
# class be used.
```

```
#
# Code included in the global scope of this file will be executed at startup
# in any Zope client. This includes Zope itself (the web interface) and zenhub.
# This makes this the perfect place to alter lower-level stock behaviour
# through monkey-patching.
```

monkeypatching is the dynamic replacement of attributes at runtime (and a method is a special case of an attribute). Great care must be taken when using this technique as it has potential to lead to unpredictable results.

In practice, it is a way of changing or extending Zenoss-supplied code.

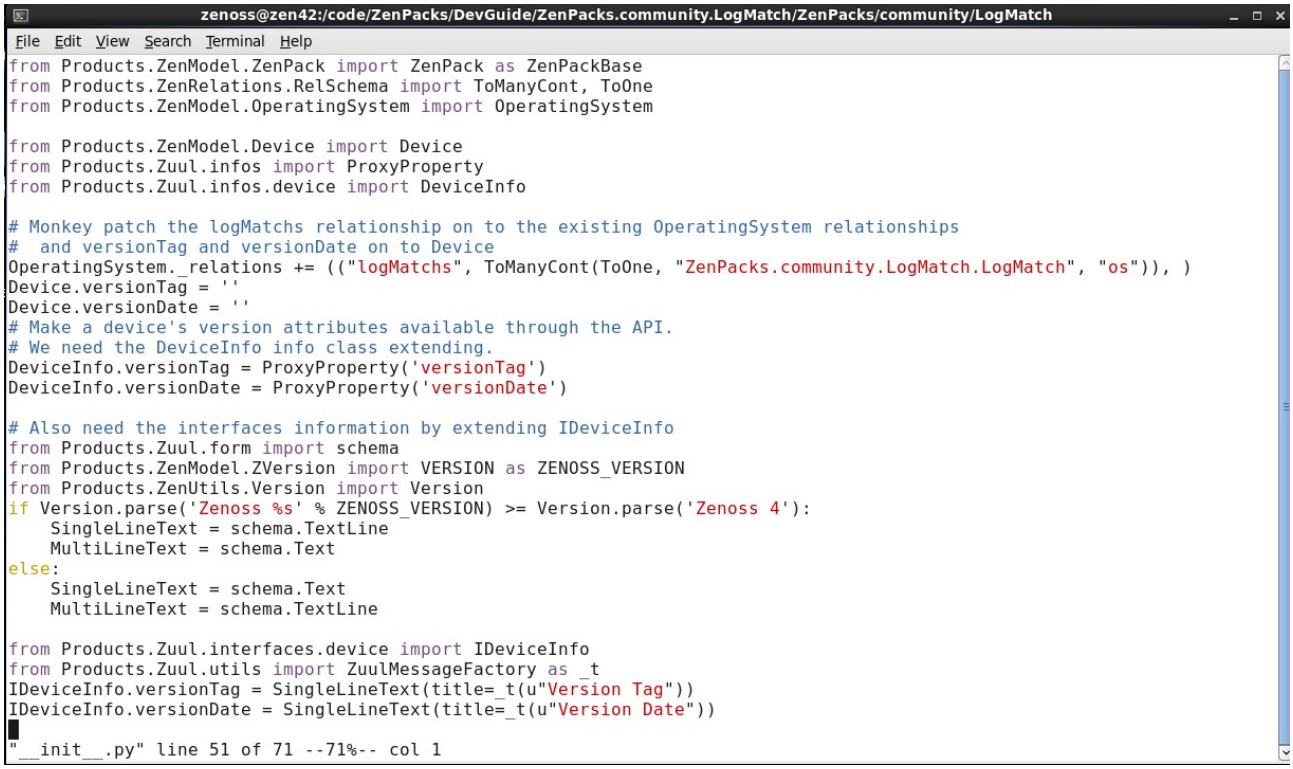


Figure 112: __init__.py for LogMatch ZenPack - monkeypatching

The requirement is to extend the existing *OperatingSystem* class, which is defined in *$ZENHOME/Products/ZenModel/OperatingSystem.py*. It already has a number of relationships defined:

```
class OperatingSystem(Software):

  totalSwap = 0L
  uname = ""

  _properties = Software._properties + (
    {'id':'totalSwap', 'type':'long', 'mode':'w'},
    {'id':'uname', 'type':'string', 'mode':''},
  )

  _relations = Software._relations + (
    ("interfaces", ToManyCont(ToOne,
      "Products.ZenModel.IpInterface", "os")),
    ("routes", ToManyCont(ToOne, "Products.ZenModel.IpRouteEntry", "os")),
    ("ipservices", ToManyCont(ToOne, "Products.ZenModel.IpService", "os")),
```

```
        ("winservices", ToManyCont(ToOne,
          "Products.ZenModel.WinService", "os")),
        ("processes", ToManyCont(ToOne, "Products.ZenModel.OSProcess", "os")),
        ("filesystems", ToManyCont(ToOne,
          "Products.ZenModel.FileSystem", "os")),
        ("software", ToManyCont(ToOne, "Products.ZenModel.Software", "os")),
      )
```

The following line in the *__init__.py* of the ZenPack achieves this:

```
OperatingSystem._relations += (("logMatchs", ToManyCont(ToOne,
       "ZenPacks.community.LogMatch.LogMatch", "os")), )
```

The standard *OperatingSystem* class is extended dynamically when the ZenPack is loaded.

Similarly, the standard *Device* object class has new attributes added for *versionTag* and *versionDate*. Since we may want to display these attributes, it is also necessary to extend both the *Device* info (*DeviceInfo*) and the interface (*IDeviceInfo*) classes; see Figure 112.

A common error is to forget the necessary imports, defining the classes for modification.

The remainder of *__init__.py* ensures that all *os* relations are rebuilt for all device instances, when the ZenPack is installed:

```
class ZenPack(ZenPackBase):

   def install(self, dmd):
     ZenPackBase.install(self, dmd)

     # Put your customer installation logic here.
     for d in self.dmd.Devices.getSubDevices():
       d.os.buildRelations()
```

and that the *OperatingSystem* relations have the *logMatchs* removed and all device instance *os* relations are rebuilt, when the ZenPack is removed:

```
def remove(self, dmd, leaveObjects=False):
  if not leaveObjects:
    # When a ZenPack is removed the remove method will be called with
     # leaveObjects set to False .This means that you likely want to
    # make sure that leaveObjects is set to false before executing
    # your custom removal code.
    OperatingSystem._relations = tuple([x for x in OperatingSystem._relations \
          if x[0] not in ['logMatchs']])
    for d in self.dmd.Devices.getSubDevices():
      d.os.buildRelations()
```

9.12.2 LogMatch.py

The only changes in *LogMatch.py* are:

1. Change the relationship to match *__init__.py*:

```
# The logMatchs relationship does not exist in the default Products.ZenModel.OperatingSystem
# It is monkey-patched in the __init__.py of this zenpack
_relations = OSComponent._relations + (
  ('os', ToOne(ToManyCont,
    'Products.ZenModel.OperatingSystem', 'logMatchs')),
)
```

2. The device method needs changing from *return self.logMatchDevice()* to:

```
def device(self):
  os = self.os()
  if os: return os.device()
```

It follows the *os* relationship to the *OperatingSystem* class and then returns **its** *device* method.

Inspecting *$ZENHOME/Products/ZenModel/OperatingSystem.py* shows:

```
def device(self):
  """Return our Device object for DeviceResultInt.
  """
  return self.getPrimaryParent()
```

Ultimately we end up at the object representing the device.

It is perfectly possible (and probably more "Pythonic") to do this in a single statement:

```
return self.os().device()
```

9.12.3 browser/configure.zcml

The stanza that links JavaScript files with device classes will need changing. The original *configure.zcml* has two *viewlet* entries, one to display *LogMatch* components and one to modify the device's Overview panel. The *LogMatch* stanza will certainly need changing to apply to all Devices. The Overview change may or may not be desirable - it is exactly the same code for each:

```
<viewlet
    name="js-LogMatchJavascriptJs"
    paths="/++resource++LogMatchJavascript/js/LogMatch.js"
    weight="10"
    for="Products.ZenModel.Device.Device"
    manager="Products.ZenUI3.browser.interfaces.IJavaScriptSrcManager"
    class="Products.ZenUI3.browser.javascript.JavaScriptSrcBundleViewlet"
    permission="zope2.Public"
    />
```

Note that the "*for*" statement effectively uses a "fully-qualified" path, starting from *Products*, to the *Device* class in the *Device* module.

Note that the "*for*" statement applies to the *Device* object class, including any device object class that inherits from *Device*. In practice this generally means that **all** device instances are eligible to use the *LogMatch* JavaScript code.

9.12.4 LogMatchMap modeler plugin

A modeler plugin fundamentally runs against a **device** instance. The original modeler specified the *logMatchs* relationship in ***relname***; the implication being that *logMatchs* was on the device.

In the new scenario, the *logMatchs* relationship is on the *os* component, not directly on the device, so a **compname** statement is required. This extra line is the only change.

```
relname = "logMatchs"
modname = "ZenPacks.community.LogMatch.LogMatch"
```

```
    # It is a component of the os component of a device that we want to populate
    compname = "os"
```

Section 9.6.1 discussed the inheritance of the modeler plugin *compname* by both *RelationshipMaps* and *ObjectMaps*. When the modeler plugin code creates a *relMap()* and then appends to it any *objectMaps* populated from the modeler data, the *compname* is implicitly used in the creation of these objects:

```
# Create a relationship map - relname above specifies the logMatch
relationship
rm = self.relMap()
# For each entry in the SNMP table, we need to create a LogMatch component
for oid, data in logMatchTable.items():
  # Use try / except to prevent nasty failures
  try:
   # Next line instantiates a LogMatch component object, populating the
   #  component object's attributes with the matching values from the LogMatchTable
   #  logMatchName, logMatchFilename, etc
   #  modname (specified above) defines the object class for the component
   om = self.objectMap(data)
    # Any attribute can then be overwritten, if required. id is an inherited
   #  attribute but we want to ensure uniqueness
   om.id = self.prepId(om.logMatchName)
   # snmpindex is also an inherited attribute. Set it to the index
   om.snmpindex = oid
   # Append this object instance to the relationship map
   rm.append(om)
```

The *LogMatchDeviceMap.py* does not require any changes as it populates attributes directly on the device instance that is being modeled (which used to be of class *LogMatchDevice* and will now be of object class *Device*).

Remember that the modeler plugins will need to be added either to Zenoss device classes or to individual device instances.

9.12.5 Remove / install ZenPack

Given the changes made to this ZenPack, it is prudent to remove the ZenPack and then install it, rather than simply doing a reinstall. Zenoss should be stopped and started entirely after both the remove and the install.

Before removing the ZenPack, as a precautionary measure, move any devices in the */Server/Linux/SimpleTest* device class (the one with the *zPythonClass* set), to */Server/Linux*.

To test the changes, add the *LogMatchMap* modeler plugin to a test device in the */Server/Linux* class and ensure that the snmp agent on that device is configured to deliver logmatch information. Modeling the device should produce *Log Match Files* components.

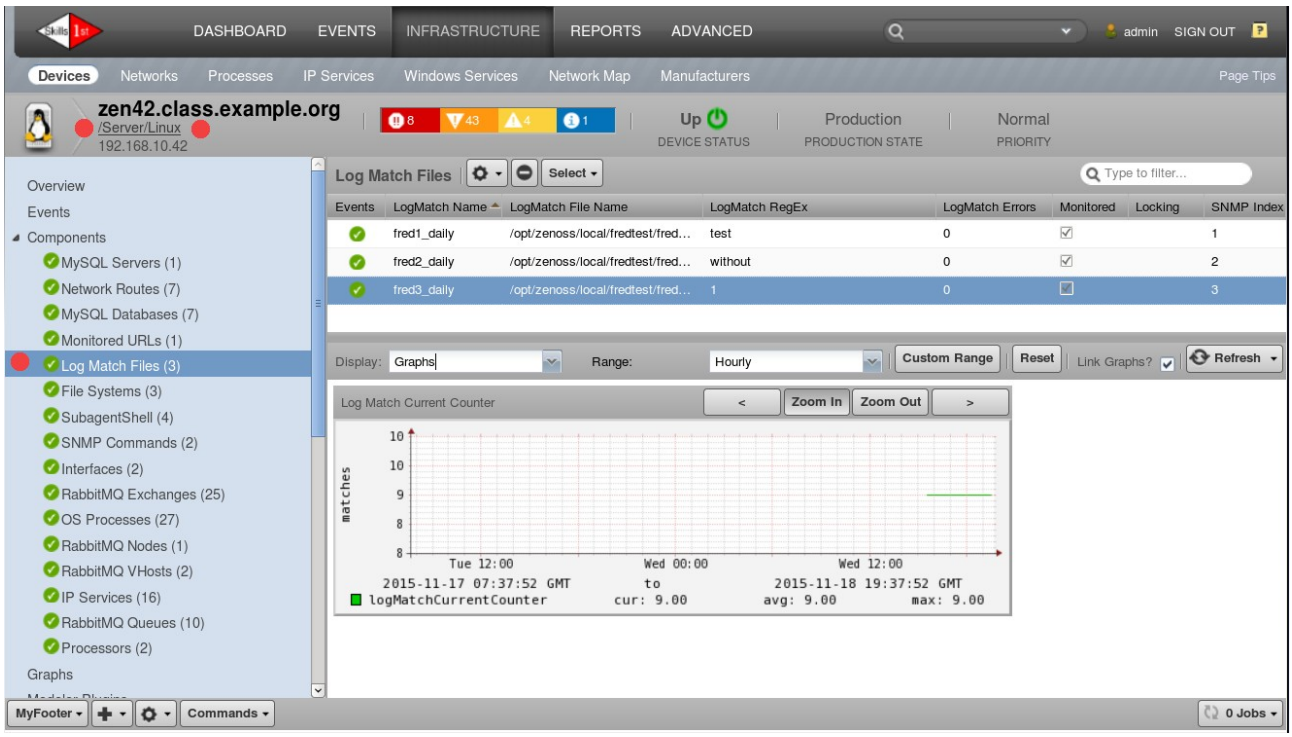


Figure 113: Test device in /Server/Linux class with Log Match Files components

The *LogMatch* component template should still be automatically bound to components of object class *LogMatch*; however, because these components are now components of ***os***, the rrd data files will be under:

```
$ZENHOME/perf/Devices/<device name>/os/logMatchs/<logMatch instance>
```

The ZMI should also show the change in relationships.

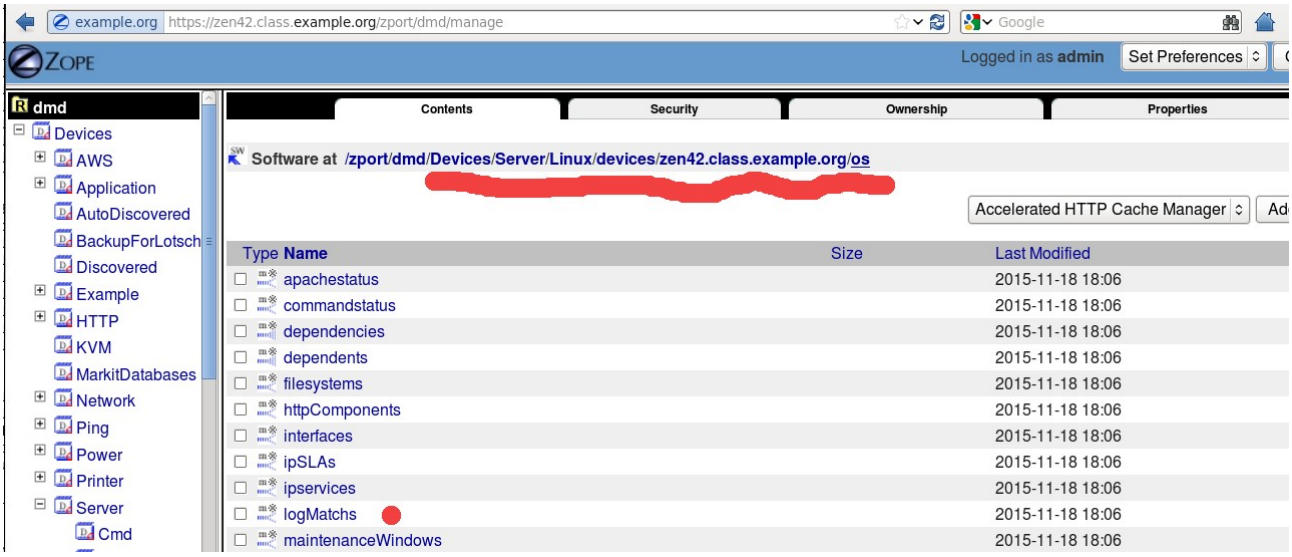


Figure 114: ZMI with logMatchs relationship under the os component of a /Server/Linux device instance

Finalise the ZenPack by updating the *README.rst,* updating the version to 1.0.2 and exporting the ZenPack to create the egg file and *objects.xml*.

10.0 Rewriting the LogMatch ZenPack with zenpacklib

In mid-2015, Zenoss delivered **zenpacklib** which is a package designed to take much of the coding effort out of ZenPacks. The area where it provides most benefit, is in largely eliminating the need for JavaScript, *info.py, interfaces.py* and *configure.zcml*. Its documentation pages can be found at http://zenpacklib.zenoss.com/en/latest .

The difficulty with zenpacklib is that it does not do everything; for example, it cannot simplify writing modeler plugins or custom datasources. In order to write such code it is necessary to **really** understand the constructs that zenpacklib simplifies for you.

10.1 Creating ZenPacks with zenpacklib

There are two ways to get started with zenpacklib. You can either use it to create a new ZenPack from the command line, or you can copy it into an existing ZenPack.

To create a ZenPack from the command line, run the following:

```
./zenpacklib.py create ZenPacks.community.zenpacklibtest
```

This will print several lines to let you know what has been created. Note that the ZenPack's source directory has been created, but it has not yet been installed.

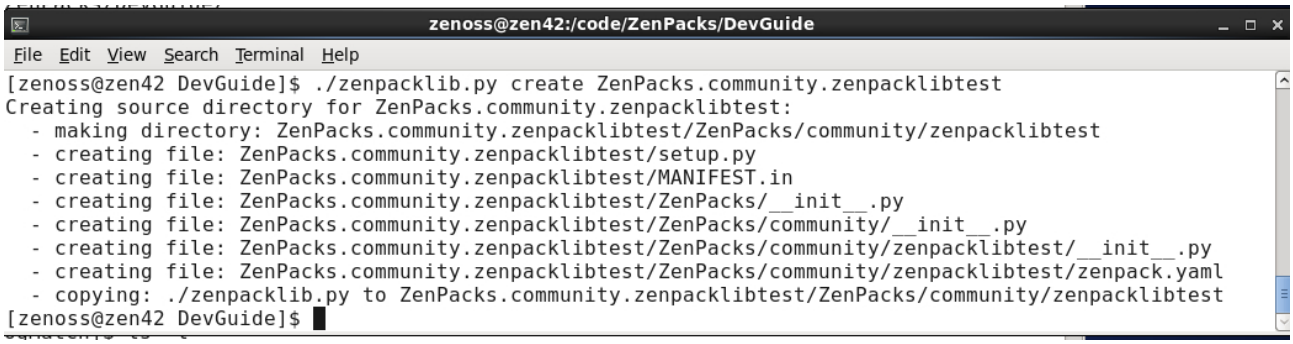


Figure 115: Creating a ZenPack with zenpacklib.py

Note that the start of the usual directory hierarchy is created, each directory having an *__init__.py*. *zenpacklib.py* is copied to the base directory of the ZenPack and a *zenpack.yaml* is created.

No other directories are created under the base directory, such as *modeler/plugin* hierarchies or *datasource* directories. This means that if you create such directories you **must** ensure you create an *__init__.py* in each directory in the subsequent hierarchy. It is adequate to use:

```
touch __init__.py
```

The *zenpack.yaml* file simply contains:

```
name: ZenPacks.community.zenpacklibtest
```

The *__init__.py* in the base directory of the ZenPack contains:

```
from . import zenpacklib
zenpacklib.load_yaml()
```

Alternatively, if a ZenPack already exists, simply copy *zenpacklib.py* to the **base** directory of the ZenPack and create *zenpack.yaml* in the base directory, ensuring there is a name entry for the ZenPack as shown above.

The existing *__init__.py* in the ZenPack's base directory will need the following lines added:

```
from . import zenpacklib
zenpacklib.load_yaml()
```

10.2 zenpacklib capabilities

The online documentation for zenpacklib facilities is at http://zenpacklib.zenoss.com/en/latest/yaml-reference.html :

- Object class definitions for devices and components
- Relationships between devices and components
- Zenoss device classes
- zProperties
- Performance template definitions

There is a presumption that the ZenPack will create **new** Zenoss device classes and **new** object classes for device and components. There is no assistance with modifying core Zenoss constructs, other than adding a global zProperty.

TODO: Is it possible to modify existing classes??

10.3 Converting the logmatch ZenPack for zenpacklib

As with the last major modification to the LogMatch ZenPack, the conversion to zenpacklib will be developed in a separate ***git*** branch, called *zenpacklib* and it will be version 1.0.3. The starting point for this version and git branch will be 1.0.1 where a new device object class is created and the Overview menu is modified to show *versionTag* and *versionDate*.

In addition, a Zenoss device class */Server/Linux/LogMatch* will be created.

10.3.1 zenpacklib benefits - items no longer required

zenpacklib will make the following files and directories redundant so they should be removed (or at least “hidden”):

- LogMatchDevice.py
- LogMatch.py
- configure.zcml
- info.py
- interfaces.py
- browser/

10.3.2 zenpack.yaml

Definitions of new object classes, relationships, device classes and templates can be done in a ***zenpack.yaml*** file that must exist in the base directory of the ZenPack. This permits definitions to be done in a kind of pseudo code that is much simpler to write and much less error prone. The other major benefit is that JavaScript is created automatically for these elements.

The *zenpack.yaml* file is rather focused around Zenoss device classes; the */Server/Linux/LogMatch* device class will be defined in *zenpack.yaml* with the *zPythonClass* zProperty set to *ZenPacks.community.LogMatch.LogMatchDevice*. *zCollectorPlugins* will also be set.

```
name: ZenPacks.community.LogMatch

device_classes:
 /Server/Linux/LogMatch:
  remove: False   # False is default - specified for clarity
  zProperties:
   zPythonClass: ZenPacks.community.LogMatch.LogMatchDevice
   zCollectorPlugins: ['zenoss.snmp.NewDeviceMap', 'zenoss.snmp.DeviceMap',
'HPDeviceMap', 'DellDeviceMap', 'zenoss.snmp.InterfaceMap',
'zenoss.snmp.RouteMap', 'zenoss.snmp.IpServiceMap',
'zenoss.snmp.HRFileSystemMap', 'zenoss.snmp.HRSWRunMap',
'zenoss.snmp.CpuMap', 'HPCPUMap', 'DellCPUMap', 'DellPCIMap',
'zenoss.snmp.SnmpV3EngineIdMap', 'community.snmp.LogMatchDeviceMap',
'community.snmp.LogMatchMap']
```

The *LogMatchDevice* object classes will be defined in *zenpack.yaml* with:

```
classes:
 DEFAULTS:
  base: [zenpacklib.Component]

 LogMatchDevice:
  base: [zenpacklib.Device]
  meta_type: LogMatchDevice # Will default to this but in for completeness
  label: LogMatch Host

  properties:
   versionTag:
    type: string
    label: Version Tag
    short_label: VerTag
   versionDate:
    type: string
    label: Version Date
    short_label: VerDate

  relationships:
   logMatchs:
    label: logMatchs
    display: false
```

The *versionTag* and *versionDate* attributes are defined; the *default* keyword could also be used but a property value defaults to *None* anyway.

Note the *relationships* stanza to explicitly name a relationship called *logMatchs*.

The *LogMatch* component class follows a similar patter, with a few extra items:

```
 LogMatch:
  label: Log Match File # NB It is label, with spaces removed, that is used
to match a component template
  meta_type: LogMatch  # Will default to this but in for completeness
  order: 60
  auto_expand_column: logMatchFilename
  monitoring_templates: [LogMatch]

  properties:
   logMatchName:
    type: string
    label: LogMatch Name
    label_width: 100
    order: 3.1
   logMatchFilename:
    type: string
    label: LogMatch Filename
    label_width: 400
    order: 3.2
   logMatchRegEx:
    type: string
    label: LogMatch RegEx
    label_width: 150
    order: 3.3
   logMatchCycle:
    type: int
    label: LogMatch Cycle
    grid_display: false
    order: 3.4
   logMatchErrorFlag:
    type: int
    label: LogMatch ErrorFlag
    label_width: 100
    grid_display: false
    order: 3.5
   logMatchRegExCompilation:
    type: string
    label: LogMatch RegExCompilation
    label_width: 100
    grid_display: false
    order: 3.6

  relationships:
   logMatchDevice:
    label: logMatchDevice
    display: true   # Show this relationship in Details dropdown
```

Note carefully that the ***label*** field for this component object class (*Log Match File*):

- Will be used in the left-hand menu as the name of the component. For a single component instance the label will be singular - *Log Match File*; if there are multiple instances the label will automatically be plural - *Log Match Files*.

- The ***label*** **not** the object class name is used, by default, to find component performance templates to automatically apply
- It is possible to define a component template to apply using the monitoring_templates keyword (which expects a list):

```
monitoring_templates: [LogMatch]
```

Note that this ZenPack continues to provide performance templates added by the GUI, in *objects.xml* (which works perfectly well for both Zenoss 4 and 5). A later example will discuss adding templates to *zenpack.yaml*.

If several component classes are defined, the *order* keyword can be used to arrange the order. The smaller the number, the nearer to the top of the list will be the component.

The *auto_expand_column* can be used exactly the same way as in a JavaScript file to select a field to take advantage of any unallocated space in the component panel width.

An *order* keyword can also used when defining properties; smaller numbers are nearer the left. There are many keywords available for defining properties which match many of the controls available in JavaScript; common ones are:

- label_width defaults to 80
- column_width defaults to label_width
- default depends on type which can be:

  string, *int*, *float*, *boolean*, *lines*, *password* or *entity*
- details_display defaults True - show in Details dropdown
- grid_display defaults True - show as column in comp display
- renderer default to None (ie string)
- api_only) used to define property or method used in
- api_backendtype) the API. Defaults are false and property
-) method must be defined in a .py file

Note that if column widths are defined that exceed 750 pixels then all the column width directives are ignored and revert to the default.

Relationships between the device and component are denoted by a *class_relationship* stanza:

```
class_relationships:
 - LogMatchDevice(logMatchs) 1:MC LogMatch(logMatchDevice)
```

Note that the relationships stanza is **not** mandatory for either a device or a component object class. Similarly, in the *class_relationships* stanza it is not mandatory to specify the relationship names (those in brackets); however, default relationship names will be created according to the following rules:

- A ToOne relationship will default by lowercasing the first letter of the class. giving *logMatchDevice*.
- A ToMany relationship will default by lowercasing the first letter of the class and adding an “s” to the end to make it plural, giving *logMatchs*.

In this case, the relationships stanzas on both object classes could be omitted and the Class_relationship could simply be:

```
LogMatchDevice 1:MC LogMatch
```

however, it is better practice to explicitly name the relationships to avoid confusion.

Use the ***lint*** parameter to *zenpacklib.py* to check the syntax of *zenpack.yaml*:

```
./zenpacklib.py lint zenpack.yaml
```

10.3.3 zenpack.yaml elements in modeler plugins

The relationship name is needed in the *LogMatchMap* modeler plugin to specify the *relname* to populate. Similarly, the *modname* in the modeler needs to match the concatenation of the ZenPack name and the component class as defined in *zenpack.yaml*.

10.3.4 Completing the ZenPack

Whenever classes or properties have been added or deleted in *zenpack.yaml*, the zenpack should be reinstalled. If properties have been changed, for example *label_width* on an attribute or *monitoring_templates* on a component class, then it is sufficient to recycle *zenhub* and *zopectl*. *zenhub* will report if the total width of defined label fields exceeds 750 pixels.

Note that there are no source files generated for the object classes, *info.py*. *interfaces.py*, *configure.zcml* or JavaScript files; these are all generated by zenpacklib and held in memory.

As with any other development ZenPack, items can be added from the menus, other code files can be added, the *README.rst* should be updated and the version should be increased. The ZenPack's *build/lib* directory should be cleared and the ZenPack should then be exported to recreate the .egg file and the *objects.xml*.

To ensure clean testing, the test device should be completely removed from Zenoss and then recreated as a */Server/Linux/LogMatch* device.

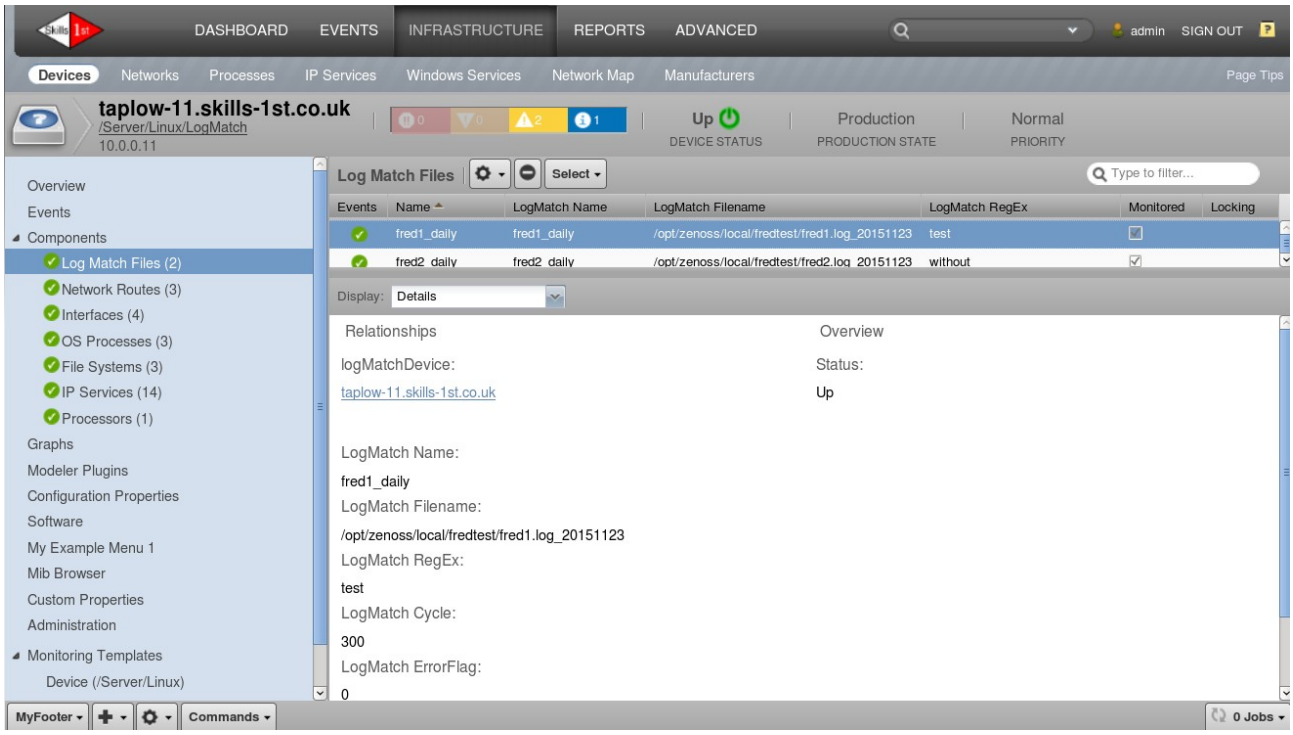


Figure 116: Log Match Files components for /Server/Linux/LogMatch device - created using zenpacklib

10.3.5 JavaScript to modify the device Overview panel

This reproduces most of the functionality from version 1.0.1 of the ZenPack; however, it does not achieve the changes on a device's Overview panel to show the *versionTag* and *versionDate* fields.

There is an expected way to do custom JavaScript using zenpacklib that doesn't involve any ZCML. If you want to have a JavaScript file loaded for all Zenoss pages, put it in *resources/global.js;* if you want it to only apply to your custom device type(s), put it in *resources/<device>.js*. If zenpacklib detects the presence of either of these types of files it will wire up the ZCML for you automatically:

- Create a *resources* directory under the base directory
- Create an *__init__.py* in *resources*
- Copy the old *custom-overview-device.js* to *LogMatchDevice.js* in *resources*
- Reinstall the ZenPack and recycle daemons

There is one major hiccup with converting an existing ZenPack to use zenpacklib. Without zenpacklib, rrd files for components are created under *$ZENHOME/perf/Devices/<host name>* and a subdirectory is created with the name of the component relationship (*logMatchs* in this case). Under this subdirectory is a further directory for each component instance (*fred1_daily* etc) and the rrd files are under there.

The code in zenpacklib.py (1.3 at least) ignores the subdirectory for the relationship and creates directories for component instances directly under *$ZENHOME/perf/Devices/<host name>*. If data already exists, it won't be used after the zenpacklib upgrade.

One way to address this is to modify the *zenpacklib.py* code for this ZenPack in the *rrdPath* method (around line 900). **Do ensure a backup is taken of the file first**.

```
def rrdPath(self):
  """Return filesystem path for RRD files for this component.

  Overrides RRDView to flatten component RRD files into a single
  subdirectory per-component per-device. This allows for the
  possibility of a component changing its contained path within
  the device without losing historical performance data.

  This requires that each component have a unique id within the
  device's namespace.

  """
  original = super(ComponentBase, self).rrdPath()

  try:
    # Zenoss 5 returns a JSONified dict from rrdPath.
    json.loads(original)
  except ValueError:
    # Zenoss 4 and earlier return a string that starts with "Devices/"
    #JC - revert back to rrd directories having a component_relationship subdir
    #return os.path.join('Devices', self.device().id, self.id)
    return original
  else:
    return original
```

10.3.6 Performance data as a component configuration attribute

When viewing components for a device, typically the top part of the window shows configuration details for the component and the lower part shows graphs, where the graphs are driven by data collected by a daemon such as zenperfsnmp, zencommand or zenpython and data is typically collected every 5 minutes. Configuration data is typically collected every 12 or 24 hours.

The standard file systems component is a good example where Free Byes and Used Bytes appear both as configuration data and are also graphed.

How can the configuration display be updated with the up-to-date information provided by the performance data collection daemon?

One solution is achieved using the zenpython daemon and will be discussed in section 13.5. A much simpler and effective solution is possible with zenpacklib. An object class attribute can be defined with the ***datapoint*** keyword.

The LogMatch ZenPack is gathering the *logMatchCurrentCounter* (1.3.6.1.4.1.2021.16.2.1.7) OID as a datapoint in the LogMatch performance template. The *LogMatch* object class does not currently have this attribute; the definition in *zenpack.yaml* can be modified to add it:

```
LogMatch:
  label: Log Match File
  meta_type: LogMatch   # Will default to this but in for completeness
  order: 60   #
  auto_expand_column: logMatchFilename
  monitoring_templates: [LogMatch]

  properties:
    logMatchName:
      type: string
      label: LogMatch Name
      label_width: 100
      order: 3.1
   ..........
   logMatchCurrentCounter:
      type: int
      label: Current Counter
      label_width: 60
      order: 3.7
      datapoint: logMatchCurrentCounter_logMatchCurrentCounter
      datapoint_default: 0
```

The *type* is defined as an integer and the *order* puts it at the far right of the component display. The *label_width* statements for several attributes have been adjusted slightly to fit the space. Note that the *type* keyword will need to be numeric - either *int* or *float*.

The *datapoint* keyword must match an existing datapoint, where the format is:

<datasource name>_<datapoint name>

```
logMatchCurrentCounter_logMatchCurrentCounter              for example
```

A default value for the datapoint can be specified with the *datapoint_default* keyword.

The modeler plugin should have a slight change so that a configuration poll does collect data for the new *logMatchCurrentCounter* attribute:

```
        snmpGetTableMaps = (
            GetTableMap('logMatchTable',
                        '.1.3.6.1.4.1.2021.16.2.1',
                        {
                            '.1': '_logMatchIndex',
                            '.2': 'logMatchName',
                            '.3': 'logMatchFilename',
                            '.4': 'logMatchRegEx',
                            '.11': 'logMatchCycle',
                            '.100': 'logMatchErrorFlag',
                            '.101': 'logMatchRegExCompilation',
                            '.7': 'logMatchCurrentCounter',
                        }
                ),
```

The rest of the modeler code needs no changes as the values from *GetTableMap* populate their corresponding fields in the component *objectMap*.

The ZenPack should be reinstalled and Zenoss recycled.

Figure 117: LogMatch component with Current Counter datapoint displayed as configuration information

Note that you should refresh the browser to see updates to the upper-window configuration data.

This feature has been incorporated into the *zpl_and_datapoint* git branch of the ZenPack and is version 1.0.4.

11.0 COMMAND DirFile sample ZenPack

The LogMatch ZenPack used the SNMP protocol both for gathering configuration data with the modeler plugins and for gathering performance data through templates.

This ZenPack sample provides a straight-forward example of using a CommandPlugin modeler and using COMMAND performance templates. Fundamentally, this means using ***ssh*** to talk to target systems.

The advantage of ssh is that it is very flexible and setting up performance templates is relatively simple. The disadvantage is that it is very inefficient of resources on the Zenoss server and potentially also on the target devices.

A later sample will discuss the merits and details of rewriting the COMMAND elements to use a PythonPlugin modeler and creating Python datasources.

This ZenPack also demonstrates the concept of sub-components where a device contains multiple components, which contain multiple sub-components.

zenpacklib is used to create and build this ZenPack.

11.1 Requirements specification

It is required to collect configuration information for the **existence** of certain Linux filesystem directories. Additionally, for each specified directory, file information is required. The directory will be specified as a fully-qualified pathname; files within the directory will be based on matching a regular expression; for example:

- Directory1 /opt/zenoss/local/fredtest
- Directory1 file regex fred1.*
- Directory2 /var/log
- Directory2 file regex .*log$

The effort (in terms of human development time) is to be minimal at the cost of efficiency of computing resources. The bash command to be run to collect directory information is:

```
find / -type d
```

The command for file information is:

```
find / -type f
```

This ZenPack, with these unmodified commands, should not be run on anything other than a small test system because of the effort required to deliver and process the output of the commands. A more realistic scenario might be to limit the scope of the bash *find* commands to a much more restricted subset of the filesystem; for example */opt/zenoss/local*. The aim is to provide a ZenPack that is very simple to test.

Upto three sets of directory / file regex pairs may be configured.

Configuration information should show directories as **components** of a device if they exist. In addition, if files matching the file regex exist, they should be **sub-components** of the directory.

Performance information should be gathered for both directory and file components using the *du* bash command to show the disk used by the element, in bytes.

11.2 ZenPack specification

The new ZenPack will be called **ZenPacks.community.DirFile**.

The ZenPack will create a new component type called **Dir** with a single attribute:

- dirName of type string

There will also be a **File** component with three attributes:

- fileName of type string
- fileDirName of type string
- fileRegex of type string

For now, the ZenPack will also create a new object class for these devices - **DirFileDevice**. The device class has no extra attributes.

A *DirFileDevice* will have a *ToManyCont* relationship with *Dir* components called **dirs**. The corresponding *ToOne* relationship from the *Dir* component will be **dirFileDevice** (note the capitalisation carefully). The *Dir* component class will also have a **files** *ToManyCont* relationship with *File* components whose corresponding *ToOne* relationship with the *Dir*, will be **dir**, thus creating a 3-tier hierarchy.

The ZenPack will create the new Zenoss device class **/Server/Linux/DirFile**.

Configuration for the three pairs of directories and optional file regex parameters, will be achieved with new zProperties:

- zMonitorDir1 zMonitorDir1File
- zMonitorDir2 zMonitorDir2File
- zMonitorDir3 zMonitorDir3File

Since much of the data is to do with directory hierarchies, many “names” will have one or more unix-style slashes in them. Such characters could be interpreted as meta-characters so a general policy will be, when creating fundamental ***id*** attributes, the Zenoss utility, *prepId*, will be used to ensure that any “unsafe” characters are replaced with an underscore. “Safe” characters are defined in *prepId* as:

```
a-z  A-Z  0-9  - _ , . $ ( )
```

A modeler plugin will be required to discover *Dir* and *File* components - **DirFileMap**.

zenpacklib will generate all the JavaScript that is required and produce some useful links between the component hierarchy. No *info.py*, *interfaces.py* or *configure.zcml* is required.

Templates will be created through the GUI and added to the *objects.xml* of the ZenPack.

Unless otherwise noted, all ZenPack instructions from here on in this chapter, will be for Zenoss 4 Core (or earlier).

11.3 Creating the ZenPack

The ZenPack will be built using zenpacklib so that will also be used for creation:

```
./zenpacklib.py create ZenPacks.community.DirFile
```

The ZenPack directory hierarchy is created down to the base directory which contains:

- *zenpacklib.py*
- *zenpack.yaml* containing the ZenPack name.
- *__init__.py* containing the import of zenpacklib and the *load_yaml()* statement

No further directories or files are created.

Create a *README.rst* in the top-level directory.

Note than the zenpacklib create command does **not** install the ZenPack. Do so with:

```
zenpack --link --install ZenPacks.community.DirFile
```

Examine the ZenPack through the Zenoss GUI and fill in the *Author* and *License* fields and any co-requisites. Note that a ZenPack created with zenpacklib will have a version *1.0.0dev* by default.

11.4 zenpack.yaml

zenpack.yaml allows the definition of new zProperties. Prior to zenpacklib, new properties could be defined fairly simply in the *__init__.py* in the base directory of a ZenPack. For example, the original VMwareESXiMonitor ZenPack created *zVSphereUsername* and *zVSpherePassword* zProperties (see Figure 118).

Note that any zProperties created by any ZenPack by any method, defines **global** zProperties. They cannot be limited to an object class or to a Zenoss device class. For this reason, the facility should be used sparingly.

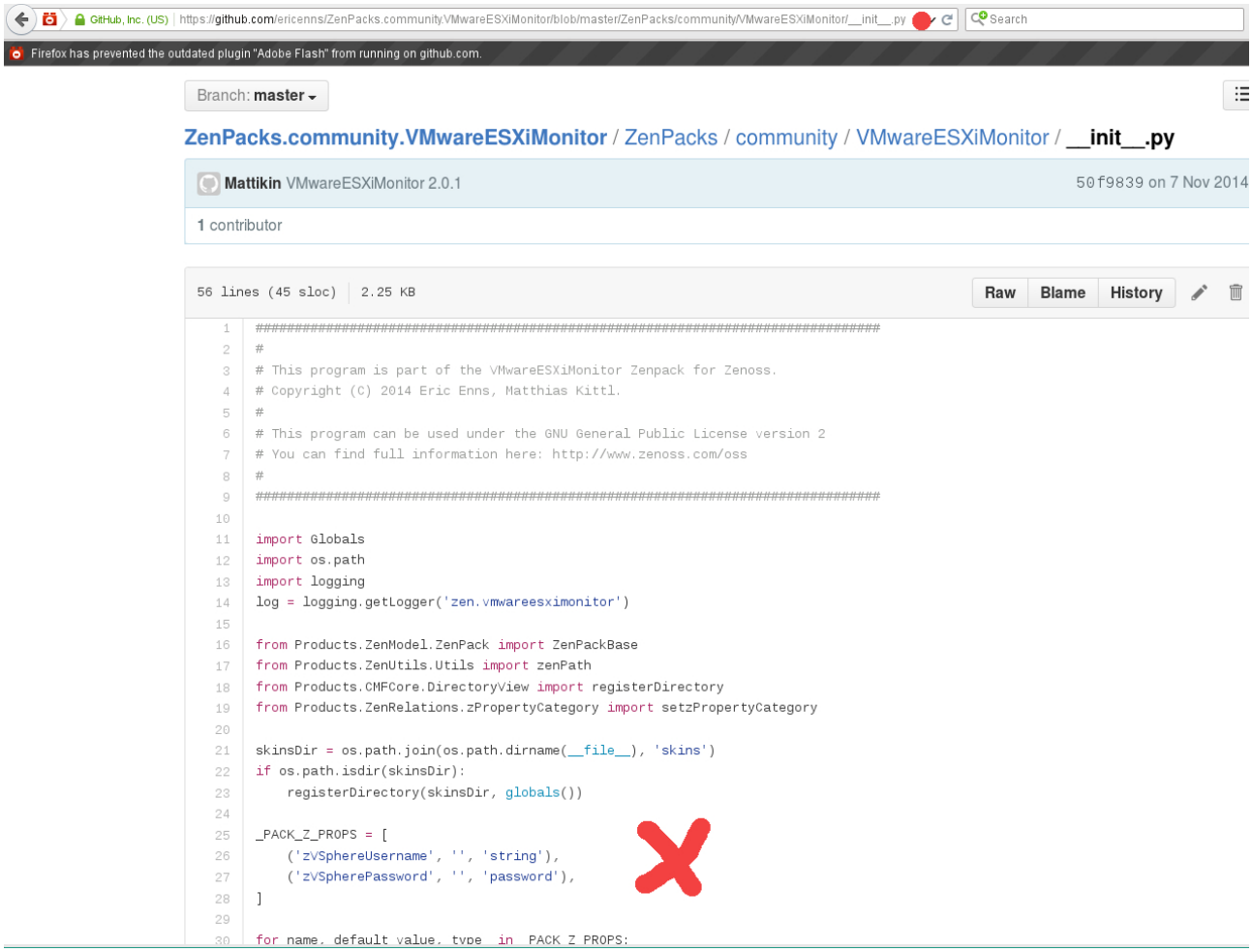


Figure 118: __init__.py for original ZenPacks.community.VMwareESXiMonitor showing creation of new zProperties

The */Server/Linux/DirFile* device class will be defined in *zenpack.yaml* with the *zPythonClass* zProperty set to *ZenPacks.community.DirFile.DirFileDevice*. *zCollectorPlugins* will also be set.

See http://zenpacklib.zenoss.com/en/latest/yaml-zProperties.html for full documentation on creating zProperties with zenpacklib, including supported types and defaults.

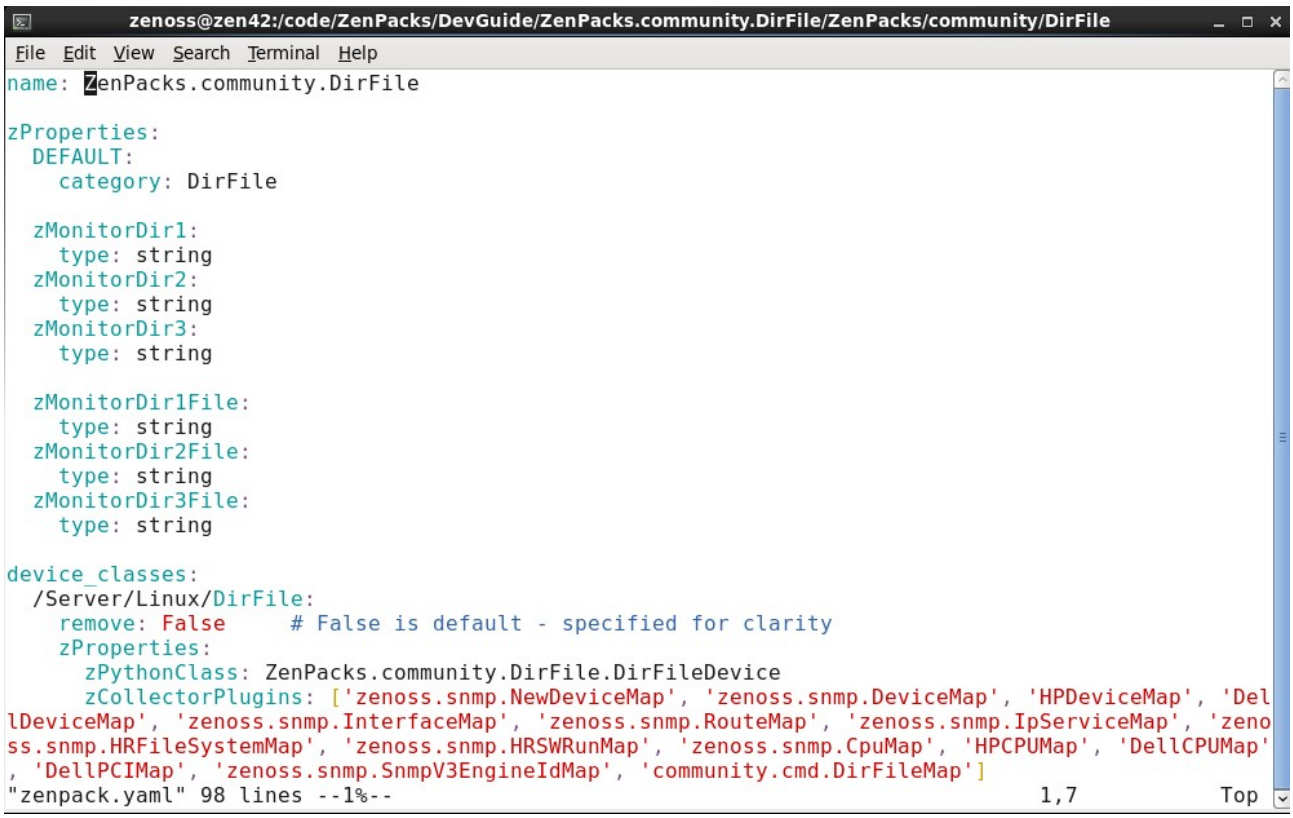


Figure 119: zenpack.yaml for DirFile ZenPack - zProperties and device class

The *DirFileDevice* object class and the *Dir* and *File* Component classes are defined next in *zenpack.yaml*.

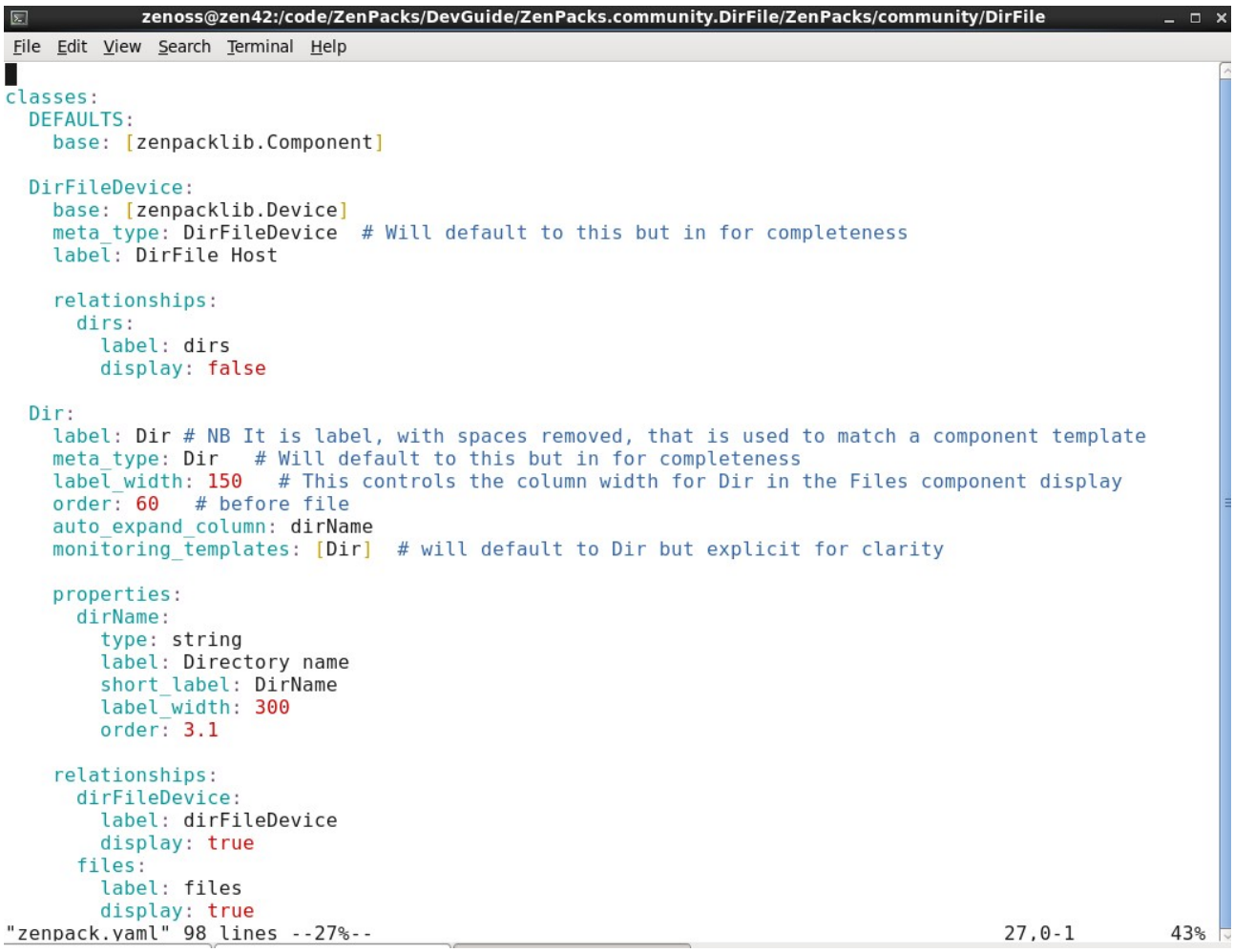


Figure 120: zenpack.yaml for DirFile ZenPack - object class for device and Dir component

Points to note in Figure 120 are:

- For the *Dir* component:
  - The **label** is used to automatically match a component template. The **monitoring_templates** keyword is redundant here but is explicit to improve understanding. The label is also used as the component name in the left-hand menu (with “s” added for more than one component instance).
  - When the directory is used as part of the *Files* component display, it is the label and column_width keywords in the ***Dir*** definition that controls the display
  - The order of components in the left-hand menu is controlled by the class definition **order** keywords, where lower numbers are nearer the top. Thus *Dirs (order: 60)* will be above *Files (order: 70)*.
  - Both relationships are defined explicitly although they would default to these same names (object class lower-cased; *ToManyCont* relationship is plural; *ToOne* relationship is singular).
  - Both relationships should be displayed in the component grid and in the *Details* dropdown.


There are two contradictions to the above notes.

- The left-hand menu component label is supposed to become singular if there is only one instance. This does not appear to happen.
- Although both relationships for *Dir* have *display: true* , only the *dirFileDevice* relationship is actually shown in the *Details* dropdown.

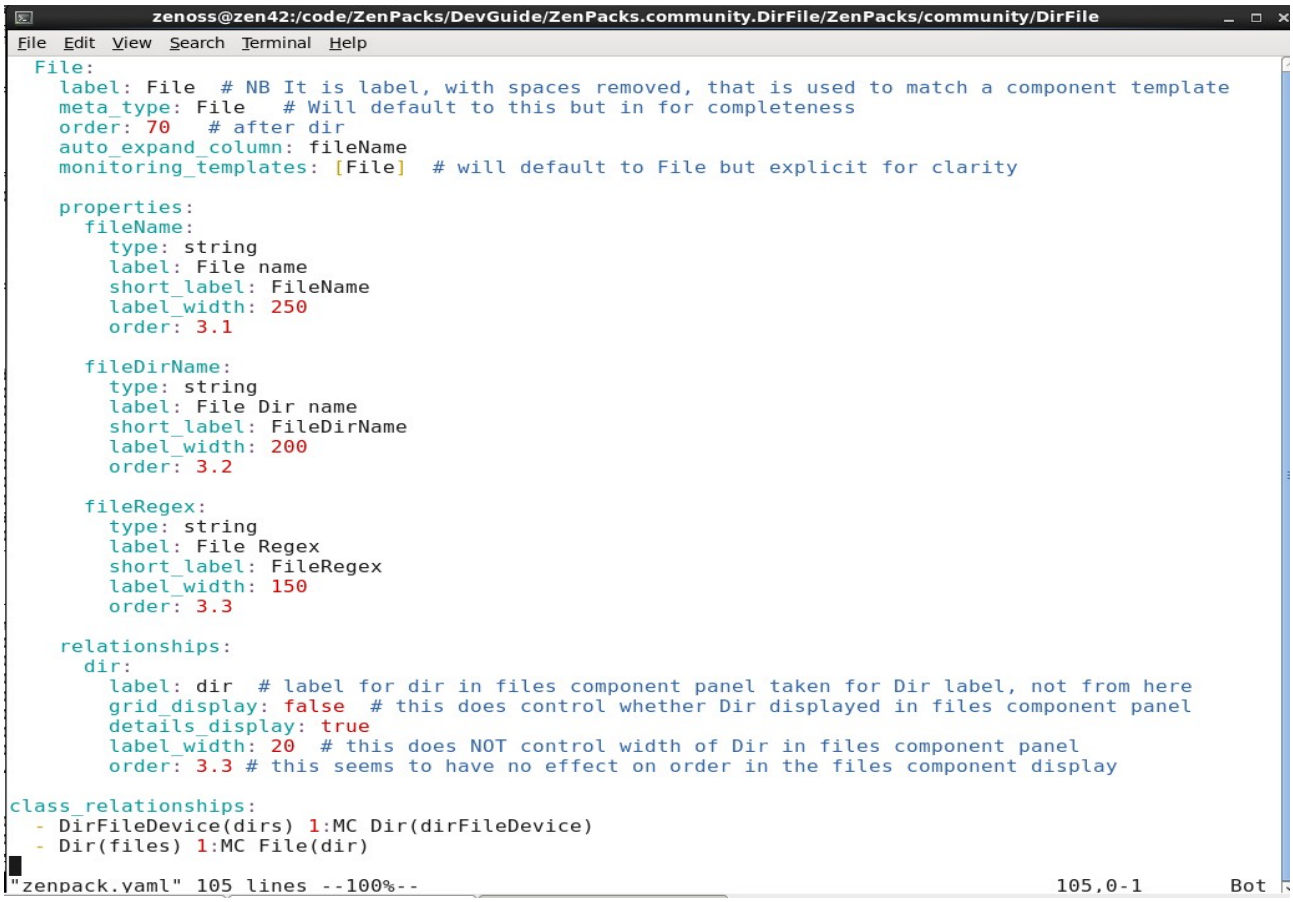


Figure 121: zenpack.yaml for DirFile ZenPack - object class for File and class relationships

Points to note in Figure 121 for the *File* component class definition are:

- The class *order: 70* places it below *Dirs* in the left-hand menu
- The order of columns is controlled by the property *order* statements with 3.1 left of 3.2 etc.
- The relationship again is defined explicitly, however:
  - Either *grid_display* or *display* set to *true* controls whether the relationship is shown as a column in the *Files* component panel
  - The *label_width* and *label* for this column are taken from the *Dir* class keywords, not from this relationship statement
  - The *order* keyword appears to have no effect

When the yaml file is complete, the ZenPack should be reinstalled and zenoss completely recycled.

```
zenpack --link --install ZenPacks.community.DirFile
zenoss restart                                    for Zenoss 4
```

```
serviced service restart Zenoss.core            for Zenoss 5 Core
serviced service restart Zenoss.resmgr          for Zenoss 5 Enterprise
```

Check that the new Zenoss device class is created and that the *zPythonClass* and *zCollectorPlugins* are correctly set.

11.5 DirFileMap modeler plugin

Modeler plugins must go under a *modeler*/*plugin* directory hierarchy under the base ZenPack directory. To follow the previous sample, there should then be a *community*/*cmd* continuation of the hierarchy, although this is not essential.

Creating a ZenPack with zenpacklib does not create any of this hierarchy so it must be done manually and each directory should have an empty *__init__.py*.

The *DirFileMap* plugin will inherit from the ***CommandPlugin*** object class (whereas the *LogMatch* example inherited from the *SnmpPlugin*).

11.5.1 * CommandPlugin code in core Zenoss

The ***CommandPlugin*** class is defined in *$ZENHOME*/*Products*/*DataCollector*/*plugins*/*CollectorPlugin.py*. It inherits from the *CollectorPlugin* class that is discussed in more detail in Section 9.6.1.

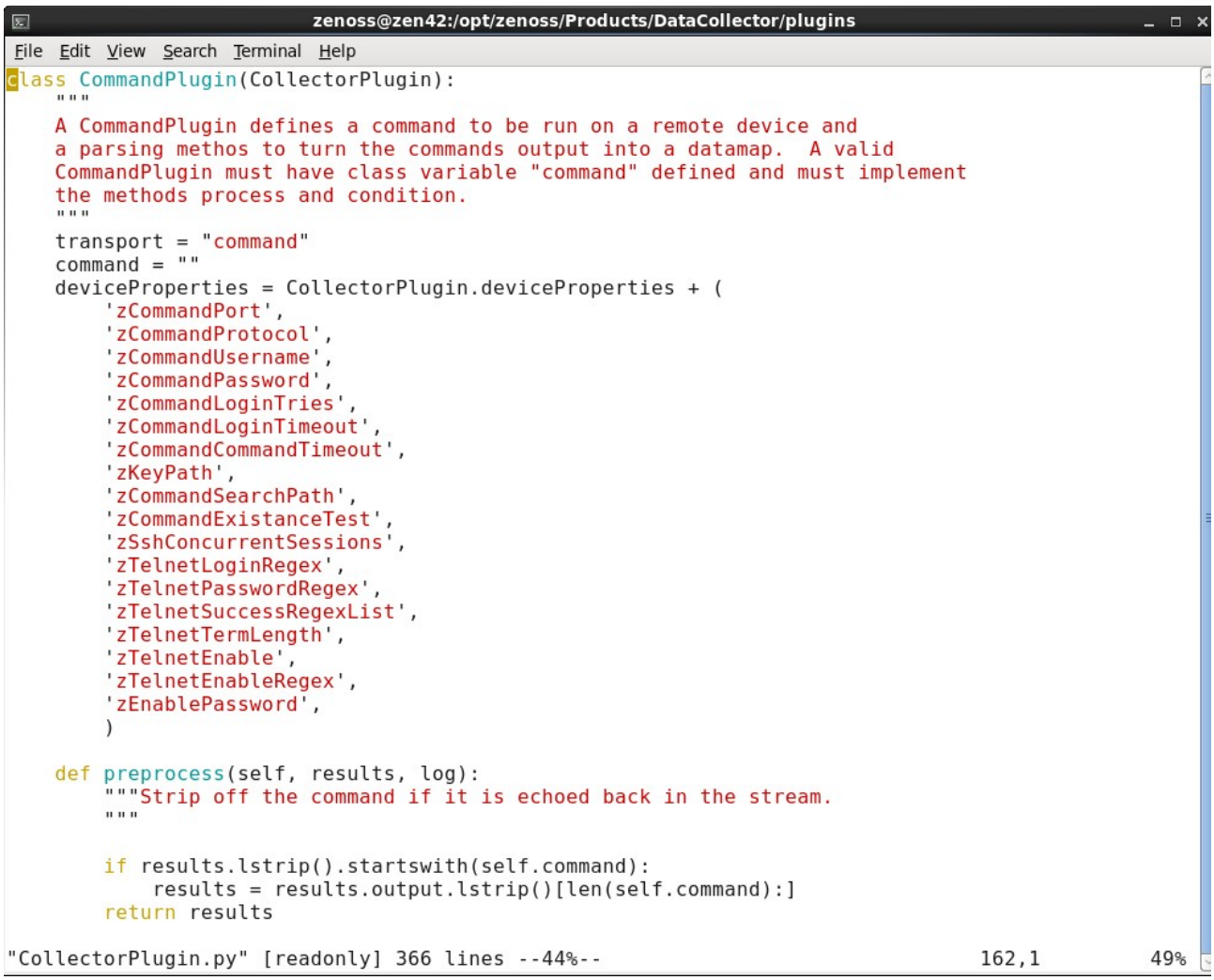


Figure 122: CommandPlugin class for ssh-based modeler plugins

As Figure 122 shows, all the Command and Telnet zProperties for a device are made available to a *CommandPlugin*. It is the *transport* variable set to *command* that instructs the *zenmodeler.py* daemon code to use *ssh* (by default, the other option for *zCommandProtocol* being *telnet*).

If the results returned are prefaced by the command being echoed back, then the *preprocess* method strips off that command.

The *command* variable in *CommandPlugin* is the null string; this must be overridden to be the correct command in the ZenPack modeler.

zenmodeler.py works out what modelers should be applied for each device and then runs the *collectDevice* method. (*zenmodeler.py* is in *$ZENHOME/Products/DataCollector*.

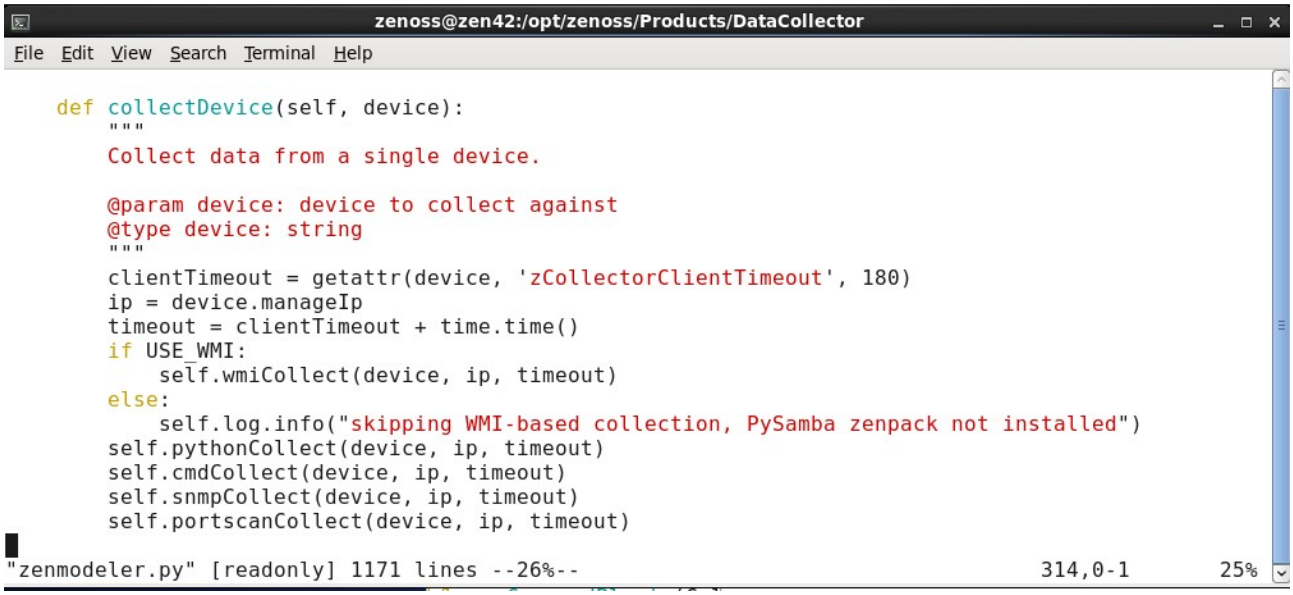


Figure 123: collectDevice method in $ZENHOME/Products/DataCollector/zenmodeler.py

Indeed, *collectDevice* calls all five collect methods, including :

```
self.cmdCollect(device, ip, timeout)
```

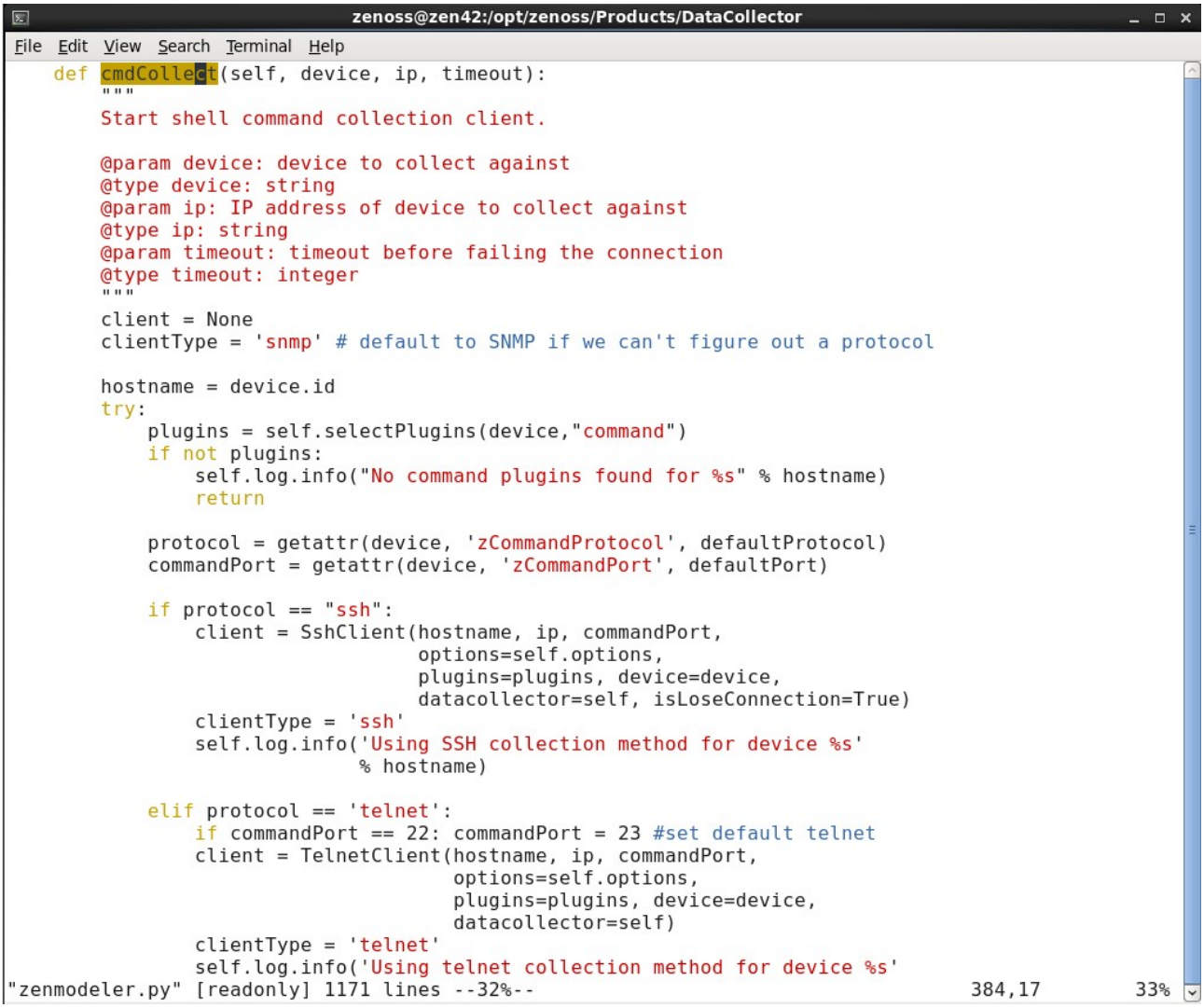


Figure 124: cmdCollect method in zenmodeler.py

cmdCollect checks for plugins of type *command*, gathers the *zCommandProtocol* and *zCommandPort* for the device (defaults are *ssh* and port *22*), and then calls *SshClient* to actually collect ssh data.

The *SshClient* method is in *$ZENHOME/Products/DataCollector/SshClient.py*. There are a complex set of methods to build a transport session to the target device, taking account of all the ssh authentication zProperties and handling any failures. The *addCommand* method documents that a new channel is opened for each command to be run, which could be very expensive if lots of commands are required.

SshClient delivers results using Python ***twisted*** libraries. twisted is a generic way of delivering results asynchronously; in other words, several requests may be sent out without having to wait for the result from the first request to be returned. twisted tracks requests and responses and ensures that the correct returning data (or failure) is associated with the appropriate request, by means of **callbacks**. Performance can be dramatically improved by this way of working as communication is less likely to be blocked.

twisted is not specific to ssh communications; twisted libraries are used by most modeler plugins and by many of the performance data collector daemons.

In summary, the *CommandPlugin* has access to all the *zCommand* zProperties for a device, and manages the session setup, data retrieval, session close and any error handling for ssh communications.

11.5.2 Using zProperties in the modeler plugin

Any plugin can extend the *deviceProperties* that are available to the modeler:

```
class DirFileMap(CommandPlugin):
  deviceProperties = CommandPlugin.deviceProperties + (
      'zMonitorDir1',
      'zMonitorDir2',
      'zMonitorDir3',
      'zMonitorDir1File',
      'zMonitorDir2File',
      'zMonitorDir3File',
    )
```

This ensures that the new zProperties defined in *zenpack.yaml* are available to the modeler. Note that this extends the zProperties from the *CommandPlugin* class, which inherits from the *CollectorPlugin* (see *$ZENHOME/Products/DataCollector/plugins/CollectorPlugin.py*):

- **CommandPlugin zProperties**
  - 'zCommandPort',
  - 'zCommandProtocol',
  - 'zCommandUsername',
  - 'zCommandPassword',
  - 'zCommandLoginTries',
  - 'zCommandLoginTimeout',
  - 'zCommandCommandTimeout',
  - 'zKeyPath',

- 'zCommandSearchPath',
- 'zCommandExistanceTest',
- 'zSshConcurrentSessions',
- 'zTelnetLoginRegex',
- 'zTelnetPasswordRegex',
- 'zTelnetSuccessRegexList',
- 'zTelnetTermLength',
- 'zTelnetEnable',
- 'zTelnetEnableRegex',
- 'zEnablePassword',

- **CollectorPlugin properties and zProperties**
  - 'id',
  - 'manageIp',
  - '_snmpLastCollection',
  - '_snmpStatus',
  - 'zCollectorClientTimeout',

Within the ZenPack plugin code, any of these can simply be referred to as attributes of ***device***; for example, *device.zMonitorDir1File*.

11.5.3 CommandPlugin command

The essential variable that a *CommandPlugin* must provide is the **command**. This can be anything that will run in a bash shell,. If the command for the plugin is not built-in shellscript code then the command must exist on every target and the correct path to the script must be known.

The command for this ZenPack plugin needs to get all directories, starting from a particular point in the filesystem, and then get all files. The output will be delivered into a **results** variable, to be processed by the **process** method. To separate the directories output from the files output, a line that contains *__SPLIT__* will be used:

```
# The command to run.
# Get directories (one per line) then a line with __SPLIT__ then files (one per line)
# Beware this has potential to return LOTS of data
command = (
    #'find /opt/zenoss/local -type d ;'
    'find / -type d ;'
    'echo __SPLIT__; '
    #'find /opt/zenoss/local -type f'
    'find / -type f'
    )
```

Note that semicolons are used to effect a newline in the “shellscript”. Each line needs to be single-quoted when using this construct.

The command can be a simple one-line shellscript as is created in the Example command modeler, *ExampleCMD.py.example*, found when a ZenPack is created from the GUI:

```
command = "/bin/cat /proc/partitions"
```

Note that zProperties **cannot** normally be used as part of the command definition.

A good sample Command-based ZenPack that is publicly available, written by Zenoss, is **ZenPacks.zenoss.RabbitMQ**, available from GitHub at https://github.com/zenoss/ZenPacks.zenoss.RabbitMQ

Remember that the command will run, taking account of all the zCommand zProperties of the device so if ssh to target devices uses usernames and passwords, then the correct values need to be configured for *zCommandUsername* and *zCommandPassword*. If public keys are used for ssh, the *zCommandUsername,* potentially the *zCommandPassword,* and the *zKeyPath* must be correct and the public key for the *zenoss* user on the zenoss server needs to have been copied to the *.ssh/authorized_keys* file for the correct user on the target systems. Note that *zCommandPassword* is used to hold the **passphrase** for the key if one has been set; otherwise *zCommandPassword* is not used with public key ssh.

Other zProperties to note that affect running commands over ssh are:

- zCommandLoginTries default 1
- zCommandLoginTimeout default 10s
- zCommandCommandTimeout default 15s
- zCommandSearchPath default is unset
- zSshConcurrentSessions 10

Note particularly *zCommandCommandTimeout* if you have a long-running command, though extending the time limit potentially just slows the whole modeling process, especially if some targets are not responding at all.

The *zCommandSearchPath* is a good way of defining a standard for where local scripts should be held, with the possibility of device-level override if necessary. If the command provided is not a fully-qualified pathname then the script will be sought for in *zCommandSearchPath*.

11.5.4 The process method of the modeler plugin

A *CommandPlugin* in a ZenPack must include a ***process*** method to decode and apply the command output. The method is passed the **device** object and the **results** from the command. Typically the first line of the method provides some logging and may often include debug logging to show the raw results:

```
def process(self, device, results, log):
  log.info("Modeler %s processing data for device %s",
    self.name(), device.id)
  #log.debug('results is %s ' % (results))
```

"*self*" in this case is the modeler plugin so the *log.info* line would provide output in *$ZENHOME/log/zenmodeler.log* like:

```
2015-11-27 09:54:53,670 INFO zen.ZenModeler: Modeler
community.cmd.DirFileMap processing data for device taplow-11.skills-
1st.co.uk
```

The first real task of *process* is to construct a dictionary of the zProperties for directories and files:

```
# Create dictionary where key is directory and value is file regex
dirRegex = {}
if device.zMonitorDir1:
  if device.zMonitorDir1File:
    dirRegex[device.zMonitorDir1.rstrip('/')] = device.zMonitorDir1File
  else:
    dirRegex[device.zMonitorDir1.rstrip('/')] = None
if device.zMonitorDir2:
  if device.zMonitorDir2File:
    dirRegex[device.zMonitorDir2.rstrip('/')] = device.zMonitorDir2File
  else:
    dirRegex[device.zMonitorDir2.rstrip('/')] = None
if device.zMonitorDir3:
  if device.zMonitorDir3File:
    dirRegex[device.zMonitorDir3.rstrip('/')] = device.zMonitorDir3File
  else:
    dirRegex[device.zMonitorDir3.rstrip('/')] = None
 log.info(' dirRegex is %s ' % (dirRegex))
```

This ensures that a null *device.zMonitorDir* property is not included in the lookup dictionary. It is permissible for a non-null dictionary to have a null file regex. If the directory zProperty has been entered by a user with a trailing "/" then this is stripped off. The directory is used as the directory key; the file regex is the value.

The main body of the *process* method builds *ObjectMaps* for components and sub-components and delivers the *RelationshipMaps* that link them together.

Where a modeler - any modeler - has to populate a "component that contains a sub-component" set of relationships, typically there is a *for* loop to process the component and then an internal loop, often handled as a separate function, to process sub-components of the component. The trick is to pass the component relationship and instance as parameters to the inner loop. An algorithmic outline would be:

```
initialise RelationshipMap for component maps
for component in list_of_components
  get relevant data for component
  modify any raw data, as required
  create an ObjectMap for the component
  add ObjectMap to component RelationshipMap
  for sub-component in list of sub-components
    initialise a list for sub-component maps
    get relevant data for sub-component
    modify any raw data, as required
    create an ObjectMap for the sub-component
    add ObjectMap to sub-component map list
    return a RelationshipMap with correct compname, relname, modname
         and the sub-component map list
return the RelationshipMap for the device with correct component relname,
    modname and the component map list
```

A process method must deliver one of:

- *None* - changes nothing. Good in error cases.
- A *RelationshipMap* for the device - component information

- An *ObjectMap* for the device - device information
- A list of *RelationshipMaps* and *ObjectMaps* - both

See the discussion in section 8.7.2.6 about delivering consistent maps where components and sub-components are created.

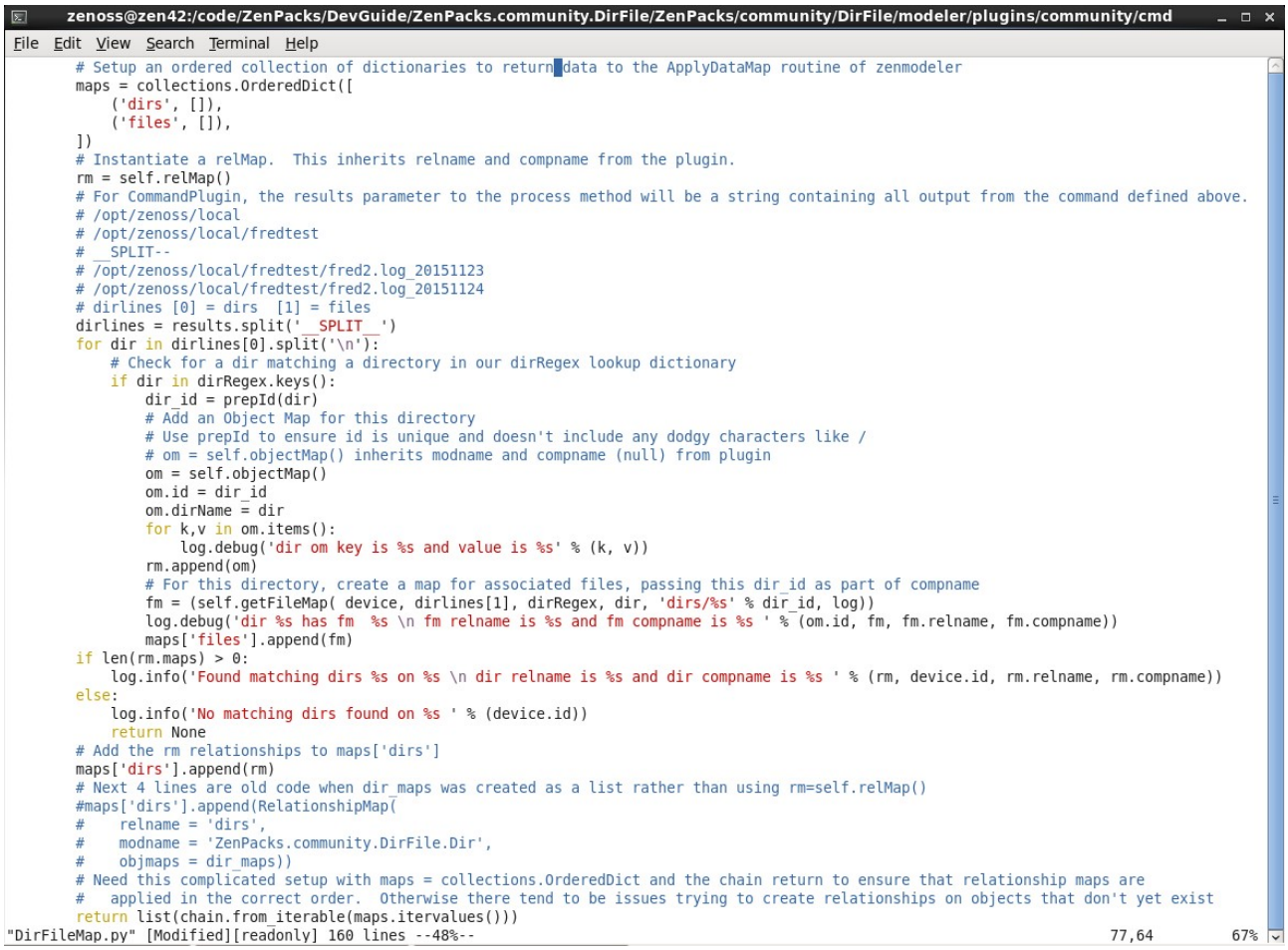


Figure 125: Main loop of DirFileMap modeler plugin

Note in Figure 125 that:

- The relationship map for *dirs* is instantiated with a call to *relMap* which ensures that the *relname* is inherited from the modeler plugin.

```
# Instantiate a relMap. This inherits relname and compname from the plugin.
rm = self.relMap()
```

- The code used to construct an object map for the directory uses the *objectMap* method (note lower-case “o”). This is a method on the *CollectorPlugin* class that automatically sets the *ObjectMap compname*, *modname* and *classname* to those specified (or defaulted) for the modeler plugin. Specifically, ***modname*** is set to *ZenPacks.community.DirFile.Dir*.

```
dir_id = prepId(dir)
om = self.objectMap()
om.id = dir_id
om.dirName = dir
rm.append(om)
```

- The *id* attribute of the *Dir* component has had *prepId* applied to ensure there are no unsafe characters in this field
- The *Files* sub-component(s) for this *Dir* are populated by calling the *getFileMap* function and passes to that function the compname as:

```
'dirs/%s' % dir_id
```

where *dirs* is the relationship on the device and the %s has substituted this particular Dir id making an example component parameter of *'dirs/opt_zenoss_local_fredtest'*

The *getFileMap* function checks the file regex against all the files in the chosen directory and returns a *RelationshipMap* with a list of *ObjectMaps*, one for each matching file.

zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.DirFile/ZenPacks/community/DirFile/modeler/plugins/community/cmd

File Edit View Search Terminal Help

```
    def getFileMap(self, device, files_string, dirRegex, dirMatch, compname, log):
        #log.debug('files_string is %s , dirRegex is %s , compname is %s ' % (files_string, dirRegex, compname))
        file_maps = []
        for file in files_string.split('\n'):
            # Split out the filename part and the directory part
            f = file.split('/')[-1]
            d = '/'.join(file.split('/')[:-1])
            # Only consider creating a file map if the directory matches the dirMatch parameter
            if d == dirMatch:
                for k, v in dirRegex.items():
                    if d == k:                  # got directory match
                        if re.search( v, f):   # check the regex
                            # Got a regex match against filename f
                            file_id = prepId(f)
                            # Don't want to inherit compname or modname from plugin as we want to set this expicitly
                            # Use ObjectMap rather than om=self.objectMap()
                            file_maps.append(ObjectMap(data = {
                                'id': file_id,
                                'fileName' : f,
                                'fileDirName' : d,
                                'fileRegex' : v,
                                }))
                            log.info('Found dir %s and file %s match' % (d, f))
                            # Get out of for k, v in dirRegex.items(): loop - don't care if matches on >1 regex
                            break

        # Return file_maps relationship map with compname passed as parameter to this method
        # Again - don't want to inherit relname, modname or compname for this relationship as we want to set them explicitly
        # Use RelationshipMap rather then rm=self.relMap()(
        return RelationshipMap(
            compname = compname,
            relname = 'files',
            modname = 'ZenPacks.community.DirFile.File',
            objmaps = file_maps)
```

"DirFileMap.py" [readonly] 167 lines --79%-- 132,33 99%

Figure 126: DirFileMap modeler plugin - getFileMap function

Note in Figure 126 that the *RelationshipMap* uses the *compname* passed from the calling code as a parameter and specifies the relname **on the component** (*files*). The *modname* is the module containing the object class definition for the *Files* component (*ZenPacks.community.DirFile.File*).

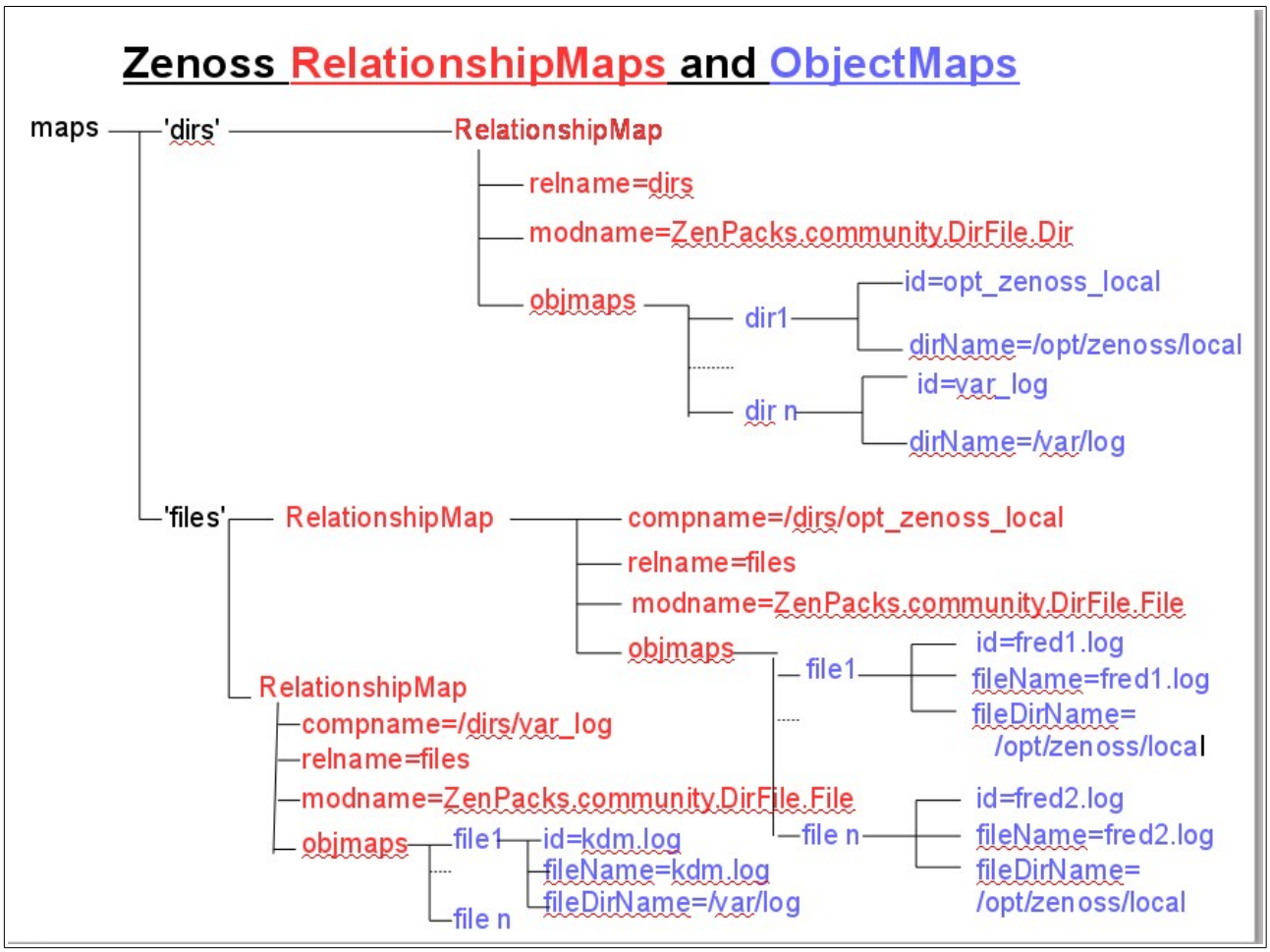


Figure 127: RelationshipMaps and ObjectMaps

Figure 127 demonstrates the resultant *maps* directory with its *dirs* and *files* RelationshipMaps.

11.5.5 * What's in an object map?

Different ZenPacks use varying techniques for constructing object maps and relation maps. The raw data is in an *ObjectMap* (upper-case "O"). The *CollectorPlugin* class in *$ZENHOME/Products/DataCollector/plugins* defines the ***objectMap*** (lower-case "o") method:

```
def objectMap(self, data={}):
  """Create an object map from the data
  """
  om = ObjectMap(data)
  om.compname = self.compname
  om.modname = self.modname
  om.classname = self.classname
  return om
```

It is good practice to specify *relname* and *modname* at the top of the ZenPack modeler plugin class if a **single** relationship is being modeled..

```
# No compname specified here as Dir is a component directly on the device
#  (defaults to null string)
# classname not required as largely deprecated. classname is the same
```

```
#  as the module name here
relname = 'dirs'
modname = 'ZenPacks.community.DirFile.Dir'
```

The *compname* is null for *Dir* objects as they are components directly on the device class.

The *classname* is effectively deprecated now that the modeler plugin classname is mandated to match its module name.

Note that **objectMap** instantiates an **ObjectMap** (note upper-case "O"). ***ObjectMap*** and ***RelationshipMap*** can be found in *DataMaps.py* in the same directory; both are, effectively, **protobufs** (raw data).

The ZenPack modeler code then instantiates an object with:

```
om = self.objectMap()
```

There is also a **relMap** method that instantiates a **RelationshipMap**.

```
  def relMap(self):
    """Create a relationship map.
    """
    relmap = RelationshipMap()
    relmap.relname = self.relname
    relmap.compname = self.compname
    return relmap
```

Again, the *relMap* instantiates a *RelationshipMap* with *relname* and *compname* being populated from the modeler plugin values. A *RelationshipMap* is just a container to store *ObjectMaps*.

```
class RelationshipMap(PBSafe):
  relname = ""
  compname = ""

  def __init__(self, relname="", compname="", modname="", objmaps=[]):
    self.relname = relname
    self.compname = compname
    self.maps = [ObjectMap(dm, modname=modname) for dm in objmaps ]
```

The ZenPack modeler code instantiates a relationship with:

```
rm = self.relMap()
 ..... create an object map .....
rm.append(om)
```

Note that to check whether *rm* contains any object maps, check the length of the *maps* list:

```
if len(rm.maps) > 0:
```

When populating sub-components of components, the modeler plugin values for *modname*, *compname* and *relname* are not useful. The *getFileMap* function sets these directly, where *compname* is passed as a parameter showing the relationship and instance of the directory:

```
return RelationshipMap(
      compname = compname,
      relname = 'files',
      modname = 'ZenPacks.community.DirFile.File',
      objmaps = file_maps)
```

The object map is created directly with an *ObjectMap* class:

```
file_maps = []
.......
file_maps.append(ObjectMap(data = {
  'id': file_id,
  'fileName' : f,
  'fileDirName' : d,
  'fileRegex' : v,
  }))
```

11.5.6 zenpacklib and the modeler plugin

It is clear from the preceding subsections that it is essential that the ZenPack writer knows the precise names of objects, relationships and modules. Although zenpacklib avoids the need to write an object class file for *Files*, a module **will** be constructed in memory, for that class whose name will be:

```
<ZenPack name>.<object class name>        eg.
ZenPacks.community.DirFile.File
```

zenpacklib defaults relationship names to be a classname with the first letter lower-cased and an “s” added for *ToMany* relationships. Whilst labour-saving, this is less than ideal from a clarity perspective. It is better practice to explicitly name the relationships as has been demonstrated in the two sample *zenpack.yaml* files.

11.5.7 Testing the DirFileMap modeler

It is strongly recommended that testing take place on a small system where the search for directories and files is limited. The test environment shown in the following log file was working against:

- Device taplow-11.skills-1st.co.uk
- zMonitorDir1 /opt/zenoss/local/fredtest
- zMonitorDir1File fred1.*
- zMonitorDir2 /var/log/
- zMonitorDir2File .*log$
- zMonitorDir3 /opt/zenoss/local/fredtest/test
- zMonitorDir3File fred2\.log.*

taplow-11 has two matching files in each of the directories.

The COMMAND string was limited by providing a subdirectory for the start of the *find*:

```
find /opt/zenoss/local -type d
```

Once the modeler code is complete, restart *zenhub* and *zopectl*. The *DirFileMap* modeler should now appear in the available list of the modeler plugins dialogue.

Move a test device to the */Server/Linux/DirFile* zenoss class and remodel it. To get debugging output, run the new modeler standalone:

```
zenmodeler run -v 10 -d taplow-11.skills-1st.co.uk --collect community.cmd.DirFileMap
```

Hopefully, after refreshing the GUI for the device in question, the two new components appear:

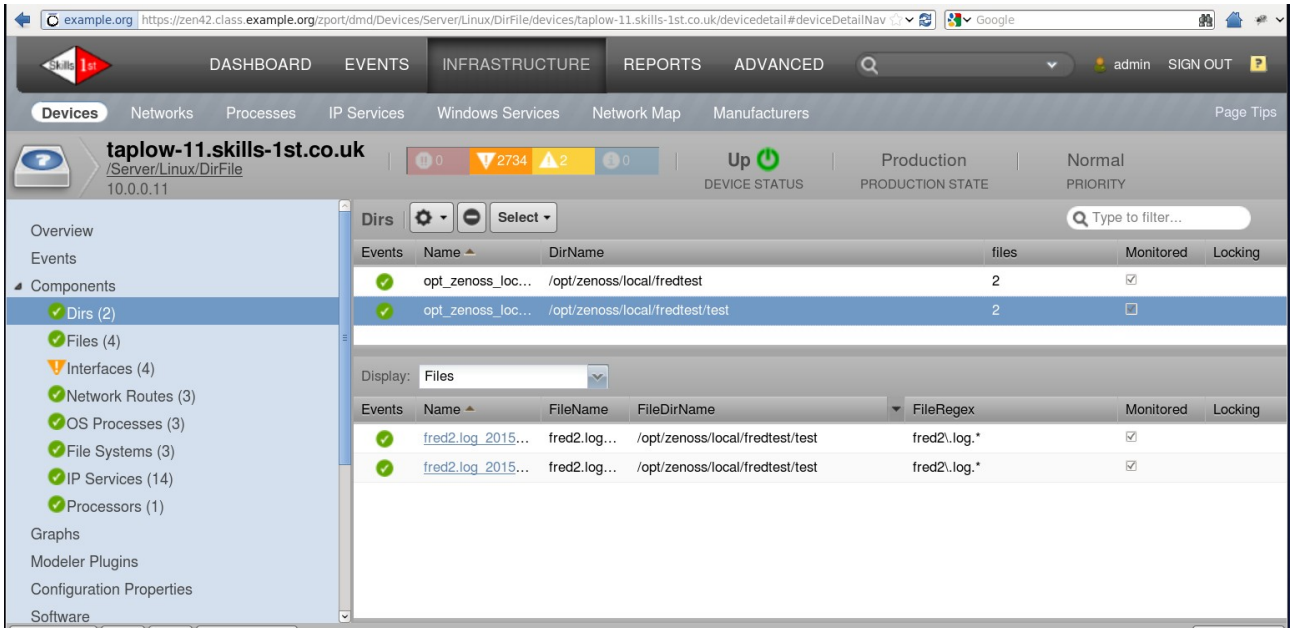


Figure 128: GUI showing Dirs and Files components

Note that both *Dirs* and *Files* are shown in the left-hand menu, even though *File* is a sub-component of *Dir*. The *Files* left-hand menu shows a correct count of 4 (2 for each directory).

Note that a really neat trick of the JavaScript code generated by zenpacklib is that it automatically creates a dropdown *Files* menu from the *Dir* component panel which shows the sub-components for that component. Before zenpacklib, this took a fair amount of complex JavaScript coding to achieve. For hand-written JavaScript samples, have a look at the A10 ZenPack at https://github.com/jcurry/ZenPacks.community.A10 or the RabbitMQ ZenPack at https://github.com/zenoss/ZenPacks.zenoss.RabbitMQ .

*11.5.7.1 * Analysing the zenmodeler log*

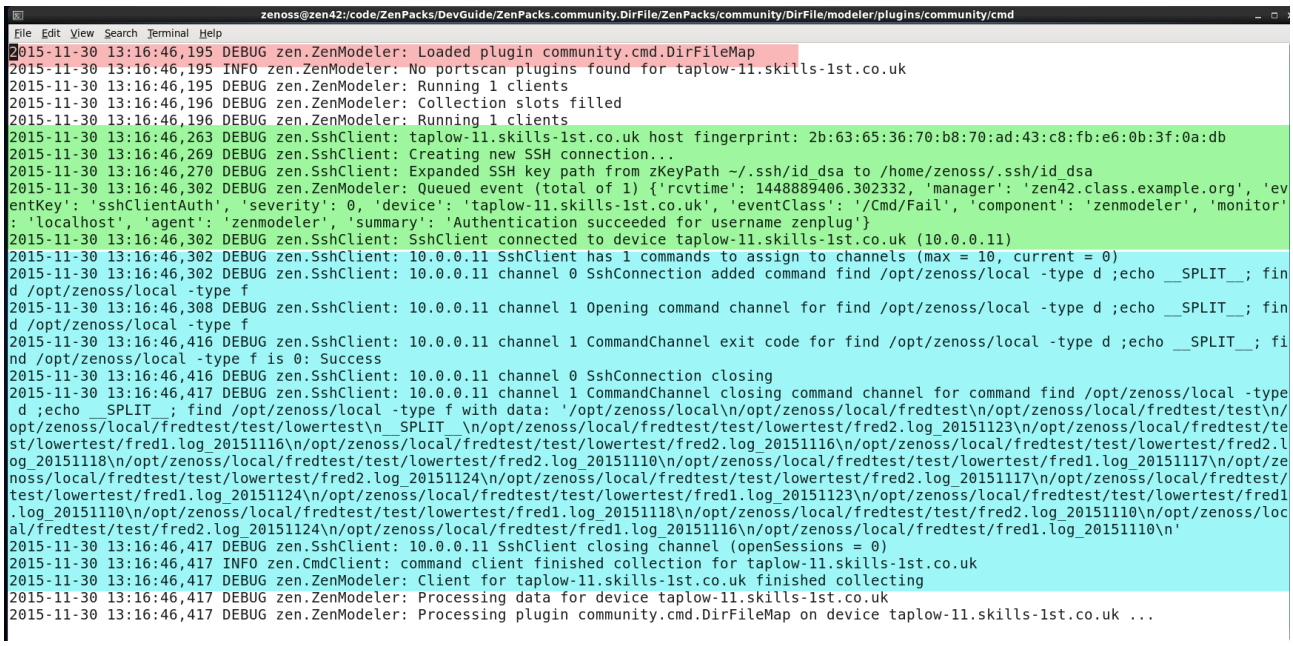


Figure 129: zenmodeler debug output for the DirFileMap modeler plugin - ssh connectivity

Figure 129 shows detailed logging of the ssh connection:

- The first (red) highlighted section confirms that the *DirFileMap* modeler has been loaded. If it does not appear in the list then suspect syntax errors in the modeler code.
- The second (green) highlighted section shows the ssh credentials being used
  - The host fingerprint is generated when the first ssh communication is established to the target host.

♦ Note that it is important to test ssh to each target **directly** as the first communication will generate the fingerprint entry and ask whether to add it to the *known_hosts* file in the *zenoss* user's *.ssh* directory. The target name must be **identical** to the name that Zenoss uses. Zenoss modelers will fail if asked this question. The direct test must be performed as the *zenoss* user.

♦ Note on Zenoss 5 the test must be performed from the *zencommand* **container** as the *.ssh/known_hosts* inside the container is **not** the same as that for the *zenoss* user on the base host.

```
serviced service attach zencommand su zenoss -l
ssh -l zenplug zenny2.class.example.org
cat .ssh/known_hosts
```

  - The *zKeyPath* zProperty is retrieved and used
  - The queued event summary has “Authentication succeeded for username *zenplug*” which gives the outcome and the username
- The third (blue) highlighted section shows the command being sent and the data being returned

The second part of the log shows the standard results output and the output from some of the log statements inserted into the modeler code.

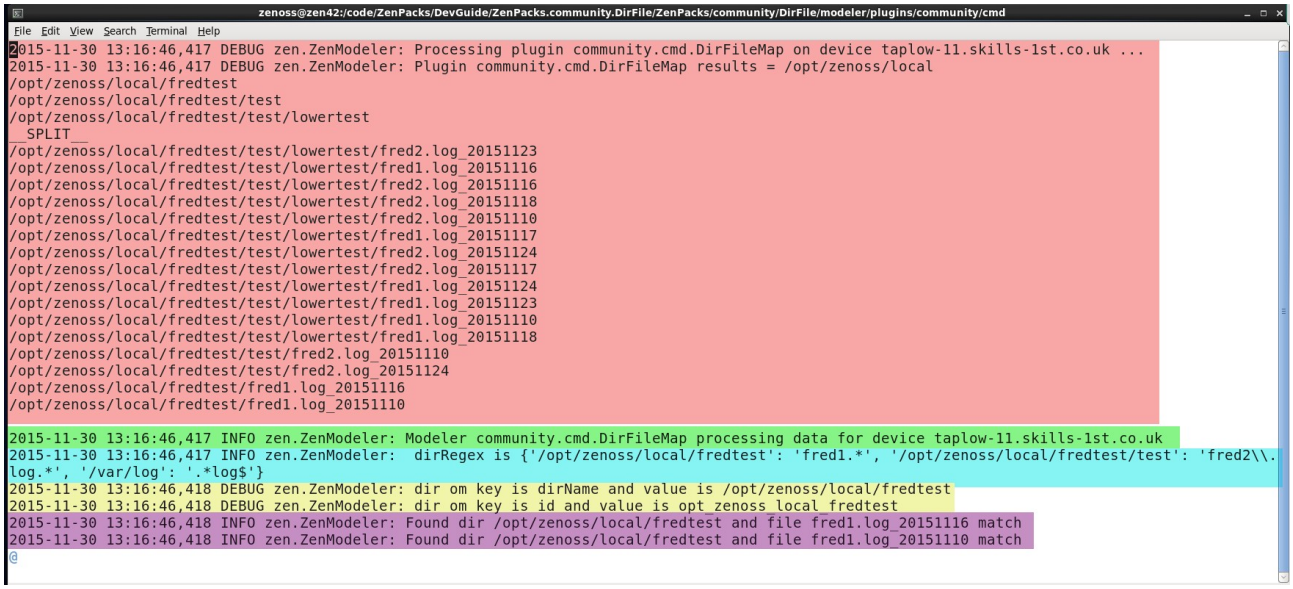


Figure 130: zenmodeler debug output for the DirFileMap modeler plugin -standard debug logging and specific ZenPack logging

The top highlighted section is standard output if zenmodeler is run in debug mode.

- Which modeler is run against which device
- The results output

The second section (green) is generated by the *log.info* statement at the start of the *process* method.

The third (blue) section is the modeler's *log.info* to display the *dirRegex* directory

The fourth (yellow) section is the result of the *log.debug* statement when creating the directory object:

```
for k,v in om.items():
  log.debug('dir om key is %s and value is %s' % (k, v))
```

The fifth (purple section) is from the *log.info* in *getFileMap*, each time a relevant directory / file pair is found.

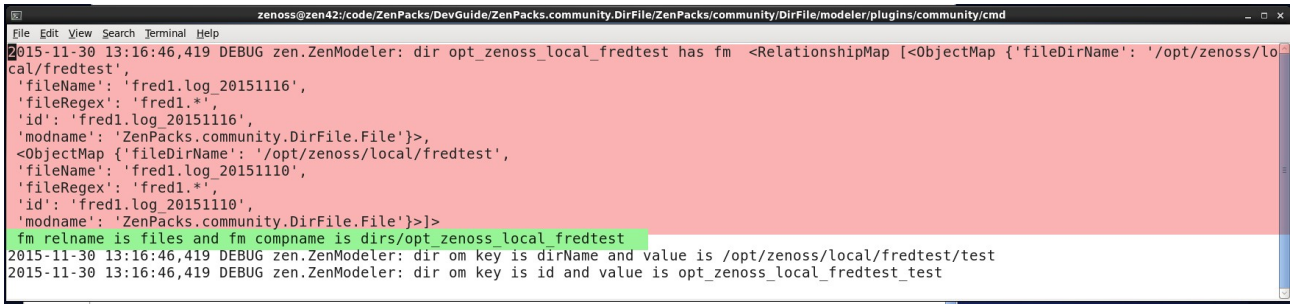


Figure 131: zenmodeler debug output for the DirFileMap modeler plugin -file relationship map plus fm rename and compname

The third part of the log is generated by the *log.debug* statement at the end of the first loop. The relationship for each directory / file matching pair is printed. A newline (*\n*) is used to ensure the *fm.relname is files and fm.compname*... is on a separate line for clarity.

```
        rm.append(om)
        # For this directory, create a map for associated files, passing this
        #  dir_id as part of compname
        fm = (self.getFileMap( device, dirlines[1], dirRegex, dir, 'dirs/%s' % dir_id, log))
        log.debug('dir %s has fm %s \n fm relname is %s and fm compname is %s ' %
              (om.id, fm, fm.relname, fm.compname))
```

The final part of the log shows the output of the *log.info* at the end of the process method, which outputs the directory relationship:

```
        log.info('Found matching dirs %s on %s \n dir relname is %s and dir \
         compname is %s ' % (rm, device.id, rm.relname, rm.compname))
```

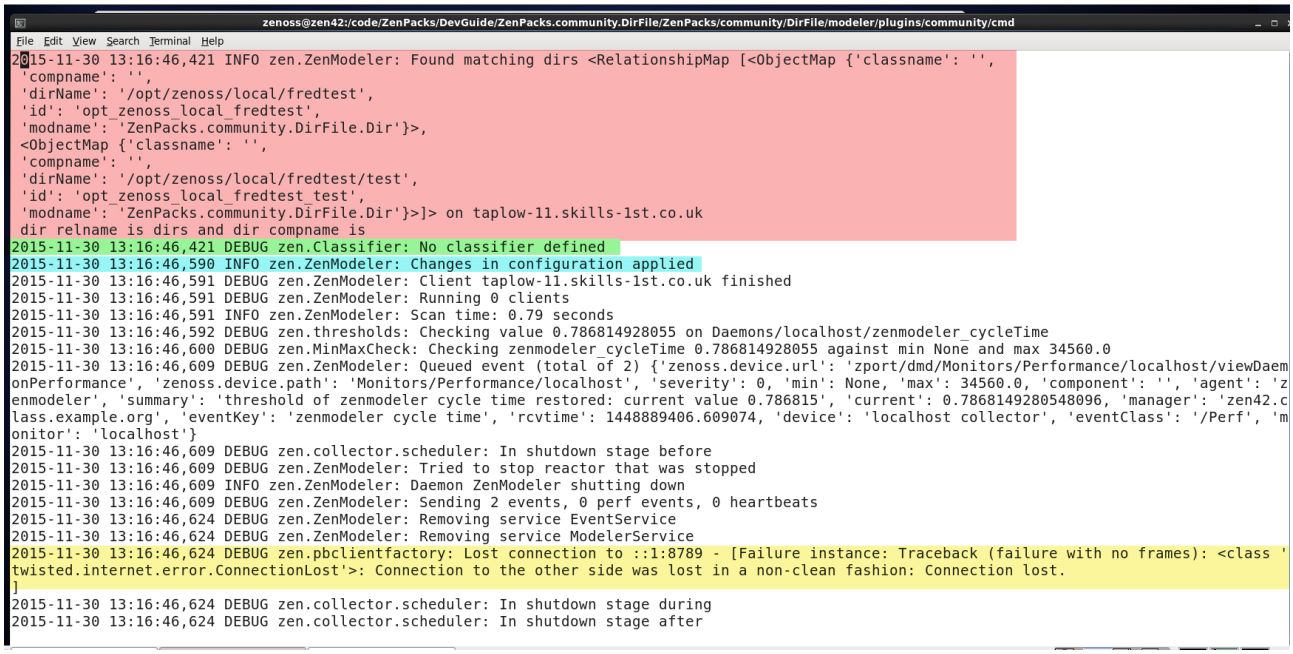


Figure 132: zenmodeler debug output for the DirFileMap modeler plugin - directory relationship and close

It is perfectly normal and not an error condition to get a line saying:

```
2015-11-30 13:16:46,421 DEBUG zen.Classifier: No classifier defined
```

Watch for a line saying “Changes in configuration applied”; this is likely to be good news.

The section at the end of the log highlighted in yellow is also perfectly normal and does **not** represent an error condition.

11.6 *monkeypatching so command modeler uses zProperties

A major drawback with a CommandPlugin modeler is that the actual command run cannot make use of a device's zProperties; the *process* method **can**, but the command definition **cannot**.

There is a very neat workaround shown as an example at https://github.com/cluther/ZenPacks.example.EvaluatedCommandModeler .

It is the *CollectorClient* class in *$ZENHOME/Products/DataCollector/CollectorClient.py* that manages the ssh connection to remote targets, to collect data. It uses a number of parameters defined in its *__init__* method:

```
def __init__(self, hostname, ip, port, plugins=None, options=None,
        device=None, datacollector=None, alog=None):
```

Note particularly the *plugins* and *device* parameters which will be populated by modeler routines, where the *device* is the object representing the target and the *plugins* is the *CommandPlugin*, including the command to be run (see Figure 122).

The *__init__.py* in the base directory of the ZenPack can monkeypatch the standard *CollectorClient* class to force the *plugin.command* to check for a "*${*" construct and perform a TALES evaluation if found. Thus the command definition in the modeler can use expressions such as *${here/zMonitorDir1}*.

This example is delivered in the *evalCommand* branch on GitHub. It starts from the original *master* branch and ZenPack version 1.0.0. The new branch generates version 1.0.3 of the ZenPack.

11.6.1 * Modifying __init__.py

The default *__init__.py* will be very minimal, simply importing zenpacklib and loading the yaml file:

```
from . import zenpacklib
zenpacklib.load_yaml()
```

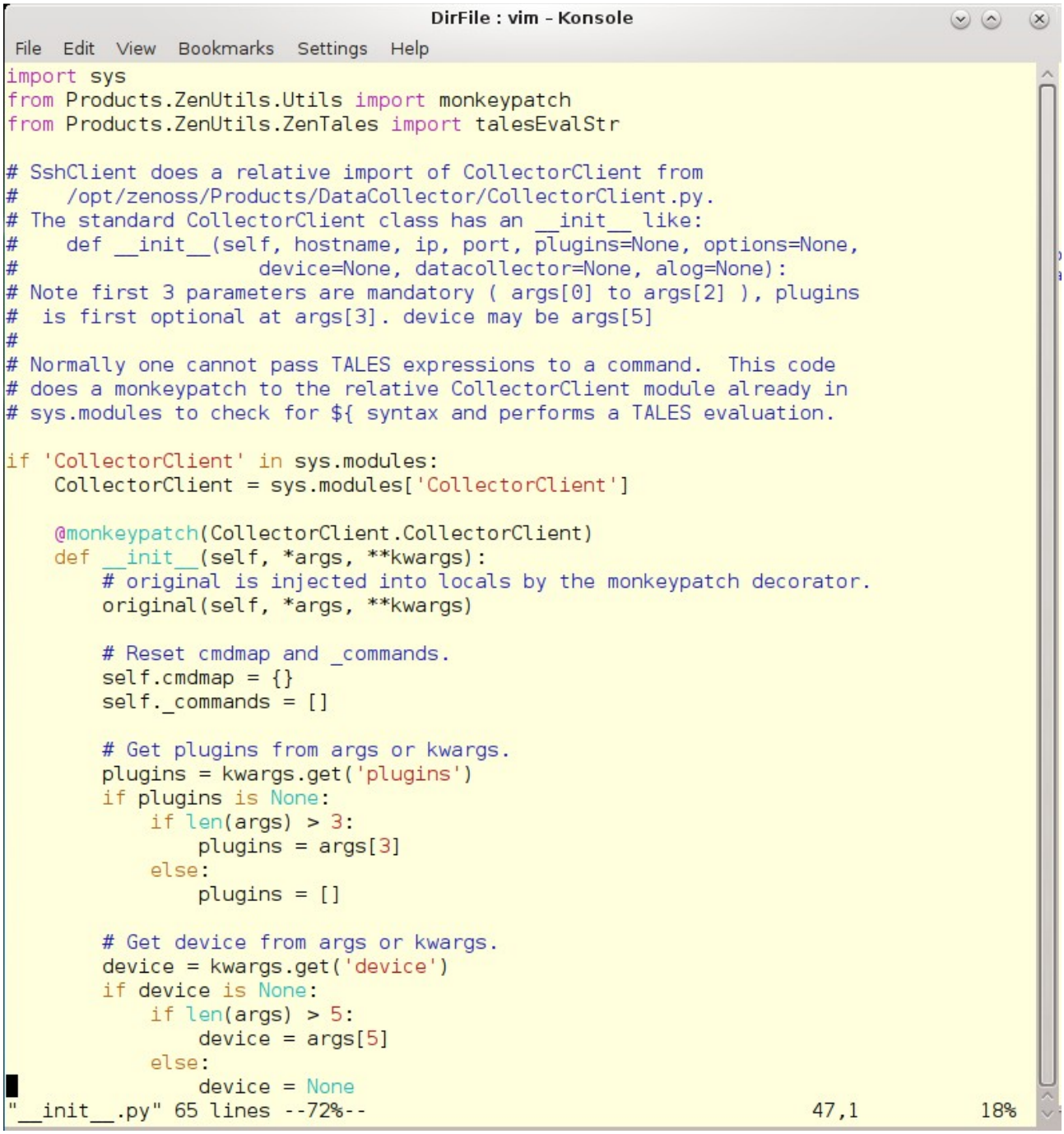


Figure 133: __init__.py (part 1) to monketpatch CollectorClient to permitnuse of zProperties in command

Note in Figure 133:

- *sys, monkeypatch* and *talesEvalStr* need importing.
- It is the *__init__* method that is being monkeypatched.
- The original arguments are accessed and the *device* and *plugins* parameters are extracted.
- The *cmdmap* and *_commands* attributes are reset to an empty dictionary and an empty list respectively.

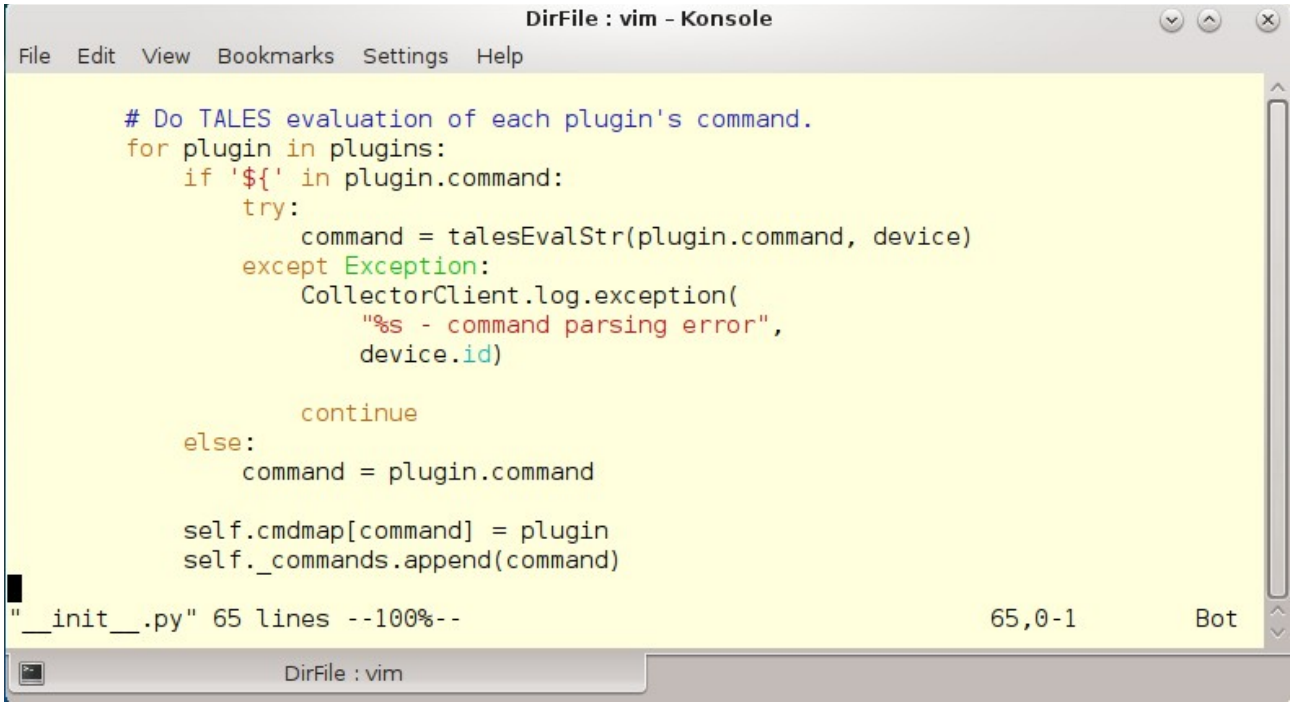


Figure 134: __init__.py (part 2) to monketpatch CollectorClient to permitnuse of zProperties in command

Each plugin is then tested for the *${* string; if found, the *talesEvalStr* method is used which takes the *plugin.command* and the *device* as parameters and performs a ZODB lookup to determine the resulting value of the zProperty specified in the calling *${here/zProp}* syntax. Otherwise the command is left in its original form.

11.6.2 * Modifying the modeler plugin code

The modeler plugin can be greatly simplified if the command has access to the zMonitorDir properties and its performance improved many-fold.

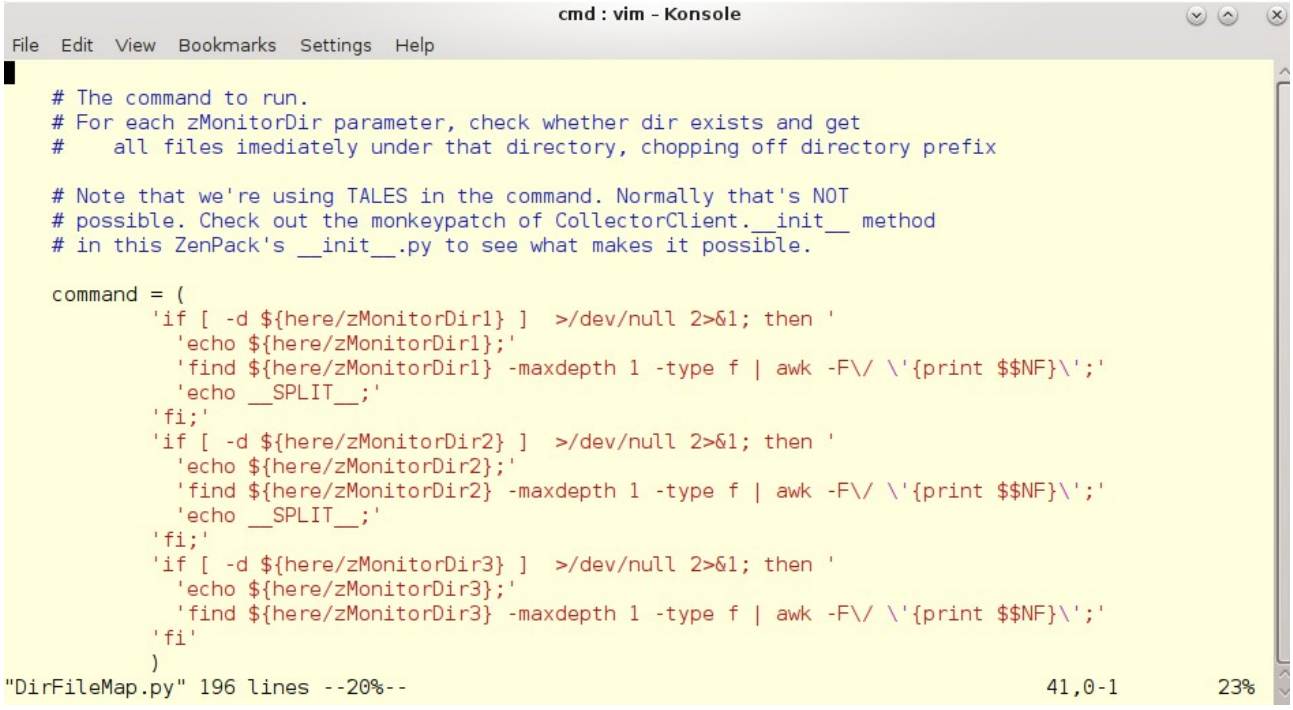


Figure 135: command definition for modeler plugin, using zProperties

For each zMonitorDir zProperty:

- The existence of the directory is checked with a simple test, discarding output to */dev/null*:

```
if [ -d ${here/zMonitorDir1} ] >/dev/null 2>&1
```

- The zMonitorDir zProperty is echoed to stdout
- The *find* command is used to list all files in this directory (avoiding directories and recursion into subdirectories). *awk* is used to return the last field when separated by "/" (which must be escaped).
- Note that the last field, normally *$NF*, needs an extra *$* to prevent $ being interpreted as a special character.
- Note that single quotes in the *awk* command need escaping with \
- The original modeler returned the full pathname of files. Since the directories are now known, this version only returns the short name of the file.

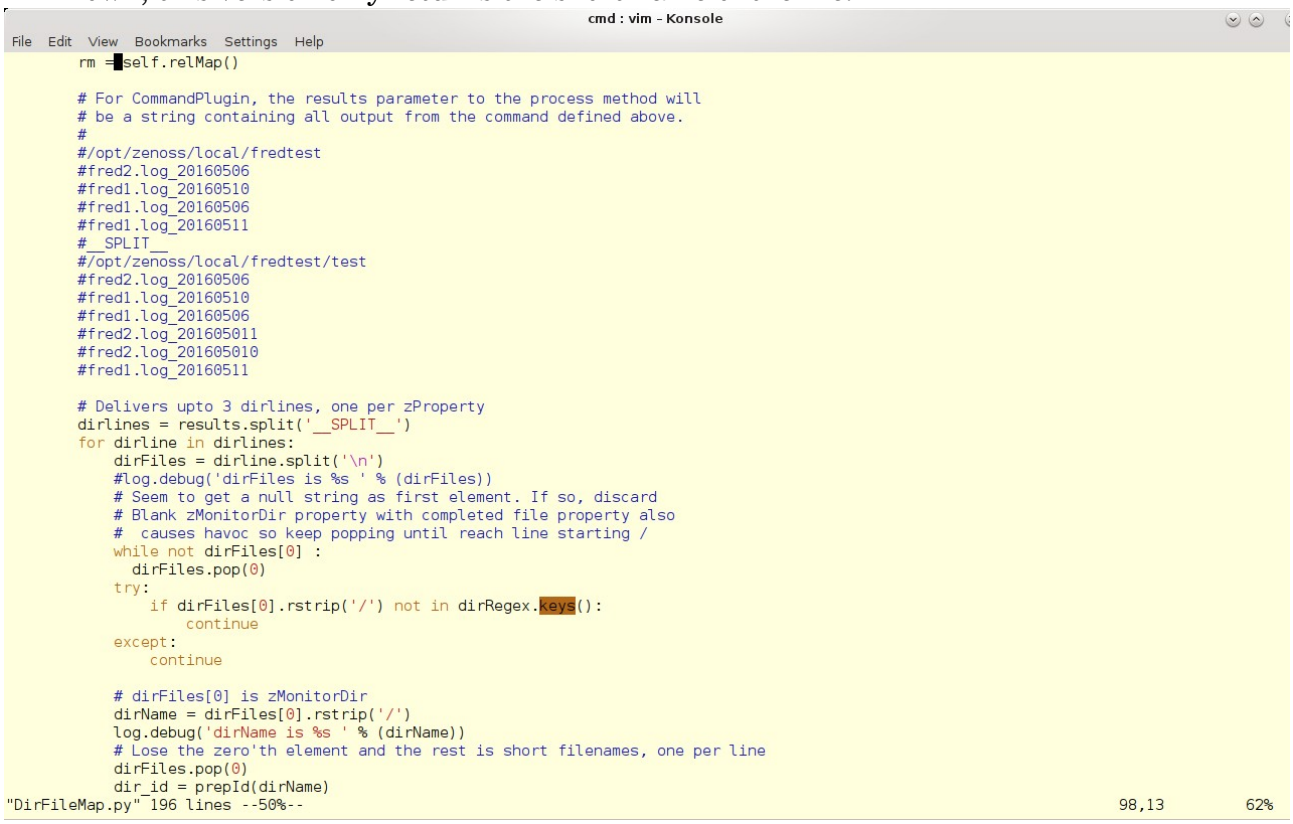


Figure 136: process method of modified DirFileMap modeler plugin- part 1

The process method of the modeler starts as before, building a *dirRegex* dictionary to hold the relevant zProperties and the maps collection is initialised.

Output from the command should be as shown in Figure 136 with a line for the directory name, followed by lines for each file in that directory. The pairs of directory / file output for each set of zProperties are separated by a line with *__SPLIT__*.

The first line for each property pair should be the directory but sometimes a blank line creeps in so they are discarded with:

```
dirFiles.pop(0)
```

A non-null first line is tested to ensure it matches one of the zMonitorDir directories.

The directory name is extracted into *dirName* for use in creating the directory object and the zero'th element containing the directory is then discarded with *pop*, leaving the remaining *dirFiles* holding a list of file names.

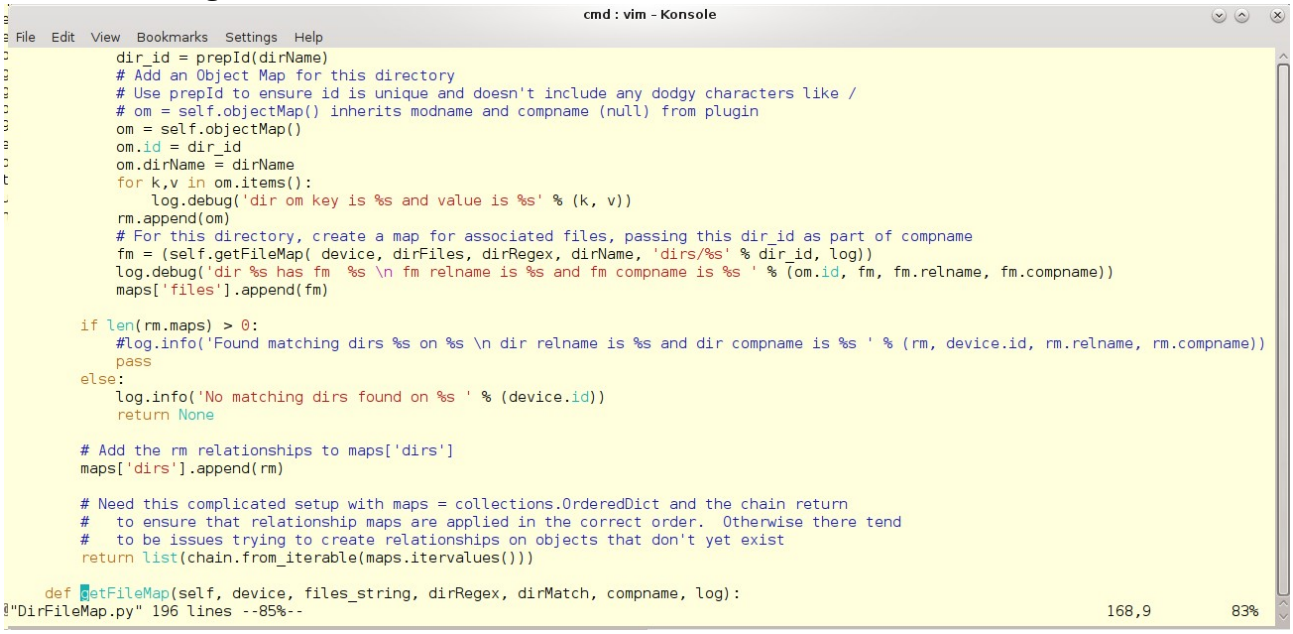


Figure 137: process method of modified DirFileMap modeler plugin- part 2

The directory object and relationship map are created exactly as before.

The *getFileMap* function is still used to test the list of files in the directory, against the regular expression given in the zMonitorDirFile zProperty. The main difference is that *dirFiles* is passed as the *files_string* parameter.

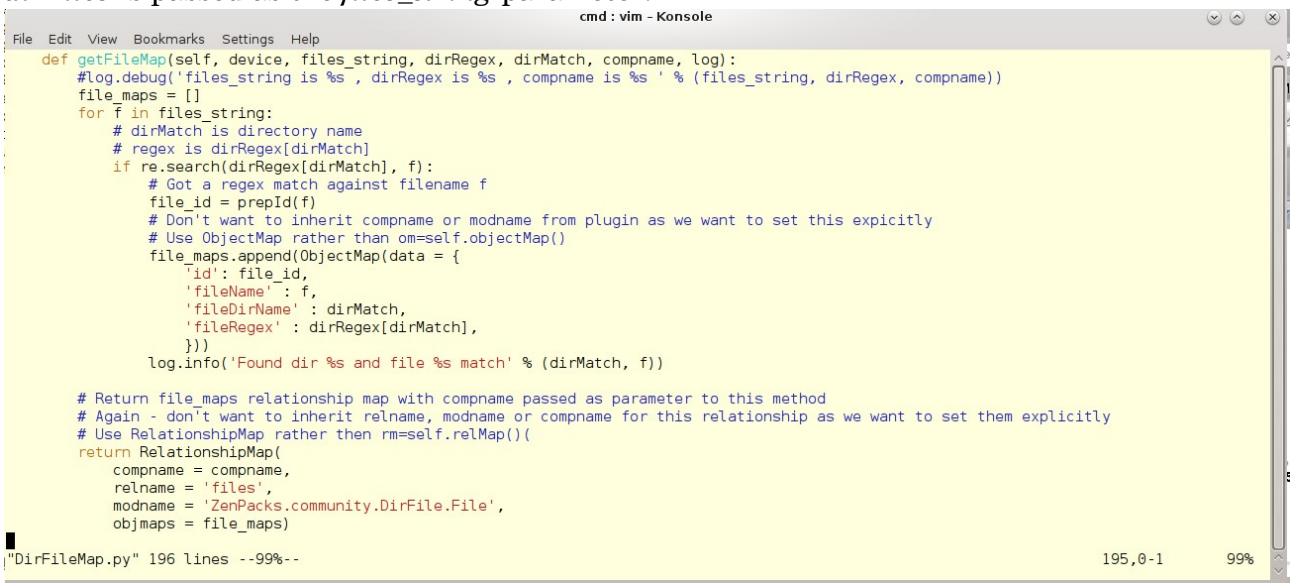


Figure 138: process method of modified DirFileMap modeler plugin-getFileMap method

The *getFileMap* function is also much simplified. Each file in the f*iles_string* parameter is tested against the regular expression for the specified directory. If a match is found then the file object is created and a *RelationshipMap* is returned.

11.6.3 * Testing the new code

Since *__init__.py* has been changed, test devices should be removed from the */Server/Linux/DirFile* class, the ZenPack reinstalled and Zenoss completely recycled. Move a test device back to the device class and remodel.

If the modeler does not appear in the modeler list then suspect syntax errors; try importing it into zendmd with:

```
import ZenPacks.community.DirFile.modeler.plugins.community.cmd.DirFileMap
```

If the modeler is shown in the list of modelers, but fails, run the modeler in debug mode:

```
zenmodeler run -v 10 -d zenny2.class.example.org --collect DirFilePythonMap > /tmp/fred 2>&1
```

11.6.4 * Caveats

The new modeler should produce exactly the same results as the original version; however it should take **far** less cycles to do so.

The only caveat with this solution, as with any ZenPack that monkeypatches core code, is that if Zenoss changes the underlying code in a new version or patch then breakage may well occur!

12.0 Collecting performance data

This section will continue to develop the DirFile ZenPack, exploring different ways of collecting performance data.

Commands give an easy way to collect performance data through monitoring templates. The **COMMAND** datasource type is run by the *zencommand* daemon.

The examples in this section are not intended as serious, production-strength monitors; the intention is to provide easy scripts with virtually no prerequisite knowledge or effort required to implement them.

12.1 Testing environment for the ZenPack

Testing was carried out against two target devices, each with two directories to be monitored, where each directory has a small handful of files, each containing several lines with specific string matches:

- *taplow-11.skills-1st.co.uk* and *taplow-30.skills-1st.co.uk* devices, both with:
  - zMonitorDir1 = /opt/zenoss/local/fredtest
  - zMonitorDir1File = fred1.*
  - zMonitorDir3 = /opt/zenoss/local/fredtest/test
  - zMonitorDir3File = fred2\.log.*

- /opt/zenoss/local/fredtest directory contains:

```
 bino:/opt/zenoss/local/fredtest # ls -l
total 16
-rw-r--r-- 1 jane users 119 Dec 2 17:36 fred1.log_20151110
-rw-r--r-- 1 jane users 559 Dec 2 17:37 fred1.log_20151116
-rw-r--r-- 1 jane users 952 Dec 3 22:40 fred1.log_20151202
drwxr-xr-x 3 jane users 4096 Dec 3 19:17 test
```

- /opt/zenoss/local/fredtest/test contains:

```
bino:/opt/zenoss/local/fredtest/test # ls -l
total 12
-rw-r--r-- 1 jane users 499 Dec 2 17:38 fred2.log_20151124
-rw-r--r-- 1 jane users 499 Dec 3 19:17 fred2.log_20151125
drwxr-xr-x 2 jane users 4096 Nov 29 18:17 lowertest
```

- Each file has a selection of lines, some with key words that some templates will be trying to match ("test 1" and "without"). For example:

```
 bino:/opt/zenoss/local/fredtest # cat fred1.log_20151110
test 1
test 2
without key word
test 1
test 1
test 1
test 1
test 1
test 1
test 1
test 1
test 1
test 1
test 1
a line with test
```

The test environment **must** be kept small. The objective is to demonstrate ZenPack features, not provide a production-strength ZenPack.

12.2 Collecting device performance data

As a simple example somewhat related to the *DirFile* scenario, the amount of free disk space on the root filesystem of a device will be gathered using a *COMMAND* datasource.

The script is ***df_root.sh***:

```
#!/bin/bash
#
# Use df to get disk free. Get result in bytes (-B 1) and use Posix flag
#  for compatibility.
# Check 6th whitespace separated field for /
#  output 3rd field (Used in KB) and make sure no duplicate lines

df -P -B 1 | awk -F " " '$6~/^\/$/ {print $3}' | uniq
```

Good practice might be to develop the script within the *libexec* directory of the ZenPack initially. Ensure the script is executable and that it runs successfully locally. The result must be numeric.

This script will need to be run against remote devices using ssh, using all the zCommand zProperties that should already be in place for modelling.

Note that *zCommandPath* will be appended to any command that does not have a fully-qualified path name. Note also that it is dubious practice to include environment variables in *zCommandPath* eg. *$ZENHOME/libexec*, as *$ZENHOME* may vary depending on remote systems and the *zCommandUsername* that is used for ssh. It is better to use explicit path names unless there is a good, well-documented reason for using environment variables.

Note that the *zCommandSearchPath* appears to be ignored by the zencommand mechanism. Use *zCommandPath* to set a path. (Tested with Zenoss 4.2.5 SUP 457).

Ensure that the correct zCommand zProperties are configured either at a device class or at a specific device level.

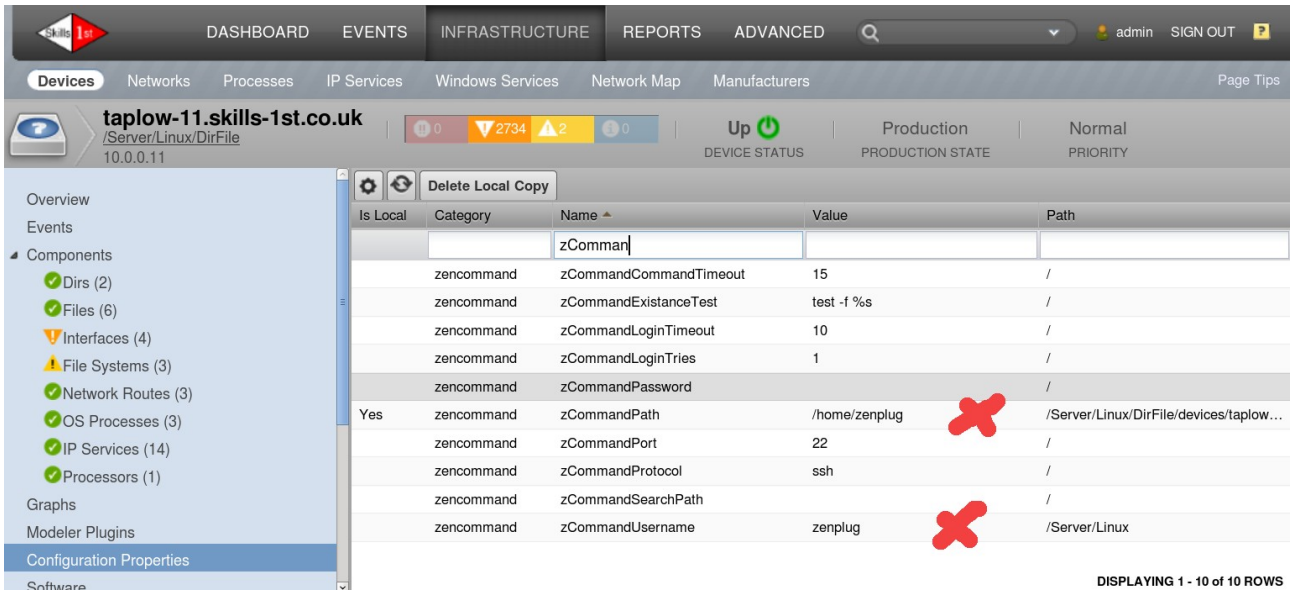


Figure 139: zCommand zProperties for ssh communication

The example here has:

- zCommandUsername zenplug
- zCommandPath /home/zenplug

Copy the script to the target device into */home/zenplug*, owned by *zenplug;* ensure that execute permission is set and test it.

Test the script manually from the Zenoss server, as the *zenoss* user:

```
[zenoss@zen42 libexec]$ ssh -l zenplug taplow-11.skills-1st.co.uk /home/zenplug/df_root.sh

The authenticity of host 'taplow-11.skills-1st.co.uk (10.0.0.11)' can't be
established.
RSA key fingerprint is 2b:63:65:36:70:b8:70:ad:43:c8:fb:e6:0b:3f:0a:db.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'taplow-11.skills-1st.co.uk' (RSA) to the list
of known hosts.
4729434112
```

Make sure you use exactly the same target name as Zenoss knows the device by, to ensure that a host key is established correctly in *known_hosts*.

Create a monitoring template, *Disk_free_df* and associate it with */Server/Linux/DirFile*. Create a datasource, *df* in the template, of type *COMMAND*.

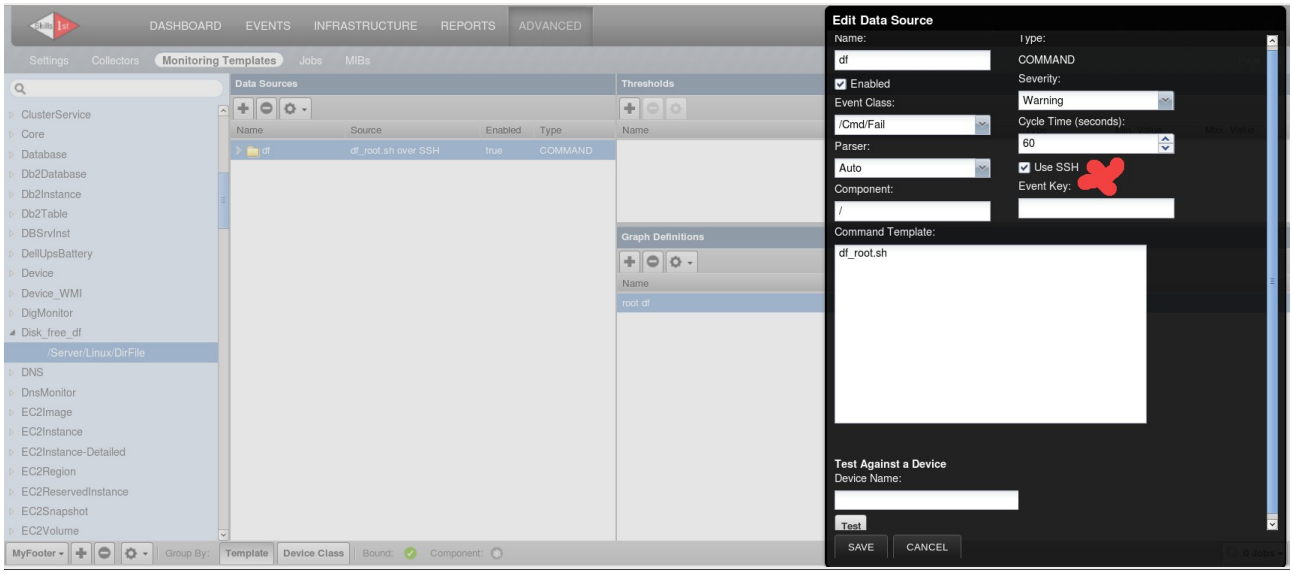


Figure 140: df datasource in the Disk_free_df performance template

Ensure that the *Use SSH* box is ticked. It is good practice to fill in the Component field if at all appropriate as this feeds through into the Events Console. Since the script is designed to only monitor the "/" filesystem, that will be used as the component.

Note that the Cycle Time is a configurable parameter for a *COMMAND* datasource, which defaults to 300s. 60s is useful for testing but obviously inappropriate in production for gathering disk space data.

With Zenoss 4, be aware that changing the Cycle Time of any datasource will invalidate any data already collected for that particular datasource and will result in no further data being collected with the new cycle time. This is because the rrd file has this parameter as its <step> value; try looking at an rrd file with:

```
rrdtool dump df_df_root.rrd | more
```

If the cycle time is changed, delete the existing rrd file and let it be recreated when the next datapoint is gathered. With Zenoss 5, the cycle time **can** be changed and the effect will be active at the next collection interval.

Make the *Command Template* box simply the executable filename, leaving Zenoss to append the *zCommandPath* before sending the command to the target.

Remember to create a datapoint for the output of the command. In this case it is trivial as there is only one result so a single GAUGE datapoint will suffice.

Create any graphs and thresholds required for the template.

Don't forget that the device template must be bound either at the device class level or to a specific device.

COMMAND datasources always have a *Test* function. Sometimes this is useful; sometimes it is misleading.

- Test works for running local scripts against devices
- Test sometimes works running local scripts against components
- Test does **not** work running against remote targets - it actually applies the zProperties and then tries to run the command locally (even though it reports that it is running against the remote device)

A much more reliable test is to run zencommand with the *--showfullcommand* option:

```
zencommand run -v 10 --showfullcommand -d taplow-11.skills-1st.co.uk
```

The *--showfullcommand* parameter is useful especially where values are substituted from zProperties. It does show all the substituted parameters of the command. In this particular example, it doesn't actually add anything significant.

Hopefully, typically after two cycle time intervals, data should appear in any graph that was defined for the template, under the left-hand *Graphs* menu. The impatient who are using Zenoss 4 and earlier, can check the rrd files under *$ZENHOME/perf/Devices/<name of device>* to see if a data file has been created. The file takes the format:

<datasource>_<datapoint>.rrd eg. df_df_root.rrd

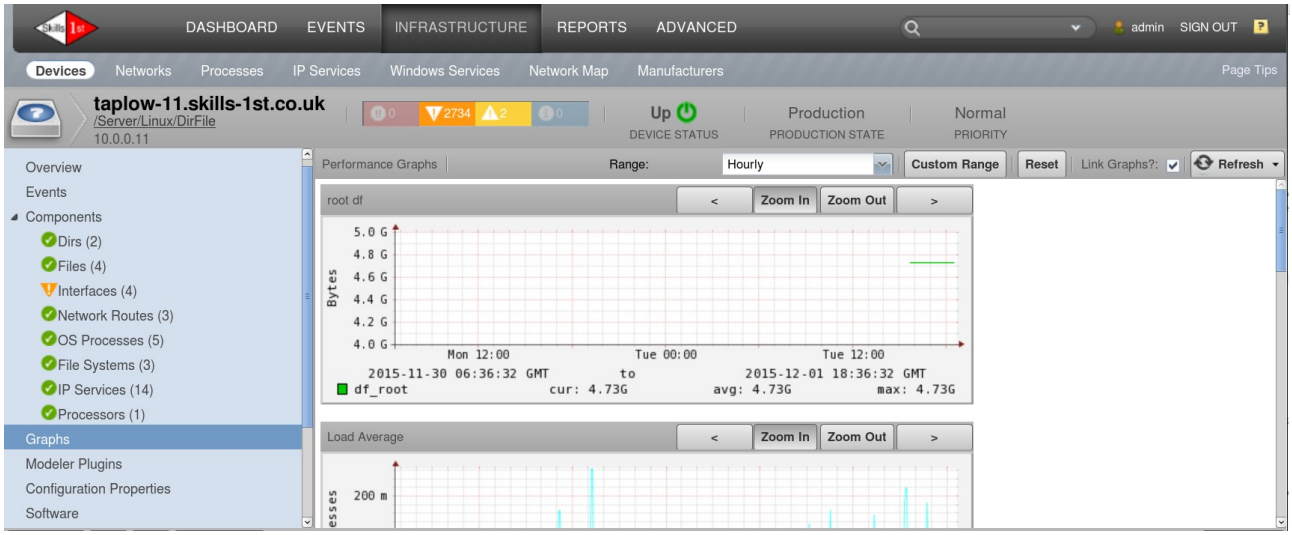


Figure 141: Graph of root df for taplow-11.skills-1st.co.uk

The template can be added to the ZenPack through the Zenoss GUI and then exported to save the new contents of *objects.xml*.

12.2.1 * Analysing the zencommand debug log

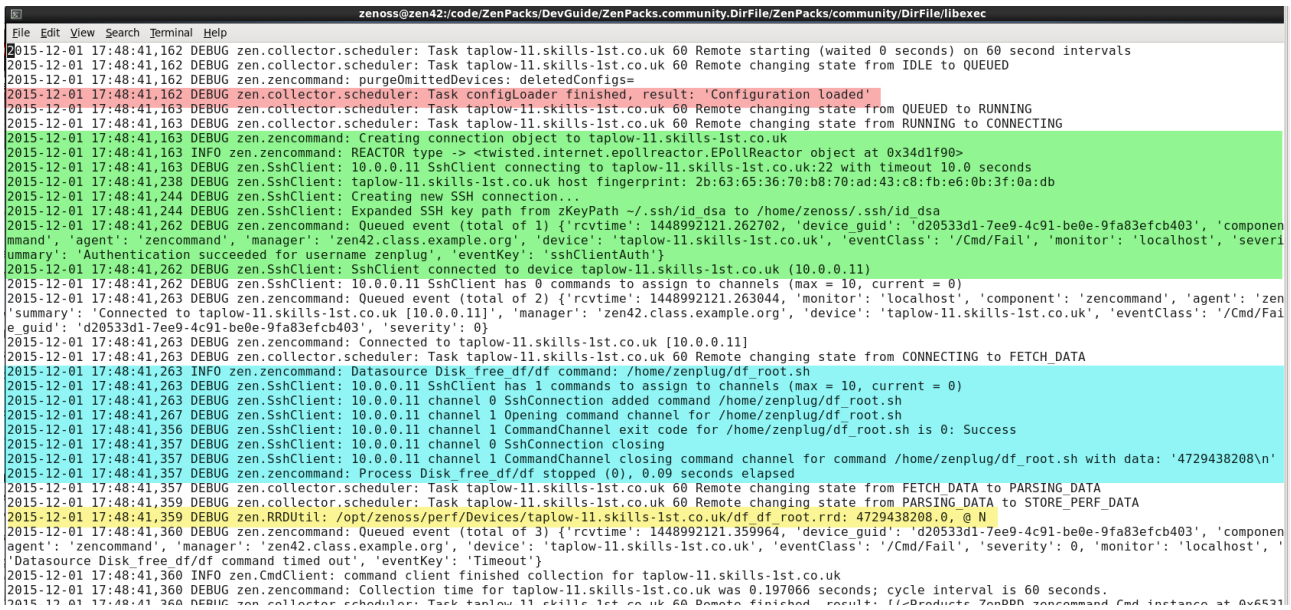


Figure 142: Output from zencommand run in debug

The first highlighted section (red) shows that the configuration has loaded. It is the responsibility of **zenhub** to analyse all templates and the devices they are bound to and then allocate tasks to different daemons for data collection. Bear in mind that it is perfectly possible to have multiple collector Zenoss systems, each typically running an instance of each of the collection daemons and looking after a subset of the devices. If there is an error in the template, especially if it includes a user-developed datasource, then the “Configuration loaded” message will be replaced by an error message.

The second (green) section shows *SshClient* (the same module used by zenmodeler) establishing an ssh session with taplow-11. Note the summary of the queued event with “Authentication succeeded for username zenplug”.

The third (blue) highlighted section shows the command being run and confirms its full path as */home/zenplug/df_root.sh*. The good news comes almost at the end of this section with:

```
zen.SshClient: 10.0.0.11 channel 1 CommandChannel closing command channel
for command /home/zenplug/df_root.sh with data: '4729438208\n'
```

Data has been found.

The ultimate good news is the final (yellow) highlighted line with a call to *zen.RRDUtil*. This means that data is actually being written to an rrd file. It is often worth searching a daemon debug log for *RRDUtil*; if there are no matches, there will be no data!

12.3 Collecting component performance data

There are a number of different scenarios with regard to collecting component performance data:

- *COMMAND* datasource command completely specifies command to be run f**or each component**. **Single** value returned.
- *COMMAND* datasource command completely specifies command to be run f**or each component**. **Multiple** values returned.

- *COMMAND* datasource command returns data for **all** components.
- *COMMAND* datasource passes **customized key values** with command which returns data for **all** components.

The first two can be handled without any serious coding.

12.3.1 Specific component command; single value returned

The requirement is to get the disk usage for each *File* component, using a command like:

```
/usr/bin/du -P -b /opt/zenoss/local/fredtest/fred1.log_20151110 | cut -f 1
```

The filename will need to be passed as a parameter for each *File* component. The *cut* command at the end ensures that only the number of bytes used, will be returned.

As with any datasource, attributes of either the device or the component can be made available. For a device-level template, the ***here*** object is the **device**; for a component, ***here*** is the **component** object. Access is also provided to the ***dev*** object, which for a device-level template, will be the same as *here*.

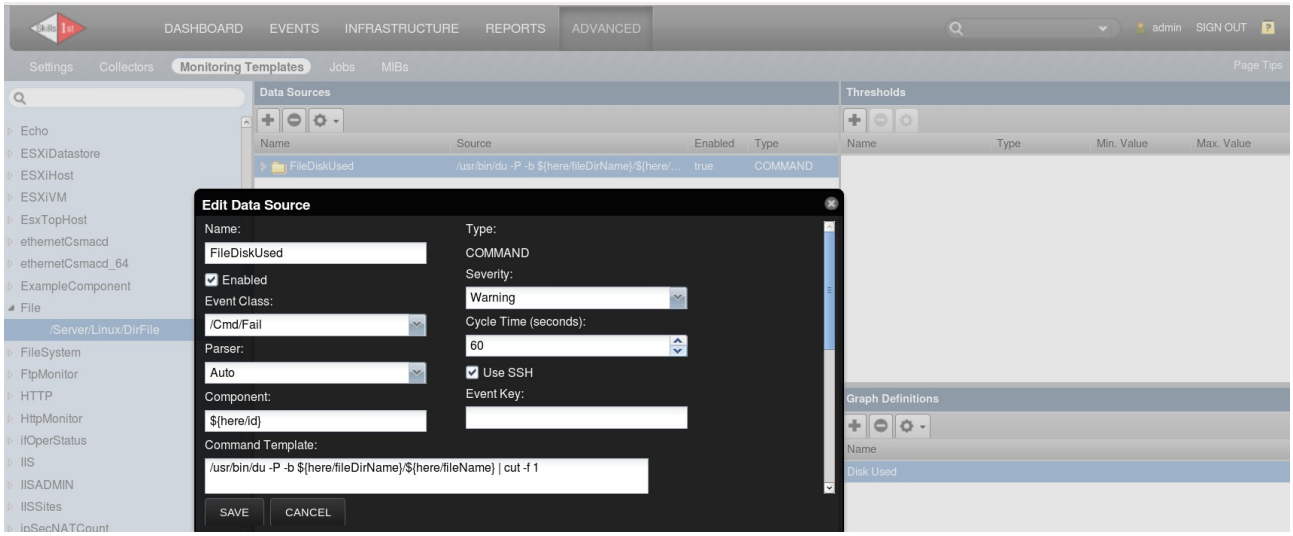


Figure 143: File datasource in the File component template

The **template** must be named *File* to match the component object and provide the automatic binding; the datasource can have any name (*FileDiskUsed*). The command has a **fully qualified** pathname and concatenates the *fileDirName* and *fileName* attributes of the *File* object to get the target parameter for the *du* command:

```
/usr/bin/du -P -b ${here/fileDirName}/${here/fileName} | cut -f 1
```

The cycle time has been reduced to 60s for testing and the *Use SSH* box is ticked. The Component field has been set to *${here/id}*. The rest of the key values are defaults.

A single datapoint, *diskUsed*, of type *GAUGE* has been created and a graph has been defined to display this datapoint.

Provided the command returns a single value and there is a single datapoint, the value automatically defaults to being associated with the defined datapoint.

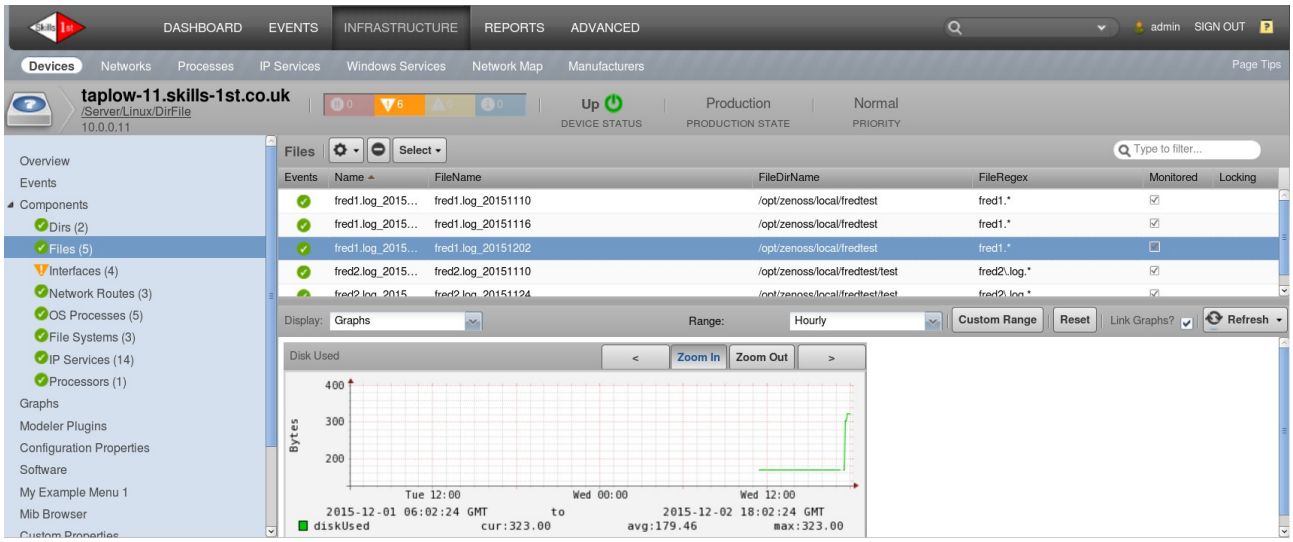


Figure 144: Graphical data for Disk Used on File component

12.3.2 Specific component command; multiple values returned

Supposing performance monitoring is to be extended to check for the presence of two specific strings in a *File* component and provide a count for the number of lines where the string appears. The strings are known in advance. A script, *file_stats.sh*, can be designed to deliver two values, with counts for each string. The script will be developed initially in the *libexec* directory of the ZenPack before being distributed to target systems.

```
#!/bin/bash
#
# Author:         Jane curry
# Date:          December 2nd 2015
# Updated:
# First parameter is filename to search for strings
#
# Nagios return codes
#set -x
STATE_OK=0
STATE_WARNING=1
STATE_CRITICAL=2
STATE_UNKNOWN=3
#
exitstatus=$STATE_OK

filename="$1"
# Substitute spaces for _
string1="test 1"
string1Name=`echo $string1 | sed -e 's/ /_/g'`
string2="without"
string2Name=`echo $string2 | sed -e 's/ /_/g'`
# Check that command is valid
stringCount1=$(grep "$string1" $filename 2>/dev/null )
if [ "$?" != 0 ]
then
  echo "Error collecting file stats"
  exit $STATE_WARNING
else
  stringCount1=$(grep "$string1" $filename | wc -l )
```

```
fi
# Check that command is valid
stringCount2=$(grep "$string2" $filename 2>/dev/null )
if [ "$?" != 0 ]
then
  echo "Error collecting file stats"
  exit $STATE_WARNING
else
  stringCount2=$(grep "$string2" $filename | wc -l )
fi

echo " File string count test ok | $string1Name=$stringCount1 $string2Name=$stringCount2"
exit $exitstatus
```

The trick for delivering multiple values to a datasource - whether it is for a component or a device - is to have the script echo output **in Nagios format,** see https://nagios-plugins.org/doc/guidelines.html#PLUGOUTPUT for further details. This basically means the output should be:

<string> | <var1>=<value1> <var2>=<value2> <var3>=<value3> .....

The <string> will appear in any event that is generated by running the command.

The exit status of the script will drive the severity of any event that is generated.

The <var1>=<value1> pairs deliver the data back to the datasource. The crucial trick is that the <var1> variable names **must exactly match the names of the datapoints** in the datasource.

In this script, the two strings are hardcoded where:

- string1="test 1"
- string2="without"

The strings are checked for white space as it is dubious practice to have datapoint names with spaces. The <var> names returned will have white space substituted by underscore, so output will be like:

```
File string count test ok | test_1=7 without=3
```

Thus, the datasource that runs this command will require the two GAUGE datapoints of *test_1* and *without*.

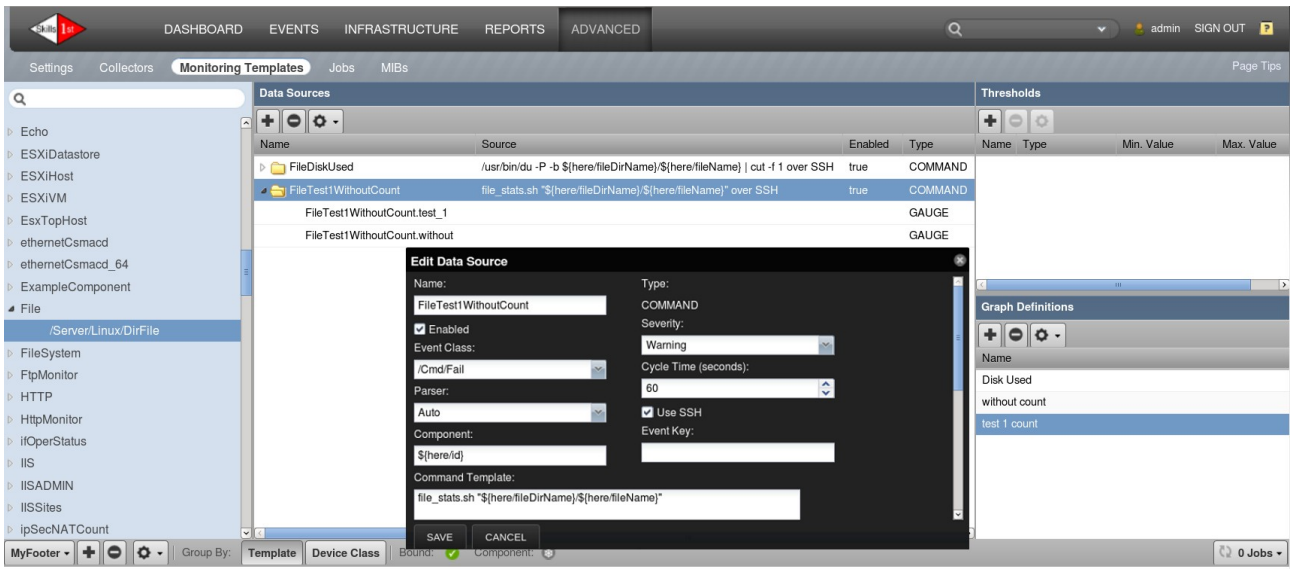


Figure 145: Datasource to run file_stats.sh with 2 datapoints and 2 graphs

Note in Figure 145:

- The template called *File* can have as many datasources as required, provided they are unique
- The datasource name is *FileTest1WithoutCount*
- The cycle time (60s) is artificially low for testing
- *Use SSH* is ticked
- The component field is set to *${here/id}*
- The command template is **not** fully-qualified so will look in the *zCommandPath* directory for the target host
- There are two datapoints. When creating the datapoint, it **must** match with the variable name in the output from the script; this is, *without* and *test_1*. The full name of the datapoint has the datasource name appended to the front giving *FileTest1WithoutCount.test_1* and *FileTest1WithoutCount.without*.
- A graph is created for *without count* and *test 1 count* each of which has the appropriate single datapoint.
- This is in a component template so no manual binding is required.

Note that it is good practice not to have underscore characters in datasource names. Especially with earlier versions of Zenoss, the underscore sometimes appeared as the divider between datasource and datapoint in a full datapoint name which could cause errors.

Test by running zencommand standalone in debug mode and redirect output to a file:

```
zencommand run -v 10 -d taplow-11.skills-1st.co.uk --showfullcommand > /tmp/fred 2>&1
```

If there are issues, it is always worth checking the Event Console for errors from the *zencommand* daemon.

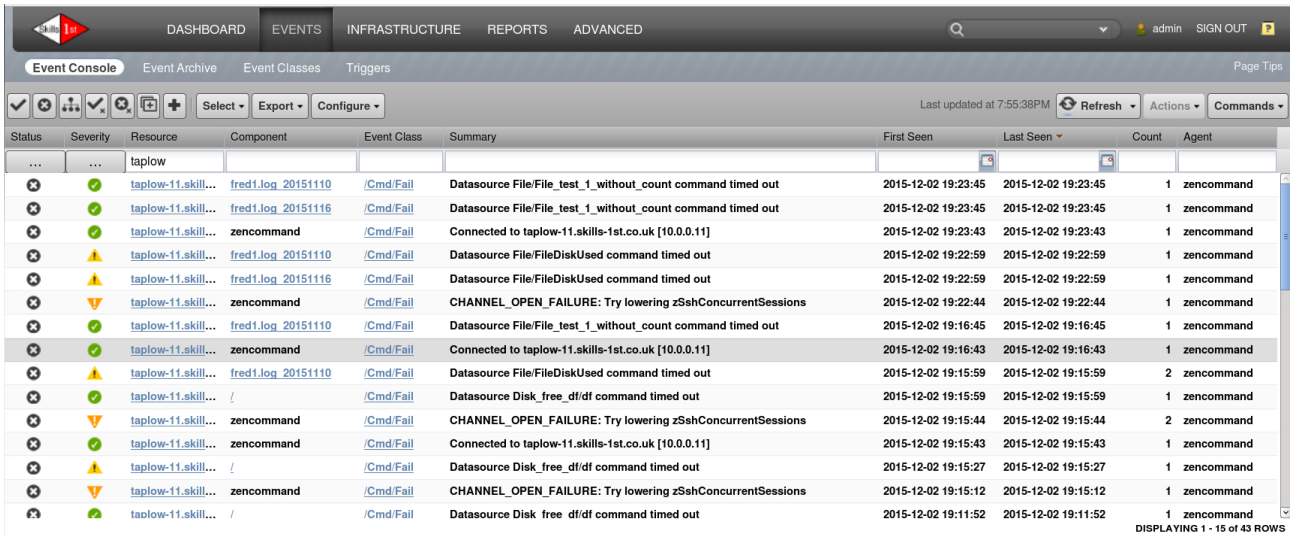


Figure 146: Event Console showing issues with running commands

As the load on both the *zencommand* daemon and on the remote target increases, occasional timeouts may be seen with subsequent clearing events. The value of customizing the *Component* field in the datasource can now be seen.

The event with a summary of *CHANNEL_OPEN_FAILURE: Try lowering zSshConcurrentSessions* was very frequent with this property set to the default of 10; reducing it to 5 eliminated this category of event.

Data should appear automatically in the graphs since this is another datasource in the *File* component template (so is automatically bound).

Figure 147: Graphs showing counts for "without" and "test 1" on the File component

With very little modification, this datasource could be copied and used to gather disk used statistics for *Dir* components, using the command:

```
/usr/bin/du -P -b -s ${here/dirName} | cut -f 1
```

12.3.3 Generic component command with parser

Take the scenario where the remote script is proscribed by someone else - part of a commercial package or the work of another department. The script delivers information about **all** components. A mechanism is needed at Zenoss to parse the command output and deliver appropriate values for appropriate components.

To demonstrate parsers, we shall use a rather contorted example to keep the Linux requirements absolutely minimal. *ls -l <directory>* provides output for each file in that directory, including, in the fifth field, the number of bytes in the file; the last field is the filename. This sample will run the *ls -l* command to bring back all files for a directory and then select the particular instance, by the ***fileName*** attribute, along with its size.

Parsers can (indeed, must) be used for any *COMMAND* datasource. A number are supplied as part of the core product under *$ZENHOME/Products/ZenRRD/parsers*:

```
[zenoss@zen42 parsers]$ pwd
/opt/zenoss/Products/ZenRRD/parsers
[zenoss@zen42 parsers]$ ls -l
total 56
-rw-r--r-- 1 zenoss zenoss 1576 Mar 11 2014 Auto.py
-rw-r--r-- 1 zenoss zenoss 1068 May 19 2015 Auto.pyc
-rw-r--r-- 1 zenoss zenoss 2504 Mar 11 2014 Cacti.py
-rw-r--r-- 1 zenoss zenoss 1982 May 19 2015 Cacti.pyc
-rw-r--r-- 1 zenoss zenoss  368 Mar 11 2014 __init__.py
-rw-r--r-- 1 zenoss zenoss  134 May 19 2015 __init__.pyc
-rw-r--r-- 1 zenoss zenoss 3159 Mar 11 2014 JSON.py
-rw-r--r-- 1 zenoss zenoss 5174 Mar 11 2014 Nagios.py
-rw-r--r-- 1 zenoss zenoss 3770 May 19 2015 Nagios.pyc
-rw-r--r-- 1 zenoss zenoss 10998 May 19 2015 ps.py
-rw-r--r-- 1 zenoss zenoss 3052 Mar 11 2014 uptime.py
```

Note in Figure 145 that there is a Parser field in the datasource dialogue which defaults to *Auto*. The *Auto* parser simply tries the *Nagios* parser first, then tries the *Cacti* parser.

There is an excellent, brief explanation of parsers in the old Zenoss 3 Developers' Guide in Chapter 12.5.2.

A ZenPack can provide a parser in the ***parsers*** subdirectory under the base directory. This directory is **not** created by default. Ensure that it also contains a *__init__.py*.

The core Zenoss code provides the *CommandParser* and *ComponentCommandParser* classes under *$ZENHOME/Products/ZenRRD*. A parser may have a complex job of decoding the output data; however, the *ComponentCommandParser* class provides a fairly easy way to decode output data, using Python regular expressions, and then matching that data to particular component instances.

The first, essential trick is that the *CommandParser* class name must match the component object class, so a parser for the *File* component must be called *File*. Note that *ComponentCommandParser* inherits from *CommandParser*.

A *ComponentCommandParser* has four attributes that need defining in the ZenPack parser to override the defaults in *ComponentCommandParser.py*:

- componentSplit = '\n'
- componentScanner = ''
- scanners = ()
- componentScanValue = 'id'

The assumption is that data will be delivered with information for each component instance on a separate line, hence *componentSplit* being set to the newline character. This holds good for *ls -l* data.

The *componentScanner* is a regular expression that picks out an element of the line that will match an attribute instance of the component. So for an example line like:

```
-rw-r--r-- 1 jane users 119 Dec 2 17:36 fred1.log_20151110
```

the componentScanner needs to match the last field to the *fileName* attribute of the component. The regular expression might be:

```
componentScanner = r'(\S+\s+)+(?P<component>.+)$'
```

The regex must contain the Python **named groups** construct to deliver a group which **must** be called ***component***. The regex above should deliver the **last** element of the line, separated by white space, into *component*.

The *componentScanValue* specifies the attribute of the component that needs to match with the *componentScanner* instance.

```
componentScanValue = 'fileName'
```

So, we are looking for a **File** component instance with a **fileName** attribute of *fred1.log_20151110*.

If this is found in the output data, the *scanners* list is another regular expression that defines what data value(s) to extract and where to put it.

```
scanners = [
    r'(\S+\s+){4}(?P<lsBytesUsed>[0-9]+)'
    ]
```

The regex passes over four whitespace-separated fields and picks out the fifth field, which must be numeric, and again uses the named groups technique to put the value into a group called *lsBytesUsed*. The group name in *scanners* can be anything but **must match a datapoint name in the datasource that calls this command**.

If the output of the command is amenable to this level of processing, then the parser only needs to contain these 4 lines in the new *ComponentCommandParser* class.

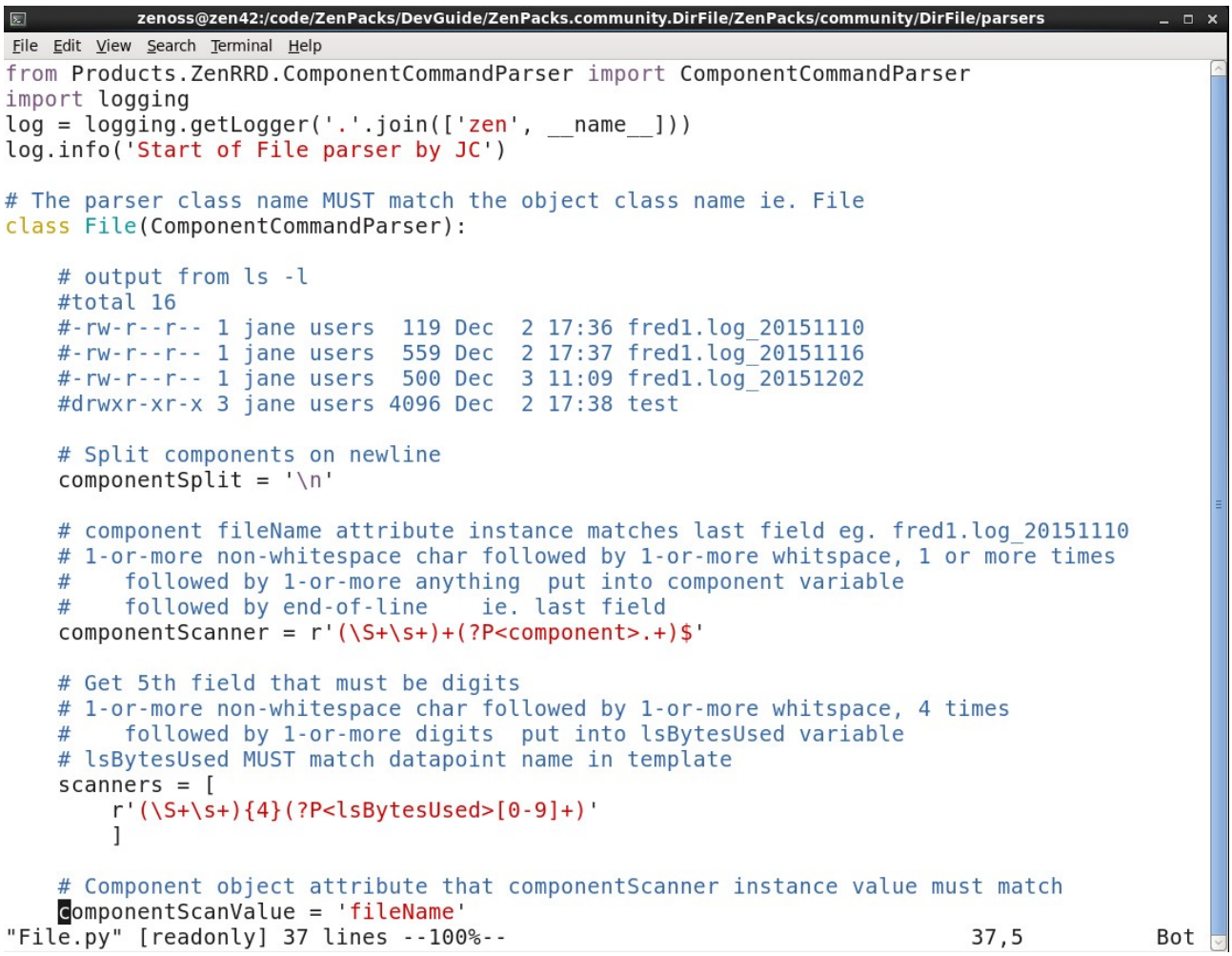


Figure 148: File ComponentCommandParser

When the parser code is complete, *zenhub* and *zopectl* need to be restarted.

The datasource to run this command will be another addition to the *File* component template. The command is artificially contorted, and wildly inefficient, to keep the Linux environment very simple; it will take the *fileDirName* attribute of a *File* component instance and run *ls - l* against that value.

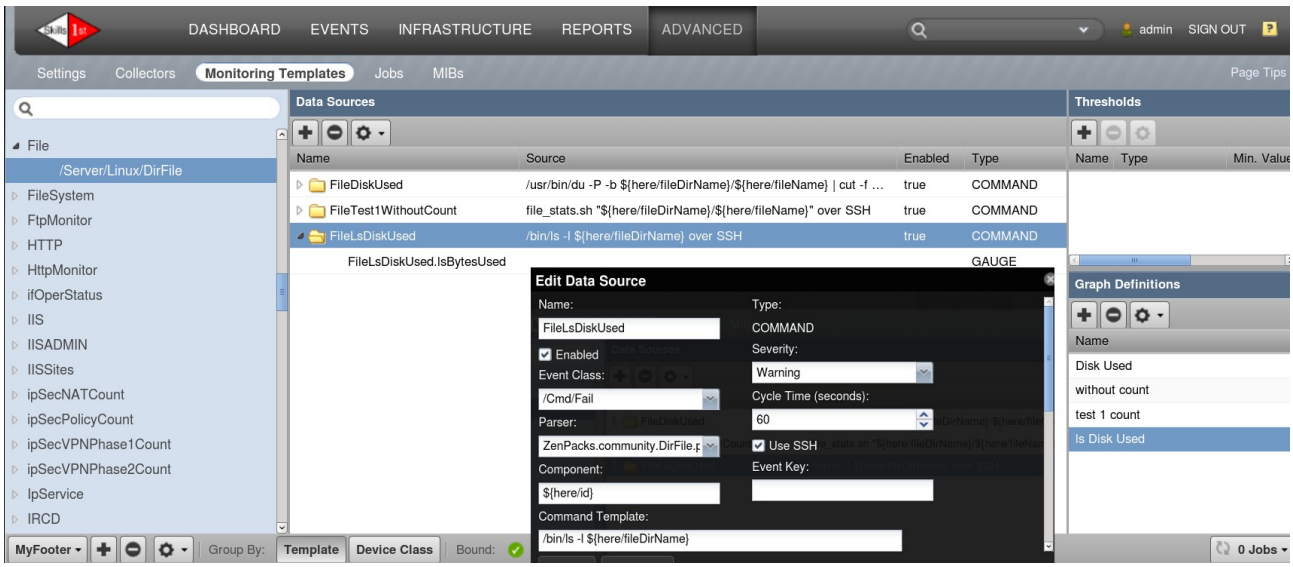


Figure 149: FileLsDiskUsed datasource for File component template using ZenPack parser

Note the dropdown box for Parsers. The first test of the new parser code is that it should appear in the list as:

ZenPacks.comunity.DirFile.parsers.File

If it does not appear, either the daemons have not been recycled or there may be a syntax error in the parser code.

Also note the datapoint for this datasource called *lsBytesUsed*. This **must** exactly match the named group in the scanner's regex in the parser code.

Debugging of parsers starts with debug level output from the *zencommand* daemon. Inspect carefully to ensure that the correct commands are run. If no data is returned then suspect that one of the regular expressions is incorrect. There is a very helpful Python regex checker application at http://www.pyregex.com/

There may also be helpful events in the Event Console along the lines of:

```
Error running parser <Products.DataCollector.Plugins.PluginLoader instance
at 0x6c977e8>
```

The *message* field contains the Python stack trace, which in this case, ended in:

```
invalid expression error: unbalanced parenthesis
```

which indeed was the case!

In the *Event Details* the event has an *output* field which contains all the output delivered by this command (the newlines get lost in the Console GUI):

```
total 16 -rw-r--r-- 1 jane users 119 Dec 2 17:36 fred1.log_20151110 -rw-r--
r-- 1 jane users 559 Dec 2 17:37 fred1.log_20151116 -rw-r--r-- 1 jane users
500 Dec 3 11:09 fred1.log_20151202 drwxr-xr-x 3 jane users 4096 Dec 2 17:38
test
```

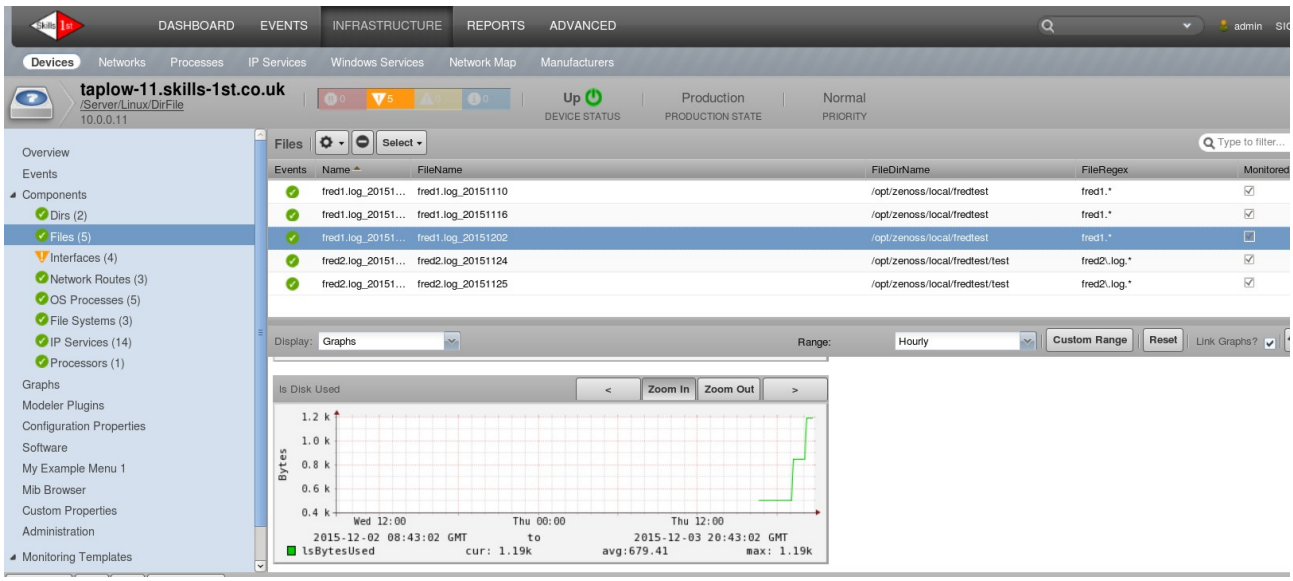


Figure 150: Graph of data from ls -l command using ZenPack parser to select data

12.3.4 customized datasource to pass customized key values

The final data collection scenario is where existing datasources do not provide all the functionality required. There are two main reasons for this:

- No suitable collection method exists
- More configuration is required in the datasource GUI dialogue

Many of the standard Zenoss core ZenPacks such as ApacheMonitor, DigMonitor, FtpMonitor and many others, create their own datasource, usually to control the dialogue. Most of these standard ZenPacks actually use a *CommandPlugin* to run a Nagios plugin, under the covers, even though the datasource type selected may be *ApacheMonitor, DigMonitor* or *FtpMonitor*. With these older ZenPacks, the new GUI is built with a mixture of new techniques and old TALES page template (.pt) files.

If a ZenPack gathers data using some form of API, like the *ZenPacks.zenoss.AWS* Amazon ZenPack or *ZenPacks.community.VMwareESXiMonitorPython* to collect VMware statistics, then the datasource will need to provide a new collection method, probably in addition to datasource GUI customisation. If developing new collection methods it is good practice to create **Python** datasources; this will be discussed in a later chapter.

Consider the earlier example for checking the number of lines where a particular word appeared in a file. The search strings were hardcoded in the script; how would one pass the desired string as a parameter from the datasource dialogue?

The requirement will be changed slightly to specify that the datasource will be limited to having a single datapoint which, by convention, will be called the same as the string to search for. The datasource dialogue needs to provide the same entry fields as a command datasource **plus** a box to specify the search string.

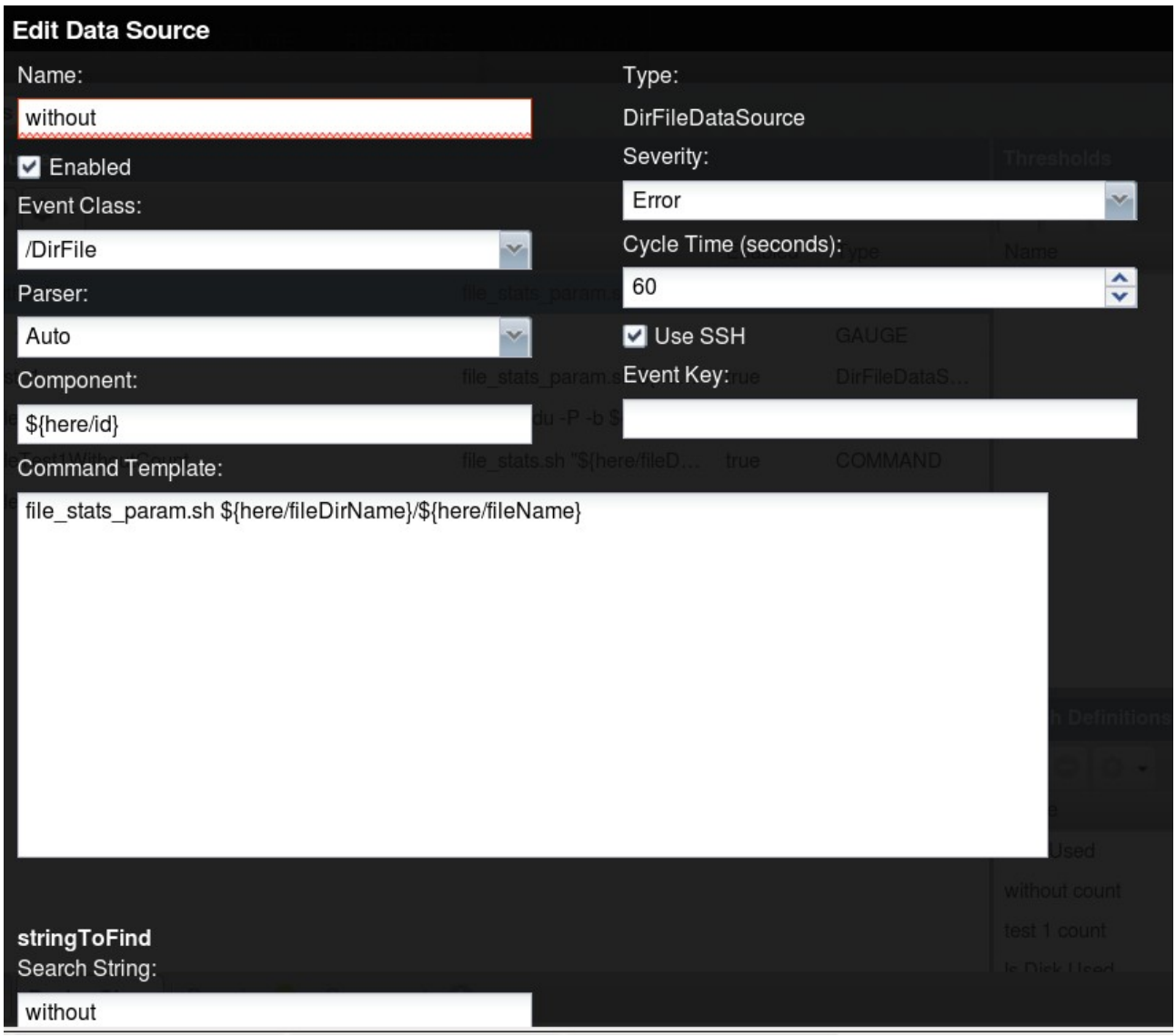


Figure 151: DirFileDataSource datasource to define "without" with single datapoint "matches"

Note the extra *Search String* field in the GUI dialogue in Figure 151. Also, the Command Template field has changed to reflect the slightly different script, *file_stats_param.sh*.

New datasources are created under the ***datasources*** directory of a ZenPack. If the ZenPack is created with zenpacklib then the directory will need manually creating along with its *__init__.py*. If the ZenPack is created from the Zenoss GUI then a *datasources* directory, with *__init__.py* will be created automatically.

The datasource file, *DirFileDataSource.py*, must be written in Python. A new class is defined for the datasource which defines:

- The name of the class
- The name of the datasource as seen in the GUI
- Any standard datasource fields whose default values are to be overridden in this ZenPack. These fields are defined in *$ZENHOME/Products/ZenModel/RRDDataSource.py* :
  - sourcetypes default = ()

    - sourcetype default = None
    - eventClass default = ''
    - cycletime default = 300
    - severity default = 3 (Warning)
    - enabled default = True
    - component default = ''
    - eventKey default = ''
    - commandTemplate default = “”
- Any new datasource fields
- Properties and relations for this new datasource class
- Any methods for the class. These may be either:
    - Overrides of existing methods inherited through the class hierarchy
    - New methods


As usual, it is good practice to setup logging so that it reflects the name of the ZenPack:

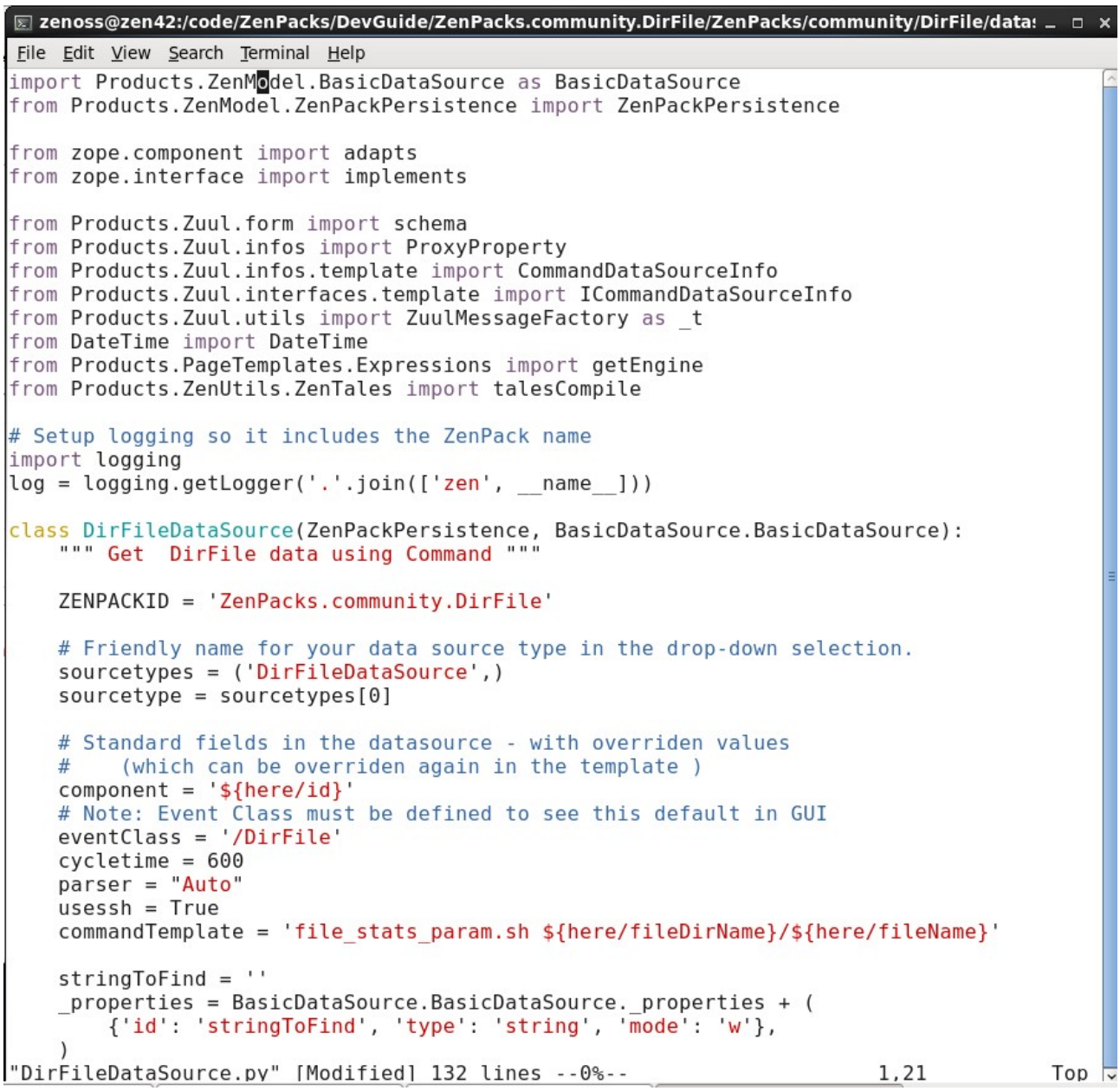


Figure 152: DirFileDataSource.py - imports, logging and attributes

```
# Setup logging so it includes the ZenPack name
import logging
log = logging.getLogger('.'.join(['zen', __name__]))
```

The *DirFileDataSource* class inherits from the ***ZenPackPersistence*** and ***BasicDataSource*** classes. *ZenPackPersistence* provides a catalog to associate objects in the ZODB with the ZenPacks that provide those object classes, to make it easy to delete all associated objects if a ZenPack is removed. Note that *ZenPackPersistence* should be the first class listed in a list of inheritance classes to ensure that its methods are not overridden.

BasicDataSource (defined in *$ZENHOME/Products/ZenModel/BasicDataSource.py*) is the base class used for SNMP and COMMAND datasources.

A common convention in datasource files is to override the *sourcetypes* and *sourcetype* attributes from RRDDataSource. Typically *sourcetypes* just contains the “friendly” name of

the datasource to be used in the Zenoss GUI; *sourcetype* (singular) is then assigned to the first element (that is, the zero'th element) of that tuple.

Any of the attributes inherited by the new datasource class can have their defaults changed. Note that if an *eventClass* is defined, that class must already exist in the ZODB database when an instance of this datasource is created; otherwise the *eventClass* field will be blank in the GUI.

The *commandTemplate* attribute reflects the new script to be run with its single parameter.

Any new attributes should be defined with a default value and added to the *_properties* stanza.

```
stringToFind = ''
_properties = BasicDataSource.BasicDataSource._properties + (
  {'id': 'stringToFind', 'type': 'string', 'mode': 'w'},
)
```

The *_relations* attribute typically just inherits relations from its parent class:

```
_relations = BasicDataSource.BasicDataSource._relations + (
    )
```

There are a number of methods defined for the base DataSource classes, some of which are already overridden in other base classes; they can be further overridden in the new class and new methods can be defined.

12.3.4.1 getDescription method

```
def getDescription(self):
  # getDescription in BasicDataSource only sets values if type = COMMAND or SNMP
  # This is the comment under Source that you see in the template against the datasource
  if self.usessh:
    return self.commandTemplate + " " + self.stringToFind + " over SSH"
  else:
    return self.commandTemplate + " " + self.stringToFind
```

getDescription in the *BasicDataSource* class only sets a useful value if the *sourcetype* is SNMP or COMMAND; otherwise it returns *getDescription* from the *RRDDataSource*, which is ***None***. The code has simply been copied from the COMMAND option in *BasicDataSource*.

12.3.4.2 useZenCommand method

```
def useZenCommand(self):
  # useZenCommand in BasicDataSource only returns True for sourcetype == 'COMMAND'
  return True
```

The new class is, in fact, going to run a command so needs the *useZenCommand* method to return *True* for the new sourcetype.

12.3.4.3 getCommand method

```
def getCommand(self, context, cmd=None):
  # No getCommand in BasicDataSource - inherits from
  #  SimpleRRDDataSource inherits from RRDDataSource
  # Duplicate getCommand from RRDDataSource and add stringToFind
  # Perform a TALES eval on the expression using self
  if cmd is None:
```

```
    cmd = self.commandTemplate
  if self.stringToFind:
    #Need to ensure any white space is wrapped in quotes
    cmd = cmd + ' "' + self.stringToFind + '"'
  if not cmd.startswith('string:') and not cmd.startswith('python:'):
    cmd = 'string:%s' % cmd
  compiled = talesCompile(cmd)
  d = context.device()
  environ = {'dev' : d,
        'device': d,
        'devname': d.id,
        'ds': self,
        'datasource': self,
        'here' : context,
        'zCommandPath' : context.zCommandPath,
        'nothing' : None,
        'now' : DateTime() }
  res = compiled(getEngine().getContext(environ))
  if isinstance(res, Exception):
    raise res
  res = self.checkCommandPrefix(context, res)
  return res
```

BasicDataSource provides a *getCommand* that is almost what is required but it needs the new *stringToFind* attribute appended to the command. *stringToFind* is enclosed in quotes so that the entire attribute is finally passed to the command on the target machine as a single parameter; thus the string can include white space.

12.3.4.4 addDataPoints method

addDataPoints for the new class defines a single datapoint called *matches*.

```
def addDataPoints(self):
  # Add a datapoint called matches if it isn't defined in the template
  if not hasattr(self.datapoints, 'matches'):
    # there is no manage_addBasicDataPoint method - only manage_addRRDDataPoint
    self.manage_addRRDDataPoint('matches')
```

The *addDataPoints* method is redefined in both *SimpleRRDDataSource* and *BasicDataSource* (*BasicDataSource* inherits from *SimpleRRDDataSource*, inherits from *RRDDataSource*), leaving the inherited method only defined for datasources of type SNMP.

12.3.4.5 Infos, Interfaces and configure.zcml

Since the ZenPack is defining new GUI elements, there also need to be ***interface*** and ***info*** definitions and an entry in ***configure.zcml*** to tie them all together. The info and interface class may be put in *info.py* and *interfaces.py*, respectively in the base directory of the ZenPack; alternatively, the new info and interface classes can be defined in the same datasource file as the main class. *configure.zcml* in the base directory **is** required and will reflect where the info and interface classes are defined. In this example, all the classes will go in the datasource file.

There is **no** need to create any JavaScript or page template code to deploy the new datasource.

Base definitions for datasource info and interface classes are under *$ZENHOME/Products/Zuul/infos* and *$ZENHOME/Products/Zuul/interfaces* respectively; in each case there is a *templates.py* file. Each file defines classes for:

- RRDDataSource
- BasicDataSource
- CommandDataSource inherits from BasicDataSource
- SNMPDataSource inherits interface from BasicDataSource
- PingDataSource inherits from RRDDataSource

Unless annotated above, inheritance is from the very basic *IInfo* / *InfoBase* classes.

There is a rather inconsistent mixture of attributes defined in these files. For example, *RRDDataSource* includes *component* and *eventKey* fields but *BasicDataSource* does not.

The *CommandDataSourceInfo* and matching *ICommandDataSourceInfo* are the best starting points for the new classes as they have all the standard fields plus those specific to running commands - *cycletime, parser, usessh* and *commandTemplate*.

If the new classes inherit from these command templates then it is only necessary to define the new field, *stringToFind*.

```
class IDirFileDataSourceInfo(ICommandDataSourceInfo):
  """Interface that creates the web form for this data source type.
    These entries define fields you see in the GUI
    The group statement is to keep attributes together on the GUI.
  """

  stringToFind = schema.TextLine(
    title = _t(u'Search String'),
    group = _t('stringToFind'))
```

Note that the *group* name can be used to keep several new GUI definition fields together.

```
class DirFileDataSourceInfo(CommandDataSourceInfo):
  """ Adapter between IDirFileSourceInfo and DirFileSource
    These entries define the default data that you see in GUI fields
  """

  implements(IDirFileDataSourceInfo)
  adapts(DirFileDataSource)

  stringToFind = ProxyProperty('stringToFind')

  # Component template doesn't run over SSH in GUI anyway, so disable
  testable = False
```

Note that *testable* = *False* removes the test button from the GUI dialogue.

As with all info and interface definitions, it is usually good practice to ensure that any attribute appears in both class types.

If there is an interface entry but no matching info then the keyword will appear in the GUI dialogue but will not show default values and changes in the GUI will not be honoured. If there is an entry for the info but not the interface then the keyword will not appear in the

GUI but default values will be honoured. The Zope Management Interface (ZMI) is a good tool to check what values exist on the object.

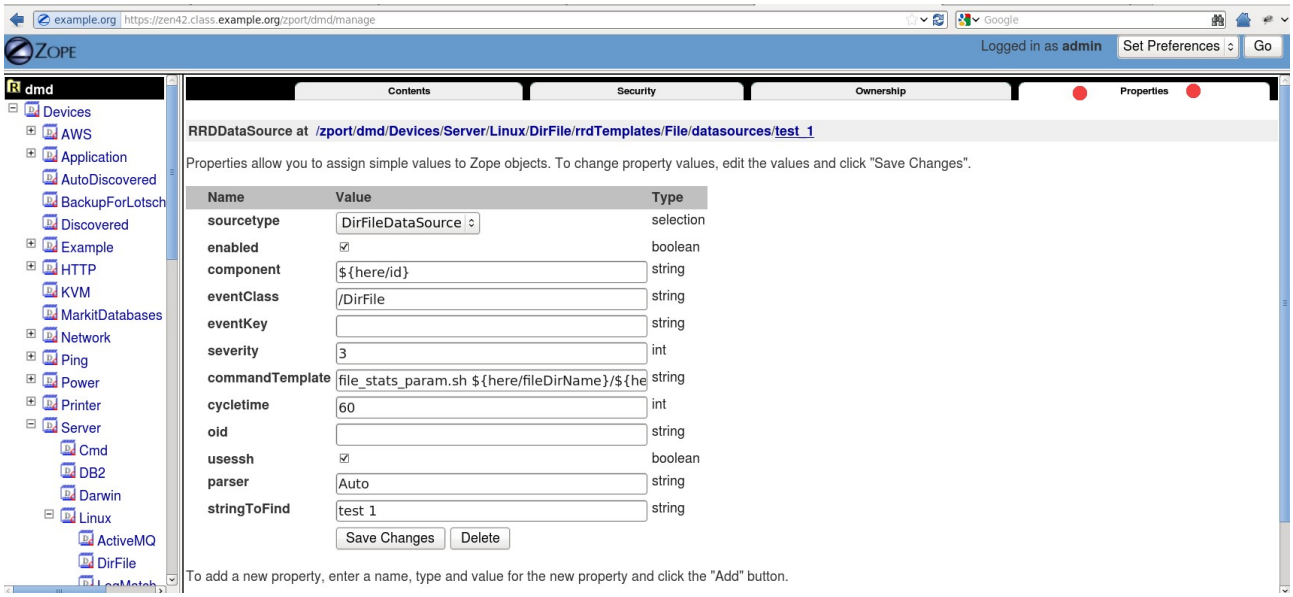


Figure 153: ZMI to check values of datasource instances - note the Properties tab

configure.zcml in the base directory of the ZenPack needs to be created if it doesn't already exist. The following lines configure the new datasource:

```
<?xml version="1.0" encoding="utf-8"?>
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:zcml="http://namespaces.zope.org/zcml">

  <!-- Info Adapters: DataSources

  For ZenPacks that add new datasource types you must register their Info
  adapter(s). The info adapters provide the API that the web interface needs
  to show information about each instance of your datasource type that is
  created. The info adapters are also used to set the properties of the
  datasource instances.
  -->

  <adapter
    provides=".datasources.DirFileDataSource.IDirFileDataSourceInfo"
    for=".datasources.DirFileDataSource.DirFileDataSource"
    factory=".datasources.DirFileDataSource.DirFileDataSourceInfo"
    />

</configure>
```

Note that all the definitions are relative to ***.datasources***; that is, they are in the *datasources* subdirectory of the ZenPack in the *DirFileDataSource* module.

12.3.4.6 Testing the new datasource

This process has created several new classes in a new *datasource* directory and, potentially, created a new *configure.zcml*. The ZenPack should be reinstalled to ensure that all these changes are picked up.

Before doing so, create the */DirFile* event class used in the template, add it to the ZenPack and ensure the ZenPack is exported so that */DirFile* is written to *objects.xml*.

Subsequent “tweaks” of the new classes only need *zenhub* and *zopectl* to be restarted. You may also need to restart *zencommand*, depending on what has changed.

The first sign of “good news” is when creating a new datasource, the *DirFileDataSource* option is available in the dropdown list. If this does not appear, suspect syntax errors. If it does appear, create a new datasource and check that all fields appear with the correct defaults. It is perfectly possible to override any values in the GUI dialogue (provided that the field has been connected with an info definition).

A datapoint called *matches* should be created automatically.

Create graphs to display data for the datasources.

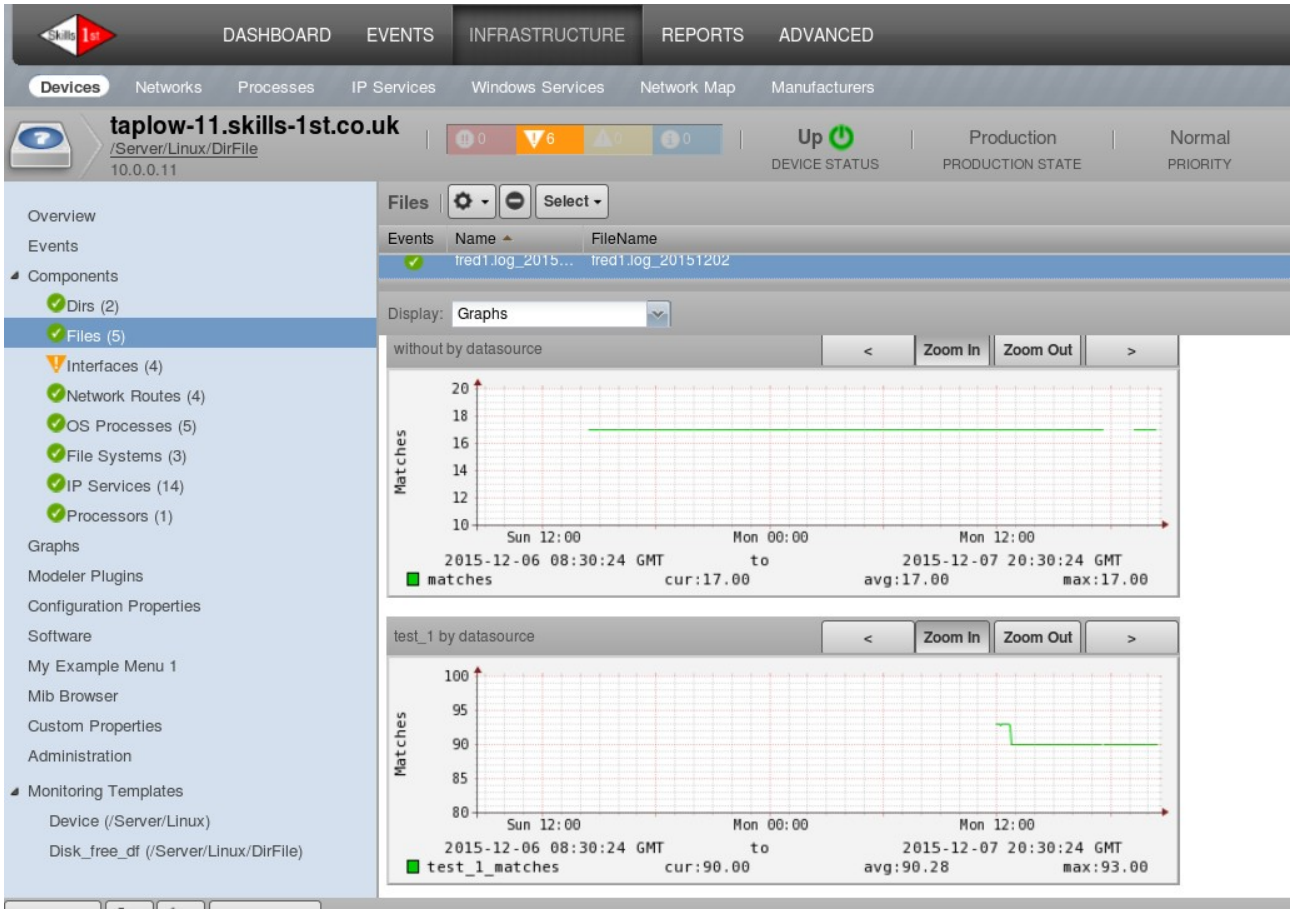


Figure 154: Graphs to count occurrences of "test 1" and "without" using the DirFileDataSource

A slightly modified version of the shellscript, f*ile_stats_param.sh*, will be required which accepts a second parameter as the string to search for and which simply delivers one Nagios-style value.

```
STATE_OK=0
STATE_WARNING=1
STATE_CRITICAL=2
STATE_UNKNOWN=3
#
exitstatus=$STATE_OK

filename="$1"
string1="$2"
# Check that command is valid
stringCount1=$(grep "$string1" $filename 2>/dev/null )
if [ "$?" != 0 ]
then
  echo "Error collecting file stats"
  exit $STATE_WARNING
else
  stringCount1=$(grep "$string1" $filename | wc -l )
fi

echo " File string count test ok | matches=$stringCount1"
exit $exitstatus
```

Note: To duplicate the earlier example, create a *without* datasource and a *test_1* datasource (using underscore to avoid white space); each should automatically get a *matches* datapoint.

Earlier versions of Zenoss used an underscore in datapoint names between the datasource and the datapoint (whereas a dot is now used). This causes a slight glitch if datasource names contain underscore. The fully-qualified names **are, in fact, correct**; datapoints can be selected in the graph dialogue from the **correct** datapoint names; but the name assigned to the datapoint loses everything before the underscore in the datasource name. Thus a datasource *test_1* with datapoint *matches* gets a datapoint called *1_matches* - and this is what appears on the legend of the graph. The workaround is to edit the graph point to change the label back to *test_1_matches*.

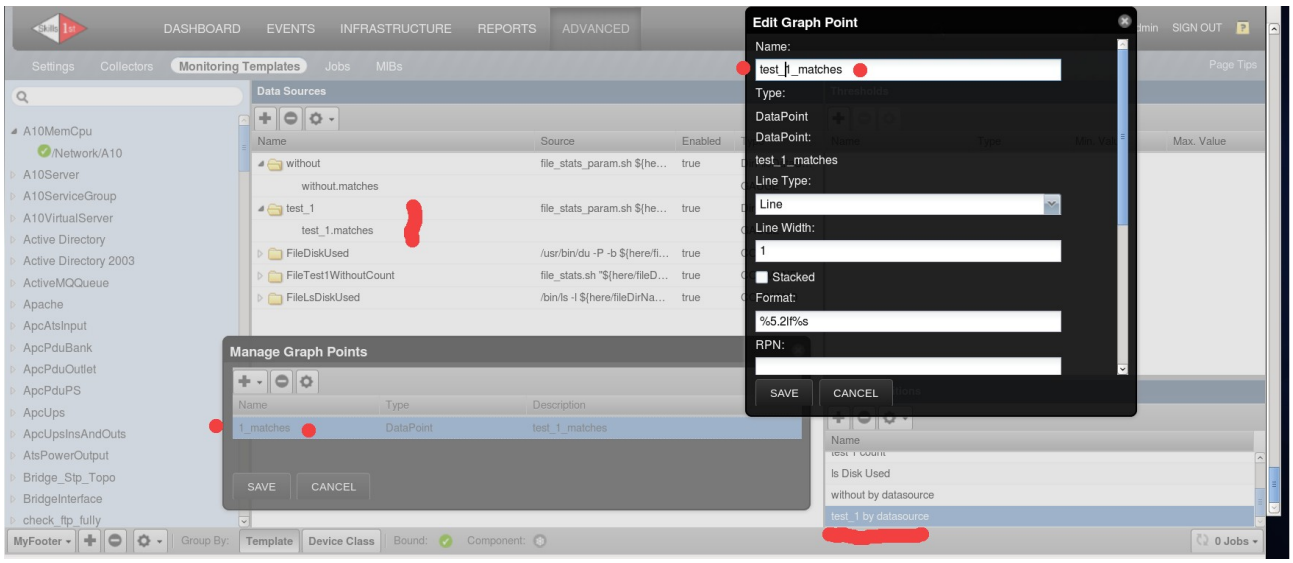


Figure 155: Changing a graph point label if datasource name contains underscore

12.4 Performance templates and zenpacklib

For a discussion of templates and zenpacklib, review section 8.9.

zenpacklib provides a means of exporting templates from an existing ZenPack, provided such templates are associated with a Zenoss device class.

```
zenpacklib.py dump_templates ZenPacks.community.DirFile > DirFileTemps.yaml
```

Templates are output to Unix stdout (ie the screen) so redirect the output to a temporary file, **not** your main *zenpack.yaml*. This output then needs incorporating into the main *zenpack.yaml,* under the appropriate class. If the ZenPack does not include the containing Zenoss device class then only the ZenPack name will be output. It may be necessary to use the GUI to add the device class to the ZenPack temporarily, in order to extract the templates.

Note one issue with exporting templates is that a template description field tends to have single quotes around it; *zenpack.yaml* requires double-quotes, otherwise subsequent lines are all interpreted as comment. The Unix *vi* editor provides automatic color coding for files with a yaml suffix, which helps spot this; red text denotes “quoted”.

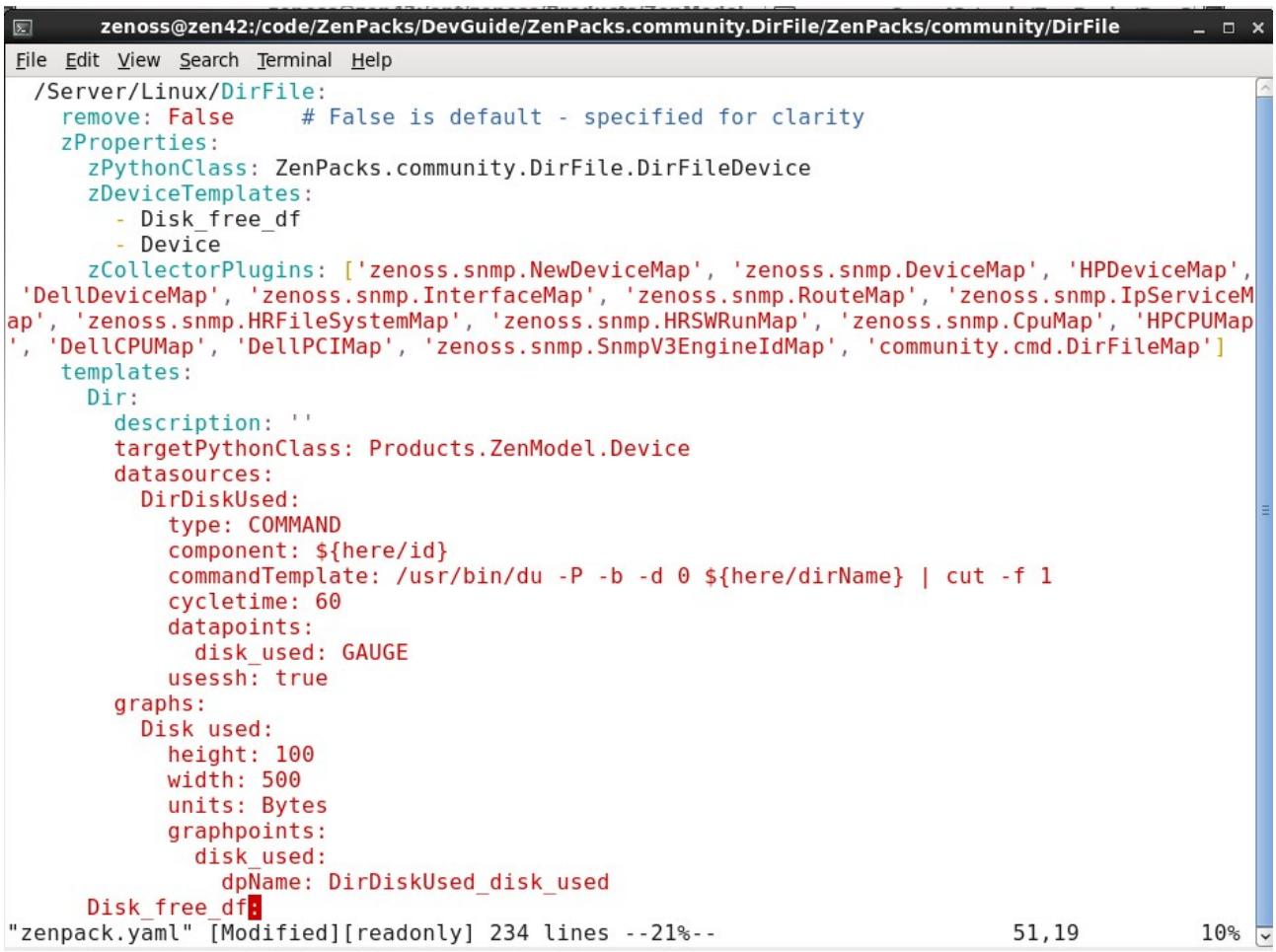


Figure 156: zenpack.yaml with incorporated templates - note red test denoting comments

Changing the two single quotes for the description field to two double quotes, resolves the issue.

Note in Figure 156 that the *zDeviceTemplates* property is specified for the Zenoss device class using the hyphen notation for a list containing *Disk_free_df* and *Device*. The order of the entries in this list will control the order of the attendant graphs. The alternative list notation, using square brackets, can be seen in the following line specifying *zCollectorPlugins*.

When the amalgamated *zenpack.yaml* is complete, the ZenPack should be reinstalled.

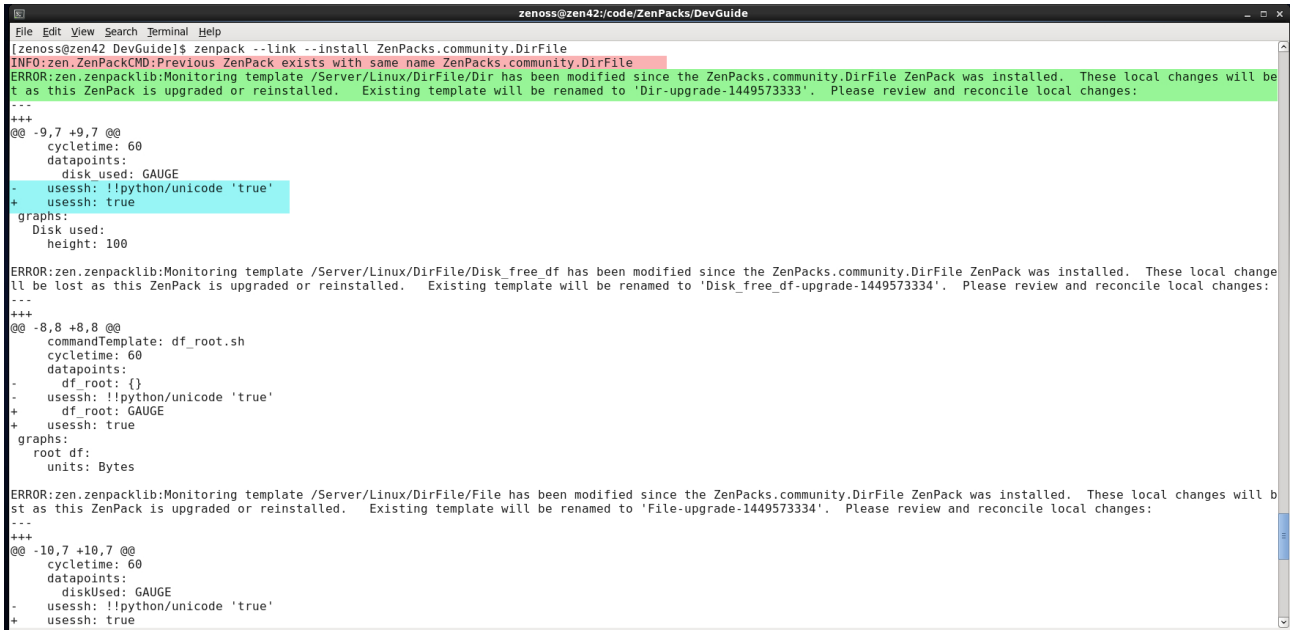


Figure 157: Output from reinstalling the ZenPack after templates changed in zenpack.yaml

If there are templates in *zenpack.yaml* that replace existing templates then an error message is generated (which seems excessive). Differences between the new and existing templates are shown in *diff* format and the old template is saved under a different name. Once the yaml versions of the templates are proven, the old templates, which have a suffix of the epoch time when created, should be deleted manually in the GUI.

If the original templates had been added to the ZenPack's *objects.xml* then these will also reflect the name change and they should be removed from the ZenPack using the GUI interface and the ZenPack re-exported. If the Zenoss device class has been added to *objects.xml* that should also be removed as it is defined in *zenpack.yaml*.

12.4.1 Where do things go wrong?

12.4.1.1 Issues with custom datasources and templates in zenpack.yaml

When using zenpacklib, there are issues when a datasource GUI field includes white space or characters that may need escaping, like $ or %. For example, without zenpacklib, the *Command Template* field for the *without* datasource worked with:

```
file_stats.sh ${here/fileDirName}/${here/fileName}
```

Dumping templates with zenpacklib produced a stanza **without** a *Command Template* at all:

```
without:
  type: DirFileDataSource
  severity: err
  cycletime: 60
  datapoints:
  matches: GAUGE
  stringToFind: without
```

If the datasource is edited through the GUI to put double quotes around the command parameters:

```
file_stats_param.sh "${here/fileDirName}/${here/fileName}"
```

and the dump_templates command is rerun, then the *CommandTemplate* keyword appears in the yaml file, but the datasource fails. *zenhub* fails to parse the string and pass it to zencommand - look in ***$ZENHOME/log/zenhub.log*** for error messages (which you only see with debug logging turned on).

More obvious is the event that appears from the Zenoss server from the *zenhub* agent, with a summary of:

```
TALES error for device taplow-11.skills-1st.co.uk datasource without
```

In the event detail, the resolution field is set to:

```
Could not create a command to send to zencommand because TALES evaluation
failed. The most likely cause is unescaped special characters in the
command. eg $ or %
```

On further investigation. this issue does not seem to be associated with white space or quoting. It would appear that the *CommandTemplate* field for our custom datasource is actually blank (as was suggested by the original *dump_templates*).

To circumnavigate this problem, the datasources and graphs associated with the custom *DirFileDataSource* will be shipped in a separate template, *FileXml*, which is delivered through the ZenPack's *objects.xml*. These definitions and graphs will be removed from *zenpack.yaml* which will have the object class for *File* modified:

```
monitoring_templates: [File, FileXml]
```

13.0 Converting COMMAND ZenPacks to PythonCollector

The DirFile ZenPack uses a *CommandPlugin* for modeling and *COMMAND* datasources to gather performance information - as do very many existing ZenPacks.

There are many drawbacks to COMMAND-driven functionality:

- Inherent overhead in running COMMAND datasources under bash which spawns a new subprocess to run each command
- Many problems with quoting and escaping characters in bash
- *CommandPlugin* modeler does not (by default) allow zProperties to be passed as part of the command, so very inefficient
- Huge overhead if running COMMAND datasources to many components

The positive aspect of using ssh commands is that it is easy and relatively well understood. This still doesn't necessarily make it a good idea!

This chapter will explore converting first the COMMAND datasources and then the modeler plugin of the DirFile ZenPack to use the **zenpython** daemon that is provided with *ZenPacks.zenoss.PythonCollector*.

13.1 ZenPacks.zenoss.PythonCollector

ZenPacks.zenoss.PythonCollector is a freely-available ZenPack written and supported by Zenoss. It was first released mid-2013 and is compatible with:

- Zenoss Core 4.2.x
- Zenoss Resource Manager 4.1.x
- Zenoss Resource Manager 4.2.x
- Zenoss Core 5.0.x
- Zenoss Resource Manager 5.0.x

The latest version (December 2015) is 1.7.3.

Be aware that this is a ZenPack still being actively developed and ensure that all the latest updates information is read whenever a new version is installed. For example, 1.7.2 / 3 have introduced two new parameters which can cause major disruption to existing environments:

- **blockingwarning**
  - The *zenpython* collector daemon executes plugin code provided by other ZenPacks. If this plugin code blocks for too long it will prevent *zenpython* from performing other tasks including collecting other datasources while the plugin code is executed. The *blockingwarning* option will cause *zenpython* to log a warning for any plugin code that blocks for the configured number of seconds or more. The default value is 3 seconds (30s in 1.7.2). Decimal precision such as 3.5 can be used.
- **blockingtimeout**
  - This option will cause zenpython to **disable** a plugin if it blocks for longer than the number of seconds specified. The *zenpython* daemon will restart itself after disabling the plugin to get unblocked. Events will be created indicating that some monitoring will not be performed due to disabled plugins on all affected devices. The default value is 5 seconds. Decimal precision such as 5.5 can be used. Setting this to 0 will disable the blocking functionality, reverting the behaviour back to pre 1.7.2 version.
  - Once a plugin is blocked, it will remain **permanently blocked** until its name is removed from either ***/var/zenoss/zenpython.blocked*** on Zenoss 5, or ***/opt/zenoss/var/zenpython.blocked*** on Zenoss 4. The *zenpython* service must be restarted after manual modifications to this file.

Slightly older additional parameters for the *zenpython* daemon are:

- twistedthreadpoolsize
  - Controls size of threads pool. Datasources can use multi-threading to run multiple requests in parallel. Increasing this value may boost performance at the cost of system memory used. The default value is 10.
- collect
  - Allows only specific plugins to run. This is primarily a developer option to help reduce the noise while developing plugins. The default is to collect all configured

plugins. The value for this option is a regular expression. Only plugins whose class name matches the regular expression will be run.

- ignore
  - Prevents specific plugins from running. This is primarily a developer option to help reduce the noise while developing plugins. The default is to not ignore any configured plugins. The value for this option is a regular expression. Only plugins whose class name doesn't match the regular expression will be run.

These options can be configured either from the Zenoss GUI from *ADVANCED -> Settings -> Daemons* (or *ADVANCED -> Settings -> Collector -> Daemons* if remote Collectors are deployed); alternatively, edit *zenpython.conf* in *$ZENHOME/etc* and then restart the *zenpython* daemon.

The PythonCollector ZenPack is a prerequisite for many other ZenPacks, both from Zenoss and developed by other people. It is recommended as an effective enabling technology and is much preferable to using command-style monitoring and modeling.

13.1.1 Using the PythonCollector ZenPack

Unless the PythonCollector ZenPack is installed solely as a prerequisite for other ZenPacks, writing code is inevitable. The wiki documentation at http://wiki.zenoss.org/ZenPack:PythonCollector provides some terse information.

> "The goal of the Python data source type is to replicate some of the standard COMMAND data source type's functionality without requiring a new shell and shell subprocess to be spawned each time the data source is collected. The COMMAND data source type is infinitely flexible, but because of the shell and subprocess spawning, it's performance and ability to pass data into the collection script are limited. The Python data source type circumvents the need to spawn subprocesses by forcing the collection code to be asynchronous using the Twisted library. It circumvents the problem with passing data into the collection logic by being able to pass any basic Python data type without the need to worry about shell escaping issues.
>
> The Python data source type is intended to be used in one of two ways. The first way is directly through the creation of Python data sources through the web interface or in a ZenPack. When used in this way, it is the responsibility of the data source creator to implement the required Python class specified in the data source's Python Class Name property field. The second way the Python data source can be used is as a base class for another data source type. Used in this way, the ZenPack author will create a subclass of PythonDataSource to provide a higher-level functionality to the user. The user is then not responsible for writing a Python class to collect and process data. "

Fundamentally, to collect performance data with the *zenpython* daemon, a **PythonDataSourcePlugin** must be written in Python. This is the "first way" described above and later on the wiki page as "Using the Python data source directly". A *PythonDataSourcePlugin* must be in a file called ***dsplugins.py*** in the base directory of the ZenPack. If a ZenPack contains several such plugins, then a ***dsplugins*** subdirectory can be created.

There are a number of ZenPacks around that demonstrate these techniques:

- zenoss/ZenPacks.zenoss.AWS

- ZenPacks.zenoss.Hadoop
- zenoss/ZenPacks.zenoss.HBase (with dsplugins directory)
- ZenPacks.community.zplib.twemproxy
- ZenPacks.community.zplib.Redis

Once the plugin is written, it can be used when defining a Python-type datasource.

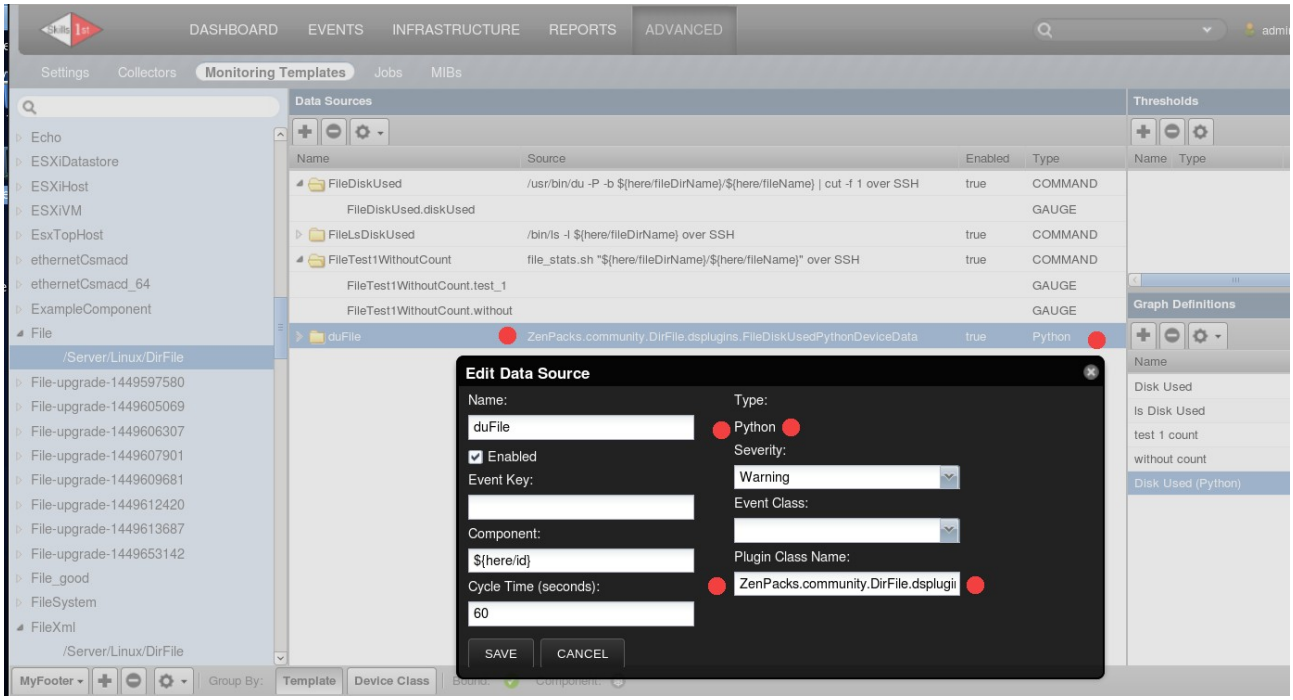


Figure 158: Defining a Python datasource, including a custom plugin

Unfortunately the Zenoss GUI does not provide a dropdown box to select the plugin so this has to be accurately typed.

```
ZenPacks.community.DirFile.dsplugins.FileDiskUsedPythonDeviceData
```

As can be seen in Figure 158, there is not a wealth of other options to specify for the Python datasource. If more configuration is required, then the second option described on the wiki page is used to create a complete datasource, as seen in the previous chapter; the datasource must **include** a *PythonDataSourcePlugin*.

The *ZenPacks.zenoss.PythonCollector* ZenPack provides three main elements:

- ***zenpython* daemon** in the base directory of the ZenPack
- ***PythonDataSource* datasource** in the *datasources* subdirectory
- ***PythonDataSourceConfig*** in the *services* directory

13.1.2 * Anatomy of a PythonDataSourcePlugin

The *PythonDataSourcePlugin*, like **all** datasource plugins, has some methods that run at the **collector** *zenpython* daemon (or *zencommand* daemon for a COMMAND datasource); other methods are actually run centrally by **zenhub**.

Fundamentally, collector daemons do *not* have access to the Zope database (ZODB). If a collector daemon needs access to attributes or methods of a device instance or component instance, they have to be gathered by *zenhub* and passed to the collector daemon in a configuration phase. This happens when a collector daemon starts up.

If these attributes change at some stage in the ZODB, then all relevant datasource configurations for that device will be re-examined and the new configuration will be pushed to the relevant collector daemons. This now happens automatically when any relevant attribute in the ZODB is changed; in earlier versions of Zenoss, the *Push Changes* menu from a device's *Action* icon was used to perform this reconfiguration on-demand.

Examination of *PythonDataSource.py* in the *datasources* directory of *ZenPacks.zenoss.PythonCollector* shows the standard attributes of a *PythonDataSource* - which inherits from the *RRDDataSource* class. Note that the cycletime attribute is changed in the *PythonDataSource* to be a **string** rather than an integer.

This file also includes the interface and info classes for the *PythonDataSource*, including the new ***plugin_classname*** field and the redefined *cycletime*. The *testable* attribute is set to *False*.

The ***PythonDataSourcePlugin*** class inherits from the very basic *object* class. It has one attribute and a number of methods.

- **proxy_attributes** = () attribute
  - This is a way to pass specific zProperties of a device from the ZODB to the *zenpython* daemon.
- **config_key(cls, datasource, context):** method
  - Run at **zenhub** to determine what config elements are grouped together into a single config object; all config objects are then sent to the zenpython daemon The default is:

```
 return (
context.device().id,
datasource.getCycleTime(context),
datasource.rrdTemplate().id,
datasource.id,
datasource.plugin_classname,
)
```

  - Each config received by the daemon will be evaluated and split into tasks based on the configured TaskSplitter. At a minimum there will be one task per config, but it is possible to have more.
  - Judicious choice of these parameters can sometimes result in a huge reduction of processing effort. For example, if one command actually elicits information for all component instances on a device, then the command should only be run once and the combined returned data should then be sorted to the correct components.

- The ***context*** parameter is the object that the template is applied to - either a device or a component.

- **params(cls, datasource, context):** method
  - Run at **zenhub** to gather data about the *context* (device or component) from the ZODB.
  - Normally this would not be used to gather zProperties; they would be passed in the *proxy_attributes* attribute.
  - For example, in the DirFile ZenPack, a *File* object class has attributes for *fileName* and *fileDirName*. These could be retrieved from the ZODB by the *params* method and passed to the *zenpython* daemon.
  - Note that a Python dictionary is returned.
  - The default *params* method returns an empty dictionary.

- The methods that are actually executed by the *zenpython* daemon are all effectively null functions in the base *PythonDataSourcePlugin* class. Note that the ***collect*** method **must** be implemented in any class that inherits from *PythonDataSourcePlugin*. Typically some of the other methods will also be overridden in any subclass.

```
def collect(self, config):
   """No default collect behaviour. Must be implemented in subclass."""
   return NotImplementedError

 def onResult(self, result, config):
   """Called first for success and error."""
   return result

 def onSuccess(self, result, config):
   """Called only on success. After onResult, before onComplete."""
   return result

 def onError(self, result, config):
   """Called only on error. After onResult, before onComplete."""
   return result

 def onComplete(self, result, config):
   """Called last for success and error."""
   return result

 def cleanup(self, config):
   """Called when collector exits, or task is deleted or recreated."""
   return
```

  - The comments in the code are fairly self explanatory as to when the methods are run.
  - Typically a subclass will at least implement its own *collect, onSuccess* and *onError* methods.

- The ***new_data*** utility method is generally called by any subclass, either in *collect* or *onSuccess* to create a new, empty data structure for returning results.

```
  def new_data(self):
   """
   Return an empty data structure.
```

```
Suitable for returning from on* methods.

This data structure should emulate the source format defined in
Products.ZenRRD.parsers.JSON.
"""
return {
  'values': defaultdict(dict),
  'events': [],
  'maps': [],
  }
```

- Fundamentally, a *PythonDataSourcePlugin* must deliver data results and/or events in a *new_data* data structure.

13.2 Twisted

This section is not marked as “hard” with an asterisk - but it is; however, anyone who is going to implement code using the PythonCollector ZenPack must get at least some understanding of the concepts of ***Twisted*** libraries.

Twisted is an event-based framework for internet applications, written in Python. Twisted projects variously support TCP, UDP, SSL/TLS, IP multicast, Unix domain sockets and others. Protocols supported include HTTP, XMPP, NNTP, IMAP, SNMP, ssh, IRC and ftp. There are also APIs into VMware VSphere and Amazon AWS among others. Twisted is based on the event-driven programming paradigm, which means that users of Twisted write **callbacks** which are called by the Twisted framework.

Relating this to the *PythonDataSourcePlugin* discussed earlier, the *zenpython* daemon is the “framework” and the “on” methods are the callbacks. The collect method implements twisted code that should return a ***Twisted Deferred***.

The basic idea is that when an application is asking for data, typically over some communications medium, the code makes the request and then has to wait until the result is delivered. If this is a request over a piece of “wet-string”, low bandwidth network, then the program may be delayed significantly waiting for the response.

A *Twisted Deferred* is a way of decoupling the request for data from its response. The program may continue with other processing or other requests. A D*eferred* is an object with a promise to call back with a result later. The deferred is returned immediately but think of it as a “place holder”. When the result is delivered, it triggers the deferred, which then calls you back at a function that you have specified. If an error occurred, an error handling function that you specified is called back; thus we have the concept of **callbacks** and **errbacks**. *onSuccess* is a callback; *onError* is an errback. Look at the *doTask* method of the *zenpython.py* code in the PythonCollector ZenPack base directory to see how these are implemented.

One of the really difficult issues with Twisted, is debugging. Putting log or print statements into code often results in errors or no data, because all there is at that stage is the D*eferred*, the “place holder”. There is no data.

There are also a number of different ways of creating twisted methods. Some Python modules and APIs inherently deliver twisted results (eg. pynetsnmp.twistedsnmp); if this is **not** the case the most frequently used technique in Zenoss appears to be to use a Python **decorator**,

@inlineCallbacks, to preface the collect method. These are demonstrated in the Python datasource examples in the zenpacklib documentation at http://zenpacklib.zenoss.com/en/latest/tutorial-http-api/datasource-plugin-datapoints.html .

Note that if an inherently twisted delivery method is used, then the *@inlinecallbacks* should **not** also be used.

An outline of a collect method might be:

```
from twisted.internet.defer import inlineCallbacks, returnValue
     .....

  @inlineCallbacks
  def collect(self, config):
  ......
  gather data from the datasource for use in the collection code
  ......
  for each datasource:
    try:
      response = yield( the routine that actually gathers data)
      do further processing on response
    except:
      log the exception
    returnValue(the processed response)
```

Points to note:

1. The *inlinecallbacks* decorator and *returnValue* are imports from *twisted*.
2. You must use *returnValue()*, (note the value is in brackets), not the usual *return*
3. The decorator must be on the line immediately preceding the collect function definition.
4. Note that the *yield* in the above code makes a Python **generator** out of a function. A generator is quite different from a function in that it does not return one value, but several values over time. In this case, it returns the deferred data as it arrives. The collect function, decorated with *inlinecallbacks*, expects a generator and calls it back as often as necessary and waits for each deferred it receives.
5. The *response = yield* code block should always be in a Python *try .. except* construct to ensure that errors are allowed to generate an errback.

A couple of useful links for Twisted and yield discussions are https://confluence.oceanobservatories.org/display/CIDev/Gotchas+with+inlineCallbacks,+yield+and+returnValue and http://stackoverflow.com/questions/3894278/twisted-deferred-addcallback-vs-yield-and-inlinedeferred .

13.3 Creating Python datasources

The DirFile ZenPack already exists with modeler plugin and COMMAND datasources. This provides an easier environment then normal as the existing COMMAND-based modeler plugin has already discovered *File* and *Dir* components.

The big difference with using the *zenpython* daemon rather than *zencommand*, is that it does not have built-in utilities for running over the ssh protocol. It is the code in *$ZENHOME/Products/DataCollector/SshClient.py* that actually performs the ssh

communication for COMMAND-based modelers and plugins; examination of that code shows that it uses the ***twisted.conch*** libraries.

In real practice, there are often better ways of gaining data from remote devices than using ssh, with the help of other Twisted libraries or APIs; however, for these samples, to keep the target functionality very simple, the *PythonDataSourcePlugin* will run scripts **local to the zenpython daemon**, which in turn, run ssh commands. The communication will be rather less flexible and forgiving than under *zencommand*, but will still benefit from using existing zProperties such as *zCommandUsername*.

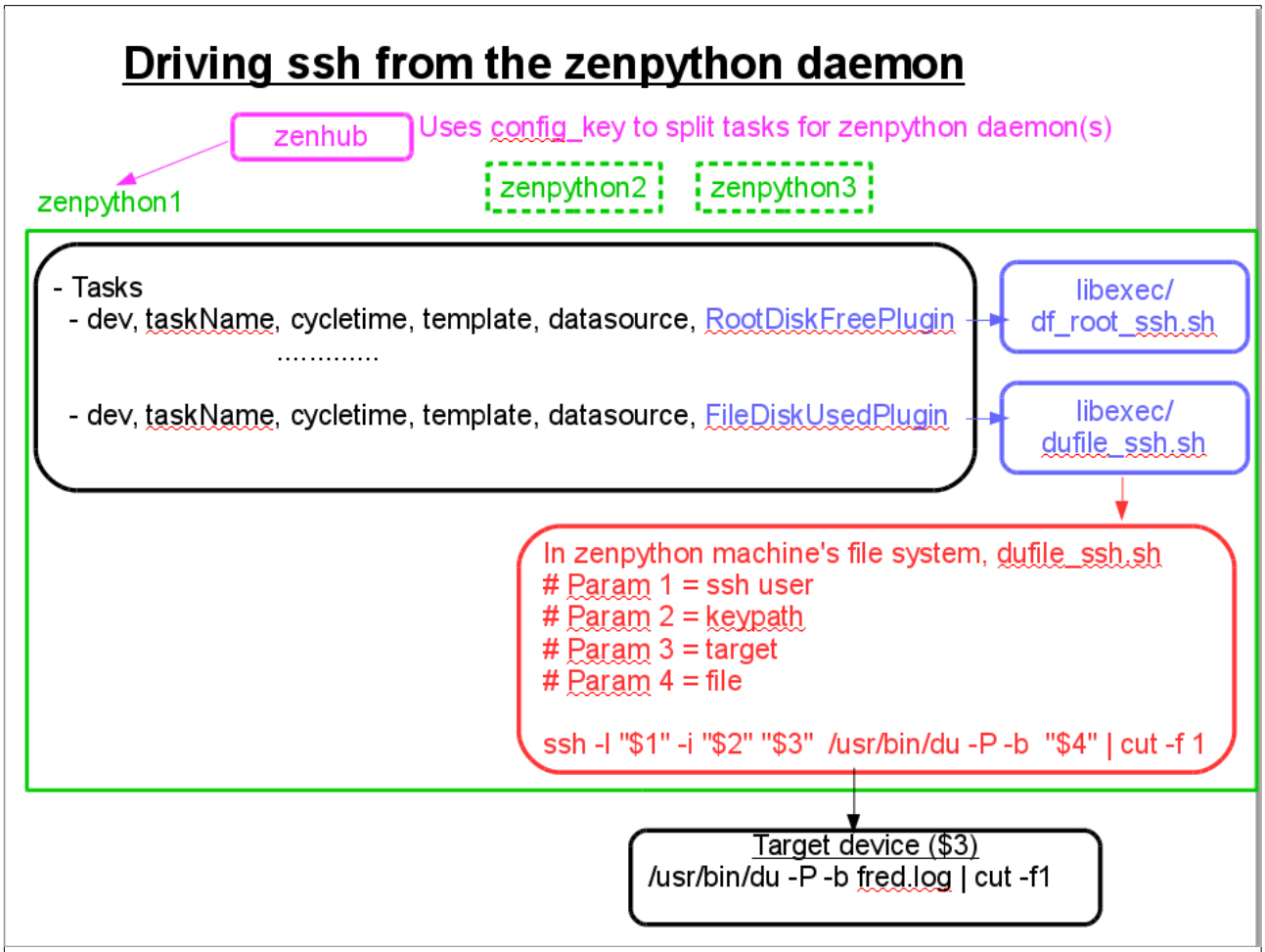


Figure 159: Driving ssh from a zenpython daemon to get disk usage for a file

Figure 159 demonstrates that:

- *zenhub* examines **all** configurations for **all** zenpython datasources for **all** devices and components in the ZODB, and works out how many different configs are are necessary to service those requirements.
- The configs are then handed to the relevant *zenpython* daemon(s) which works out how many tasks are required. It is perfectly possible with both Zenoss Service Dynamics and Zenoss Core 4 to have multiple collectors on different systems, each running a set of collection daemons, including *zenpython*.

- Each *zenpython* examines its task list and schedules tasks, depending on the cycletime of the datasources.
- Each task will call a *PythonDataSourcePlugin*. For these examples, the plugin calls a shellscript **local** to this *zenpython*. The file location is hardcoded in the plugin to be relative to the ZenPack's plugin directory; in practice this is in the *libexec* directory of the ZenPack.
- The plugin in the example passes *zCommandUsername* and *zKeyPath* to the script, along with the device IP address and the filename for which to get *du* information.
- The script, *<ZenPack base dir>/libexec/dufile_ssh.sh* is run by the *collect* method of the plugin, in asynchronous fashion, courtesy of Twisted.

```
# Param 1 = ssh user
# Param 2 = keypath
# Param 3 = target
# Param 4 = file

ssh -l "$1" -i "$2" "$3" /usr/bin/du -P -b "$4" | cut -f 1
```

- The target device runs the *du* command over the ssh session, returning data to the plugin via Unix stdout.

This solution loses the control offered by the *zencommand* daemon to take advantage of other zProperties such as zCommandPath, zCommandLoginTimeout, zCommandCommandTimeout and zCommandPort, although they could be implemented into a more robust solution with more effort.

13.3.1 Collecting *device* performance data

The first section in Chapter 12 was a simple example to collect the amount of free disk space on the root filesystem of a device, using the script ***df_root.sh***:

```
#!/bin/bash
#
# Use df to get disk free. Get result in bytes (-B 1) and use Posix flag
for compatibility.
# Check 6th whitespace separated field for /
#  output 3rd field (Used in Bytes) and make sure no duplicate lines

df -P -B 1 | awk -F " " '$6~/^\/$/ {print $3}' | uniq
```

To run this command with *zenpython* implementing the ssh, the script, **df_root_ssh.sh**, is modified thus:

```
#!/bin/bash
#
# Use df to get disk free.
# Param 1 = ssh user
# Param 2 = keypath
# Param 3 = target

#Get result in bytes (-B 1) and use Posix flag for compatibility.
# Check 6th whitespace separated field for /
#  output 3rd field (Used in Bytes) and make sure no duplicate lines
```

```
ssh -l "$1" -i "$2" "$3" df -P -B 1 | awk -F " " '$6~/^\/$/ {print $3}' | uniq
```

The *PythonDataSourcePlugin* will be implemented in the *dsplugins.py* file in the base directory of the ZenPack.

13.3.1.1 Imports for the PythonDataSourcePlugin

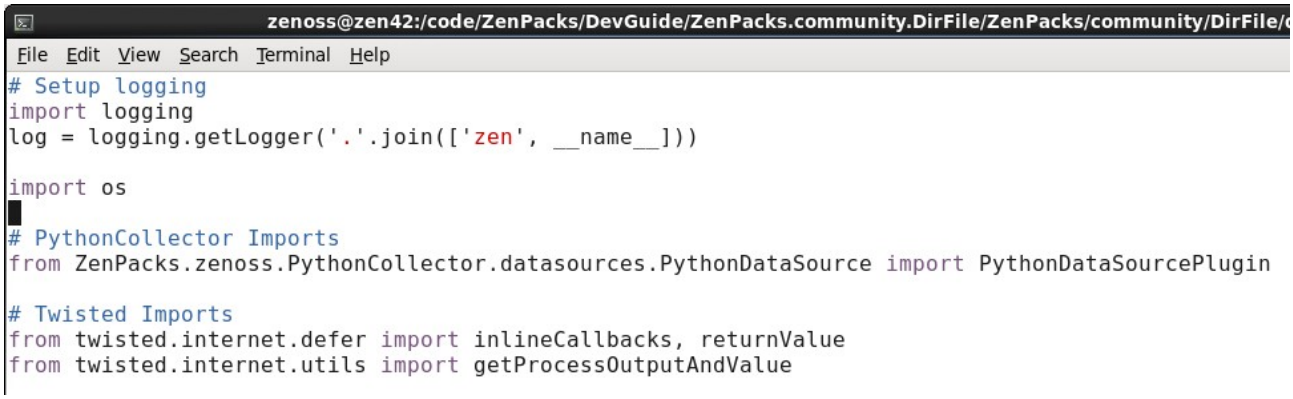


Figure 160: Imports for dsplugins.py

Figure 160 shows the logging being setup and then imports for:

- os — to get directory information
- PythonDataSourcePlugin — base class from PythonCollector ZenPack
- inlinecallbacks and returnValue — from twisted
- getProcessOutputAndValue — runs a command, returning a ***Twisted Deferred***

13.3.1.2 proxy_attributes and config_key method for the PythonDataSourcePlugin

In the sample test environment, public keys have already been setup between Zenoss and targets so the only essential zProperties required are *zCommandUsername* and *zKeyPath*.

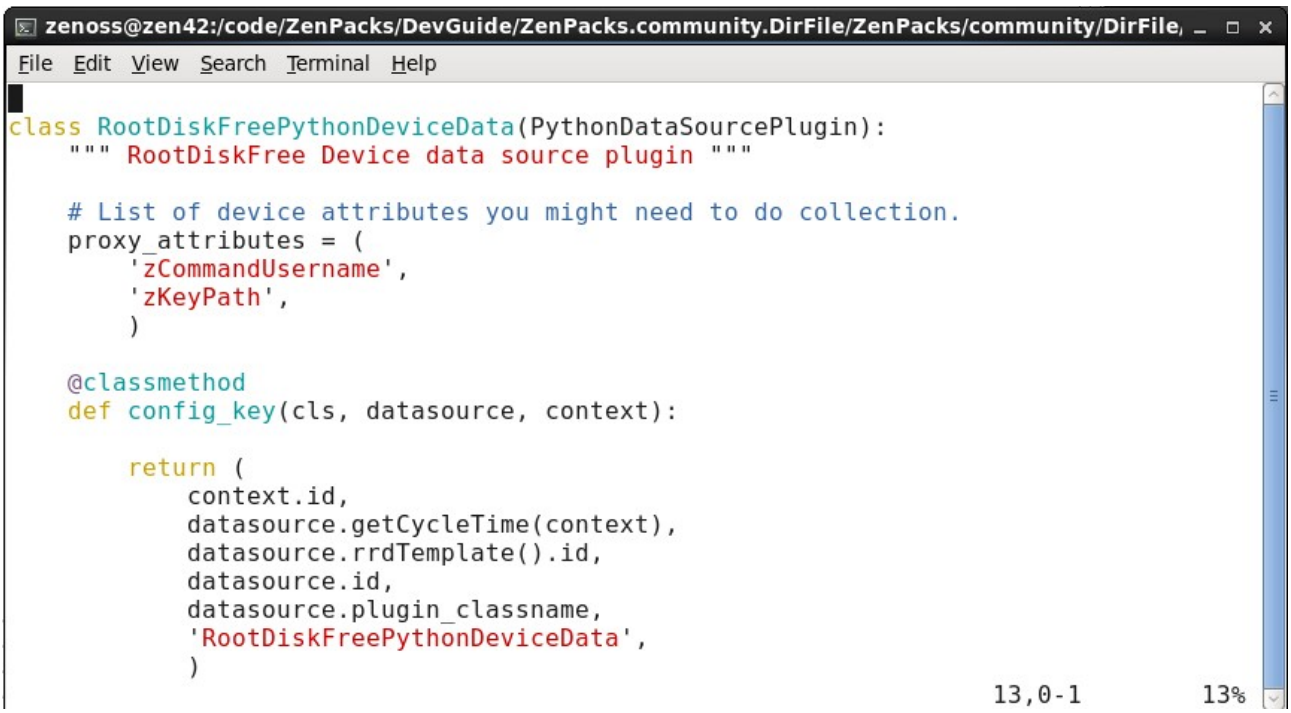


Figure 161: proxy_attributes and config_key method

The *config_key* method, to be run by *zenhub*, creates separate configs based on:

- context.id - this is a device template so context.id will be hostname
- datasource.getCycleTime(context) - the cycletime for this device for this datasource
- datasource.rrdTemplate().id - the performance template name
- datasource.id - the datasource name
- datasource.plugin_classname - the datasource plugin class name
- 'RootDiskFreePythonDeviceData' - this is simply a “good-practice” string to help identification in log files

Ultimately this plugin will be driven by a device-level performance template.

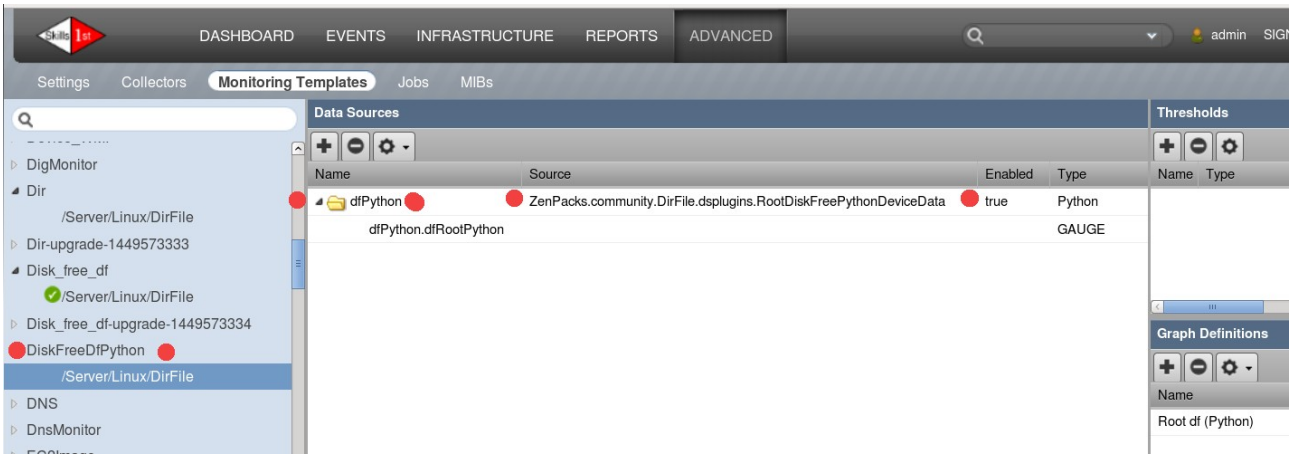


Figure 162: DiskFreeDfPython performance template

Note in Figure 162 that:

- The **template** is *DiskFreeDfPython*
- The **datasource** is *dfPython*
- **Plugin** is *ZenPacks.community.DirFile.dsplugins.RootDiskFreePythonDeviceDat*a

To see these parameters in use, run *zenpython* in debug mode and examine the output:

```
zenpython run -v 10 -d taplow-11.skills-1st.co.uk > /tmp/py 2>&1
```

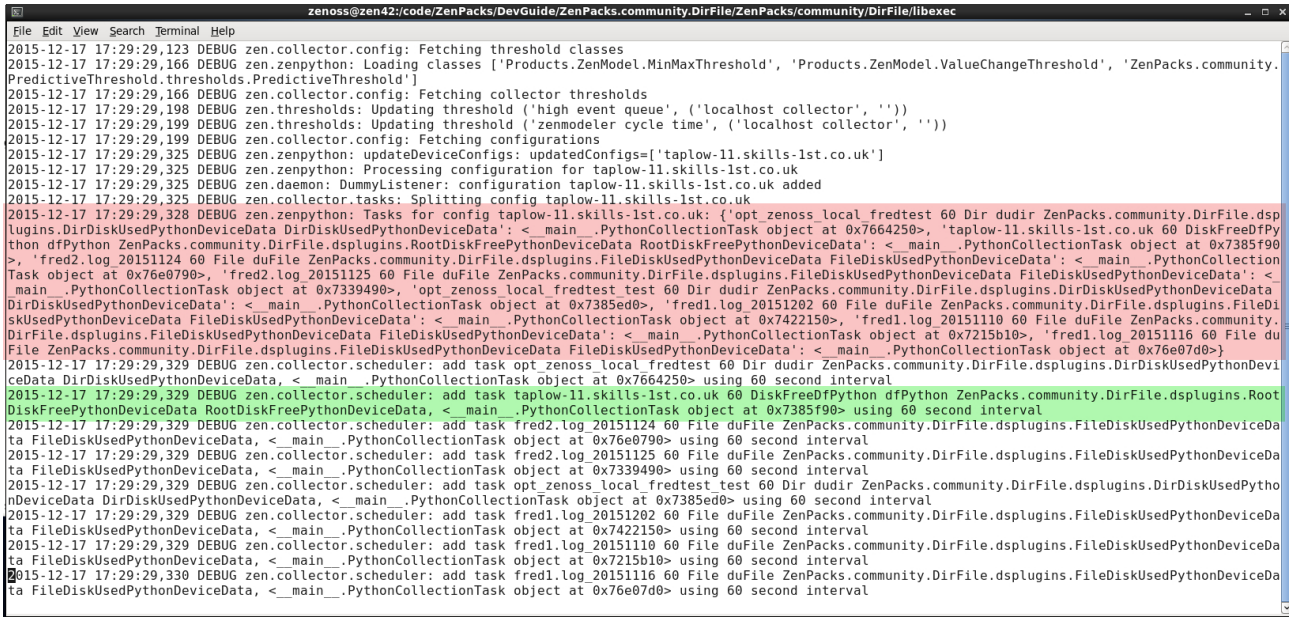


Figure 163: Debug log for zenpython showing all tasks for this device and the RootDiskFree task

In Figure 163 the first section highlighted in red is all the tasks to be run by this *zenpython* for the device taplow-11.skills-1st.co.uk.

A separate *add task* is then logged for each task, showing the *config_key* parameters that have been used (second highlighted section in green).

> add task taplow-11.skills-1st.co.uk **60 DiskFreeDfPython dfPython ZenPacks.community.DirFile.dsplugins.RootDiskFreePythonDeviceData RootDiskFreePythonDeviceData**

where:

- 60 cycletime in the datasource
- DiskFreeDfPython template name
- dfPython datasource name
- **ZenPacks.community.DirFile.dsplugins.RootDiskFreePythonDeviceData**
  - plugin
- **RootDiskFreePythonDeviceData** comment string

13.3.1.3 collect method for the PythonDataSourcePlugin

File Edit View Search Terminal Help

```
    @inlineCallbacks
    def collect(self, config):

        log.debug('config is %s ' % (config))
        log.debug('config.datasources is %s ' % (config.datasources))
        for c in config.datasources:
            log.debug('config.datasource element is %s ' % (c))
        ds0 = config.datasources[0]
        for k,v in ds0.__dict__.items():
            log.debug('ds0 k is %s and v is %s ' % (k,v))

        # Get path to executable file, starting from this file
        #    which is in ZenPack base dir/dsplugins
        #    Executables are in ZenPack base dir / libexec
        # NOTE: If using dsplugins directory, then need to go up one to get to libexec
        thisabspath = os.path.dirname(os.path.abspath(__file__))
        libexecdir = thisabspath + '/../libexec'
        # If ds0.zKeyPath starts with ~ then it doesn't expand properly so convert to full path
        #  expanduser gives $HOME including trailing /
        homedir = os.path.expanduser("~")
        if ds0.zKeyPath.startswith('~'):
            keyPath = ds0.zKeyPath.replace('~', homedir)
        else:
            keyPath = ds0.zKeyPath
        # script is df_root_ssh.shh taking 3 parameters, zCommandUsername, keyPath, host address
        cmd = os.path.join(libexecdir, 'df_root_ssh.sh')
        #cmd = os.path.join(libexecdir, 'df_root_ssh_bad.sh')
        args = ( ds0.zCommandUsername, keyPath, ds0.manageIp)
        log.debug(' cmd is %s \n ' % (cmd) )

        try:
            cmd_stdout = yield getProcessOutputAndValue(cmd, args = args )
            log.debug( 'RootDiskFree collect. stdout is %s and stderr is %s and exit code is %s ' % (cmd_stdout[0], cmd_stdout[1], cmd_stdout[2]))
        except Exception:
            log.exception('Error in collect gathering RootDiskFree info - %s ' % (Exception))
        returnValue(cmd_stdout)
```

Figure 164: collect method

The ***collect*** method in Figure 164 starts with the Python *@inlineCallbacks* decorator. It is passed the datasource configuration as parameter.

The *config* parameter (which is generally the hostname of the device) has a datasources element which is a list of ***PythonDataSourceConfig***, the definition for which can be found in the PythonCollector ZenPack in *services/PythonConfig.py*.

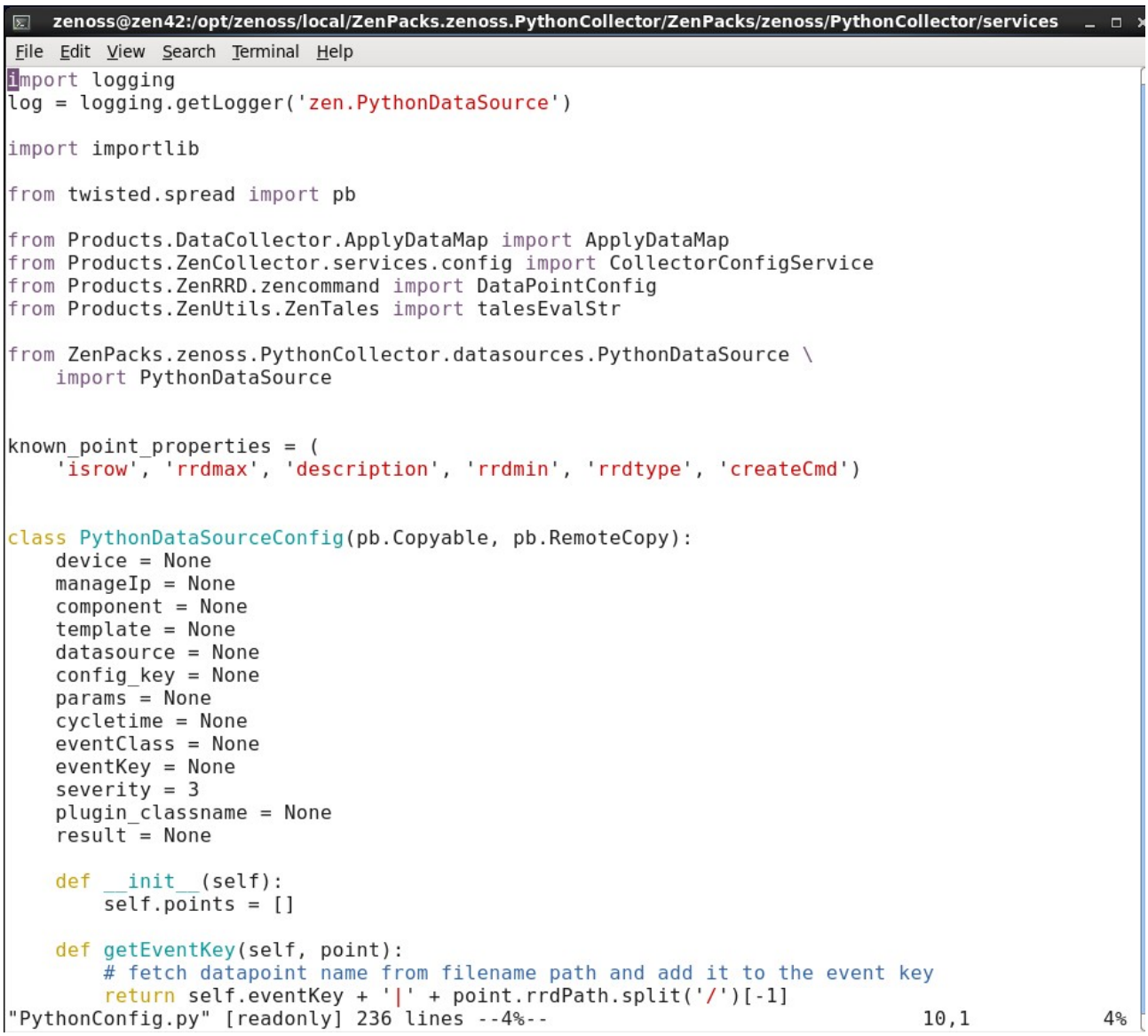


Figure 165: PythonDataSourceConfig class definition in PythonCollector ZenPack

In addition to the attributes seen in Figure 165, any proxy_*attributes* that are declared are also part of the class attributes.

To demonstrate this, extra *log.debug* statements can be used around the start of the *collect* method:

```
def collect(self, config):

  log.debug('config is %s ' % (config))
  log.debug('config.datasources is %s ' % (config.datasources))
  for c in config.datasources:
    log.debug('config.datasource element is %s ' % (c))
  ds0 = config.datasources[0]
  for k,v in ds0.__dict__.items():
    log.debug('ds0 k is %s and v is %s ' % (k,v))
```

This results in entries in a debug log like Figure 166

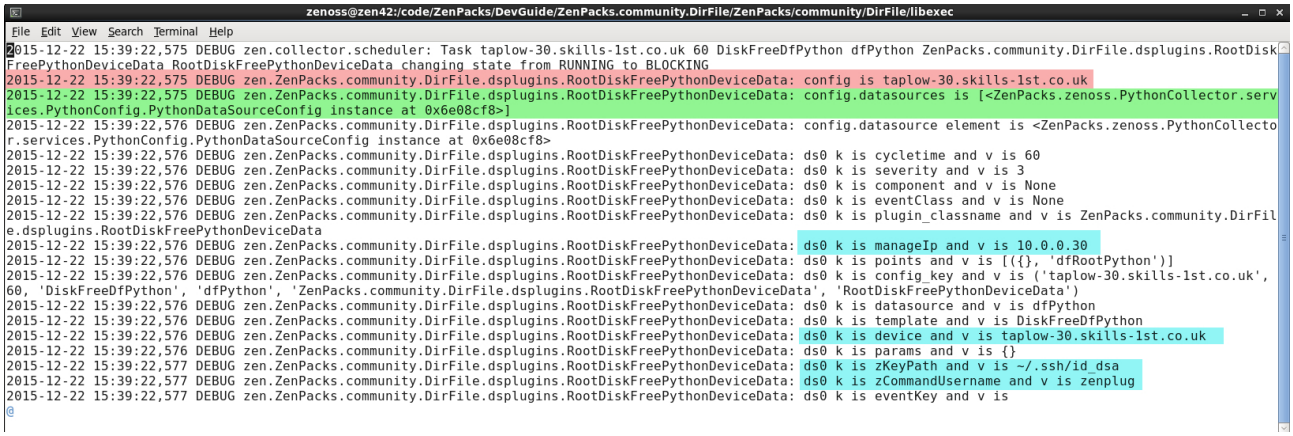


Figure 166: Debug log for zenpython showing keys and values for a PythonDataSourceConfig

Note that the *manageIp* attribute of the device is available as part of a *PythonDataSourceConfig*. Those zProperties that are declared as *proxy_attributes* are also accessible, as is the *device* parameter (which is set to the device *id*).

The datasource is accessed in the line:

```
ds0 = config.datasources[0]
```

The next few lines are concerned with finding the directory holding the script that *zenpython* will execute:

```
# Get path to executable file, starting from this file
#  which is in ZenPack base
#  Executables are in ZenPack base dir / libexec
thisabspath = os.path.dirname(os.path.abspath(__file__))
libexecdir = thisabspath + '/libexec'
```

The standard Python *os* module is used to ascertain the "current" directory and then append */libexec*.

If using the *zKeyPath* zProperty, typically it has a value of *~/.ssh/id_dsa* where "~" gets translated to the home directory of the Zenoss user. When passed as a parameter, the "~" needs expanding to be the full path or the ssh command fails:

```
# If ds0.zKeyPath starts with ~ then it doesn't expand properly so convert
#    to full path
# expanduser gives $HOME including trailing /
homedir = os.path.expanduser("~")
if ds0.zKeyPath.startswith('~'):
  keyPath = ds0.zKeyPath.replace('~', homedir)
else:
  keyPath = ds0.zKeyPath
```

The next stage uses the *os* module to construct the complete command to be run, with parameters:

```
# script is df_root_ssh.shh taking 3 parameters, zCommandUsername, keyPath, host address
cmd = os.path.join(libexecdir, 'df_root_ssh.sh')
args = ( ds0.zCommandUsername, keyPath, ds0.manageIp)
```

The script name, *df_root_ssh.sh* is hardcoded here. The *zCommandUsername* and *manageIp* attributes from the *PythonDataSourceConfig ds0* element are passed as parameters one and three to the script. The second parameter is the full path to the public-key file.

The remainder of the collect method uses the imported Python twisted *getProcessOutputAndValue* method to run the command, returning either the output or raising an exception if the command failed.

```
try:
  cmd_stdout = yield getProcessOutputAndValue(cmd, args = args )
  log.debug( 'RootDiskFree collect. stdout is %s and stderr is %s and exit
    code is %s ' % (cmd_stdout[0], cmd_stdout[1], cmd_stdout[2]))
except Exception:
  log.exception('Error in collect gathering RootDiskFree info - %s ' % (Exception))
returnValue(cmd_stdout)
```

getProcessOutputAndValue is used to run the command as this method spawns a process and returns a Deferred that will be called back with its output (from stdout and stderr) and it's exit code as (out, err, code) If a signal is raised, the Deferred will errback with the stdout and stderr up to that point, along with the signal.

The result from *getProcessOutputAndValue* is a tuple with 3 elements:

```
( <command stdout> , <command stderr>, <command exit code > )
```

Note that the code that actually retrieves the data needs to be in a *try / except* clause so that any failure raises an errback; because the whole *collect* method has the *@inlineCallbacks* decorator, the *yield* function delivers communications value(s), as they arrive.

returnValue() must be used to deliver a result from the collect method, **not** a simple *return*; this is part of the twisted module.

13.3.1.4 onResult method for the PythonDataSourcePlugin

onResult is called immediately on result and before either onSuccess or onError.

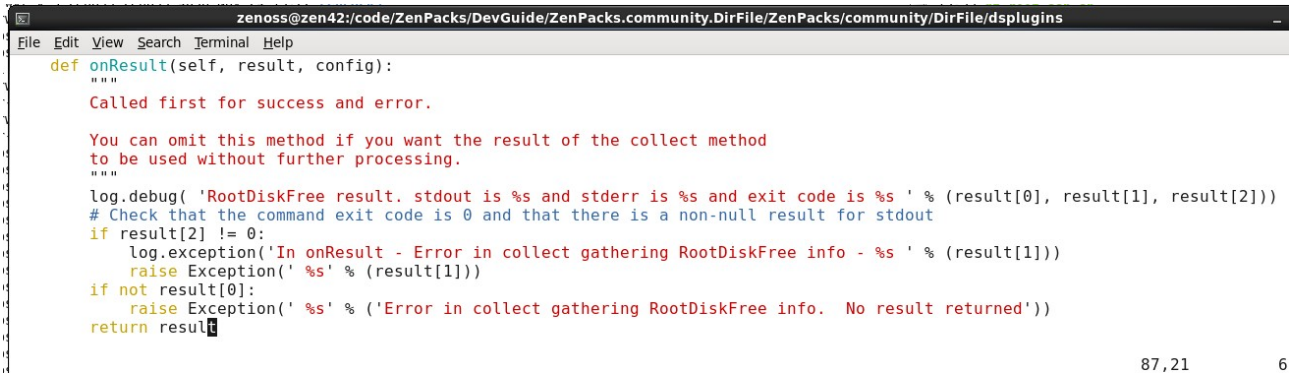


Figure 167: onResult method

It is used to check whether the command exit code was non-zero or the command stdout was null; in either case, an exception is raised.

13.3.1.5 onSuccess method for the PythonDataSourcePlugin

If the *collect* method successfully yields results then the *onSuccess* callback will be called. Ultimately, the *zenpython* daemon should return a dictionary datastructure:

```
return {
  'values': defaultdict(dict),
  'events': [],
  'maps': [],
  }
```

where the *values* element is a d**ictionary of dictionaries** and any *events* and / or *maps* are Python **lists** (both may be empty).

The values dictionary has component id as key name. Since this example is a device-level datasource, the component is *None*. The inner dictionary has the **key as a datapoint name and the value as the corresponding value**.

This example is further simplified by hard-coding the datapoint name, *dfRootPython*, as the key of the inner dictionary and the value is the integer result returned by the collect method.

```
def onSuccess(self, result, config):
  """
  Called only on success. After onResult, before onComplete.
  """
  log.debug( 'In success - result is %s and config is %s ' % (result, config))
  data = self.new_data()
  data['values'] = {}
    for ds in config.datasources:
    # We are forcing a single value returned from collect to populate
    #   a single known datapoint called dfRootPython
    data['values'][None] = {'dfRootPython' : result}

  log.debug( 'data is %s ' % (data))
  return data
```

13.3.1.6 onError method for the PythonDataSourcePlugin

If the *collect* method fails for any reason then the errback, *onError* will be called. Typically this will perform some logging and probably deliver an event to be passed to the user through the Event Console.

Again, as a first example, the event is largely hardcoded but it does pass the result stderr message as part of the event summary field.

```
def onError(self, result, config):
  """
  Called only on error. After onResult, before onComplete.
  You can omit this method if you want the error result of the collect
  method to be used without further processing. It recommended to
  implement this method to capture errors.
  """
  log.debug( 'In OnError - result is %s and config is %s ' % (result, config))
  return {
    'events': [{
      'summary': 'Error getting root df data with zenpython: %s' % result,
      'eventClass': '/DirFile',
```

```
            'eventKey': 'RootDiskFreePythonDeviceData',
            'severity': 4,
            }],
        }
```

Note that in this case, there is no *values* or *maps* components of the datastructure returned, and that the *events* element is a list containing one or more dictionaries, where the **dictionary key is a valid event field name** and the value is a suitable value.

It is good practice to deliver in the event, some indication of where the error has occurred. In this example, the *eventKey* field is set to the string *RootDiskFreePythonDeviceData*.

It is good practice to set an *eventClass* in the event, otherwise it will have the *Unknown* class in the event console.

No other methods are required for the *PythonDataSourcePlugin*. It will inherit the *onComplete* and *cleanup* methods from its parent *PythonDataSourcePlugin* class, all of which simply return the result.

13.3.1.7 Testing the new PythonDataSourcePlugin

When a new plugin has been created, it should only be necessary to restart:

- zenhub
- zopectl
- zenpython

Similarly if any changes are made to the plugin file.

13.3.1.8 Performance template to drive the PythonDataSourcePlugin

A device-level performance template is required to drive the new *PythonDataSourcePlugin*. In section 13.3.1.5 the *onSuccess* method hard-coded the name of the datapoint as ***dfRootPython*** (to keep things simple for now). The template must define a datasource of type *Python* with a datapoint called *dfRootPython*.

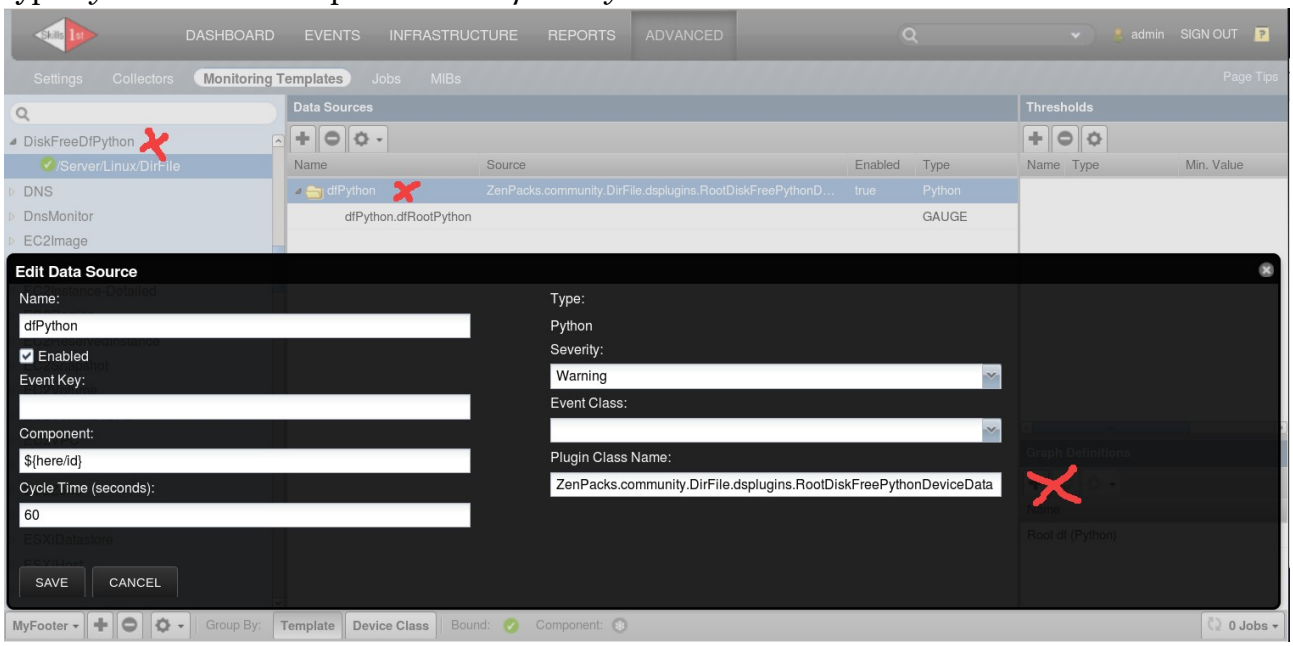


Figure 168: DiskFreeDfPython performance template with PythonDataSourcePlugin

The template:

- Has name *DiskFreeDfPython*
- Is associated with Zenoss device class */Server/Linux/DirFile*
- Has a datasource called *dfPython* of type *Python*
- Has a cycle time of 60 seconds (for testing - probably too fast for production)
- Has Plugin Class Name of *ZenPacks.community.DirFile.dsplugins.RootDiskFreePythonDeviceData*
    - Note that the plugin field, strictly, is the module path
    - Unfortunately there is no dropdown selection list for this field so care is necessary when typing
    - The module name is the same whether a *dsplugins* directory is used or not (see later)
- The *dfPython* datasource has a datapoint called *dfRootPython* (to match the plugin onSuccess datapoint coding) which gives a fully-qualified datapoint name of *dfPython.dfRooyPython*
- A graph is created for the template called *Root df (Python)* which includes the datapoint

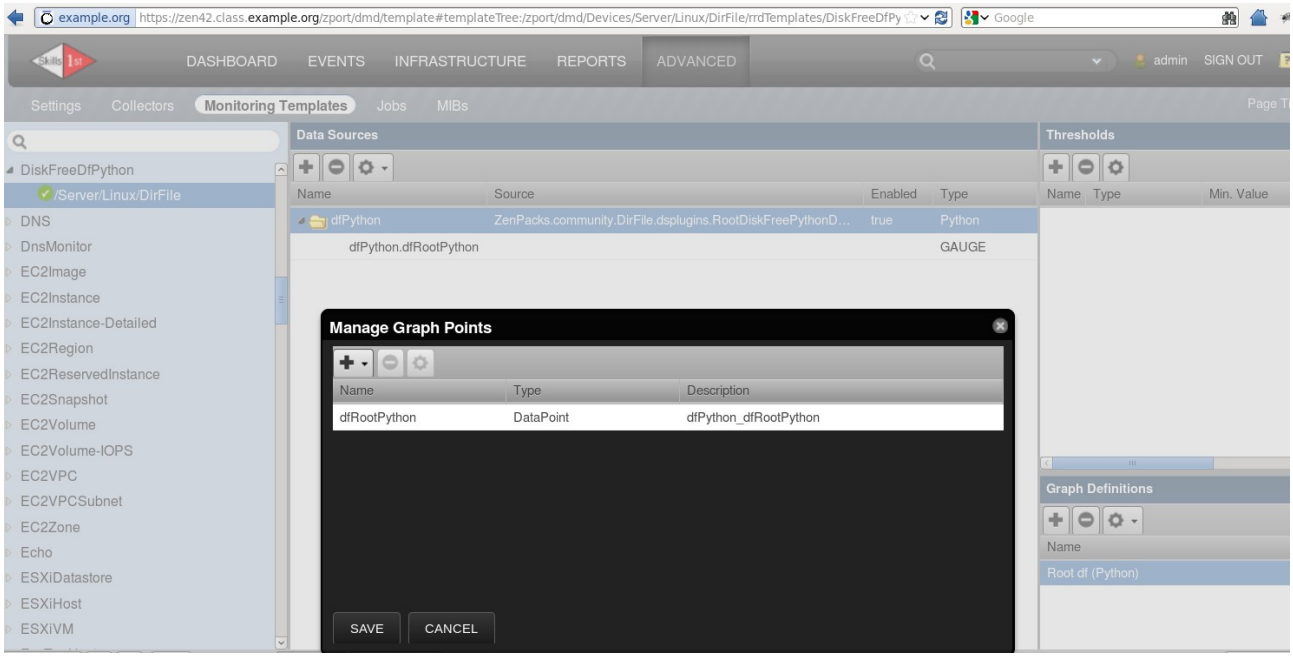


Figure 169: Datapoint and graph for DiskFreeDfPython performance template

The performance template must be bound to a device class or specific device before data will be gathered. This could be done in the ZenPack's *zenpack.yaml* file.

```
device_classes:
 /Server/Linux/DirFile:
  remove: False   # False is default - specified for clarity
  zProperties:
   zPythonClass: ZenPacks.community.DirFile.DirFileDevice
```

```
zSshConcurrentSessions: 5
zDeviceTemplates:
 - Disk_free_df
 - Device
 - DiskFreeDfPython
```

If *zenpack.yaml* is modified then it is safest to reinstall the ZenPack and restart Zenoss.

13.3.2 * Blocking and non-blocking in Twisted

The *collect* method of a *PythonDataSourcePlugin* should return a ***Twisted Deferred***. There is an excellent commentary on the Zenoss forums by Chet Luther at http://www.zenoss.org/forum/136876 which discusses *PythonDataSourcePlugins* from the context of blocking and non-blocking code. It is slightly paraphrased here.

> Prior to version 1.7.2 of ZenPacks.zenoss.PythonCollector, several users reported problems with zenpython not collecting to varying degrees. In some cases it would appear to get completely deadlocked and stop collecting all datasources for all devices, and a /Status/Heartbeat event would eventually be created indicating zenpython was no longer functioning. In other cases it was more subtle, lots of tasks were missed or delayed which resulted in gaps in graphs and delays in detecting problems.
>
> In all of these cases the cause was eventually found to be PythonDataSourcePlugin subclasses in ZenPacks (zenpython plugins) using **blocking** IO instead of **non-blocking** IO using Twisted's mechanisms. In the extreme case of a completely deadlocked zenpython the cause was a plugin that made a synchronous MySQL query that had no timeout set, so in some cases where the database would never finish responding to a query, zenpython would remain blocked and unable to perform any collection until it was restarted. In the less severe cases it was more typical things where sometimes making API calls takes many seconds or even many minutes, but even these would accumulate to making zenpython not be able to do any other collection until those blocking API calls completed.
>
> PythonCollector 1.7.2 was released with the "blockingwarning" option, a new "percentBlocked" datapoint, and detailed task state tracking for how long tasks spent blocked. The idea was to get a better idea of how prevalent the problem was, and which common plugins were affected. It didn't do anything to keep zenpython from getting wedged by a badly behaved plugin. Some issues were found with plugins in the MySqlMonitor and AWS ZenPacks using blocking IO. An update has been released to MySqlMonitor that fixed its plugins, and an update is being worked on to the AWS ZenPack that fixes its issues. That said, some users were still concerned that a badly behaved plugin could wedge zenpython.
>
> To address this concern, PythonCollector 1.7.3 was released with the "blockingtimeout" option which defaults to 5 seconds (that may seem too short, more on that later.) The blockingtimeout option is a mechanism for zenpython to protect itself from getting wedged by a plugin that uses blocking IO, and it works as follows.
>
> A timer is started each time the collect method of any PythonDataSourcePlugin is called. The timer is cancelled as soon as that collect method returns. If the timer reaches blockingtimeout before the collect method returns, **the zenpython process completely restarts itself with the plugin that failed to return in time**

disabled. This may sound extreme, but for technical reasons was really the only way to recover from being blocked.

To understand why zenpython must restart itself you have to understand how **zenpython is a single-threaded process** that uses asynchronous IO via the Twisted library to efficiently perform lots of parallel collection. Any time non-asynchronous code is being executed it means that no other collection or process can be performed until it yields back to the reactor (Twisted's event loop). The catch here is that zenpython is executing arbitrary code provided by ZenPacks. The zenpython code essentially gives up control to that ZenPack code as soon as it calls it, until that code returns. The way that zenpython manages to restart itself when it calls to ZenPack code that doesn't return is by running a watchdog thread that keeps an eye on that timer and restarts the process if it expires.

If the plugin is disabled (and logged in */var/zenoss/zenpython.blocked* on Zenoss 5, or */opt/zenoss/var/zenpython.blocked* on Zenoss 4) then it is disabled for **all** devices, not just the device where the plugin timed out.

To understand why that is done you have to imagine a scenario where you have a plugin that executes on all Linux servers. Let's say you have 1,000 Linux servers being monitored, and let's say that 50 of them develop some condition that causes the plugin to block on them for 10 seconds. Assuming it takes zenpython only 1 minute to restart when monitoring 1,000 servers, it would take (60 seconds + ~10 seconds) * 50 servers, or ~11 minutes before zenpython would have all of the plugin+device executions disabled. During this time zenpython wouldn't be doing anything. So we are forced to assume that if a plugin blocks once, that it's not written correctly and using blocking IO. So the plugin is completely disabled for all devices that use it.

So now back to why the default blockingtimeout is set to 5, which may seem way too low given everything above. To understand this, you again have to understand why the PythonDataSourcePlugin.collect() method must return a Deferred. After all, it's easy to imagine making an API call to some application that takes more than 5 seconds. When using Twisted's non-blocking networking APIs you **immediately** get a Deferred object returned when you make a call like getPage("http://example.com/api/really-slow-thing"), even if the response doesn't come back for 10 minutes. When the response comes back, the Deferred's callback (or errback if there's an error) is called. The PythonDataSourcePlugin.collect() must do the same thing. Usually by returning the Deferred object that something like getPage("http://...") returned to you.

If you properly use Twisted's networking libraries to perform non-blocking IO in this way, your collect method will return nearly instantaneously even if the device being monitored is down, or slow, or anything like that. So your API call can take way longer than blockingtimeout, but your collect method still returns immediately.

It's almost always possible to create a Twisted way to get your stuff. If you can't find a way to do it natively within the process there are a couple of less-optimal options. You can use *twisted.internet.utils.getProcessOutputAndValue()* to run an external script that returns a Deferred that will callback when the process exits with output and an exit code. Alternatively you can use *twisted.internet.threads.deferToThread()* to run some blocking code in-process in another thread. It also will return a Deferred immediately that will callback when the call returns with the return value.

https://twistedmatrix.com/documents/13.1.0/api/twisted.internet.utils.getProcessOutputAndValue.html

https://twistedmatrix.com/documents/13.1.0/api/twisted.internet.threads.deferToThread.html

getProcessOutputAndValue is suboptimal because it requires spawning another process and passing all parameters via command line options and parsing its text output.

deferToThread is suboptimal because the threadpool is of a limited size, so only so many things can be running on the threadpool at a time. It can also consume considerable extra memory.

*13.3.2.1 * Comparing blocking and non-blocking collect methods*

The collect method discussed so far, using *getProcessOutputAndValue* is:

```
@inlineCallbacks
def collect(self, config):

 ds0 = config.datasources[0]
 for k,v in ds0.__dict__.items():
   log.debug('ds0 k is %s and v is %s ' % (k,v))

 thisabspath = os.path.dirname(os.path.abspath(__file__))
 libexecdir = thisabspath + '/../libexec'
 # If ds0.zKeyPath starts with ~ then it doesn't expand properly - convert to full path
 # expanduser gives $HOME including trailing /
 homedir = os.path.expanduser("~")
 if ds0.zKeyPath.startswith('~'):
   keyPath = ds0.zKeyPath.replace('~', homedir)
 else:
   keyPath = ds0.zKeyPath
 # script is df_root_ssh.sh taking 3 parameters, zCommandUsername, keyPath, host addr
 cmd = os.path.join(libexecdir, 'df_root_ssh.sh')
 args = ( ds0.zCommandUsername, keyPath, ds0.manageIp)
 log.debug(' cmd is %s \n ' % (cmd) )

 try:
   cmd_stdout = yield getProcessOutputAndValue(cmd, args = args )
 except Exception:
   log.exception('Error in collect gathering RootDiskFree info - %s ' % (Exception))
 returnValue(cmd_stdout)
```

This is non-blocking. *getProcessOutputAndValue* delivers a **Deferred**.

If the collect method is written with something that is probably more familiar like using the Python *subprocess* module to run a command, the plugin will block until the results are returned. Note that *subprocess* requires its command to be a list with command and arguments included.

```
@inlineCallbacks
def collect(self, config):

  ds0 = config.datasources[0]
  for k,v in ds0.__dict__.items():
    log.debug('ds0 k is %s and v is %s ' % (k,v))

  thisabspath = os.path.dirname(os.path.abspath(__file__))
  libexecdir = thisabspath + '/../libexec'
  # If ds0.zKeyPath starts with ~ then it doesn't expand properly so convert to full path
```

```
# expanduser gives $HOME including trailing /
homedir = os.path.expanduser("~")
if ds0.zKeyPath.startswith('~'):
  keyPath = ds0.zKeyPath.replace('~', homedir)
else:
  keyPath = ds0.zKeyPath
# script is df_root_ssh.sh taking 3 parameters, zCommandUsername, keyPath, host addr
cmd = [ os.path.join(libexecdir, 'df_root_ssh.sh'),
    ds0.zCommandUsername, keyPath, ds0.manageIp ]
log.debug(' cmd is %s \n ' % (cmd) )

value = None
try:
  cmd_process = yield(subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE))
  # cmd_process.communicate() returns a tuple of (stdoutdata, stderrordata)
  cmd_stdout, cmd_stderr = cmd_process.communicate()
  log.debug(' stdout is %s and stderr is %s ' % (cmd_stdout, cmd_stderr))
  if not cmd_stderr:
    value = int(cmd_stdout.rstrip())
  else:
    raise Exception('%s ' % (cmd_stderr))
except:
  log.exception('Error gathering RootDiskFree info - %s ' % (cmd_stderr))
  raise Exception(' %s' % (cmd_stderr))
   returnValue(value)
```

Under light load, with good communications response, both variations will work in a similar fashion. If *cmd_process = yield(subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE))* provides data inside the *blockingtimeout* then the same results will be seen (albeit, the overall performance of *zenpython* will still be less effective).

However, if *subprocess* does not yield results within *blockingtimeout* then the plugin will be permanently disabled, for all devices. If there are 1000 devices using the plugin and one device happens to be down, **this plugin will still be disabled for all devices**.

13.3.3 Collecting *component* performance data; specific component command; single value returned

The COMMAND version of collecting component datasources is discussed in section 12.3. 12.3.1 discusses running a command template datasource where a filename is passed and the *du* command is run to determine disk used in bytes. The Unix command is configured as part of the datasource:

```
/usr/bin/du -P -b ${here/fileDirName}/${here/fileName} | cut -f 1
```

To run the equivalent under *zenpython*, a small shellscript, *dufile_ssh.sh*, can be constructed which, as with the previous example in this section, takes ssh parameters in addition to the filename:

```
#!/bin/bash
#
# Use du to get disk used for a directory in bytes.
# Param 1 = ssh user
# Param 2 = keypath
# Param 3 = target
# Param 4 = file
```

```
echo $1 $2 $3 $4 > /tmp/dufile.tmp
#ssh -l zenplug -i ~/.ssh/id_dsa taplow-11 /usr/bin/du -P -b \
# /opt/zenoss/local/fredtest/fred1.log_20151202
ssh -l "$1" -i "$2" "$3" /usr/bin/du -P -b "$4"
```

Note that the script returns the full output - there is no *cut* in the new script so output should be of the format:

```
'499\t/opt/zenoss/local/fredtest/test/fred2.log_20151125\n'
```

where \t is a tab and \n is a newline.

A new *PythonDataSourcePlugin* class will be created to drive this script.

13.3.3.1 Using a dsplugins directory

Rather than expanding the existing *dsplugins.py*, a ***dsplugins*** directory will be created under the ZenPack base directory; thus plugins can be maintained in separate files making them easier to manage and debug.

Create the directory and then create an *__init__.py* in the *dsplugins* directory. This file needs an entry for each plugin file in the format:

```
from RootDiskFreePythonDeviceData import RootDiskFreePythonDeviceData
from FileDiskUsedPythonDeviceData import FileDiskUsedPythonDeviceData
```

where the first name is the filename (strictly, the module name) and the second name is the class within the module. It is generally good practice for the two names to be the same.

The only other small modification required to move the *RootDiskFreePythonDeviceData* class into its own *RootDiskFreePythonDeviceData.py* file under *dsplugins*, is to change the *libexec* directory that was carefully constructed.

```
thisabspath = os.path.dirname(os.path.abspath(__file__))
libexecdir = thisabspath + '/../libexec'
```

The *libexec* directory is one up from the current (*dsplugins*) directory and then down to *libexec*.

13.3.3.2 Imports, proxy_attributes, config_key and params

The imports and proxy_attributes for the *FileDiskUsedPythonDeviceData* plugin are exactly the same as for *RootDiskFreePythonDeviceData*.

The *config_key* method to determine the separate configs to be distributed to *zenpython* daemons, needs to be slightly different. To make component configs unique, both the device and the component need to be included:

```
@classmethod
def config_key(cls, datasource, context):
  # context will be a File.

  return (
    context.device().id,
    datasource.getCycleTime(context),
    datasource.rrdTemplate().id,
    datasource.id,
    datasource.plugin_classname,
    context.id,
```

```
        'FileDiskUsedPythonDeviceData',
        )
```

Since *context.id* is the component id for a component template, and several devices may have the same filename, the parent device is required to make configs unique; hence the addition of *context.device().id*.

Note that the cycle time parameter **must** remain as the second field (more on this in the next section).

The *params* method needs to retrieve data from the ZODB database for use in the later collect method.

```
@classmethod
def params(cls, datasource, context):
  # context is the object that the template is applied to - either a device or a component
  # Use params method to get at attributes or methods on the context.
  # params is run by zenhub which DOES have access to the ZODB database.
  params = {}
  params['fileName'] = ''
  if hasattr(context, 'fileName'):
    params['fileName'] = context.fileName
  params['fileDirName'] = ''
  if hasattr(context, 'fileDirName'):
    params['fileDirName'] = context.fileDirName
  params['fileId'] = context.id
  # Need to run zenhub in debug to see log.debug statements here
  log.info(' params is %s ' % (params))
  return params
```

fileDirName and *fileName* will be used to create the fully-qualified filename to run the *du* command against. *fileId* is required as the component index for the values returned by the collect method.

13.3.3.3 * A closer look at the usage of config_keys

Care is needed when constructing the tuple to be returned from the *config_key* method. The data is used by *zenhub* to create the separate configs for *zenpython* daemons. The code that actually uses this structure is ***tasks.py*** in *$ZENHOME/Products/ZenCollector*.

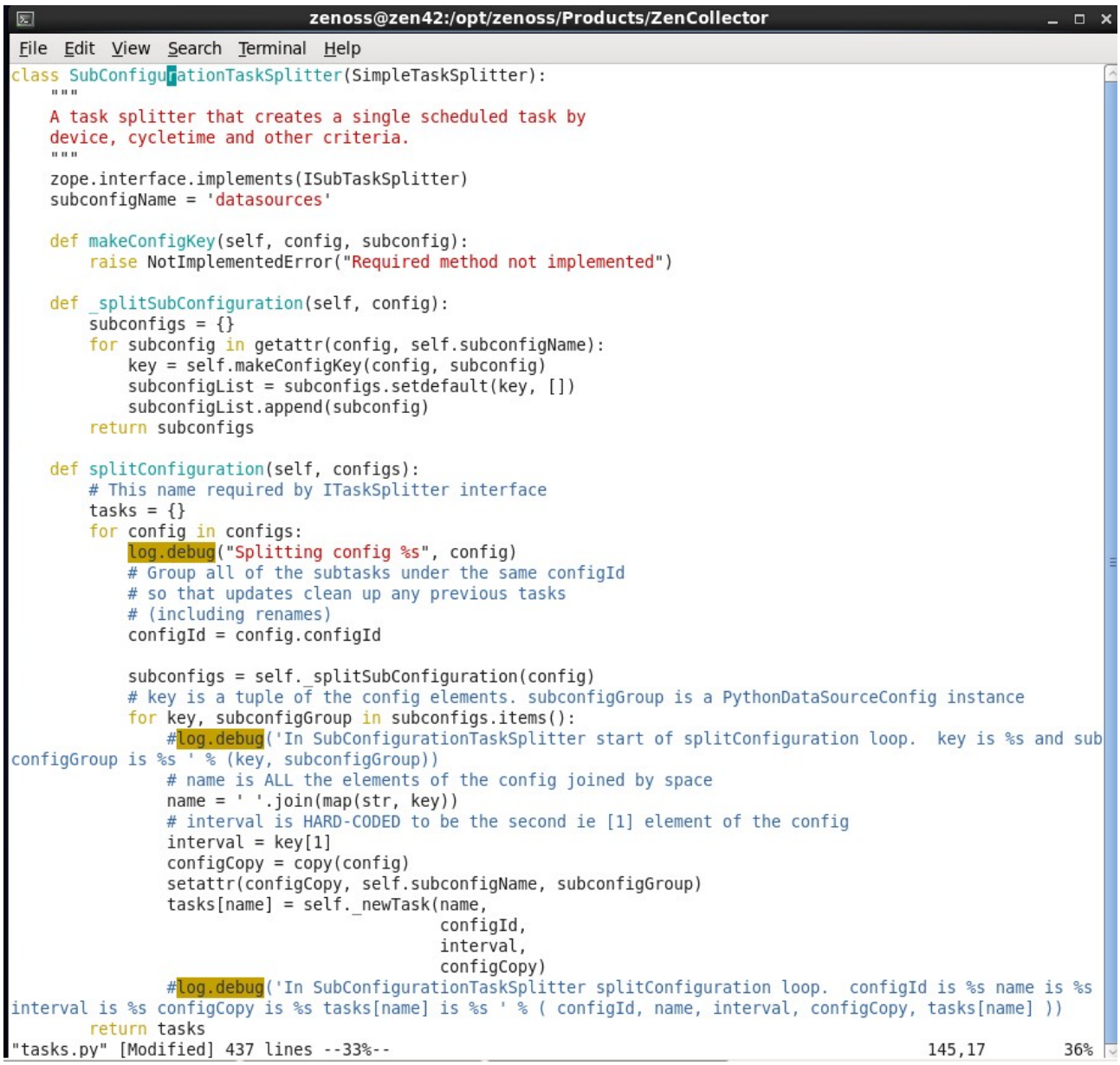


Figure 170: SubConfigurationTaskSplitter class in $ZENHOME/Products/ZenCollector/tasks.py

Note in the *splitConfiguration* method that:

- key is a Python tuple of the config_key elements
- subconfigGroup is a *PythonDataSourceConfig* instance
- interval is set to key[1]

This means that the ***interval*** parameter that is passed to the *_newTask* method is absolutely position dependent.

If *config_key* in the dsplugin file is:

```
@classmethod
def config_key(cls, datasource, context):
  # context will be a File.

  return (
    context.device().id,
```

```
        datasource.getCycleTime(context),
        datasource.rrdTemplate().id,
        datasource.id,
        datasource.plugin_classname,
        context.id,
        'FileDiskUsedPythonDeviceData',
        )
```

then the *interval* parameter will be set to *datasource.getCycleTime(context)* - the second field, that is the [1] element, in the Python tuple.

If config_key is coded as:

```
    @classmethod
    def config_key(cls, datasource, context):
      # context will be a File.

      return (
        context.device().id,
        context.id,
        datasource.getCycleTime(context),
        datasource.rrdTemplate().id,
        datasource.id,
        datasource.plugin_classname,
        'FileDiskUsedPythonDeviceData',
        )
```

then *zenpython.log* will be full of errors for such tasks as the *interval* parameter (which, incidentally, is a digit, not a string) will be set to *context.id*.

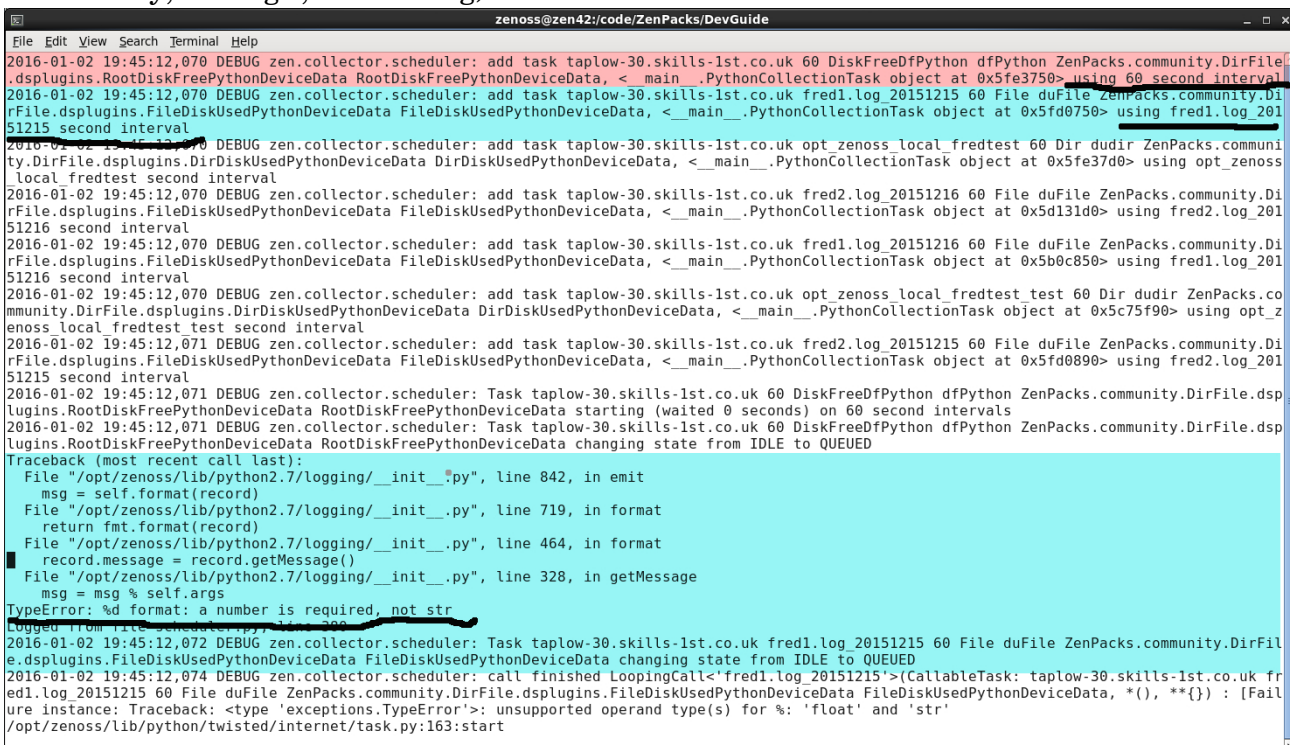


*Figure 171: zenpython debug log where interval is **not** second element of config_key*

In Figure 171 the task for the **device** using *RootDiskFreePythonDeviceData* is correct with an interval of 60 seconds (first highlighted section); see Figure 161 for this *config_key* method. Tasks for the **components** using *FileDiskUsedPythonDeviceData* (second highlighted section)

show that the second field is *fred1.log_20151215* so that is used for the *interval* parameter, which then results in the subsequent errors, starting with a “TypeError: %d format: a number is required, not str” message.

The *libexec* directory of *ZenPacks.community.DirFile* includes a slightly modified copy of *tasks.py* with extra l*og.debug* lines to help understand the task splitting process.

13.3.3.4 collect method

The *collect* method for the *FileDiskUsedPythonDeviceData PythonDataSourcePlugin* is similar to the previous method for *RootDiskFreePythonDeviceData*; the differences come towards the end where data has to be returned for each component.

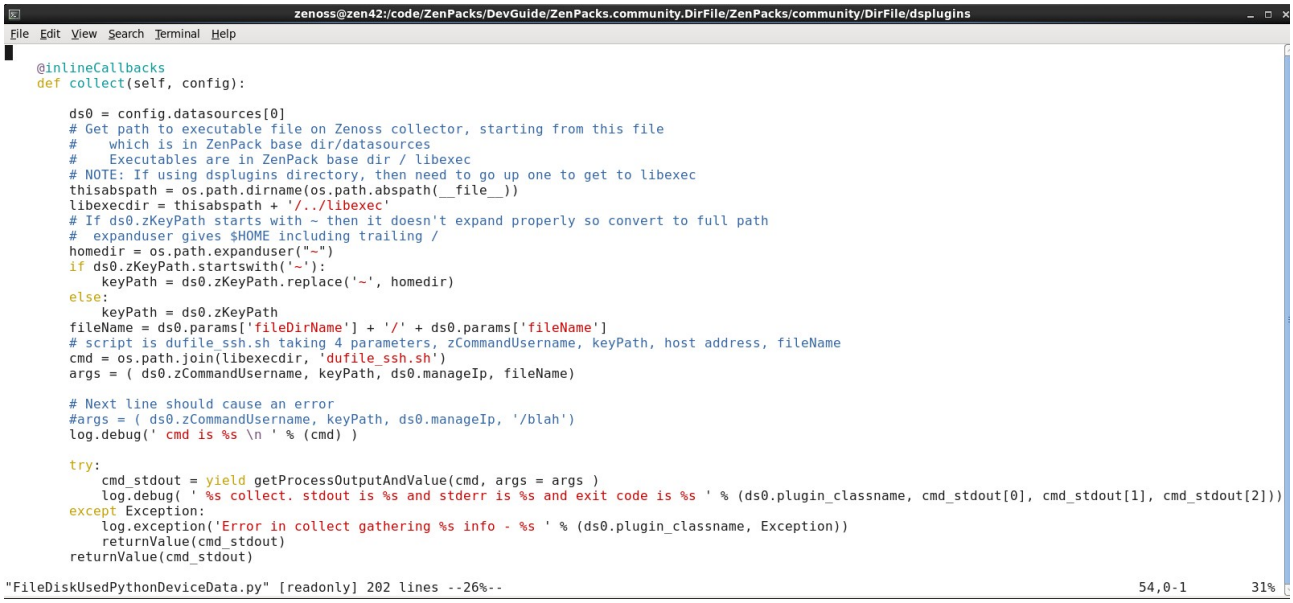


Figure 172: collect method for FileDiskUsedPythonDeviceData

In Figure 172:

- *fileName* is constructed from the datasource params list, concatenating *fileDirName* and *fileName* with a slash in between. It is then used as the final argument in the command to be run.
- The command name is hardcoded to be *dufile_ssh.sh* in the *libexec* directory of the ZenPack.
- The command arguments are passed in the *args* variable.
- *getProcessOutputAndValue* is used to deliver a Deferred result.
- The *yield* in the *try .. except* clause is a generator that delivers the output from the *du* command.
- If something goes wrong then an exception is delivered as an errback
- *ds0.plugin_classname* is used to parameterise logging and exception statements

The *onResult* method is almost exactly the same as in the previous example.

13.3.3.5 onResult, onSuccess and onError methods

The *onSuccess* method must deliver a *new_data* dictionary where the *values* element is itself a dictionary with keys being component id and values is a dictionary of {datapointname : <value returned from du command>}.

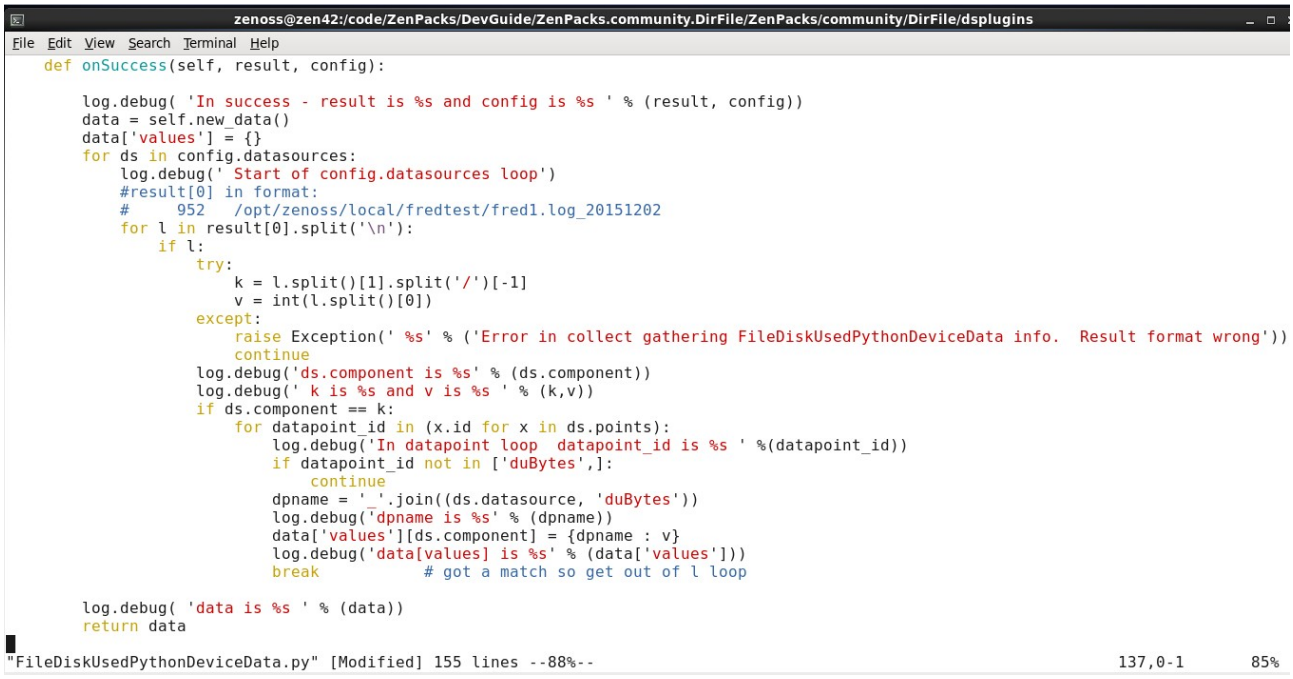


Figure 173: onSuccess method for FileDiskUsedPythonDeviceData

In the onSuccess method in Figure 173:

- For each datasource config:
  - result[0] should hold one or more lines in the format:
    - 952 /opt/zenoss/local/fredtest/fred1.log_20151202
  - Split result[0] by whitespace and:
    - Split the second element on '/' and assign the last element to *k* (the filename)
    - Assign the integer value of the first element to *v* (the number of bytes used)
  - The *k* value from the result is compared against the datasource component attribute.
  - Where it matches, the datapoints for the datasource are cycled through, looking for a datapoint called *duBytes* (which is hardcoded in the code here). The performance template that uses this plugin must be constructed to ensure that the *duBytes* datapoint exists. When a match is found, break out of this loop.
  - *dpname* is constructed by joining the datasource name with *duBytes*, linked with an underscore, to give the fully-qualified datapoint name.

- The *values* element of the data dictionary is constructed with the component id as the key and the corresponding value being the dictionary with *dpname* as its key and the returned du number as the value.

The onError method is almost exactly the same as in the previous example. The *summary* has the exception error message appended to the end. The *eventKey* field is set to *FileDiskUsedPythonDeviceData*.

13.3.3.6 Performance template to drive the PythonDataSourcePlugin

As ever, a performance template is required to drive data collection. The default, automatically-bound template for a *File* component must match the label name of the component object type ie. *File*. However, *zenpack.yaml* permits specifying a list of templates for a component so, for clarity, the template to drive the *FileDiskUsedPythonDeviceData* will be specified in its own template called ***FilePythonXml*** and will be stored in the ZenPack's *objects.xml* file.

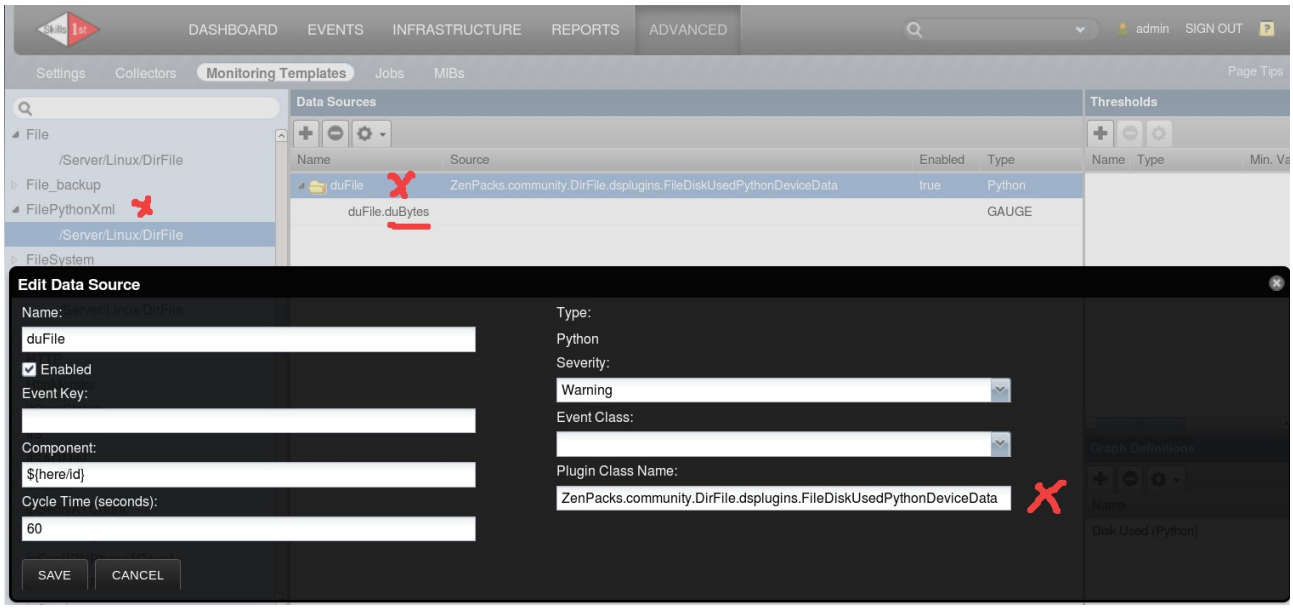


Figure 174: FilePythonXml template to drive .FileDiskUsedPythonDeviceData component plugin

Note in Figure 174:

- Template is called *FilePythonXml*
- Datasource is called *duFile*
- Datapoint name is *duBytes*, giving a fully-qualified datapoint name of *duFile.duBytes*. This must match the datapoint name constructed in the onSuccess method of the plugin.
- The Plugin Class Name must be typed carefully to match the module path to the plugin. Note that it is irrelevant whether a *dsplugins* directory is used.
- The cycle time is set to 60s for testing

A graph, *Disk Used (Python)* is created to display the datapoint for the component.

The *zenpack.yaml* file will have the *File* object class amended to include the *FilePythonXml* template:

```
File:
 label: File # NB It is label, with spaces removed, that is used to match a component template
 meta_type: File  # Will default to this but in for completeness
 order: 70  # after dir
 auto_expand_column: fileName
 # monitoring_templates defaults to File. Also need FileXml, shipped in objects.xml
 # to circumnavigate issue whereby custom datasource CommandTemplate appears blank when
 #  shipped in zenpack.yaml. Also added on FilePythonXml.
 monitoring_templates: [File, FileXml, FilePythonXml]
```

13.3.4 Collecting component performance data; specific component command; multiple values returned. Nagios plugin conversion.

This section converts the COMMAND datasource scenario which checks for the presence of two specific strings in a *File* component and provides a count for the number of lines where the string appears. The strings are known in advance. The COMMAND-based solution ran a command on the remote target (over ssh) and utilised the Nagios parser to decode the output returned into different <datapoint name>=<datapoint value> pairs.

As with the last example, a local shellscript will run an ssh command, driven by by the twisted *getProcessOutputAndValue* method. A big difference with this scenario is that the command to be run remotely by ssh is not a built-in command but a script that has been placed on the target in a specific directory; thus we have the **local** *file_stats_ssh.sh* running a **remote** command, *file_stats.sh* and the remote command is found in the **zCommandPath** directory. The filename to be checked for the presence of the hard-coded strings "*test 1*" and "*without*" is passed as the fourth parameter to the local script.

The local shellscript, *file_stats_ssh.sh* is:

```
#!/bin/bash
#
# Runs command, file_stats.sh, on remote target to return number of lines
#  in file containing "test 1" and "without"
# file_stats.sh expects one parameter - filename
#
# Returned output in format:
# " File string count test ok | $string1Name=$stringCount1 $string2Name=$stringCount2"
# or error output starting with "Error"
#
# Param 1 = ssh user
# Param 2 = keypath
# Param 3 = target
# Param 4 = file to test
# Param 5 = path to file_stats.sh

echo $1 $2 $3 $4 $5 > /tmp/filestats.tmp
CMD="$5/file_stats.sh $4"
ssh -l "$1" -i "$2" "$3" "$CMD"
```

The remote *file_stats.sh* is exactly the same as in section 12.3.2.

The new plugin, *FileStatsPythonDeviceData.py*, will be implemented in the *dsplugins* directory of the ZenPack. Remember to add a statement for it to the *dsplugins/__init__.py*:

```
from FileStatsPythonDeviceData import FileStatsPythonDeviceData
```

13.3.4.1 Imports, proxy_attributes, config_key and params

Imports are very similar to the last example. *re,* the Python regular expression module, is also imported as that will be used to help decode the Nagios-style output that is returned.

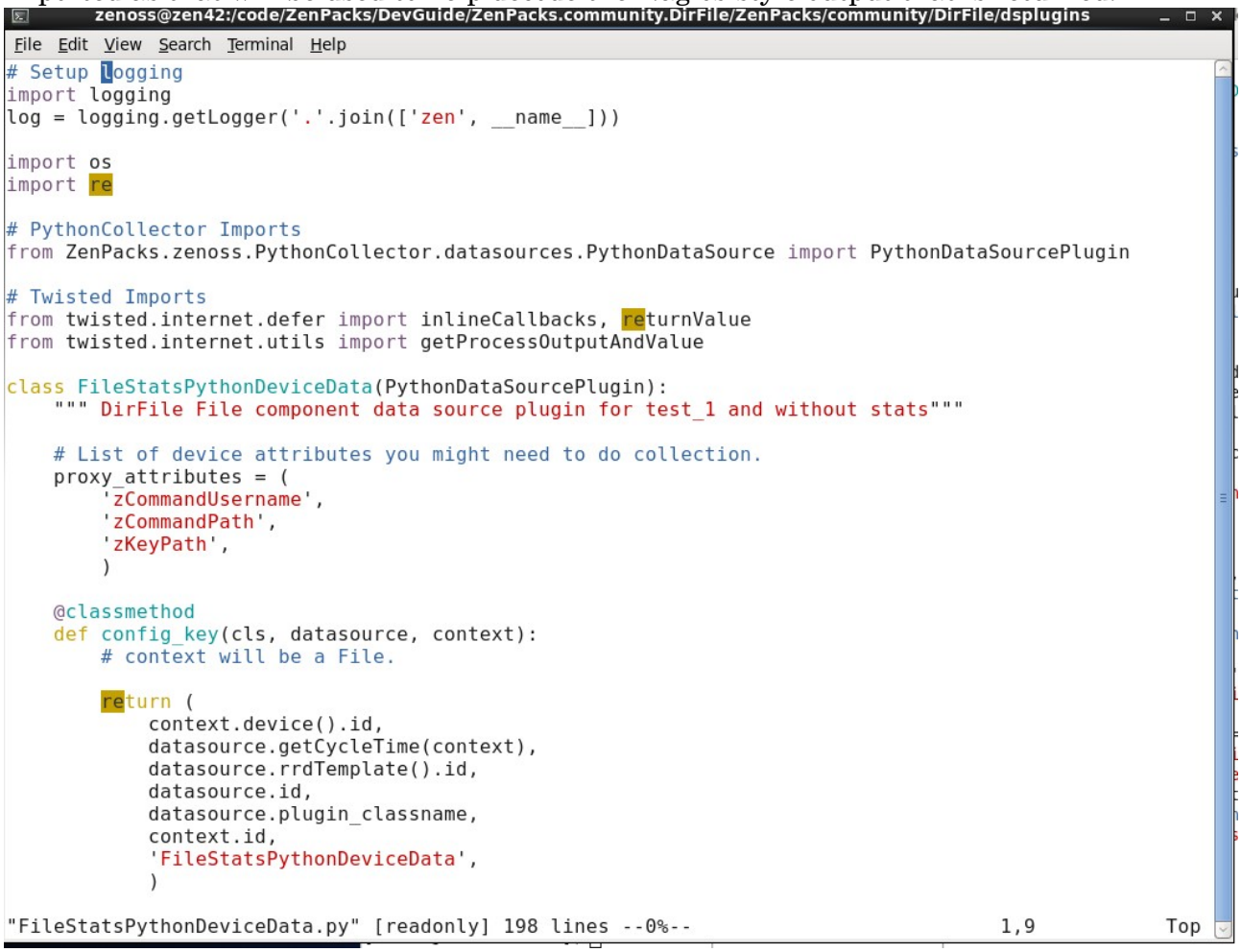


Figure 175: imports, proxies and config_key method

The local shellscript needs to pass the *zCommandPath* property to the remote script so it must be included in the *proxy_attributes* statement.

The config_key method is identical to the previous example, passing both *context.device().id* and *context.id* as part of the unique task definition.

The params method is identical to the previous example, delivering *fileName*, *fileDirName* and *fileId* for use in the methods to be run by *zenpython*.

13.3.4.2 collect method

The collect method is very similar to the previous example. It simply differs in the command to be called and its arguments.

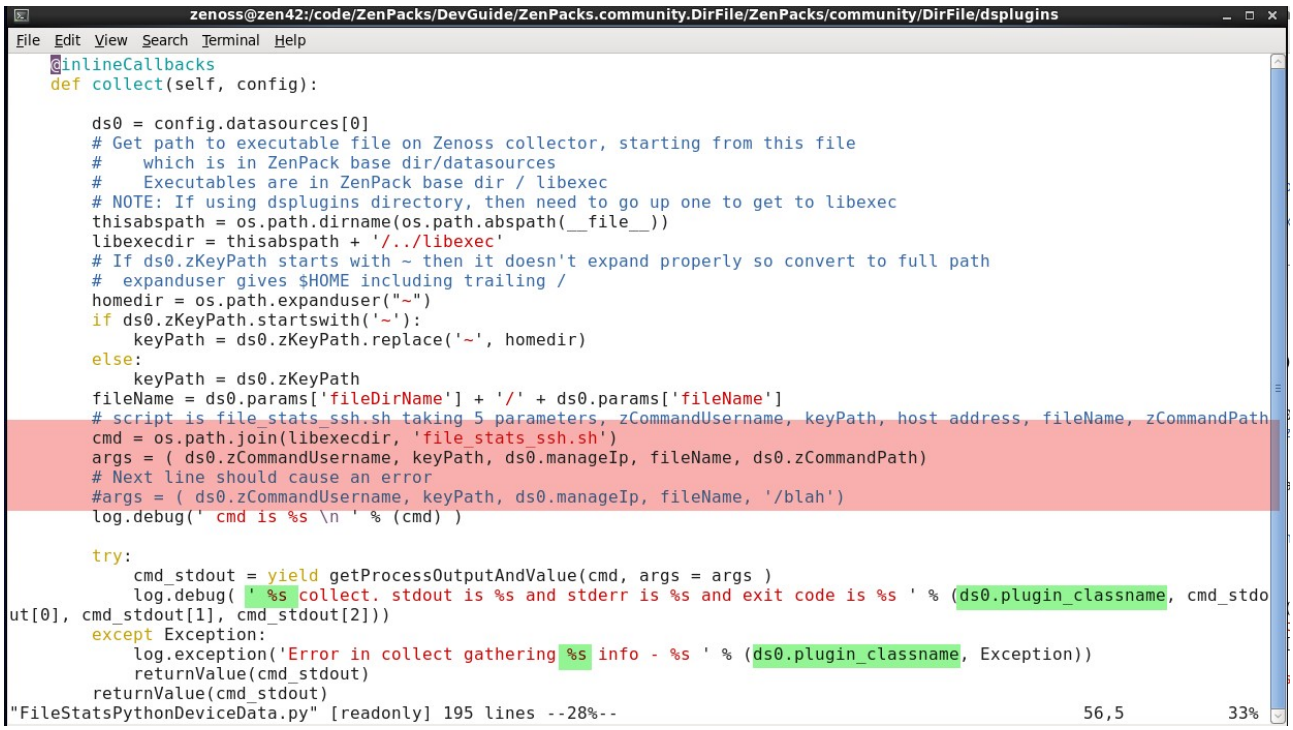


Figure 176: FileStatsPythonDeviceData collect method

Note that by using *ds0.plugin_classname* as a parameter in logging and exception statements, far more commonality can be achieved between similar collect methods.

The collect method is deliberately only concerned with retrieving data, not with processing it.

13.3.4.3 onResult, onSuccess and onError methods

onResult is identical to the previous example, returning an exception to onError if the command exit is non-zero or the command stdout is null. It also uses *ds0.plugin_classname* to parameterise date in logging and exceptions.

The onSuccess method should be used to decode the output from the script.

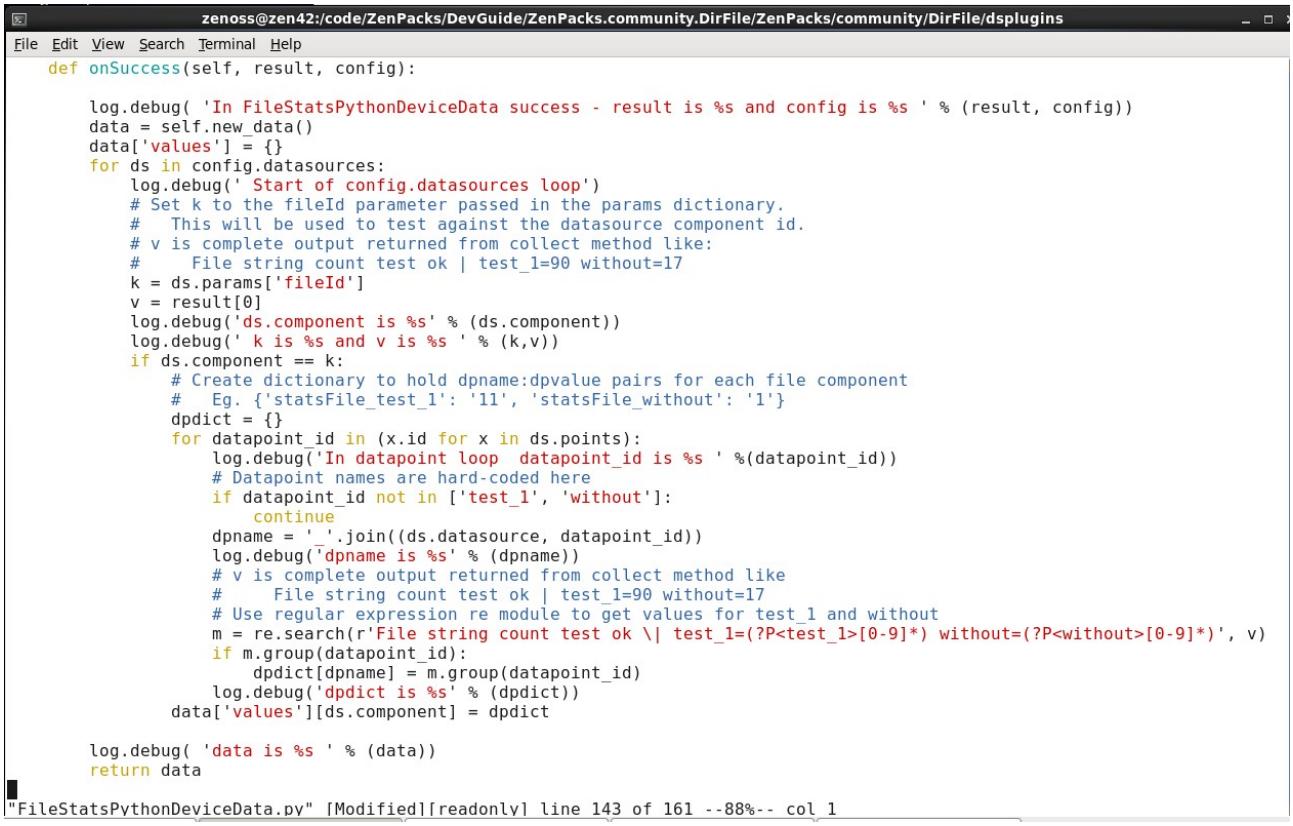


Figure 177: FileStatsPythonDeviceData onSuccess method

In Figure 177:

- The component id is gathered into variable *k* from the *params['fileId']*
- The variable *v* is set to the entire stdout output from the ssh command
- Each component (file) may have several search strings to be checked so a dictionary, *dpdict*, is created to hold *dpname*:*dpvalue* pairs for each file component.

- It is good practice (and enormously helpful) to include sample output when writing code to parse output.
- The Python *re* regular expression module is used to parse the stdout string

```
m = re.search(r'File string count test ok \| test_1=(?P<test_1>[0-9]*) without=(?P<without>[0-9]*)', v)
if m.group(datapoint_id):
  dpdict[dpname] = m.group(datapoint_id)
```

  - The search string to match, starts with the constant text:

    ```
    File string count test ok
    ```

  - Followed by a space, followed by a literal pipe symbol, which needs to be escaped with a backslash
  - Followed by a space, followed by the hard-coded first match string *test_1*= (white space has been replaced by an underscore by this stage)
  - *(?P<test_1>[0-9]*)* is a Python **named group**. The expression to match is *[0-9]** ; that is, zero or more digits and a matching value is assigned to the group name called *test_1*.
  - Followed by hard-coded literal text for the second match string " *without*=" - note the single space before *without* which is delivered by the script output.

- *(?P<without>[0-9]*)* is a second named group delivering a second set of digits to the named group called *without*.
- *v* (the entire output from the shellscript) is the string that the regular expression is applied to.
- The variable *m* holds the result of the regular expression *search* method.
- If a named group matching the *datapoint_id* exists as a result of the search, then create an entry in the *dpdict* dictionary for:

```
dpname: m.group(datapoint_id)
```

- Finally, for each file component, append an entry to the *data['values']* dictionary, to be returned, ultimately, to the template:

```
data['values'][ds.component] = dpdict
```

The onError method is very similar to the previous example.

```
File  Edit  View  Bookmarks  Settings  Help
    def onError(self, result, config):
        """
        Called only on error. After onResult, before onComplete.

        You can omit this method if you want the error result of the collect
        method to be used without further processing. It recommended to
        implement this method to capture errors.
        """
        ds0 = config.datasources[0]
        plugin = ds0.plugin_classname.split('.')[-1]
        log.debug( 'In OnError - result is %s and config is %s ' % (result, config))
        return {
            'events': [{
                'summary': 'Error getting file stats data with zenpython: %s' % result,
                'eventClass': '/DirFile',
                'eventKey': plugin,
                'severity': 4,
                }],
            }

"FileStatsPythonDeviceData.py" 164 lines --87%--                    144,17        99%
```

Figure 178: FileStatsPythonDeviceData onError method

Restart *zenhub*, *zopectl* and *zenpython* to test the new plugin.

13.3.4.4 Performance template to drive the PythonDataSourcePlugin

A new performance **template** *FileStatsPythonXml* will be created to drive the new plugin. The **datasource** is called *statsFile* and, as with the COMMAND datasource in section 12.3.2, **datapoints** *test_1* and *without* must be hard-coded to match the datapoint code in the onSuccess method.

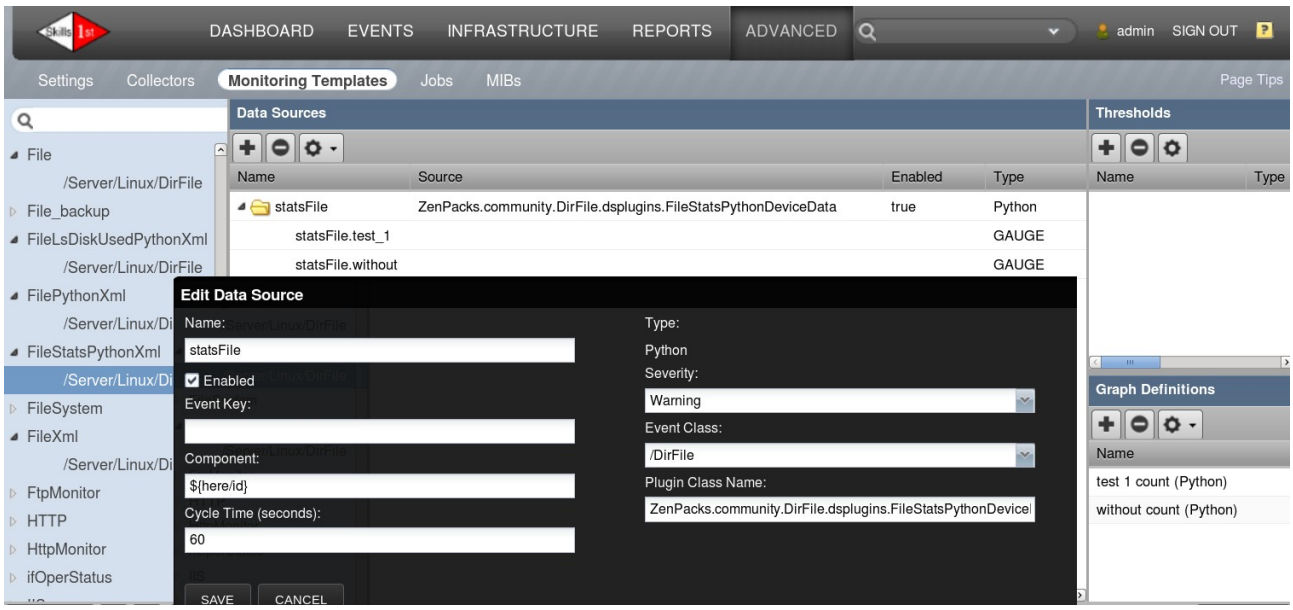


Figure 179: Template for driving the FileStatsPythonDeviceData plugin with hard-coded datapoints

This is a component template so needs to be included in *zenpack.yaml* in the *monitoring_templates* attribute for the *File* object class:

```
monitoring_templates: [File, FileXml, FilePythonXml, FileStatsPythonXml]
```

The template will then be automatically bound to any *File* component.

13.3.5 Collecting component performance data; generic component command with parser

This example converts the COMMAND example in section 12.3.3 where the remote script is proscribed by someone else - part of a commercial package or the work of another department. The script delivers information about **all** components.

The COMMAND example created a bespoke parser to parse the output of l*s -l*; a Python datasource has no built-in parsers so has to do that work anyway. This example differs from previous Python ones by demonstrating the power of the **config_key** method to reduce the work necessary to fulfil component templates.

ls -l <directory> provides output for each file in that directory, including, in the fifth field, the number of bytes in the file; the last field is the filename. This sample will run the *ls -l* command to bring back all files for a directory and then select the particular instance, by the ***fileName*** attribute, along with its size.

Given an example line to parse like:

```
-rw-r--r-- 1 jane users 119 Dec 2 17:36 fred1.log_20151110
```

from section 12.3.3, a *ComponentCommandParser* uses four attributes:

- componentSplit = '\n'
  - The delimiter to split the complete command output into elements per component
- componentScanner = r'(\S+\s+)+(?P<component>.+)$'

- Match on the last element of non-white-space followed by white-space
- where *component* is the named group to match the component id

- scanners = [r'(\S+\s+){4}(?P<lsBytesUsed>[0-9]+)']
  - Match on non-white-space followed by white-space 4 times then take the next numeric field into:
  - *lsBytesUsed,* the named group that must match the datapoint name
- componentScanValue = 'id'
  - In the COMMAND parser, this defines the **attribute** of the component that must match the componentScanner field

Since much of the work has already been done in the old COMMAND parser, significant parts of it will be used in the new plugin.

The new *PythonDataSourcePlugin*, *LsFileDiskUsedPythonDeviceData* , will be added to the ZenPack's *dsplugins* directory with the attendant entry in *dsplugins/__init_.py*:

```
from LsFileDiskUsedPythonDeviceData import LsFileDiskUsedPythonDeviceData
```

13.3.5.1 Imports, proxy_attributes, config_key and params

Imports will be exactly the same as the last example, including the Python *re* module for regular expressions.

The *proxy_attributes* statement only lists *zCommandUsername* and *zKeyPath*; *zCommandPath* is not required as the remote command to be run is a built-in (*ls -l*).

The procedure is to run *ls -l* against a **directory** and then get filename and space used for each **file**. When considering configs to be created, the *context_id* (that is, the individual files) are superfluous; what determines the unique tasks are the target device and the directory that files exist in, not the individual files. Thus the config_key method is:

```
@classmethod
def config_key(cls, datasource, context):
  # context will be a File.
  # One command to a device supplies data for all context.id's
  #  so context.id not needed in config_key

  return (
    context.device().id,
    datasource.getCycleTime(context),
    datasource.rrdTemplate().id,
    datasource.id,
    datasource.plugin_classname,
    context.fileDirName,
    'LsFileDiskUsedPythonDeviceData',
    )
```

The params method is exactly the same as the previous example.

The collect method is almost identical to the previous example; only the command and arguments are changed:

```
cmd = os.path.join(libexecdir, 'lsFileDiskUsed_ssh.sh')
args = ( ds0.zCommandUsername, keyPath, ds0.manageIp, ds0.params['fileDirName'])
```

The onResult method is identical to the previous example.

13.3.5.2 onSuccess method

The onSuccess method capitalises on the custom parser code from the COMMAND example and then uses the component id and the bytes used value to populate entries in the *data['values']* dictionary, similar to previous Python examples.

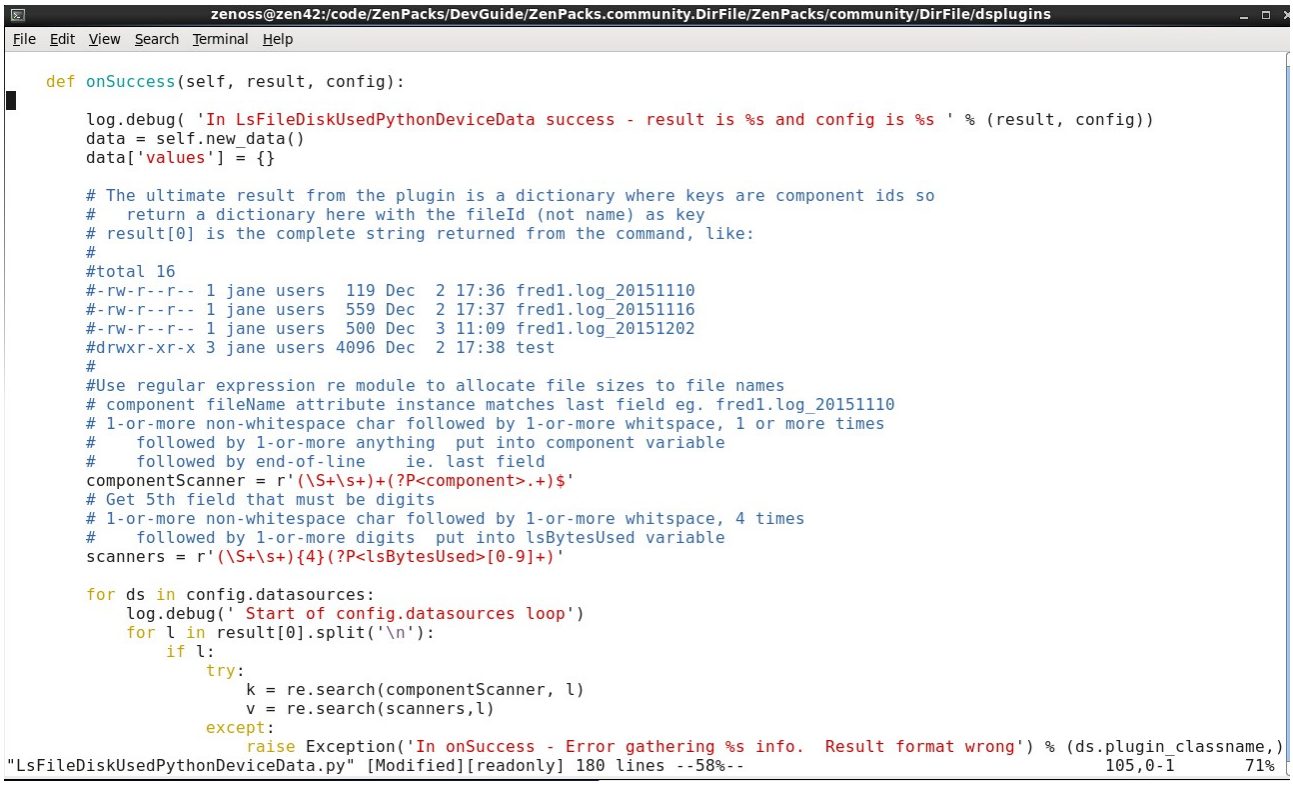


Figure 180: LsFileDiskUsedPythonDeviceData onSuccess method - part 1 - parsing the data for each component

Again note that having an example of the output that is to be parsed, is incredibly useful in the code.

The entire stdout from the command is split on the newline ('\n') character.

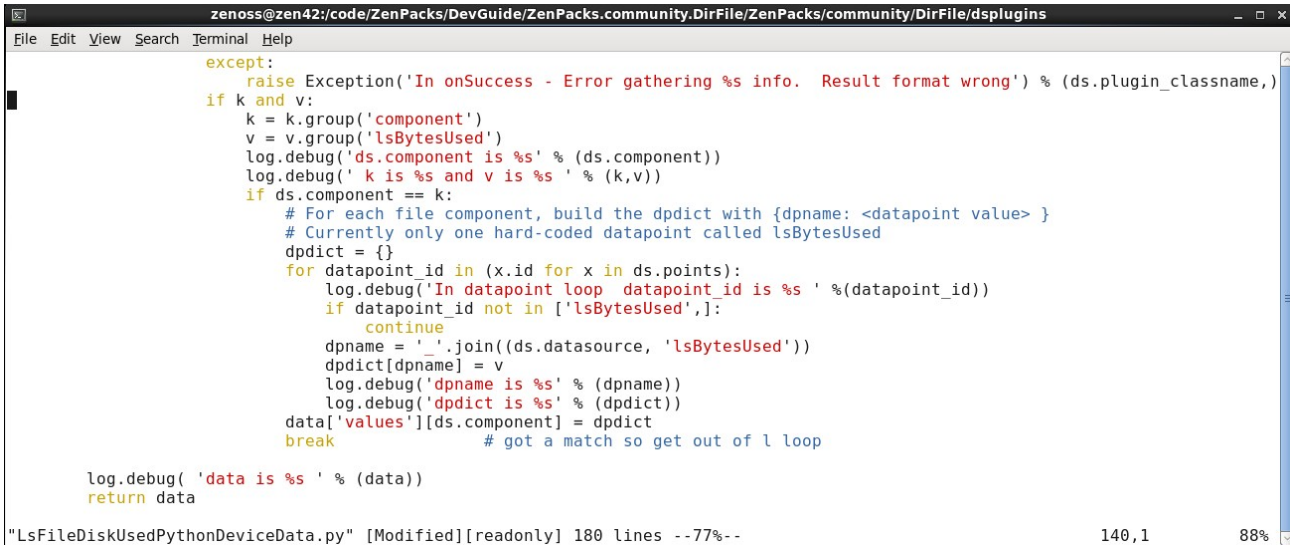


Figure 181: LsFileDiskUsedPythonDeviceData onSuccess method - part 2 - applying the parsed data to datapoints

In Figure 181 if the output line being processed has valid component id and value fields, then the component is compared with the current datasource component. If the datasource has a (hard-coded) datapoint called *lsBytesUsed* then the *dpdict* dictionary is populated with *{dpname: value}*.

As with previous examples, this process is slightly over-complex to provide for further expansion of datasources and datapoints. Since this example is looking for a filename match from an *ls -l* listing, once a match has been achieved there is no point in examining further lines in the *ls - l* listing; hence the *break*.

As usual, the *data['values']* dictionary is populated for each file component.

The onError method is virtually identical to previous Python plugin examples.

Restart *zenhub*, *zopectl* and *zenpython* to test the new plugin.

13.3.5.3 Performance template to drive the PythonDataSourcePlugin

A new performance **template** *FileLsDiskUsedPythonXml* will be created to drive the new plugin. The **datasource** is called *FileLsDiskUsedPython* and, as with the COMMAND datasource a specific **datapoint** *lsBytesUesd* must be hard-coded to match the datapoint code in the onSuccess method.

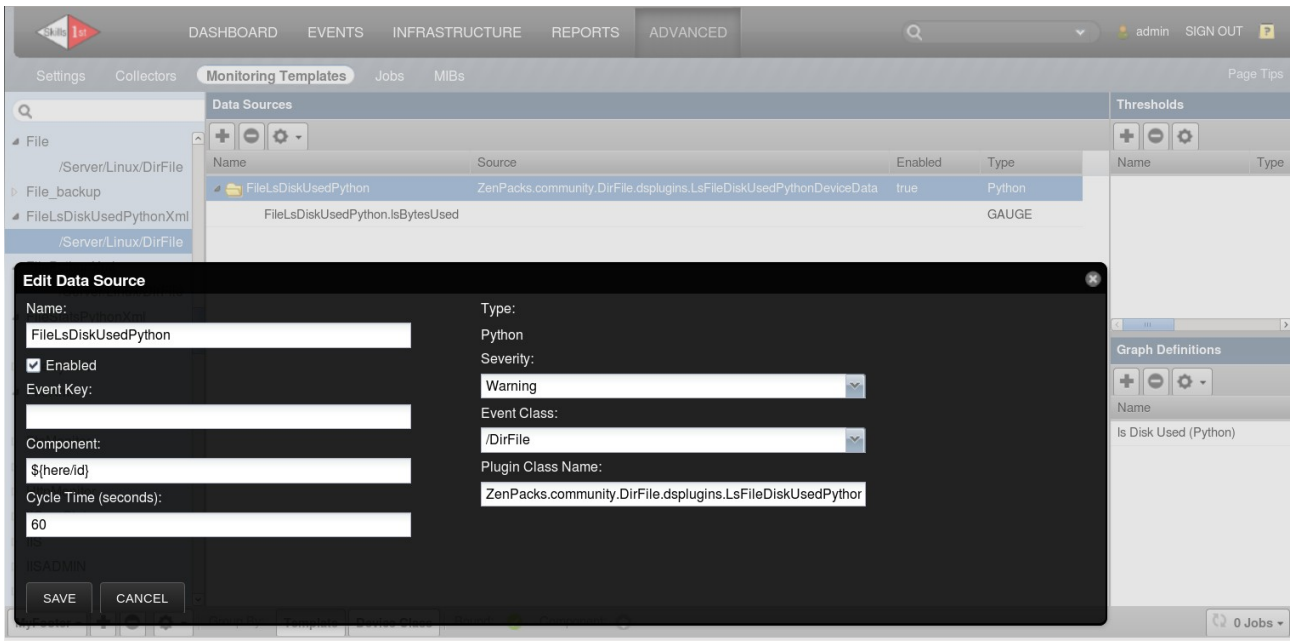


Figure 182: Template to drive LsFileDiskUsedPythonDeviceData plugin

This template can be automatically bound to *File* object classes by adding it to the *monitoring_templates* attribute for *File* in *zenpack.yaml*:

```
monitoring_templates: [File, FileXml, FilePythonXml, FileStatsPythonXml, FileLsDiskUsedPythonXml]
```

Note that for each template associated with a new Python plugin, the naming convention has included “Python” to distinguish from similar templates that are driven by COMMAND datasources.

Similarly, the graphs created have all included Python on the title so that data collected by the two different methods can be distinguished and compared.

Thus far, each of the new Python-based templates should be added to the ZenPack's *objects.xml* file by adding the template through the GUI options and re-exporting the ZenPack.

13.3.6 Collecting component performance data; customized datasource to pass customized key values

The last sample in the COMMAND section discussed creating a complete new datasource in section 12.3.4. It extended the word-checking scenario so that strings to be searched for were configurable in the datasource, and each datasource has a single datapoint for the string to be checked.

The identical **remote** script, *file_stats_param.sh,* will be run to deliver string match results, driven by a **local** ssh script, *file_stats_param_ssh.sh*.

```
#!/bin/bash
#
# Runs command, file_stats_param.sh, on remote target to return number of lines
#  in file containing supplied string parameter
# file_stats_param.sh expects two parameters - filename and string
#
```

```
# Returned output in format:
# " File string count test ok | matches= $stringCount1
# or error output starting with "Error"
#
# Param 1 = ssh user
# Param 2 = keypath
# Param 3 = target
# Param 4 = file to test
# Param 5 = path to file_stats.sh
# Param 6 = string to search for
#set -x

echo "$1 $2 $3 $4 $5 $6" > /tmp/filestatsparam.tmp
# Ensure that the string is quoted to preserve spaces
CMD="$5/file_stats_param.sh $4 \"$6\""
# file_stats_param_ssh.sh zenplug ~/.ssh/id_dsa taplow-11
#  /opt/zenoss/local/fredtest/fred1.log_20151202 /home/zenplug "test 2"
ssh -l "$1" -i "$2" "$3" "$CMD"
```

13.3.6.1 Building the Python datasource

The new datasource in the ZenPack's *datasources* directory, will be *DirFilePythonDataSource.py*. This file requires very little new development as it is largely a combination of the existing datasource developed for the COMMAND scenario, with the *PythonDataSourcePlugin* developed in *dsplugins/FileStatsPythonDeviceData.py*. The only major difference is that the new datasource will inherit from the *PythonDataSource* class rather than from the *BasicDataSource*.

The *PythonDataSource* can be inspected from the PythonCollector ZenPack base directory under the *datasources* directory. Standard properties for a *PythonDataSource* are:

```
plugin_classname = None

# Defined instead of inherited to change cycletime type to string.
_properties = (
  {'id': 'sourcetype', 'type': 'selection', 'select_variable': 'sourcetypes', 'mode': 'w'},
  {'id': 'enabled', 'type': 'boolean', 'mode': 'w'},
  {'id': 'component', 'type': 'string', 'mode': 'w'},
  {'id': 'eventClass', 'type': 'string', 'mode': 'w'},
  {'id': 'eventKey', 'type': 'string', 'mode': 'w'},
  {'id': 'severity', 'type': 'int', 'mode': 'w'},
  {'id': 'commandTemplate', 'type': 'string', 'mode': 'w'},
  {'id': 'cycletime', 'type': 'string', 'mode': 'w'},
  {'id': 'plugin_classname', 'type': 'string', 'mode': 'w'},
  )
```

Note that cycletime is redefined here to be a *string* rather than an *int*. *plugin_classname* is unique to the *PythonDataSource*.

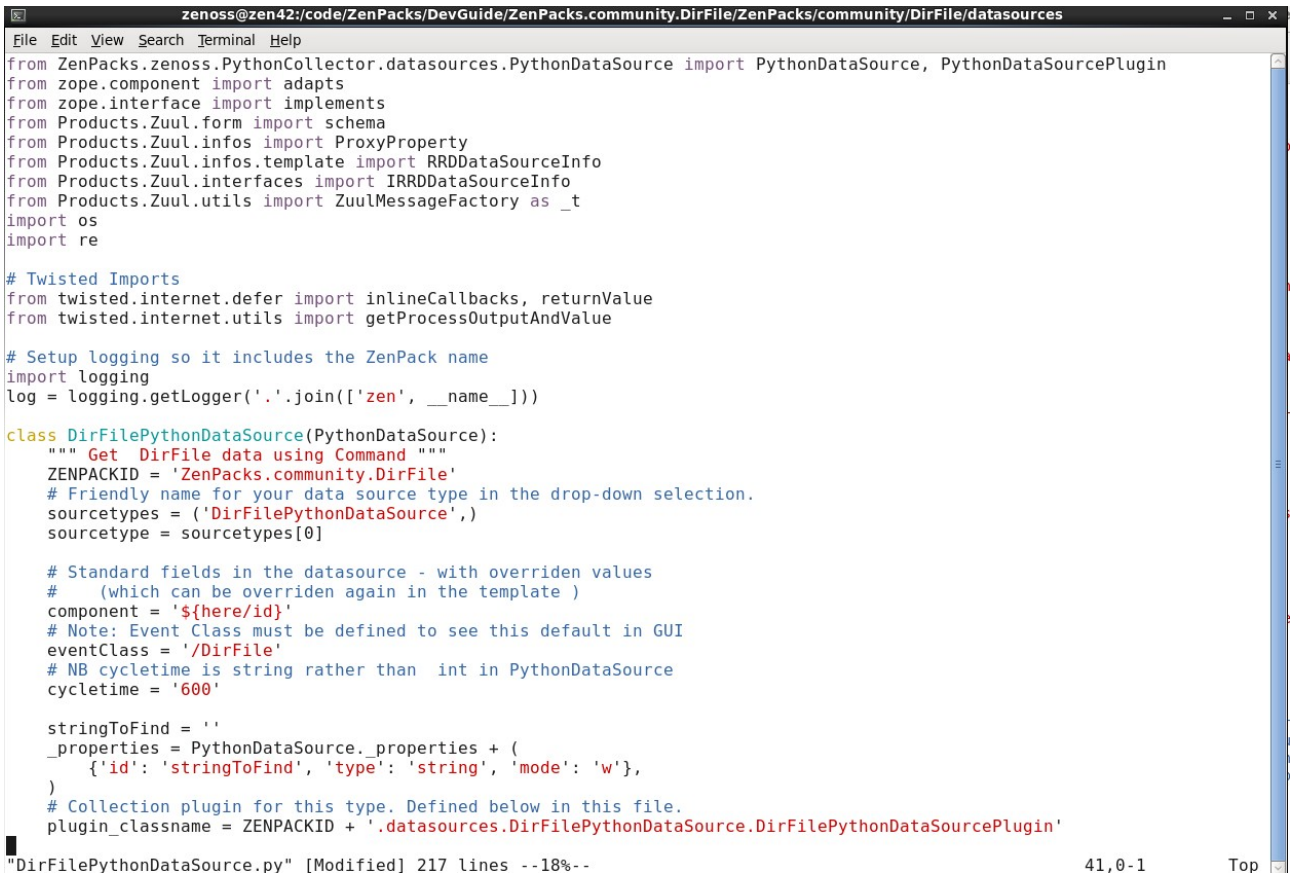


Figure 183: DirFilePythonDataSource.py - part 1

In Figure 183 note that:

- *PythonDataSourcePlugin* is imported from the Python Collector ZenPack
- The info and interface imports are from ***RRDDataSourceInfo*** and ***IRRDDataSourceInfo*** respectively. The COMMAND-based datasource used *CommandDataSourceInfo* and *ICommandDataSourceInfo*.
- The new datasource will have a “friendly name” of *DirFilePythonDataSource* (in the ***sourcetypes*** statement).
- The default cycletime is set to 600s.
- A new attribute is defined for the datasource, *stringToFind*, which will appear in the GUI as a parameter to be supplied.
- The *plugin_classname* attribute is set as the path to this ZenPack with *.datasources.DirFilePythonDataSource.DirFilePythonDataSourcePlugin* appended; that is, follow the *datasources* directory, into the *DirFilePythonDataSource* file (module) to the *DirFilePythonDataSource* class.

A datasource defines new elements of the GUI so needs to have info and interfaces entries.

Figure 184: DirFilePythonDataSource - part 2 - info and interface definitions

The rest of the datasource file defines the *PythonDataSourcePlugin* and is almost identical to that in section 13.3.4. The only changes are:

- The *PythonDataSourcePlugin* is called ***DirFilePythonDataSourcePlugin*** to match the definition earlier in the datasource file.
- The params method has an extra element which passes the *stringToFind* attribute from the GUI to the *zenpython* daemon.

```
params['stringToFind'] = datasource.talesEval(datasource.stringToFind, context)
```

- The collect method has cmd and args modified for the new script:

```
# script is file_stats_ssh.sh taking 6 parameters, zCommandUsername, keyPath,
#      host address, fileName, zCommandPath, searchstring
cmd = os.path.join(libexecdir, 'file_stats_param_ssh.sh')
args = ( ds0.zCommandUsername, keyPath, ds0.manageIp, fileName,
       ds0.zCommandPath, ds0.params['stringToFind'])
```

- The onSuccess method is modified slightly as the remote script returns a **single** <varName>=<varValue> pair where the *varName* is *matches*.

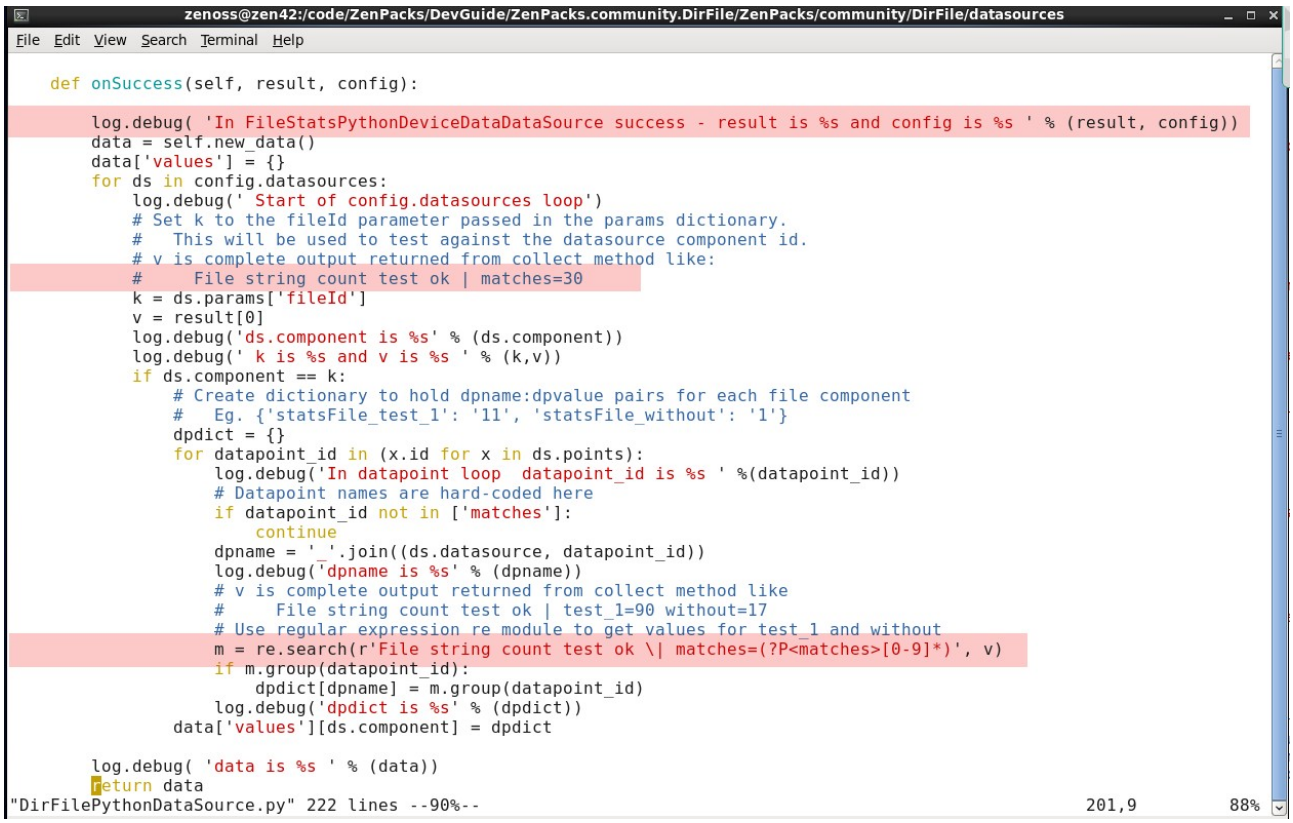


Figure 185: DirFilePythonDataSource onSuccess method

The datasource file has added new ***info*** and ***interface*** definitions so *configure.zcml* in the ZenPack's base directory, must be edited to “glue” these in.

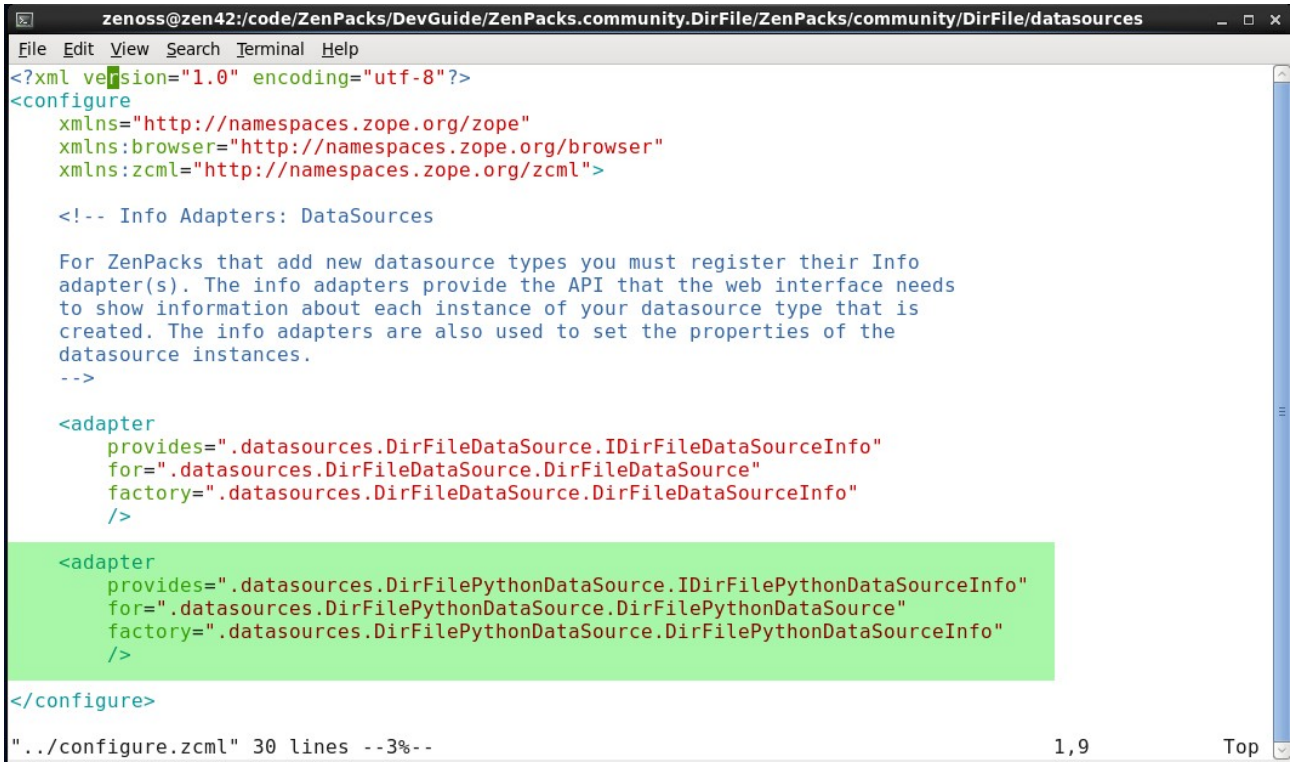


Figure 186: Modifications to configure.zcml for the DirFilePythonDataSource datasource

13.3.6.2 Deploying the new datasource

Having created a new datasource, the ZenPack must be reinstalled and zenoss completely stopped and restarted.

The first test is to create a new performance template, *FileStatsParamPythonXml* with a device class location of */Server/Linux/DirFile*. Within the template, create a new datasource and check that the new Python datasource is in the dropdown list. The convention for this ZenPack is to call the datasource <the string to search for>Python, eg. *withoutPython*.

Remember that datasource instances need to be unique because, prior to Zenoss 5, the RRD data for each device is saved in a filename that concatenates **datasource_datapoint** eg. *withoutPython_matches.rrd*.

A *matches* datapoint will need creating for each datasource (to match the datapoint name used in the onSuccess method).

Graphs should have a unique title to distinguish them from graphs created by earlier examples.

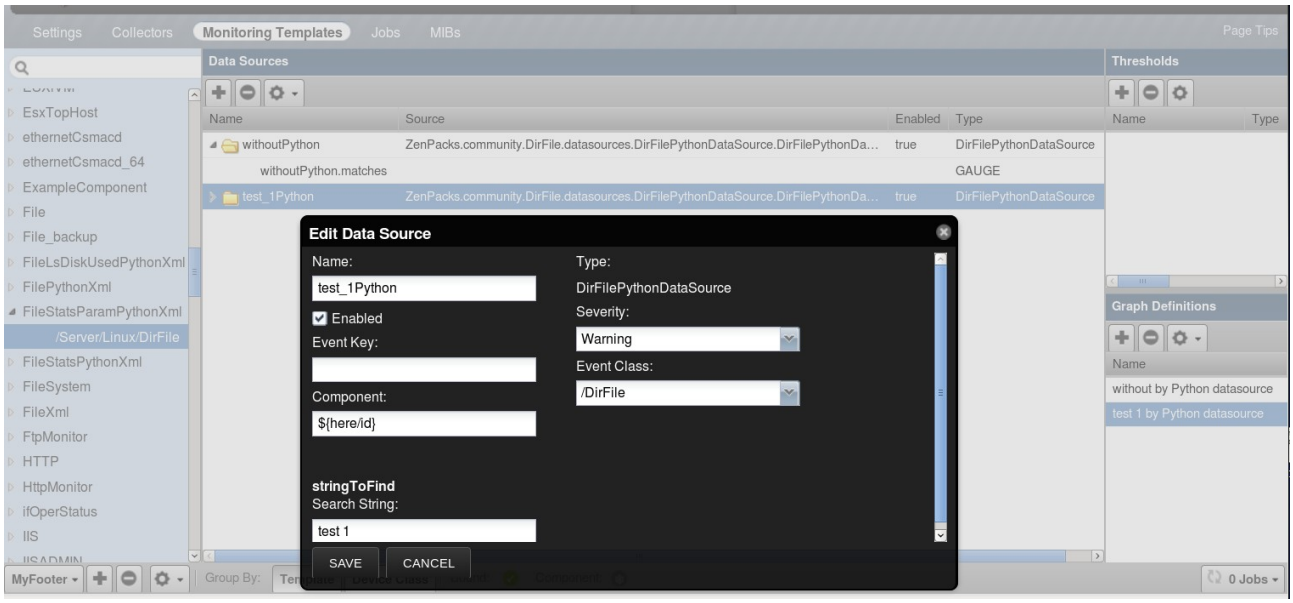


Figure 187: Template to drive new DirFilePythonDataSource

Don't forget to add this template to the ZenPack using the GUI and re-export the ZenPack.

To ensure that the template is bound automatically to all *File* object classes, re-edit *zenpack.yaml* and add this template to the *monitoring_templates* attribute.

```
monitoring_templates: [File, FileXml, FilePythonXml, FileStatsPythonXml,
       FileLsDiskUsedPythonXml, FileStatsParamPythonXml]
```

13.4 Converting the modeler to use the PythonCollector ZenPack

There are advantages and disadvantages with the COMMAND-based modeler plugin. On the plus side, it inherently handles the ssh parameters and the transport session. On the minus side, there is no mechanism for passing other parameters to the command to be run.

The COMMAND-based plugin specified the command as:

```
command = (
    'find /opt/zenoss/local -type d ;'
    #'find / -type d ;'
    'echo __SPLIT__; '
    'find /opt/zenoss/local -type f'
    #'find / -type f'
    )
```

Although the ZenPack provides for *zMonitorDir1* and *zMonitorDir1File* properties, there is no simple way to pass these parameters into the command; hence the extremely wasteful use of the *find* command above.

Using a *PythonPlugin* in the modeler means that the ZenPack writer creates the **collect** method with all the flexibility that has been seen with datasources.

Reiterating again, there are often better ways of getting twisted deferred data than using *getProcessOutputAndValue()* to run ssh sessions but since we have this mechanism and the ssh driver scripts, a *PythonPlugin* modeler will be created to demonstrate the mechanism.

The existing *dudir_ssh.sh* takes a fully-qualified directory name as a parameter , along with the username, keypath and device target, and returns:

```
<bytes used>  <Fully qualified filename>
```

if the directory exists; otherwise an error is returned. This will be used to model directories. In a later development, the <*bytes used*> will also be incorporated.

The existing *lsFileDiskUsed_ssh.sh* takes the same parameters and delivers an *ls -l* output for the directory supplied. This will be used to model files.

Under the ZenPack's *modeler/plugins* hierarchy, create a *python* subdirectory and *touch* an *__init__.py* file. The modeler file will be *DirFilePythonMap.py*.

13.4.1 Imports

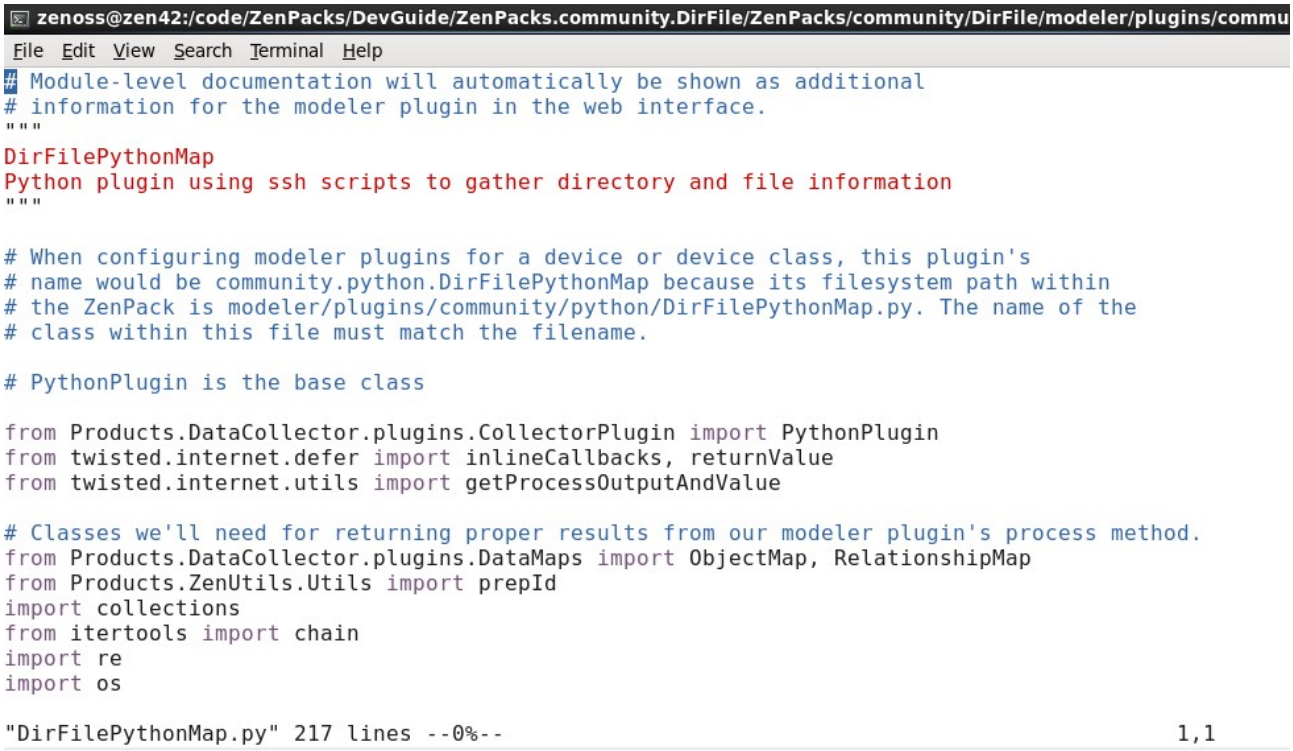


Figure 188: DirFilePythonMap imports

The modeler plugin file needs imports for the usual twisted modules and utilities. It also specifically needs to import *PythonPlugin* from *Products.DataCollector.plugins.CollectorPlugin*.

⭐ Note that *PythonPlugin* is now provided in the core code; this definition is not in the PythonCollector ZenPack. The definition mandates **collect** and **process** methods as a minimum.

```
class PythonPlugin(CollectorPlugin):
  """
  A PythonPlugin defines a native Python collection routine and a parsing
  method to turn the returned data structure into a datamap. A valid
  PythonPlugin must implement the collect and process methods.
  """
  transport = "python"

  def collect(self, device, log):
    """Dummy collector to be implemented by the actual collector.
    """
    pass
```

13.4.2 Creating a dirRegex directory from zProperties

The same code as was used in the COMMAND modeler will be used to build a dictionary of directories and file regular expressions.

Figure 189: DirFilePythonMap create_dirRegex function

The function is passed the device parameter so has access to all attributes, methods and zProperties for that device. The result is a dictionary where keys are the fully-qualified directory represented by the zMonitorDir<x> property and the values are the file regex associated with the zMonitorDir<x>File. For example:

```
{'/opt/zenoss/local/fredtest': 'fred1.*',
'/opt/zenoss/local/fredtest/test': 'fred2\\.log.*'}
```

13.4.3 DirFilePythonMap class attributes

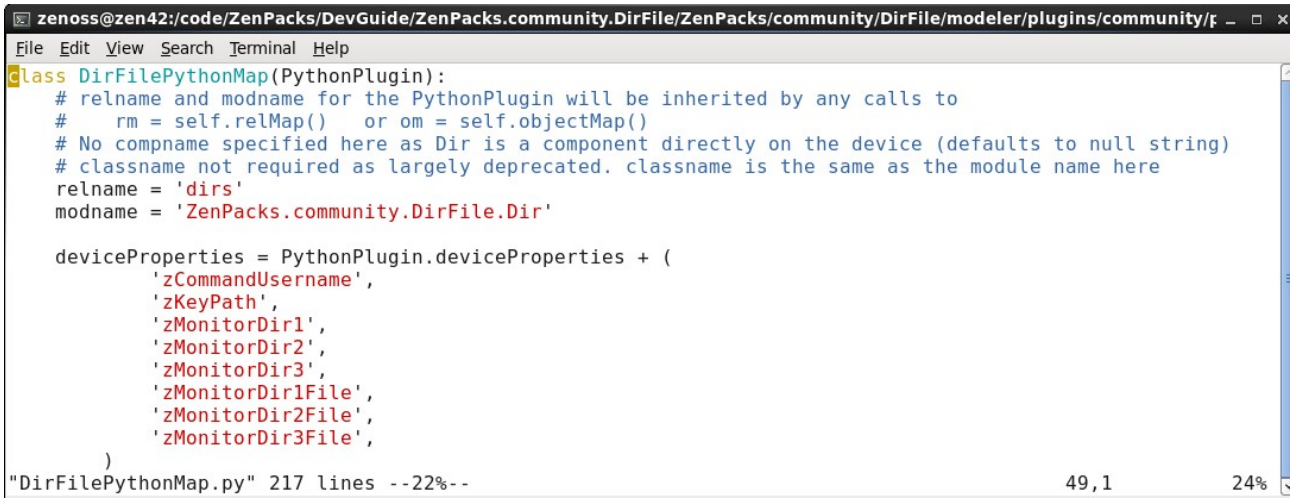


Figure 190: DirFilePythonMap attributes

The modeler will build directory components of object class *Dir* (found in *ZenPacks.community.DirFile)* by fulfilling the device relation of *dirs*.

The device properties pertaining to ssh communications and the file and directory properties, are all made available to the *PythonPlugin*.

13.4.4 collect method

The unique method for the PythonPlugin is **collect**.

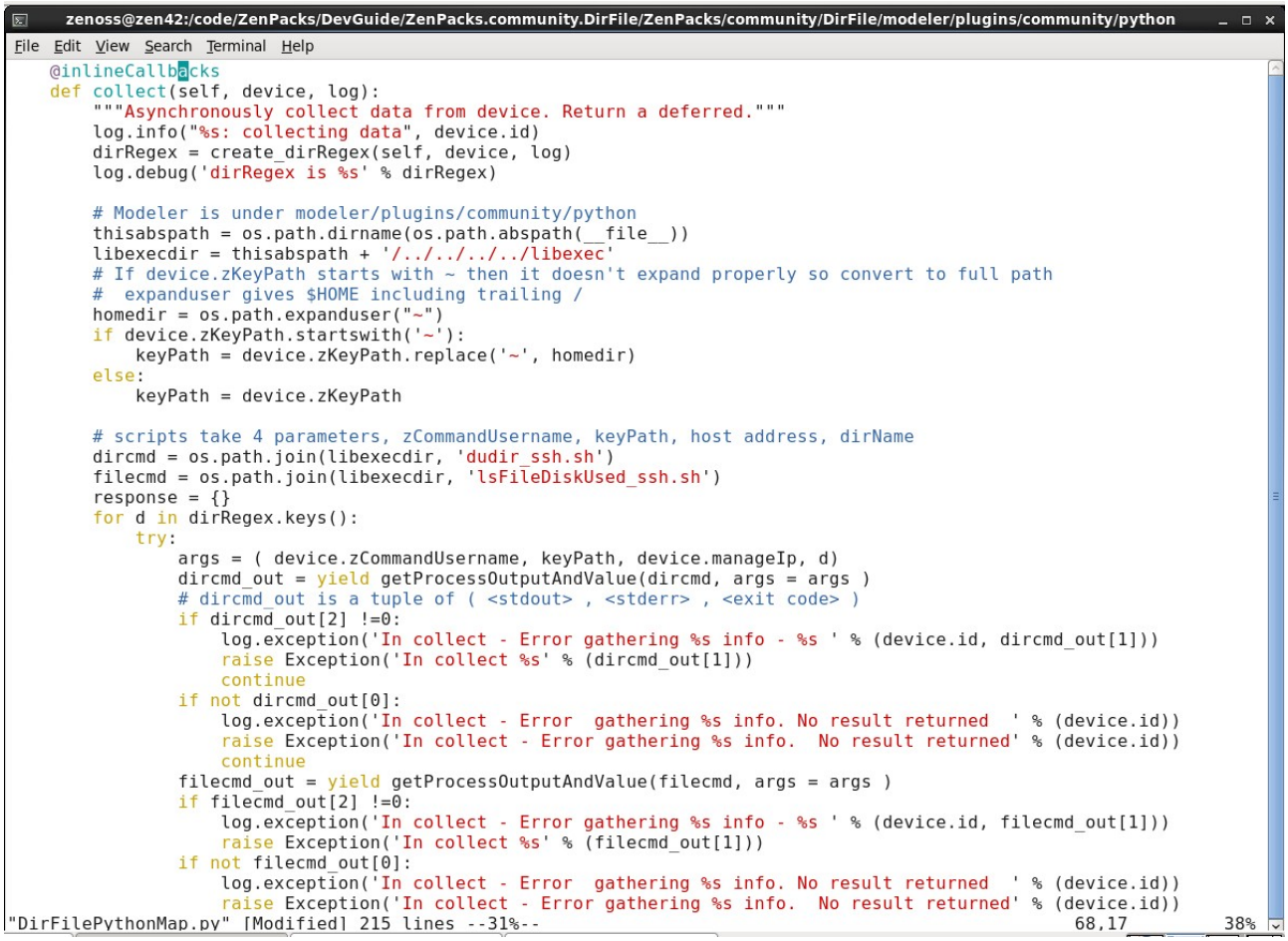


Figure 191: DirFilePythonMap collect method - part 1

Much of this code is lifted from earlier *PythonDataSourcePlugin* methods to establish the ssh keypath and the *libexec* path to the ssh file that the modeler will run.

A *dircmd* and *filecmd* are setup:

```
dircmd = os.path.join(libexecdir, 'dudir_ssh.sh')
filecmd = os.path.join(libexecdir, 'lsFileDiskUsed_ssh.sh')
```

A loop is established for *dirRegex.keys()*; that is each zMonitorDir<x> property. This ensures a small number of iterations. *args* includes the name of the directory for this iteration. *yield getProcessOutputAndValue(dircmd, args = args)* is used to deliver the **twisted Deferred** results. As with dsplugin methods, the output is checked for valid stdout values for both *dircmd* and *filecmd*.

```
response = {}
for d in dirRegex.keys():
  try:
    args = ( device.zCommandUsername, keyPath, device.manageIp, d)
    dircmd_out = yield getProcessOutputAndValue(dircmd, args = args )
    # dircmd_out is a tuple of ( <stdout> , <stderr> , <exit code> )
    if dircmd_out[2] !=0:
```

```
            log.exception('In collect - Error gathering %s info - %s ' %
                 (device.id, dircmd_out[1]))
            raise Exception('In collect %s' % (dircmd_out[1]))
            continue
          if not dircmd_out[0]:
            log.exception('In collect - Error gathering %s info. No result
                  returned ' % (device.id))
            raise Exception('In collect - Error gathering %s info. No
                  result returned' % (device.id))
            continue
          filecmd_out = yield getProcessOutputAndValue(filecmd, args = args )
          if filecmd_out[2] !=0:
            log.exception('In collect - Error gathering %s info - %s ' %
                  (device.id, filecmd_out[1]))
            raise Exception('In collect %s' % (filecmd_out[1]))
          if not filecmd_out[0]:
            log.exception('In collect - Error gathering %s info. No result
                  returned ' % (device.id))
            raise Exception('In collect - Error gathering %s info. No
                  result returned' % (device.id))
```

The successful output of this part of the collect method should be:

- *dircmd_out* a tuple of (<stdout> , <stderr> , <exit code>) from *dudir_ssh.sh*
- *filecmd_out* a tuple of (<stdout> , <stderr> , <exit code>) from *lsFileDiskUsed_ssh.sh*

A *response* dictionary has also been initialised.

The end of the collect method populates the *response* dictionary or raises an exception.

```
for d in dirRegex.keys():
  try:
    .
    .
    # Check dircmd stdout. Should be like:
    #  12345 /opt/zenoss/local/fredtest
    # Get the bytes used into named group bytesUsed
    # Pass the filecmd stdout as the fileOutput element to be sorted out by process
    check = re.search(r'(?P<bytesUsed>[0-9]+)\s+/\S+$', dircmd_out[0])
    if check:
      response[d] = {'bytesUsed': check.group('bytesUsed'), 'fileOutput': filecmd_out[0]}
    #  log.debug('response is %s ' % (response))
  except Exception, e:
    log.error(
      "%s: %s", device.id, e)
    continue
#log.debug('Response is %s \n' % (response))
returnValue(response)
```

dircmd is checked against a regular expression to ensure a valid response. The size is extracted into the *bytesUsed* named group; the entire response from *ls -l* is passed as the second element of the entry in the *response* dictionary. *response* is a twisted deferred result. For example:

```
Response is {'/opt/zenoss/local/fredtest': {'fileOutput': 'total 20\n-rw-
r--r-- 1 jane users 126 Jan 14 14:40 fred1.log_20151110\n-rw-r--r-- 1 jane
users 434 Jan 14 14:40 fred1.log_20151116\n-rw-r--r-- 1 jane users 1047 Jan
14 14:41 fred1.log_20151202\n-rw-r--r-- 1 jane users 961 Jan 18 19:10
fred1.log_20160118\ndrwxr-xr-x 3 jane users 4096 Dec 3 19:17 test\n',
```

```
'bytesUsed': '21251'}, '/opt/zenoss/local/fredtest/test': {'fileOutput':
'total 12\n-rw-r--r-- 1 jane users 499 Dec 2 17:38 fred2.log_20151124\n-rw-
r--r-- 1 jane users 499 Dec 3 19:17 fred2.log_20151125\ndrwxr-xr-x 2 jane
users 4096 Nov 29 18:17 lowertest\n', 'bytesUsed': '14587'}}
```

13.4.5 process method

The process method of *DirFilePythonMap.py* is very similar to the previous incarnation with the COMMAND plugin except that the processing of the output has now been simplified.

zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.DirFile/ZenPacks/community/DirFile/modeler/plugins/community/python

File Edit View Search Terminal Help

```
    def process(self, device, results, log):
        log.info("Modeler %s processing data for device %s",
            self.name(), device.id)
        # Create dictionary where key is directory and value is file regex
        dirRegex = create_dirRegex(self, device, log)

        # Setup an ordered collection of dictionaries to return data to the ApplyDataMap routine of zenmodeler
        maps = collections.OrderedDict([
            ('dirs', []),
            ('files', []),
        ])
        # Instantiate a relMap.  This inherits relname and compname from the plugin.
        rm = self.relMap()

        for k, v in results.items():
                dir_id = prepId(k)
                # Add an Object Map for this directory
                # Use prepId to ensure id is unique and doesn't include any dodgy characters like /
                # om = self.objectMap() inherits modname and compname (null) from plugin
                om = self.objectMap()
                om.id = dir_id
                om.dirName = k
                om.bytesUsed = int(v['bytesUsed'])
                for k1, v1 in om.items():
                    log.debug('dir om key is %s and value is %s' % (k1, v1))
                rm.append(om)
                # For this directory, create a map for associated files, passing this dir_id as part of compname
                fm = (self.getFileMap( device, v['fileOutput'],  dirRegex, k, 'dirs/%s' % dir_id, log))
                maps['files'].append(fm)
        if len(rm.maps) > 0:
            pass
        else:
            log.info('No matching dirs found on %s ' % (device.id))
            return None

        # Add the rm relationships to maps['dirs']
        maps['dirs'].append(rm)

        # Need this complicated setup with maps = collections.OrderedDict and the chain return
        #   to ensure that relationship maps are applied in the correct order.  Otherwise there tend
        #   to be issues trying to create relationships on objects that don't yet exist
        return list(chain.from_iterable(maps.itervalues()))
"DirFilePythonMap.py" [Modified] 209 lines --62%--                    130,9         75%
```

Figure 192: DirFilePythonMap process method

The *results* dictionary has directory name as keys. *results* values are dictionaries of *bytesUsed* and the file command output, which is passed to the *getFileMap* function.

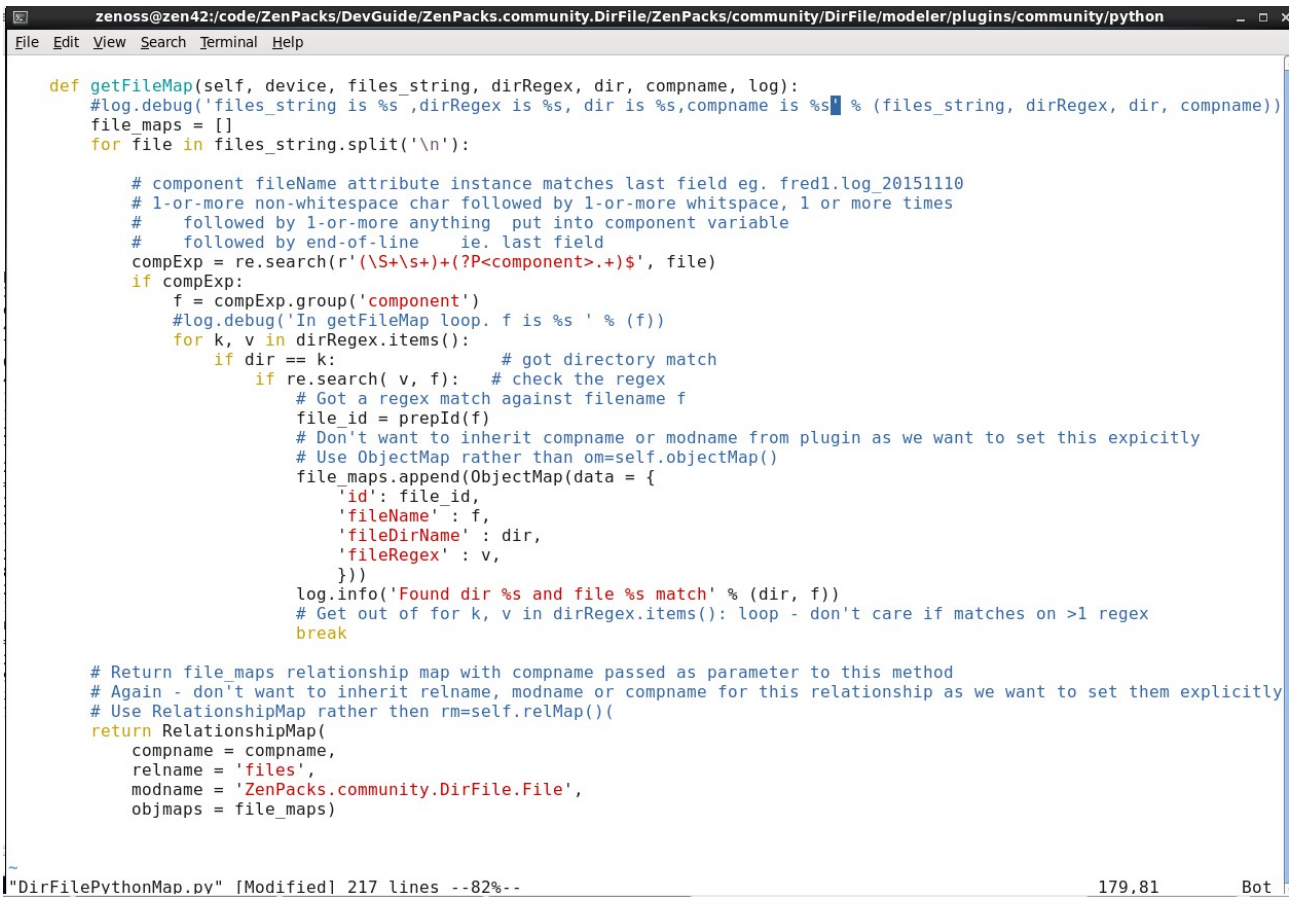


Figure 193: DirFilePythonMap getFileMap function in process method

The output string is split on newlines and the last element of the line (the filename) is then compared against the regex file expression for the matching directory. If the output line matches then a *File* component object is created and added to the list of *file_maps*.

A *RelationshipMap* is returned to instantiate the *files* relationship using the object class definition in *ZenPacks.community.DirFile.File* with the component created from the concatenation of '*dirs*/' and the directory id (passed in the *compname* parameter in the previous screenshot); eg. *dirs*/*opt_zenoss_local_fredtest* .

The end result is identical to that achieved with the COMMAND plugin.

13.4.6 Testing the new modeler

When testing, ensure that the COMMAND plugin *DirFileMap* is removed from the modeler plugin list of test devices and add *DirFilePythonMap*. Delete all existing components by selecting them and using the middle icon (-) at the top of the GUI. This will **not** delete existing performance data for the components.

The only daemons that need recycling are *zenhub, zopectl* and *zenpython*. Ultimately *zenmodeler* will also need recycling but it can be run in one-off mode for testing.

Remodel with:

```
zenmodeler run -v 10 -d taplow-11.skills-1st.co.uk --collect DirFilePythonMap > /tmp/fred1 2>&1
```

and check the output file. Add *log.debug* statements in for debugging.

The first test is that the new modeler does appear in the available list of modeler plugins for a device. If this does not appear or if the zenmodeler output appears to have skipped the new modeler, check *zenhub.log* for errors.

13.5 Combining performance data and modeler data

Some monitoring templates shipped with other ZenPacks and with the core product, retrieve **performance** data as part of the **modeler** cycle. For example, the Interfaces component includes Administrative Status and Operational Status; the File Systems component includes Used Bytes and Free Bytes. A snapshot of these values can be useful but, given that *zenmodeler* typically only runs once or twice a day, these values in the top half of a component display, may be very out of date.

Fundamentally, these values are held as attributes of the component object so are updated by the modeler plugin applying *ObjectMaps*. The PythonCollector ZenPack provides a way to update such maps as part of the *PythonDataSourcePlugin*.

Remember that the result returned from a *PythonDataSourcePlugin* is a dictionary with *events*, *values* and *maps* elements. There is a good example demonstrated in the zenpacklib documentation at http://zenpacklib.zenoss.com/en/latest/tutorial-http-api/datasource-plugin-model-5.html .

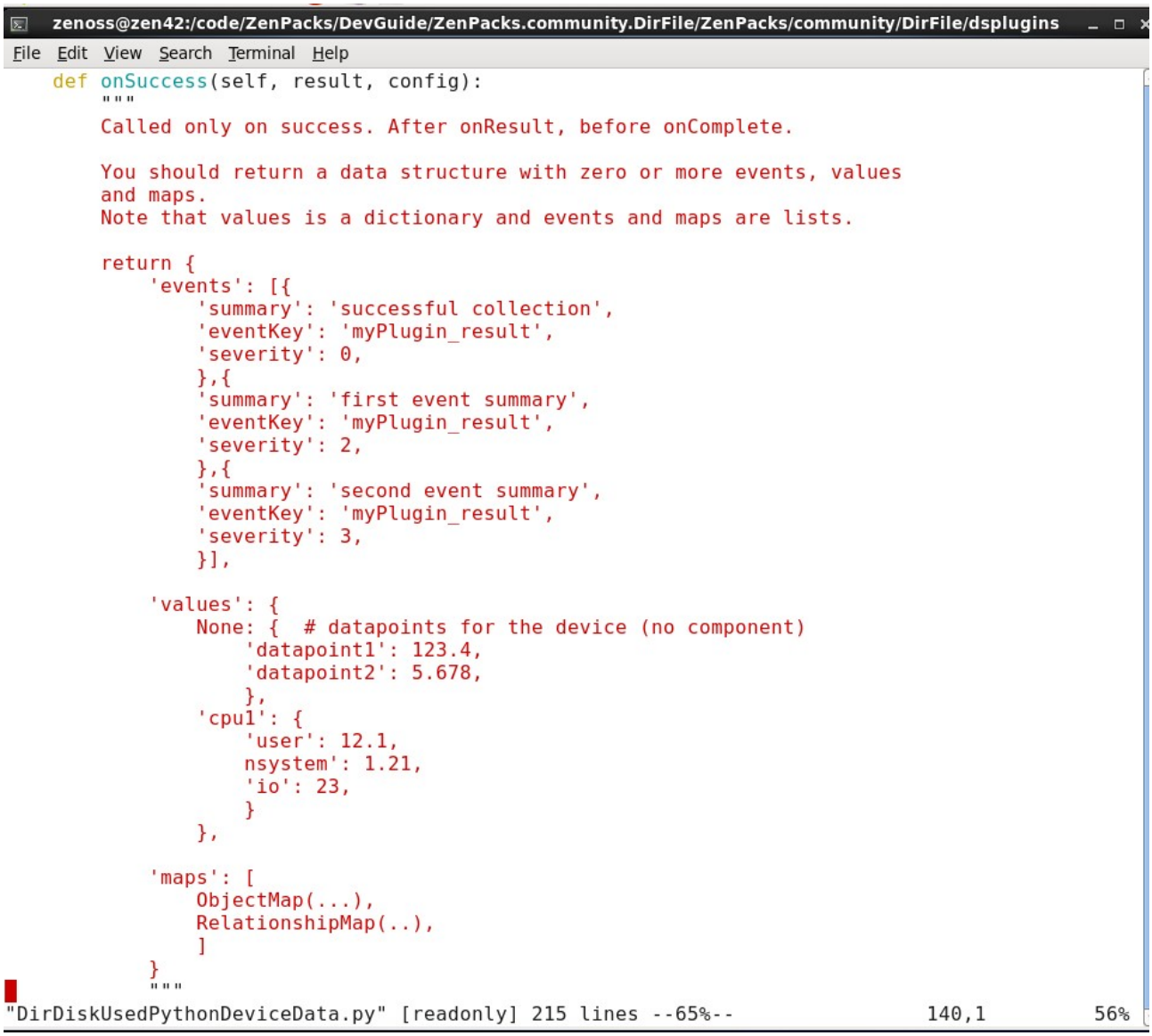


Figure 194: Datastructure returned by a PythonDataSourcePlugin

The *DirDiskUsedPythonDeviceData* plugin could easily be used to populate a new *bytesUsed* attribute of a *Dir* component object.

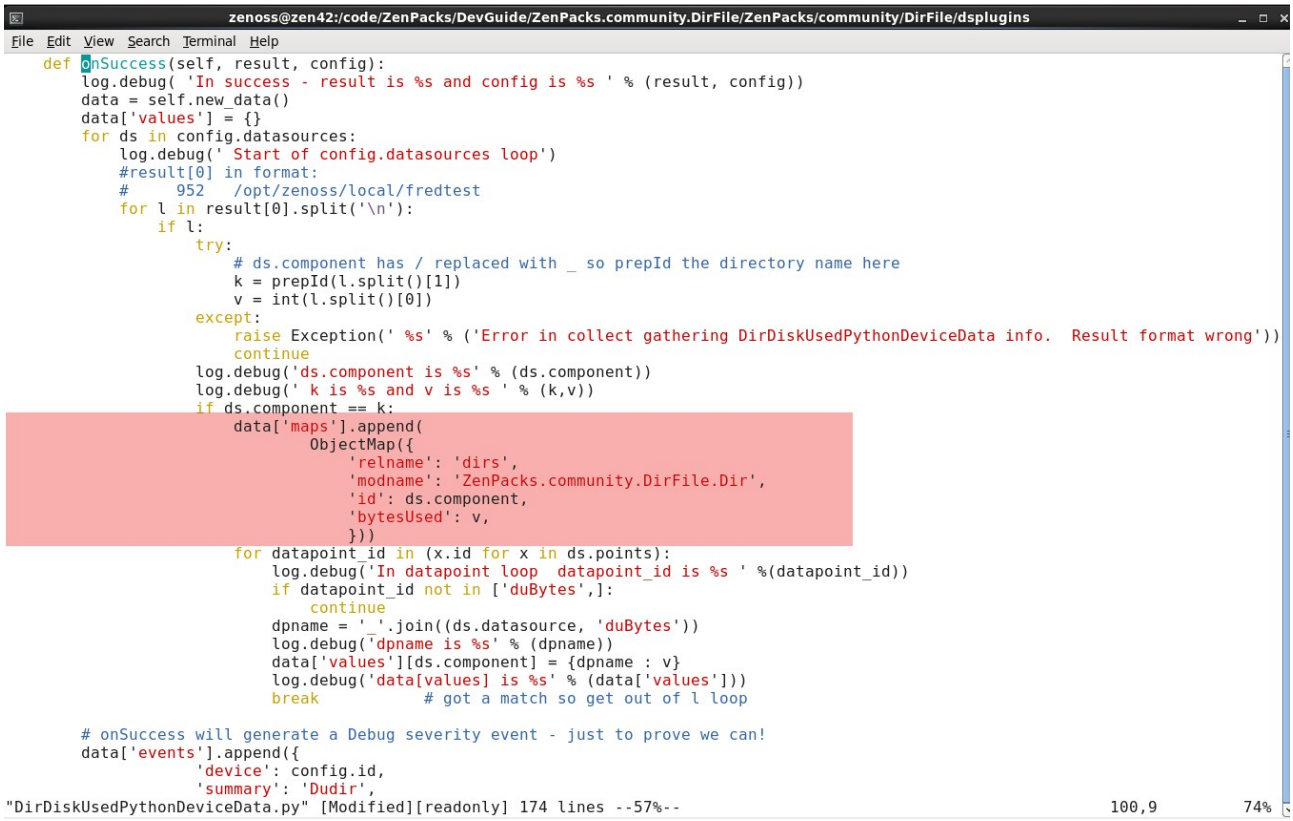


Figure 195: Creating a data['maps'] element for a PythonDataSourcePlugin

Exactly the same parameters are used in the **datasource** plugin as were used in the **modeler** plugin. The directory *id* field is the datasource component; the *bytesUsed* field is readily available in the plugin.

Don't forget to import the *ObjectMap* method from *Products.DataCollector.plugins.DataMaps* at the top of the file.

zenpack.yaml will need modifying to create the *bytesUsed* attribute of the *Dir* object class:

```
Dir:
 label: Dir # NB It is label, with spaces removed, that is used to match a component template
 meta_type: Dir  # Will default to this but in for completeness
 label_width: 150  # This controls the column width for Dir in the Files component display
 order: 60  # before file
 auto_expand_column: dirName
 monitoring_templates: [Dir, DirPythonXml] # will default to Dir but explicit for clarity

 properties:
  dirName:
   type: string
   label: Directory name
   short_label: DirName
   label_width: 300
   order: 3.1

  bytesUsed:
   type: int
   label: Bytes
   short_label: Bytes
   label_width: 100
   order: 3.2
```

The ZenPack should be reinstalled and Zenoss completely restarted, having modified *zenpack.yaml*. The result should be that the *Dirs* component display shows a *Bytes* field that is instantiated and updated by the modeler but is also updated by the datasource plugin on each cycle.

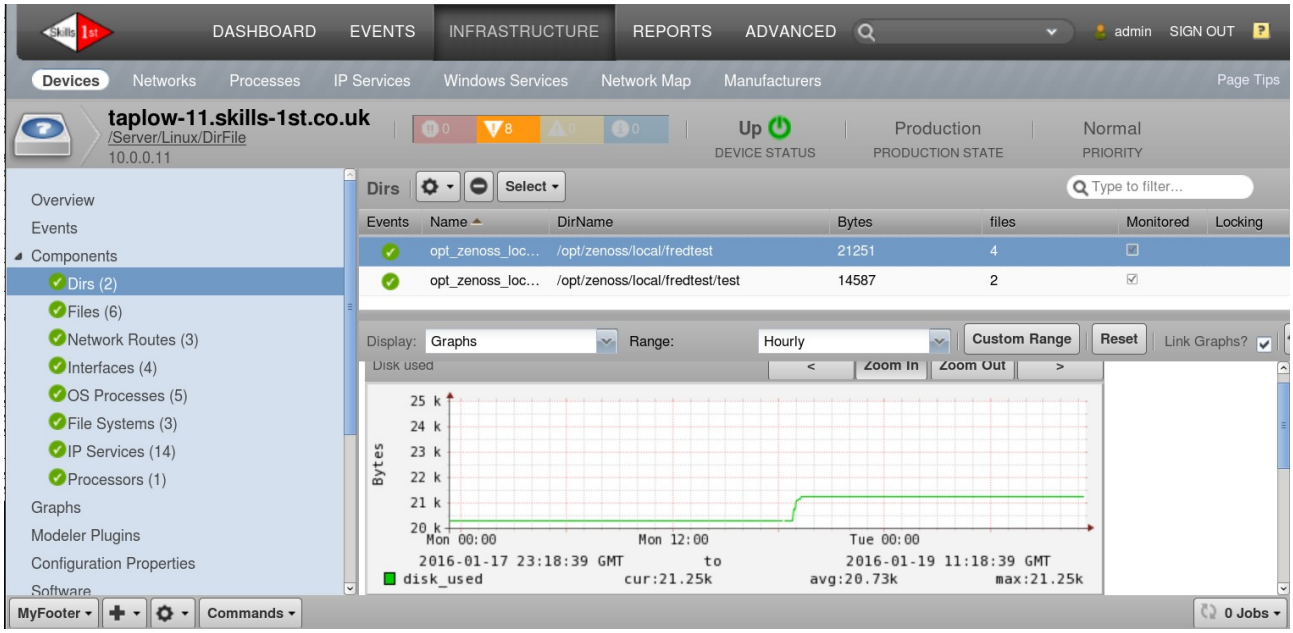


Figure 196: Dirs components with Bytes field

Remember that ZenPacks built with zenpacklib have another alternative for displaying performance data with the configuration data. The *datapoint* keyword described in section 10.3.6 is a much simpler and less resource-intensive approach.

Making changes to modeled attributes in the ZODB database during performance collection can (and usually does) cause a huge amount of configuration invalidation churn that is entirely unnecessary and may impact performance.

14.0 Events in ZenPacks

It is good practice to include new events in a ZenPack if new conditions have been created. It is easiest to include Event Class definitions in the *objects.xml* file by simply adding from the GUI menus.

14.1 Detecting duplicate events

Several standard fields of an event are key to **detecting duplicate** events and to implement event **auto-clearing** mechanism.

Fields that contribute to the event **fingerprint** and are used in the **dedupid** field for detecting repeated events, are:

- device
- component
- eventClass

- eventKey
- severity
- [summary]

If the ***eventKey*** field is null then ***summary*** becomes part of the fingerprint; otherwise *summary* is not considered. (Note it is ***eventKey***, not ***eventClassKey*** that is used here).

Standard templates for many datasource types, including *COMMAND* and *Python*, include fields in the GUI dialogue for *Component, Event Class* and *Event Key;* it is good practice to always populate these fields; by default, these fields tend to be null with the exception of a *COMMAND* template which has a default event class of */Cmd/Fail*.

If the *eventClass* field is not populated then the event will come to the Event Console as ***/Unknown***; this means that further event mapping will need to be configured before de-duplication or auto-clearing can take place.

The decision whether to populate the *eventKey* field is crucial. This determines whether the *summary* field is part of determining a repeated event. If unique distinguishing information is in the *summary* field then do not include *eventKey*.

14.2 Event auto-clearing mechanism

Zenoss has a built-in mechanism whereby an event with *severity* of ***Clear*** (0) **automatically** clears **all** previous events where:

- component UUID
- eventClass
- eventKey

are the same. The **componentUUID** incorporates both device and component but this field is often blank (it was new in Zenoss 4 and many event classes pre-date that). Thus, if *componentUUID* is null then the **auto-clear fingerprint** fields are:

- device
- component
- eventClass
- eventKey

Note the use of *eventKey* again in either definition of the auto-clear fingerprint.

14.3 Exploring the use of event class attributes

The *ZenPacks.community.DirFile* ZenPack will be used to explore the effect of event attributes.

To demonstrate the use of the event **fields** (or **attributes** - the words are used interchangeably), the *file_stats.sh* script on a remote target was hidden. This file is driven both by a *COMMAND* template datasource and a *Python* datasource.

Template	DataSource	Type	Remote Script
File	FileTest1WithoutCount	COMMAND	file_stats.sh
FileStatsPythonXml	statsFile	Python	file_stats.sh

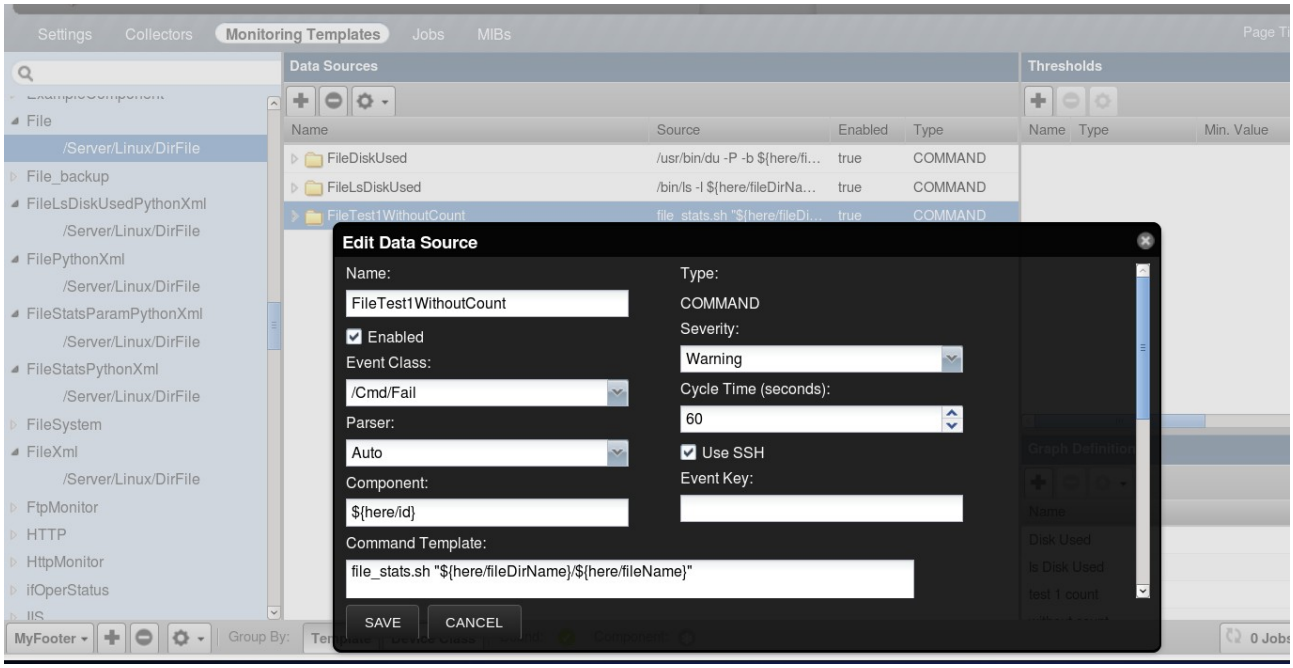


Figure 197: FileTest1WithoutCount COMMAND datasource in File template

The COMMAND-based template drives the remote script over ssh. Note that the *Component* field has been filled in but the *Event Key* field has not. *EventClass* contains the default */Cmd/Fail*.

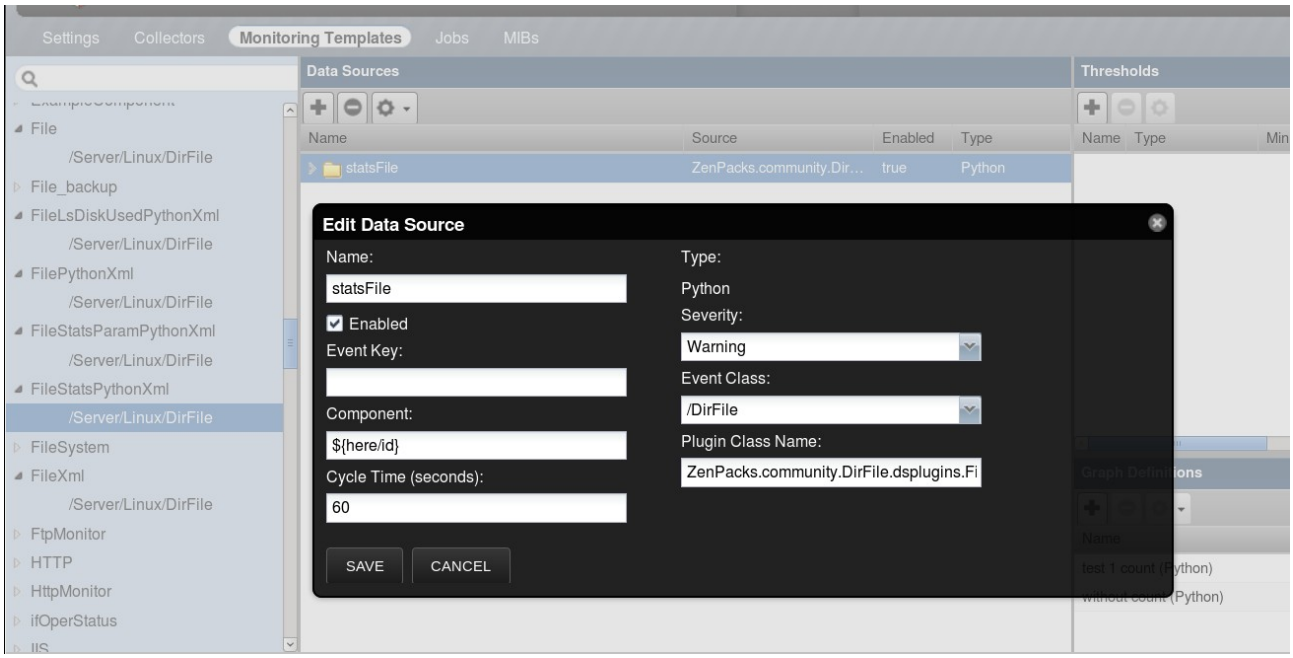


Figure 198: statsFile Python datasource in FileStatsPythonXml template

The *statsFile* Python datasource has had *Component* and *Event Class* configured but *Event Key* is left blank. The *file_stats.sh* remote script is called by the Plugin.

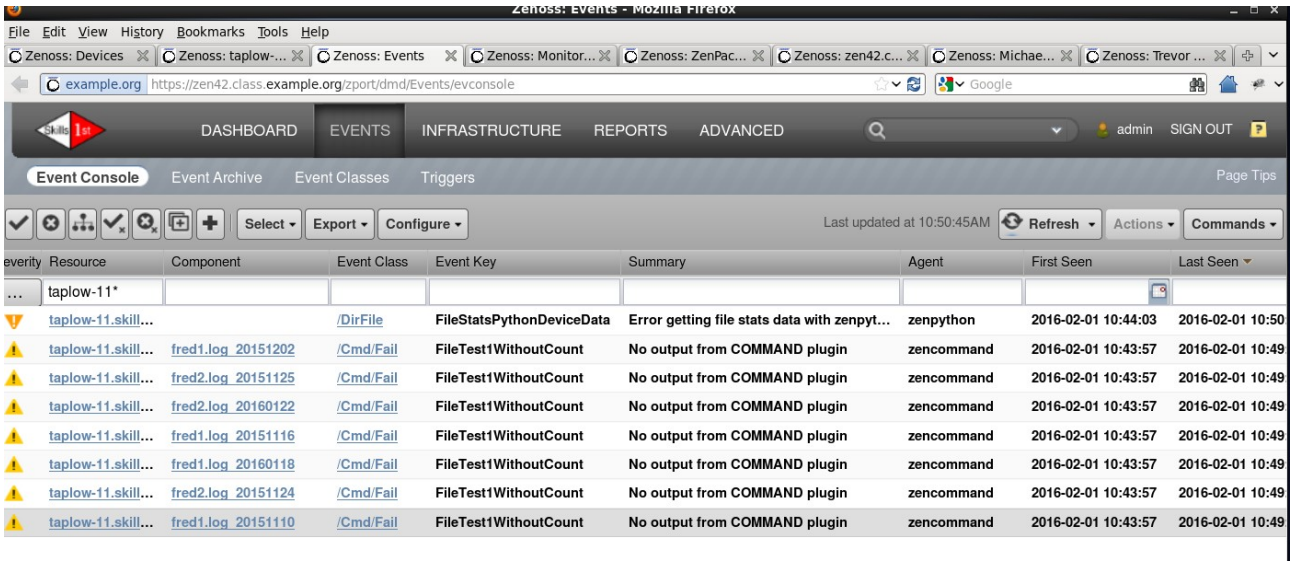


Figure 199: Events generated when remote file_stats.sh script is missing

Note in the Event Console output in Figure 199 that the events driven by the *zencommand* daemon have *Component* and *Event Key* populated; the *zenpython* event has the *Event Key* but not the *Component*.

Code in *$ZENHOME/Products/ZenHub/services/CommandPerformanceConfig.py* will populate an empty *Event Key* field with the datasource *Name* field for a *COMMAND* datasource. If *Event Key* is explicitly configured, it will override the default. The *summary* field is populated from *$ZENHOME/ZenRRD/parsers/Nagios.py* if no output is detected.

The event generated by *zenpython* comes from the *onError* method of the *PythonDataSourcePlugin*:

```
def onError(self, result, config):
  """
  Called only on error. After onResult, before onComplete.

  You can omit this method if you want the error result of the collect
  method to be used without further processing. It recommended to
  implement this method to capture errors.
  """
  ds0 = config.datasources[0]
  plugin = ds0.plugin_classname.split('.')[-1]
  log.debug( 'In OnError - result is %s and config is %s ' % (result, config))
  return {
    'events': [{
      'summary': 'Error getting file stats data with zenpython: %s' % result,
      'eventClass': '/DirFile',
      'eventKey': plugin,
      'severity': 4,
      }],
```

The event that is constructed **does** include *eventKey*, set to the plugin name, but does **not** include *component*. This is easily rectified by adding the component field to the event above:

```
'events': [{
  'summary': 'Error getting file stats data with zenpython: %s' % result,
  'eventClass': '/DirFile',
  'eventKey': plugin,
  'severity': 4,
  'component': ds0.component,
  }],
```

Also note that the event summary is the literal string "*Error getting file stats data with zenpython:* " followed by the message from the Exception in the *results* variable.

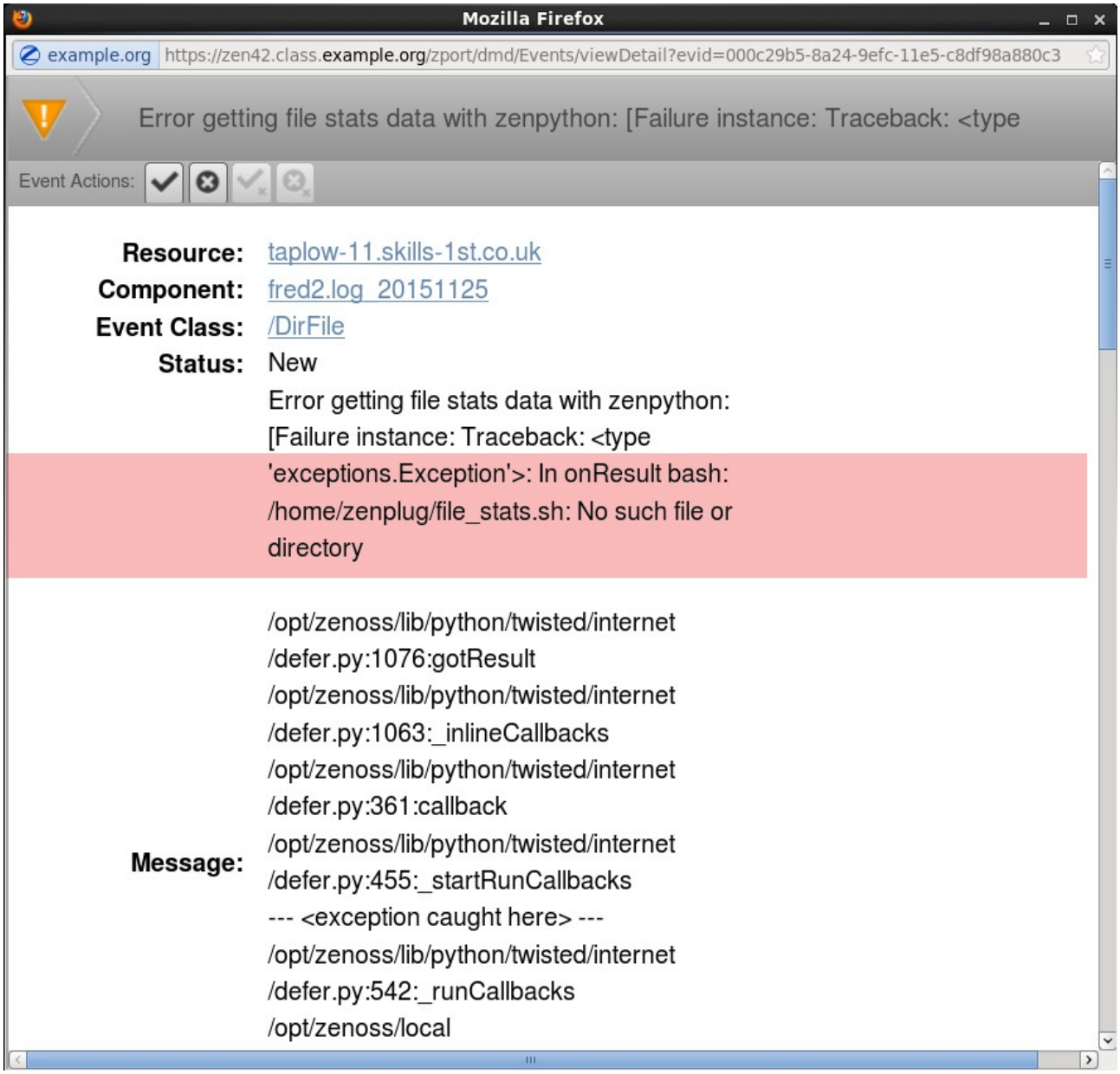


Figure 200: Detailed event when remote file_stats.sh script is missing for zenpython daemon

14.3.1 Detecting "repeat" events

Since *eventKey* is set by both *COMMAND* and *Python* datasources, the ***dedupid*** field is defined as the concatenation of:

- device
- component
- eventClass
- eventKey
- severity

separated by vertical bars.

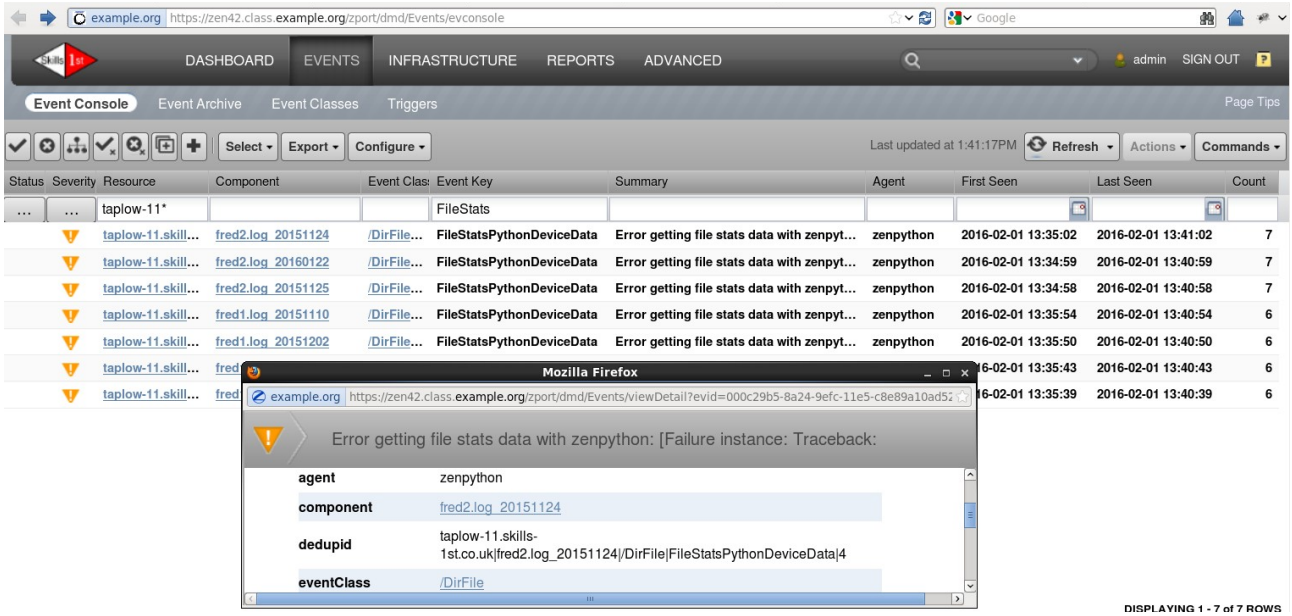


Figure 201: Events from the Python plugin with repeat count and dedupid

Figure 201 demonstrates an increasing repeat count for each event with a *dedupid* field of:

```
taplow-11.skills-1st.co.uk|fred2.log_20151124|/DirFile|
FileStatsPythonDeviceData|4
```

14.3.2 Auto-clearing events

The COMMAND-driven events will automatically clear (note **clear** status not **closed**) when the command runs successfully again, as *zencommand* always sends a default clearing event unless it is overridden by an error.

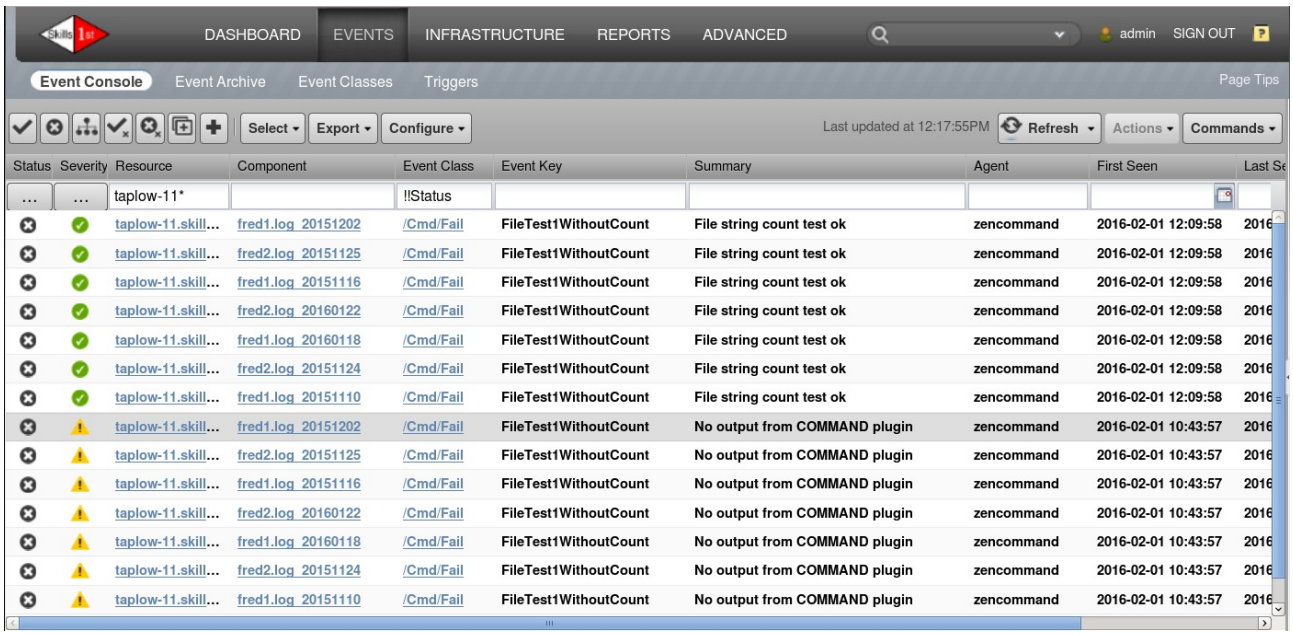


Figure 202: Events after file_stats.sh remote script is restored

When the *zencommand* command runs successfully the *summary* field is the output from *file_stats.sh* that precedes the pipe symbol.

```
echo " File string count test ok | $string1Name=$stringCount1 \
                $string2Name=$stringCount2"
```

The corresponding *PythonDataSourcePlugin* will not currently auto-clear events as the *onSuccess* method delivers no events. This could easily be added to ensure that error events from *zenpython* are automatically cleared when the *file_stats.sh* script is restored.

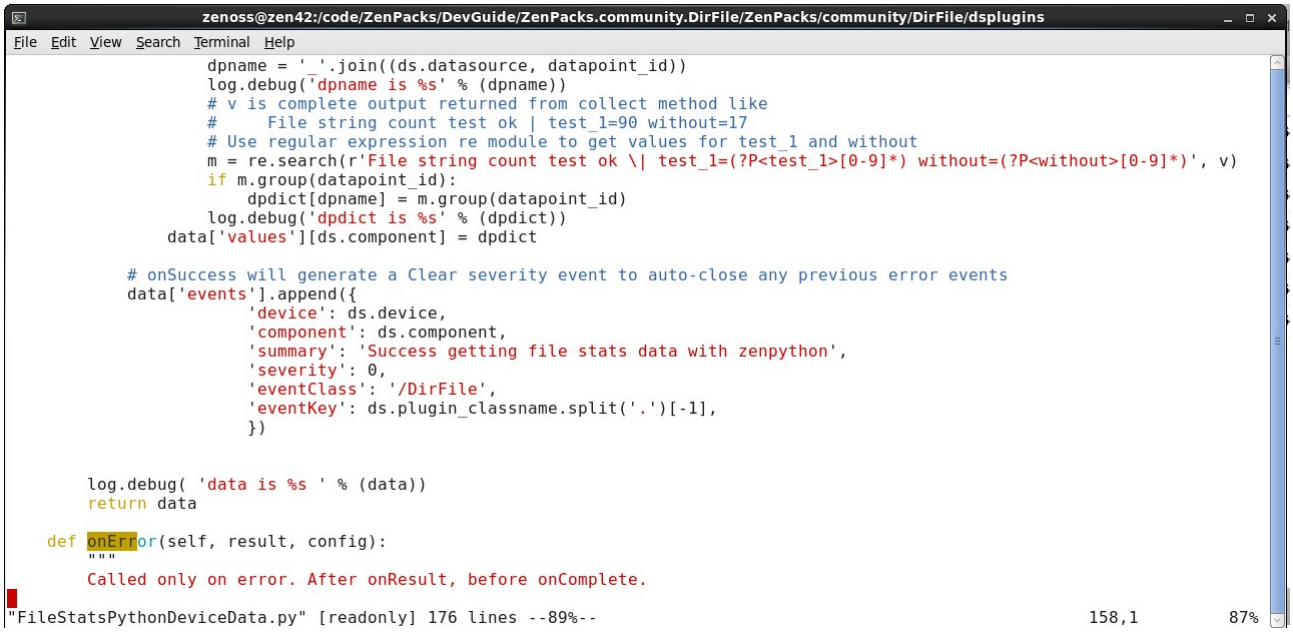


Figure 203: FileStatsPythonDeviceData plugin with clearing event added to onSuccess

After editing the dsplugin file, ensure that *zenhub*, *zopectl* and *zenpython* are recycled.

Note in Figure 203 that the clearing event has a *severity* of ***0*** (***Clear***). **The auto-clear fingerprint** is constructed from:

- device
- component
- eventClass
- eventKey

The “good news” clearing event will be a distinct event (not a repeat) as it differs from the “bad news” event in its *severity* field.

Do be aware that clear severity events that do not auto-clear any existing events, will be **silently dropped** by the *zeneventd* daemon so there should never be repeated clearing events.

Clear severity events are automatically set to ***Closed*** status (not ***Cleared***).

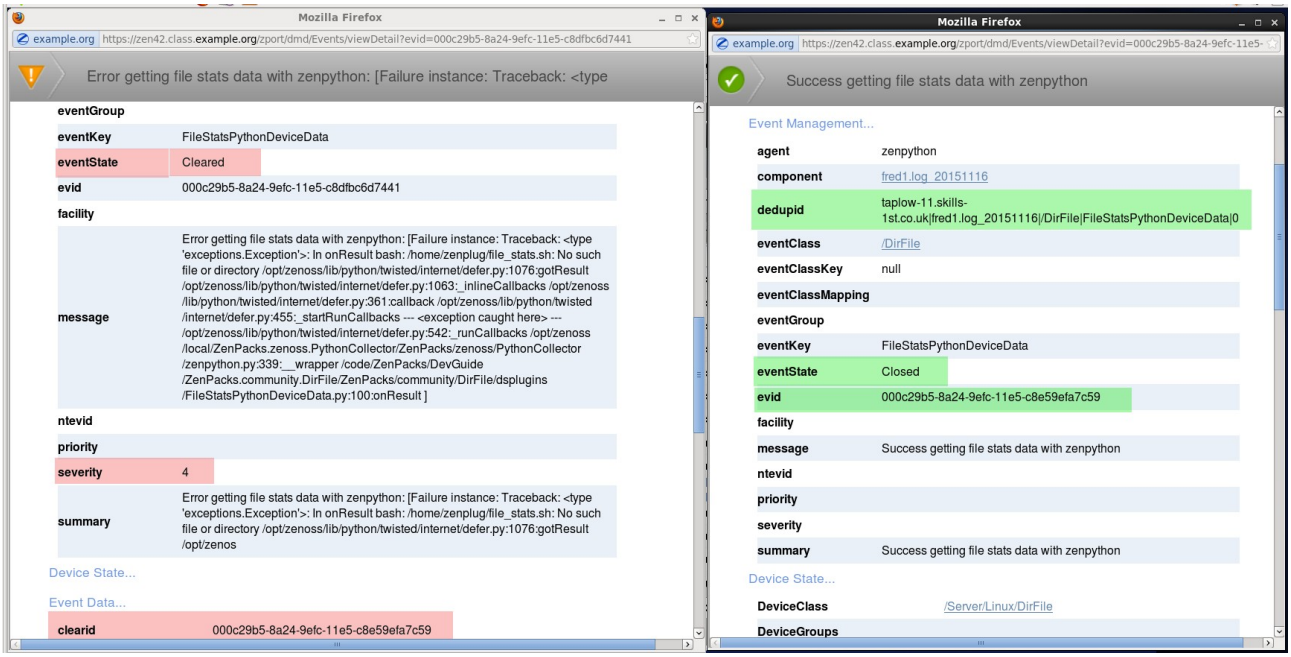


Figure 204: "Good news" event auto-clears "bad news" event

When a clearing event clears “bad news” events, the *evid* field of the “good news” event is copied to the *clearid* field of the “bad news” event.

14.4 Adding transforms to Event Classes

The main reason for adding new event classes is to make it easier to process them, ensuring that important events trigger notifications to specific users. Any event class may have a **class transform** which is code that may do one or more of:

- Modify attributes of the event
- Access the ZODB database for attributes or methods of the device that caused the event

- Create new user-defined event attributes
- Run Python code

Fundamentally, an event transform is Python code.

As an example, consider the event generated by *zenpython* when the remote *file_stats.sh* script is missing:

- eventClass /DirFile
- agent zenpython
- eventKey FileStatsPythonDeviceData
- component < file name being tested >
- summary Error getting file stats data with zenpython: < result >

It might be useful to include in the event, the directory containing the file under test. This is the *fileDirName* attribute of the *File* object instance. The *fileRegex* attribute is also available, defined in *zenpack.yaml*. A class transform for the */DirFile* event class can be constructed as shown in Figure 205.

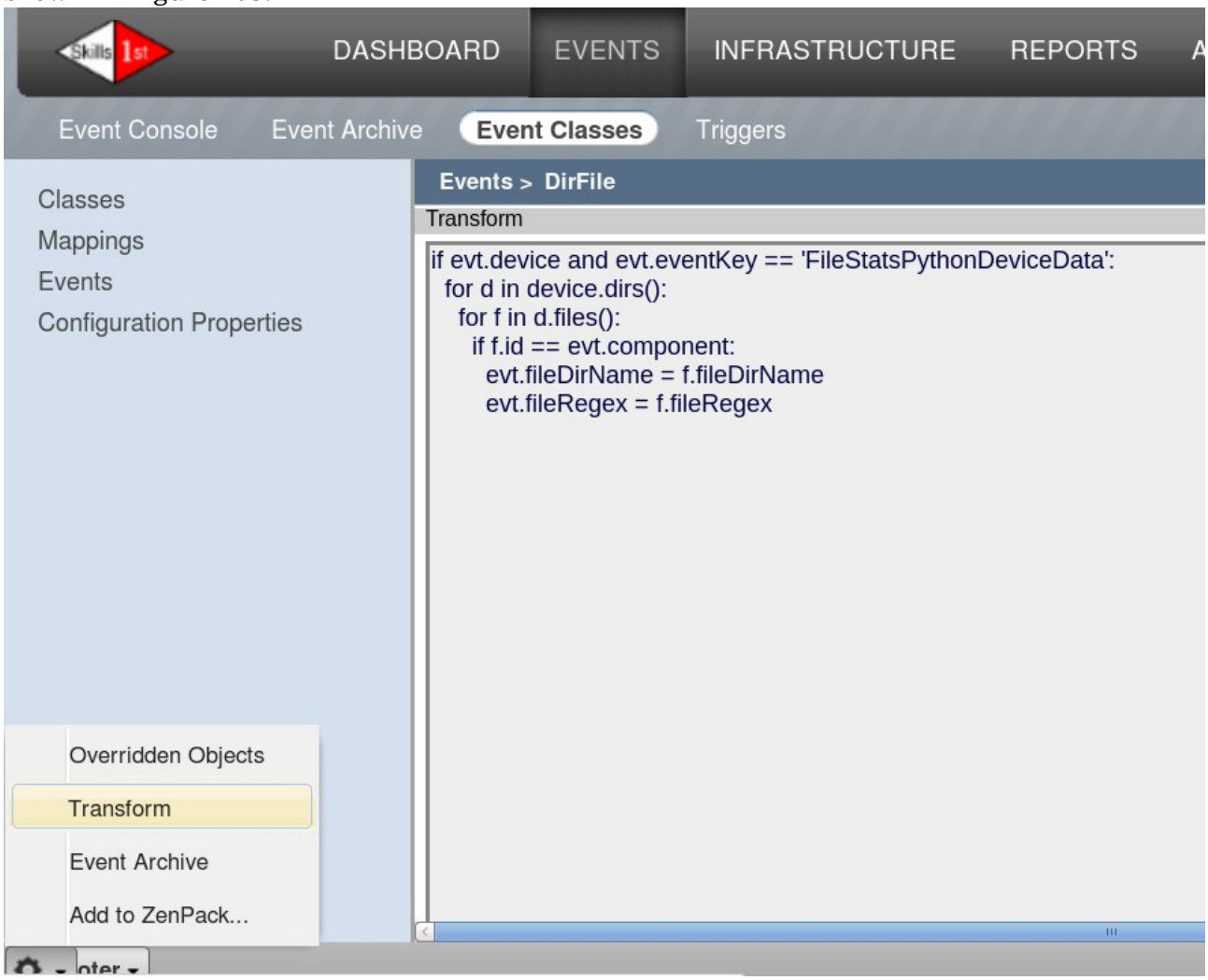


Figure 205: /DirFile event class transform

Two new user-defined event attributes are created for *fileDirName* and *fileRegex*.

Note that the transform has access to the *device* object and can query the ZODB database for its *dirs* relationships and the associated *files* relationships to check whether any match the existing *evt.component* field.

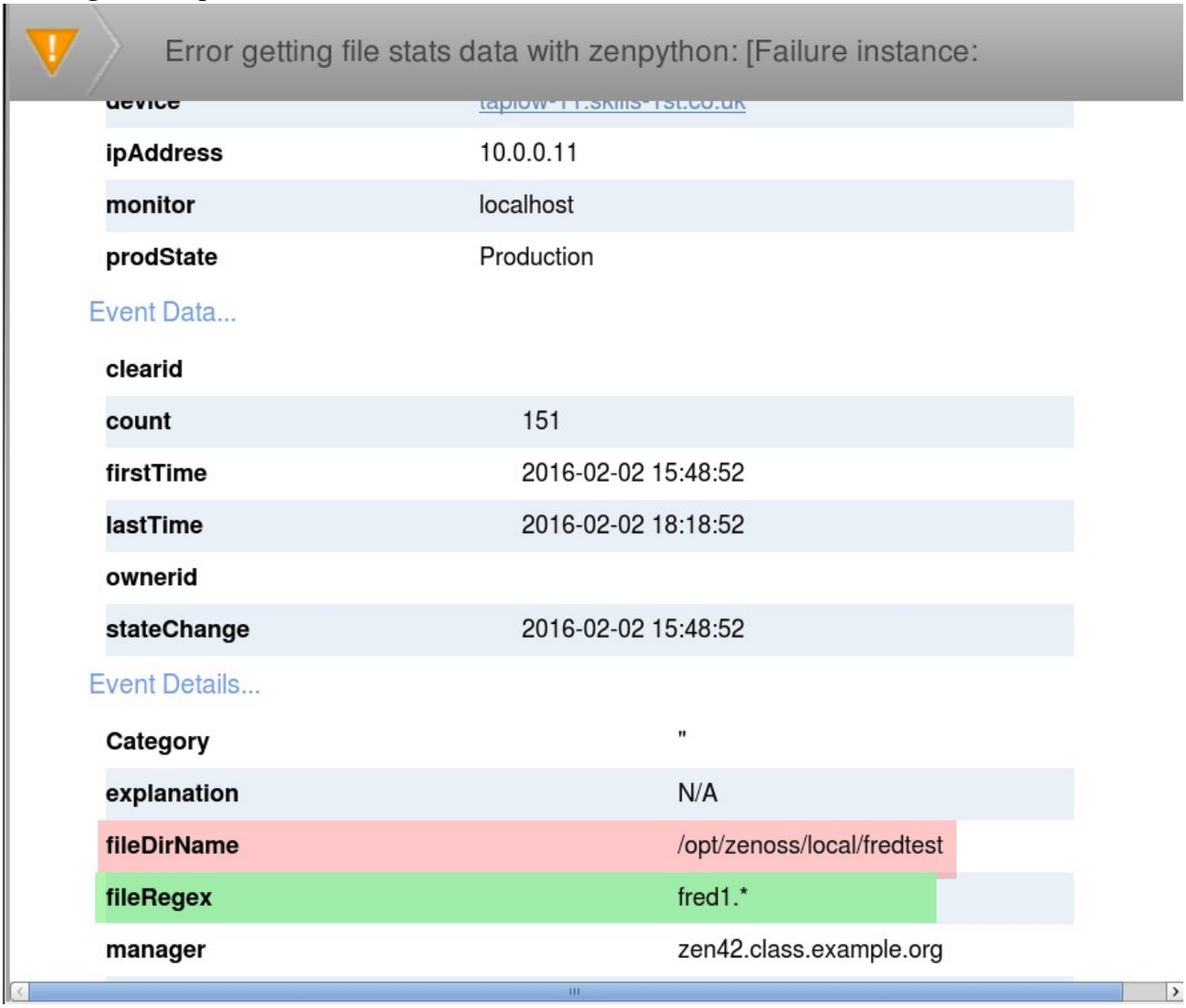


Figure 206: Event detail for /DirFile event class with new user-defined event attributes

14.5 Providing event details in a ZenPack

The Event Console permits configuration of fields to be displayed. In older versions of Zenoss, this was reached by clicking the down-arrow on an event field header. Latterly, the *Configure* button with *Adjust Columns* does the same job. Zenoss 4.2.5 with SUP 457 and later, even sorts the available columns alphabetically though earlier versions added new fields to the bottom of the list.

If a ZenPack has introduced new attributes that are required in the Event Console, these can be configured in the *zep.json* file in the ZenPack's *zep* directory. If a ZenPack has been created through the GUI then a *zep* directory will exist containing:

- zep.json.example
- actions.json.example

If the ZenPack has been created with zenpacklib then manually create the *zep* directory and *touch* the *__init__.py file*.

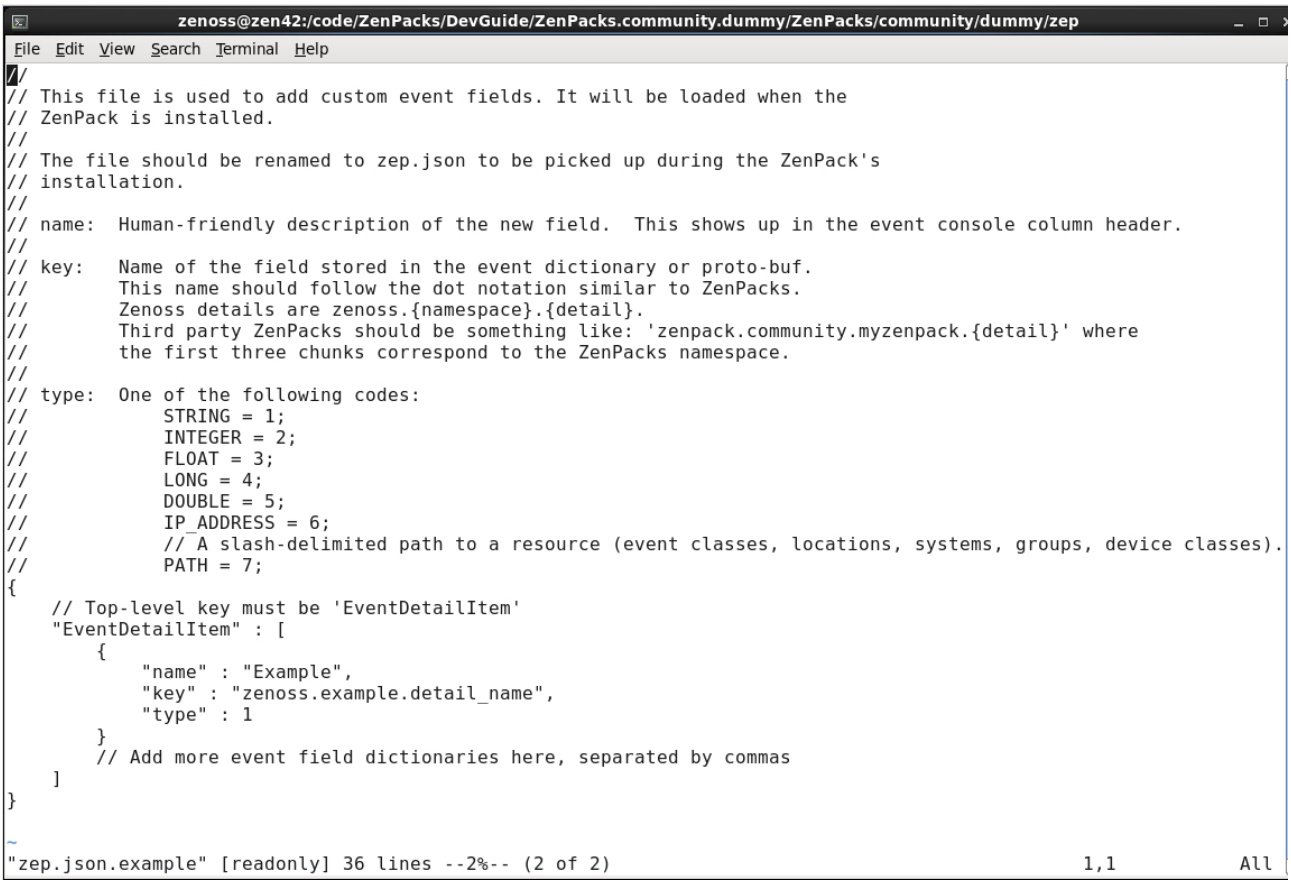


Figure 207: zep.json.example created automatically when ZenPack created from GUI

There is an item on the Zenoss wiki that helps both with defining event details fields and with providing triggers and notifications - http://wiki.zenoss.org/Providing_Triggers_Notifications_and_Event_Details_in_ZenPack .

Key points are:

- The file for defining event detail fields must be called *zep.json*
- There must be no comments in the file (prefaced with //). This will cause an error when the ZenPack is loaded such as:

```
ValueError: No JSON object could be decoded
```

- Ensure there are no commas on the final line of a clause - JSON is not as forgiving as Python - same error as above.
- *EventDetailItem* is the required reserved keyword and is a list of dictionaries where each dictionary specifies a field.
- The *key* attribute **must** match the name of the event attribute - this does not appear to match with the information provided in the default example file.

- If *key* **is** a fully qualified name such as *ZenPacks.community.DirFile.fileDirName* then the associated *name* **does** appear as a selectable field in the *Adjust Columns* dialogue of the Event Console but one cannot use *evt.ZenPacks.community.DirFile.fileDirName* in a transform as it tries to find a *ZenPacks* attribute of *evt* (and, of course, fails).
- If *key* is *ZenPacks.community.DirFile.fileDirName* and a transform references *fileDirName* then no association is made between the two.

- The *name* field is the user-friendly name used in the GUI.
- Help for the *type* field is given in the sample *zep.json.example*, where a type of 1 defines a string value.

The ZenPack should be reinstalled after introducing the *zep.json* file. This produced the following warnings but the code does appear to work:

```
2016-02-04 11:05:15,804 WARNING zen.AddToPack: Error trying to evaluate
true at line 239
2016-02-04 11:05:15,818 WARNING zen.AddToPack: Error trying to evaluate
true at line 278
2016-02-04 11:05:15,831 WARNING zen.AddToPack: Error trying to evaluate
true at line 317
```

zenoss should be completely restarted.

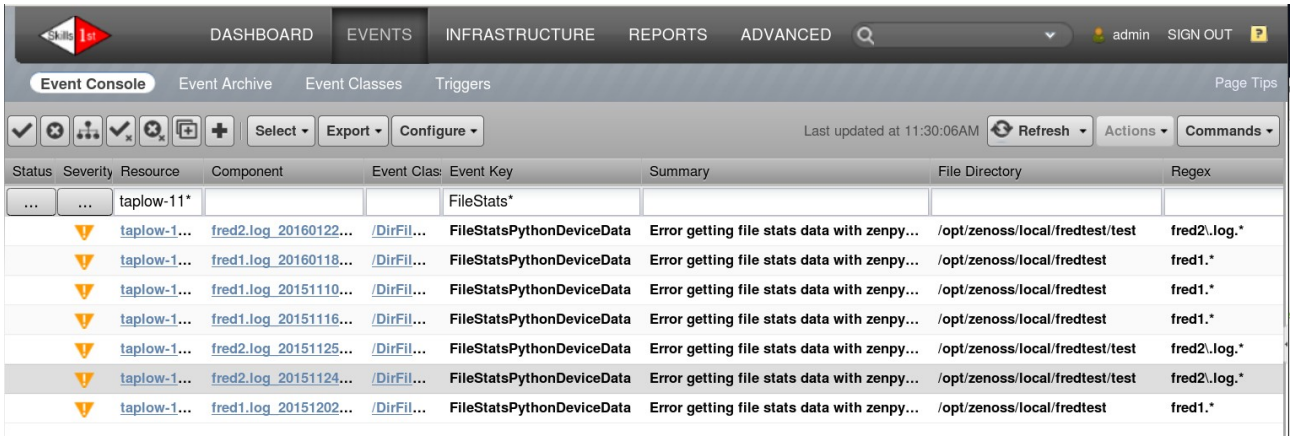


Figure 208: Event Console showing new File Directory and Regex event details

Note in Figure 208 that the column header matches the *Name* field in *zep.json*.

Ensure that the Event Console is refreshed and the web cache cleared to see the new event details fields in the Column configuration.

Occasionally, it seems to be necessary to reinstall the ZenPack a second time before the new fields appear.

14.6 Providing triggers and notifications in a ZenPack

Starting with version 4.0, Zenoss provides **Triggers** and **Notifications** for driving external alerts to users, where:

- A *trigger* defines the **conditions** for raising an alert (combination of event fields plus whether the trigger is enabled and details of users that can manage the trigger).
- A *Notification* is the **action taken** (email, page, command, trap), plus the trigger that activates the notification and the users to inform. There are also fields for:
  - Enabled
  - Send Clear
  - Send only on initial occurrence
  - Delay
  - Repeat
  - Content (eg. email parameters, command to be run, TRAP details)

Prior to Zenoss 4, **Alerting Rules** which were very similar in appearance to triggers and notifications, had to be defined; the big difference was that Alerting Rules were defined on a **per-user** basis.

In practice, triggers are evaluated by the Java *zeneventserver* daemon (sometimes referred to as *zep*); notifications are processed by *zenactiond*.

14.6.1 * Trigger and notification architecture

Consider the Event Architecture diagram in Figure 3 on page 7. When an event has been processed by the *zeneventd* daemon, it is then passed to the *zeneventserver* (sometimes referred to as *zep*) daemon via the *zenevents* queue. *zeneventserver* holds all the trigger definitions and evaluates the event against each active trigger. **Each and every** trigger that matches the event, results in a message being placed on the ***signal*** queue to be processed by the *zenactiond* daemon.

14.6.1.1 Finding trigger details

$ZENHOME/Products/ZenModel/Trigger.py only defines a “stub” object for managing user permissions for the trigger.

Some information on the structure of triggers can be deduced from *$ZENHOME/Zuul/facades/triggersfacade.py*. For example, the *synchronize* method:

> “ will first synchronize all triggers that exist in ZEP to their corresponding objects in ZODB. Then, it will clean up notifications and remove any subscriptions to triggers that no longer exist”

Triggers exist both in the *zenoss_zep* MySQL database and in the *ZODB* database; the *synchronize* method ensure that both are in step with each other.

Examining the MySQL database *zenoss_zep* and showing details of the various trigger tables, provides understanding of relevant fields from the perspective of the *zeneventserver* daemon.

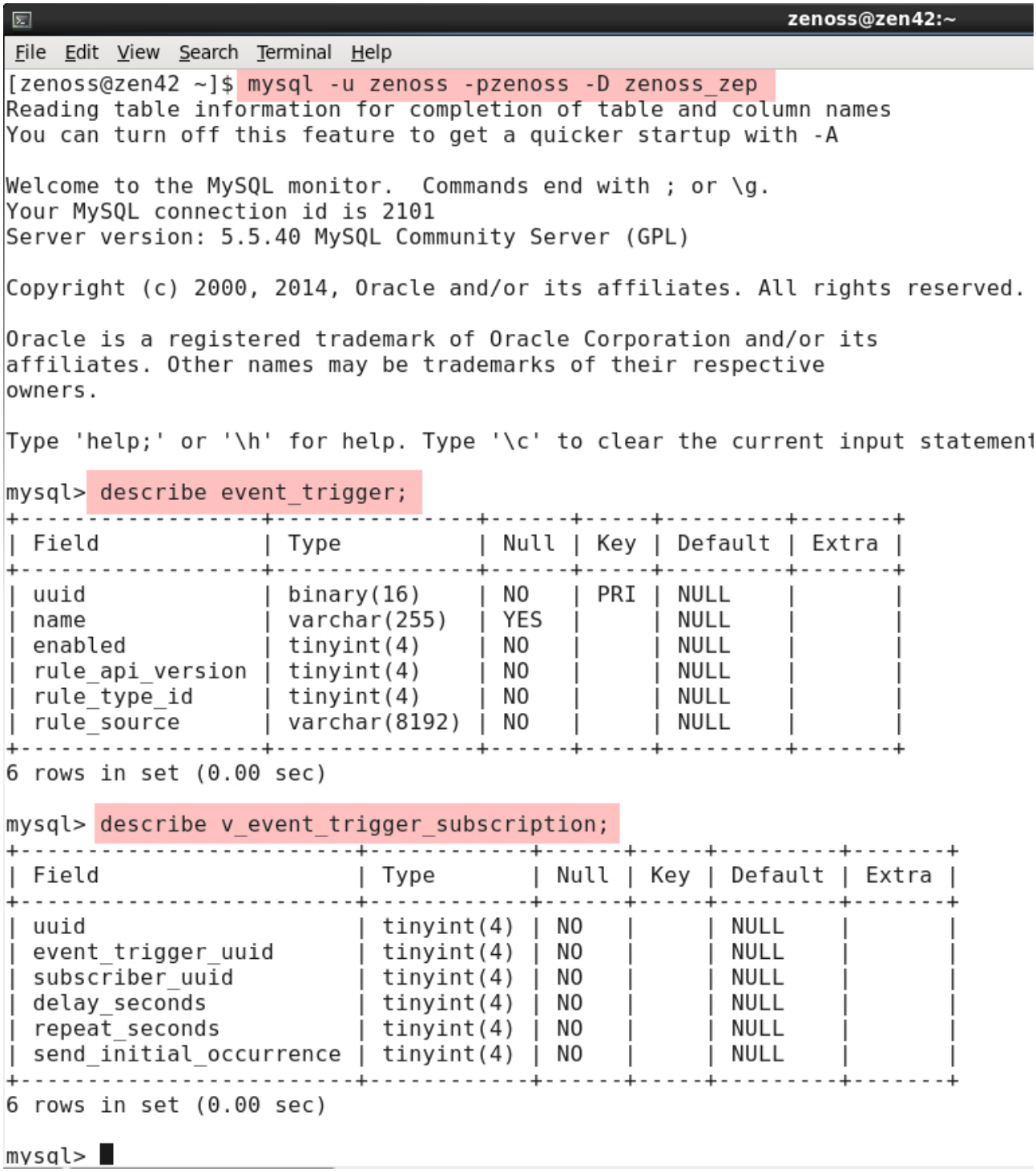


Figure 209: Examining the the MySQL zenoss_zep database - trigger tables

The *createTrigger* method in *triggersfacade.py* provides further insights into the attributes of a trigger in the ZODB database (which one cannot inspect with MySQL utilities as all data is held in an encoded format).

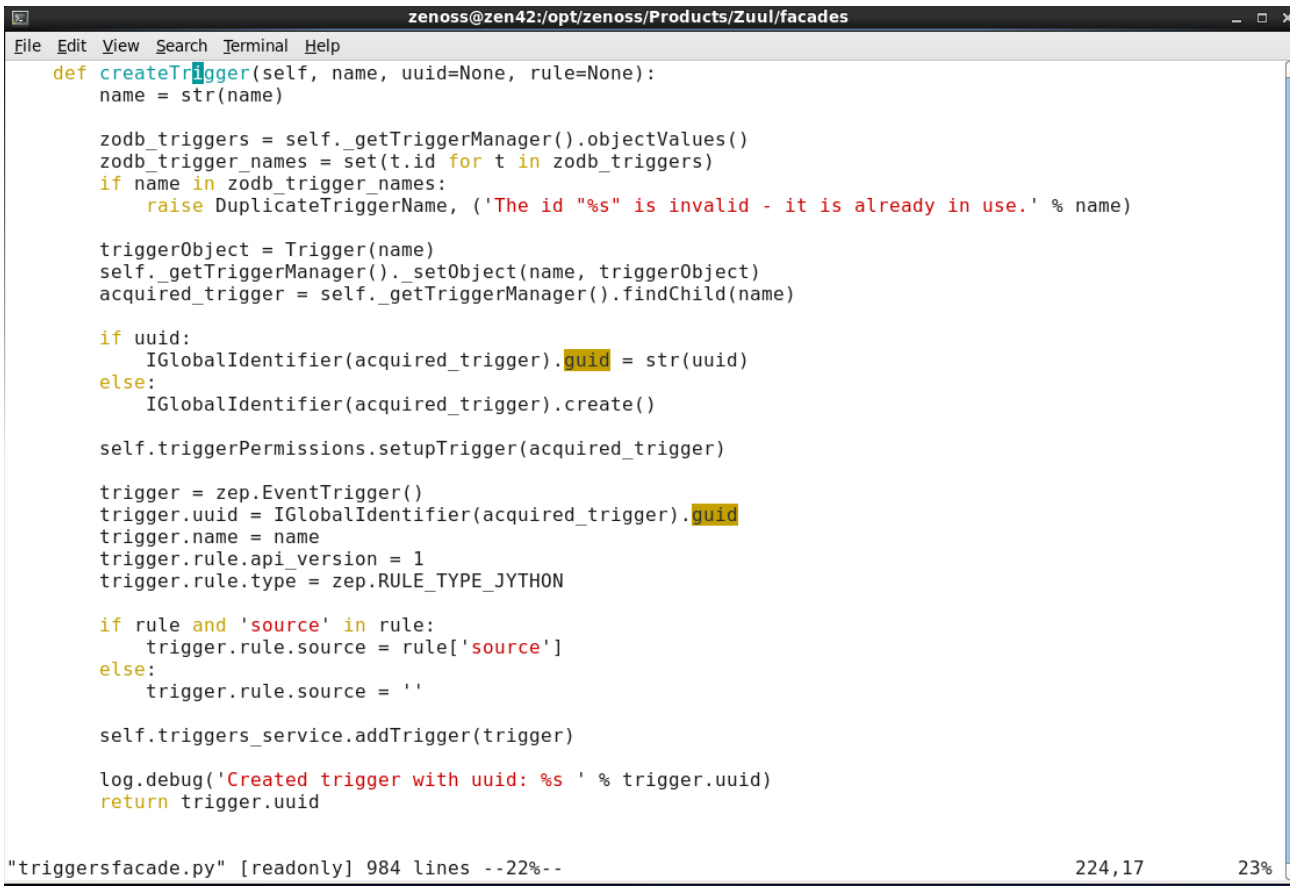


Figure 210: createTrigger method in $ZENHOME/Products/Zuul/facades/triggersfacade.py

A trigger has attributes for:

- UUID
- name
- rule (which is a dictionary), with attributes:
  - api_version
  - type
  - source
- enabled
- subscriptions (which is a list of dictionaries, each containing a subscribed notification):
  - [{ delay_seconds, repeat_seconds, send_initial_occurrence,

    subscriber_uuid, trigger_uuid, uuid},

    {.....},

    ]
- users (which is a list of dictionaries, each containing a user with permissions on this trigger):
  - [{ label, manage, type, value, write },

    { .....},

    ]

- where the *value* field is the UUID of the user or group

Fundamentally, when working with a trigger in the ZODB database, treat it as a **dictionary**.

14.6.1.2 Finding notification details

$ZENHOME/Products/ZenModel/NotificationSubscription.py defines the *NotificationSubscription* class, showing that a notification has the following attributes:

- id
- name
- enabled
- action
- delay_seconds
- repeat_seconds
- send_initial_occurrence
- send_clear
- subscriptions (which is a list of dictionaries, each representing a trigger); ie:
  - [{ name, uuid}, ... ,]
- recipients (which is a list of dictionaries, each representing a user to receive the notification). This list of dictionaries is identical in format to that used by triggers; ie:
  - [{ label, manage, type, value, write },

    { .....},

    ]
  - where the *value* field is the UUID of the user or group
- globalRead
- globalWrite
- globalManage
- content

The *content* attribute is the details for an email or pager or the command and environment details for a Command notification. Several files give help with permissible fields for the different content types; see *$ZENHOME/Products/Zuul/interfaces/actions.py, $ZENHOME/Products/Zuul/infos/actions.py* and *$ZENHOME/Products/ZenModel/actions.py*.

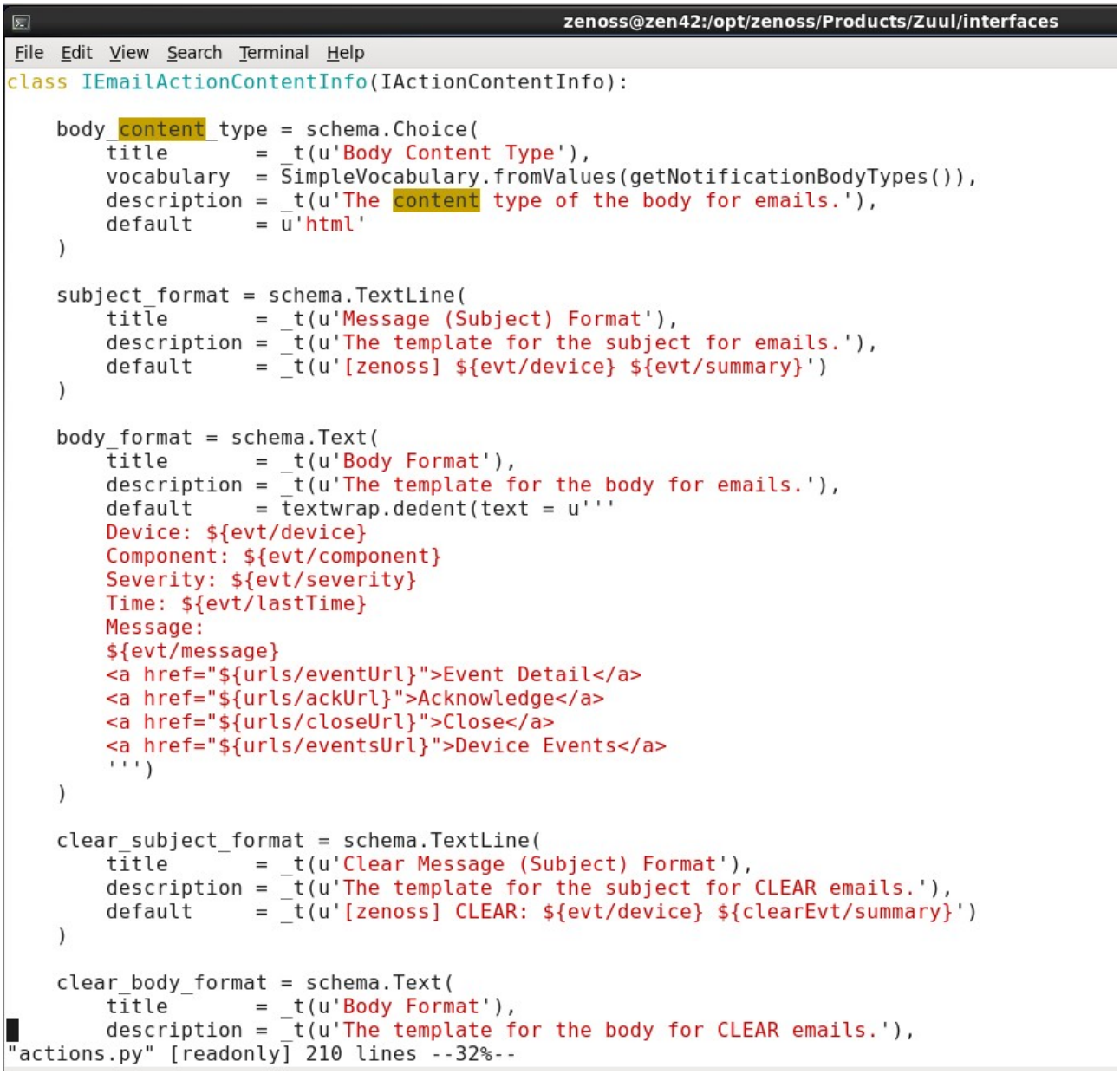


Figure 211: $ZENHOME/Products/Zuul/interfaces/actions.py showing fields for email content

Note that there are no tables in the *zenoss_zep* MySQL database for notifications; they are handled entirely in ZODB by the *zenactiond* daemon.

Fundamentally, when working with a notification in the ZODB database, treat it as an **object class** .

14.6.1.3 Dumping trigger and notification details

There is an Audit package on GitHub (https://github.com/jcurry/Audit) that provides several scripts to dump various aspects of the Zenoss ZODB database. One of these is *trigs_and_notifs.py* which outputs all aspects of all triggers and notifications. It takes a single *-f <filename>* parameter, specifying the output file:

```
./trigs_and_notifs.py -f trigs_and_notifs.out
```

14.6.2 ZenPack file for triggers and notifications

To provide triggers and notifications in a ZenPack, implement the ***actions.json*** file in the ZenPack's *zep* directory. Again, a default sample file is provided with a GUI-created ZenPack.

- The file must be called ***actions.json***
- There must be no comments
- Take care with punctuation - no comma on the last line of a clause
- Double quotes as part of a value must be escaped with backslash. For example:

```
  "rule": {
      "api_version": 1,
      "source": "(dev.production_state == 1000) and (evt.severity >= 4) and
 (evt.event_class.startswith(\"/DirFile\")) and (evt.status == 0)",
      "type": 1
       }
```

-

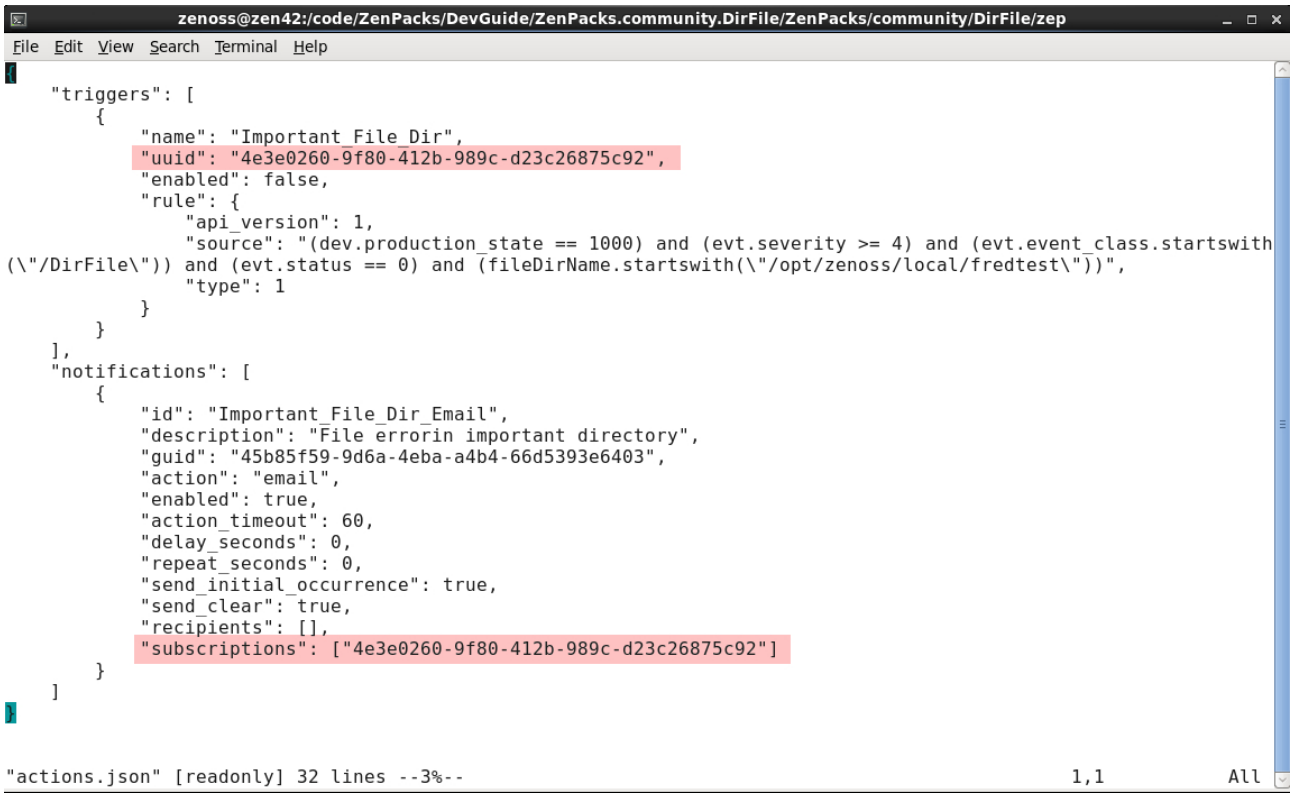


Figure 212: actions.json in zep directory of ZenPack to define trigger and notification

The crucial fields that need to associate in Figure 212 is the *subscriptions* field of the notification must be a **list** of the required trigger *uuid* fields.

New triggers and notifications require Universally Unique IDs (UUIDs) which can be generated using Python:

```
python -c "import uuid; print uuid.uuid4()"
```

Copy the resulting UUID into *actions.json*.

$ZENHOME/Products/ZenUtils/guid/guid.py has code to generate globally unique UUIDs.

The hardest part of creating an *actions.json* is understanding what the required field names are and what data structure they must implement.

In practice, the easiest way to do this is to create the trigger and notification through the GUI, dump the structures with the *trigs_and_notifs.py* file referred to in 14.6.1.3, and then create the *actions.json* file using the names and structure from that output file.

14.7 Resolving issues with triggers and notifications

If the ZenPack installation fails, suspect syntax issues.

If a trigger or notification cannot be opened in the GUI suspect a specification error such as bad fields.

Ensure that double quotes as part of a value string must be escaped with backslash.

14.8 Known issues with event fields, notifications and triggers

With Zenoss 4.2.5 SUP 457 there are two known issues:

- No defaults are provided for the content of notifications
  - GUI-created notifications **do** have defaults eg. email body, subject, etc
  - Documented in Zenoss JIRA as https://jira.zenoss.com/browse/ZEN-15367?filter=10510
  - This means that email customisation through *actions.json* such as email host, user, port **and password** all have to be hardcoded in *actions.json*
- Custom event fields can be specified in the Trigger GUI but result in errors in *zeneventserver.log*.
  - For example:

```
 2016-02-10T11:35:52.253 [INDEXER_EVENT_SUMMARY] WARN
org.zenoss.zep.impl.TriggerPlugin - exception raised while evaluating rule:
(dev.production_state == 1000) and (evt.severity >= 4) and
(evt.event_class.startswith("/DirFile")) and (evt.status == 0) and
(fileDirName.startswith("/opt/zenoss/local/fredtest")), NameError: global
name 'fileDirName' is not defined
```

  - This is in Zenoss JIRA as https://jira.zenoss.com/browse/ZEN-21963?filter=10510 and https://jira.zenoss.com/browse/ZEN-7910
  - SUP 671 (the latest 4.2.5 patch in March 2016) still does **not** fix this issue but it **is** fixed in Zenoss 5.0.7.
    - Trigger rule has the source format:

```
"source": "(dev.production_state == 1000) and (evt.severity >= 4)
and (evt.event_class.startswith(\"/DirFile\")) and (evt.status == 0)
and (hasattr(zp_det, \"fileDirName\") and \"fredtest\" in
zp_det.fileDirName)"
```

- Incorrect customisation of triggers in *actions.json* not only produces the error message in *zeneventserver.log*; custom event attributes prevent the opening in the GUI of any trigger that contains them.

- With both Zenoss 4 and Zenoss 5, event field details in a ZenPack in *zep.json* sometimes appears to take “a little while” or a second ZenPack installation before the new field can be seen in the GUI Event Console *Adjust columns* or as a selectable field in the Trigger GUI.

TODO: Resolve status of JIRA tickets to fix these issues for both Zenoss 4 and Zenoss 5.

15.0 Creating menus in ZenPacks

It is perfectly possible to extend menus in Zenoss. The GUI provides a simple way to extend command menus; code is required for more advanced techniques.

The Zenoss 3 Developer's Guide has some useful information in Chapters 13 and 14. Also consult the “Creating Zenoss ZenPacks” documents from Skills 1st at https://www.skills-1st.co.uk/papers/ .

The Zenoss GUI is built on top of Zope version 2, upto and including Zenoss 5. Extending menus is one small topic of GUI extension, several of which have already been seen in earlier chapters; for example, the display of device components, attributes and graphs.

The zenpacklib tool is excellent for creating some GUI code automatically; unfortunately it has nothing to offer by way of menu creation.

15.1 The jargon

15.1.1 Zenoss 2 (some of which is still relevant!)

Zenoss 2.x used Zope Page Templates (ZPT), Template Attribute Language (TAL), Macro Expansion for TAL (METAL), TAL Expression Syntax (TALES), Cascading Style Sheets (CSS) and a little JavaScript. Fundamentally most of these are based on HyperText Markup Language (HTML).

- HyperText Markup Language (HTML) - is the most basic formatting language available on the Web, and some version of HTML is understood by every Web browser. HTML is in practice a sloppy variant of eXtensible Markup Language (XML) which divides up a page into elements (tags such as title, head or h3) and content (for example, the things that you actually care about). Common HTML tags found in Zenoss skins files include:
  - <th> table header
  - <td> table data
  - <tr> table row
  -
 break
  - <block> creates larger structures that can include other blocks
  - <form> for user input
  - <input> input directive
- Zope Page Templates (ZPT) - are in essence HTML pages which are well-formed and have extra XML attributes (ie the bits after the element name in-between the < and >

characters). The extra XML bits (attributes) are not a part of any HTML standard and are ignored by HTML editors, meaning that ZPT pages live happily with HTML. These attributes and the programming functionality that they deliver are called the Template Attribute Language (TAL). Zenoss skins files all have a *.pt* extension for Page Template.

- Template Attribute Language (TAL) - the TAL attributes allow you to add dynamic content using information from inside the Zope database (ZODB). From a Zenoss perspective, this allows you to write a query that you can use to build a table, or show different items depending on what objects or devices exist in a particular state. In other words, TAL is the Zope way of accomplishing what you would normally need to do in a CGI inside of a plain web server like Apache. It should be noted that inside TAL it is also possible to use a restricted subset of Python. The restrictions include not being able to load certain standard libraries, as well as operations like reading and writing to disk. This is done intentionally for security reasons. See http://docs.zope.org/zope2/zope2book/source/AppendixC.html for a Zope Page Template reference. TAL includes statements such as:
  - tal:define define variables
  - tal:condition test conditions
  - tal:content replace the content of an element
  - tal:repeat repeat an element
  - tal:replace replace content of an element
  - tal:attributes dynamically change element attributes
- Macro Expansion for TAL (METAL) - because TAL is hidden away inside HTML, there's no way to reuse blocks of HTML and TAL for your site just by using TAL. METAL allows page templates to define *macros* (which are essentially sub-templates that may be called by other templates) and *slots* (which may be filled by other templates). Several METAL macros are provided with Zenoss such as:
  - page1 provides web page with breadcrumbs and content
  - page2 page 1 plus standard breadcrumbs and navigation tabs
  - page3 page 1 plus standard breadcrumbs, no tabs
  - zentable creates tables of data for display
  - navbodypagedevice macro to support sorting, filtering, multi-pages
- TAL Expression Syntax (TALES) - TALES allows access to the template's namespace, including useful properties such as the *here* context object. TALES accepts paths (e.g. *here*/*id*) which it resolves into object properties. It will attempt to resolve the final path element as a key index, a key name, an attribute, or a method. For example, if getSomething() is a method on the context, here/getSomething will return the result of that method. TALES statements are what normally provides the dynamic content for a page template, delivering data from the ZODB database.
- JavaScript - JavaScript (nothing really to do with Java!), can be written directly on the Web page inside a script element anywhere in an HTML page, or it can be stored on a server and accessed from a script element using the name specified in the src attribute.

- JavaScript Library: ExtJs- the Zenoss Web interface extensively uses the Ext Js JavaScript library from Sencha. This framework is used to make a desktop-style user application within the constraints of HTML/CSS and JavaScript. Zenoss 2 and 3 used version 3 of the library; Zenoss 4.2.x changed to ExtJs 4, requiring many ZenPacks to be updated. See https://github.com/zenoss/Zenoss-User-Interface-API-Docs/tree/master/guides/component_grid_upgrade for a discussion on this.

In Zenoss 2, GUI code was created largely in Page Template files, ending with a **.pt** suffix - and significant amounts of these Page Template files are still used in the latest versions of Zenoss. Core Zenoss GUI code is found under *$ZENHOME/Products/ZenModel/skins/zenmodel*; ZenPacks that delivered new GUI code had a directory hierarchy under the base directory called *skins* with a subdirectory of the full ZenPack name.

15.1.2 Zenoss 3 / 4 / 5

Zenoss 3 made extensive changes to the GUI, moving to far more emphasis on JavaScript and removing the need to write ZPT files; however, other files are now required to link the GUI code with the data in the ZODB database.

- JavaScript files - with ***.js*** extension. Define what to display and how to display it. Typically a ZenPack will ship *.js* files under the base directory in a *browser/resources/js* directory hierarchy.

- infos - abstracts object attribute information saved in the ZODB, that will be displayed to the user. Defines **what** will be displayed **not how** it will be displayed. May be in a separate *info.py* file in the base directory of a ZenPack; alternatively, it is perfectly acceptable to include info classes in other Python files of the ZenPack (especially if very few definitions are required).

- interfaces - describes elements of **how** the data is displayed. May be in a separate *interfaces.py* file in the base directory of a ZenPack; alternatively interface classes may be included in other Python files.

- *configure.zcml* - provides the “glue” between interfaces and JavaScript display code and this exact name will be searched for by the Zope mechanisms. Zope Configuration Markup Language (ZCML) is Zope 3's XML-based component configuration language for “wiring” together application policy and component registrations. It is documented at the Zope site at http://docs.zope.org/zopetoolkit/codingstyle/zcml-style.html .

15.2 Extending Command menus with the GUI

Zenoss has always provided a simple, GUI method to add to the *Command* menu at the bottom of the left-hand menu of a device or device class. Use the *Administration* menu to display existing commands and the dropdown from *Define Commands* to add or delete commands; clicking on a command permits editing (provided the user has sufficient authority).

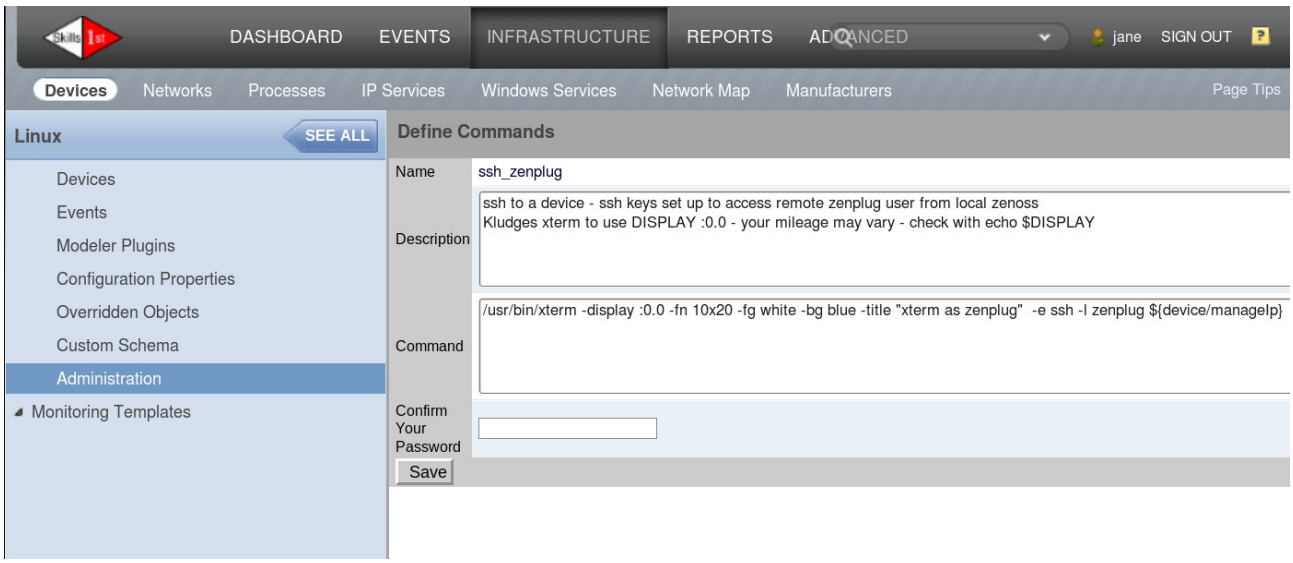


Figure 213: Define a new ssh command for the /Server/Linux device class

Any command defined at a device class level is available to all devices in that class or subclasses.

Any command can be run. The commands shipped with the core product simply produce output in a new window, for example *snmpwalk*, *ping;* they do not offer a mechanism to solicit input from a user. A trick demonstrated in Figure 213 that can work for Linux target systems, is to open an xterm window for general purpose use. Parameters can be used within the command as it has access to the *device* object so the command sets up an xterm that runs ssh to *${device/manageIp}*. (Note that with xterm you may need to kludge the *DISPLAY* variable and run *xhost +* from a normal command window before this Zenoss command works).

```
/usr/bin/xterm -display :0.0 -fn 10x20 -fg white -bg blue \
  -title "xterm as zenplug" -e ssh -l zenplug ${device/manageIp}
```

The *Define Commands* menu also has an *Add to ZenPack* option so such customisation can be transported.

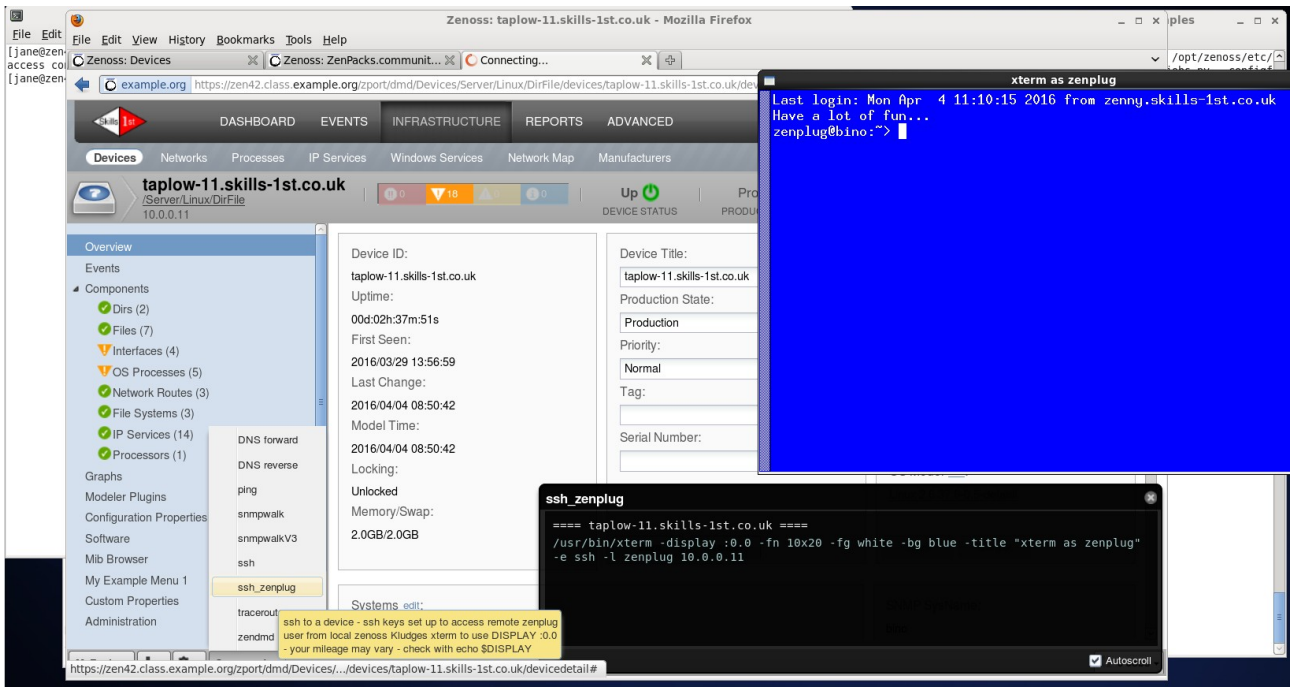


Figure 214: Output of new ssh command

15.3 ZenPacks.community.MenuExamples

ZenPacks.community.MenuExamples is a slightly modified version of an earlier ZenPack called *ZenPacks.skills1st.MenuExamples*. The main difference is that customisation of the *Modifications* menu has been removed as that is no longer relevant since later versions of Zenoss 3.x.

The ZenPack should be treated as a number of samples, not as production code. It started life with Zenoss 2.x and has been updated for Zenoss 3 and Zenoss 4; it also installs on Zenoss 5.0.x. One of its main purposes is to assist porting old ZenPacks from earlier Zenoss versions upto the latest version, hence the inclusion of so many examples with older techniques.

The ZenPack creates some new object classes and modelers as demonstration classes that can then have their menus modified:

- A new Zenoss device class */Example/TestClass* is created
- A new device object class called *ExampleDevice* is created with a new component object class of *ExampleComponent*. *ExampleDevice* has a new method, *createComment*.
- The *__init__.py* in the base directory also creates an **instance** of *ExampleDevice* in */Example/TestClass* .
- Modeler plugins are created:
  - ExampleSNMP gathers device memory & swap from UCD MIB
  - ExampleCMD gathers component disk info using COMMAND
  - ExampleHostResourcesSNMP gathers component information from HR MIB

The ZenPack demonstrates a number of different techniques for creating menus:

- *My Example Menu 1* on left-hand device menu

    - Active for all device types
    - Has three windows displaying:
        - Several SNMP attributes for the device
        - Performance graphs for the device
        - A window with Zenoss Group information and further menu dropdown examples
- *My Example Menu 2* on left-hand device menu
    - Active only for devices of object class *ExampleDevice*
    - Has two windows displaying:
        - Several (slightly different) SNMP attributes for the device
        - Performance graphs for the device
- *My Example Menu 1* has two dropdown menus which are shipped as part of *objects.xml*:
    - *My Drop down menu 1* requests input and updates device attribute
    - *My Drop down menu 2* delivers device attribute information
- Additional dropdown menus from *Display* for *ExampleDevice* device components - *Example Component Template* and *Amazing Stuff*
- From *INFRASTRUCTURE -> Devices* a new item is added to the + dropdown to *Add Example Device*
- From *INFRASTRUCTURE -> Devices* a new item is added to the left-hand *Action* menu - *Run My Predefined Shell Command*
- From a device's main panel, two new actions are added to the *Action* menu for all devices - *Example Device Action* and *Another example device action*
- A new menu is added to the footer bar, *MyFooter*, with three actions:
    - The standard *Model Device* action
    - *Jane's Predefined Command* to run a fixed command in a new window
    - *Set device comment / rackSlot* which prompts for input and then modifies attributes of the device
- Some of the menus use new functionality defined in *routers.py* and *facades.py*

15.3.1 New device class, device object class and component class

When the ZenPack is installed the *__init__.py* of the ZenPack creates two new Zenoss device classes, */Example* and */Example/TestClass*. Unusually, the ZenPack also create an **instance** of a device in */Example/TestClass* called *ExampleDevice1*.

It is not normally good practice for a ZenPack to include device **instances** as they are probably irrelevant in any other organization and may cause a name duplication. It is included here as an example and to permit some menu customisation techniques to be demonstrated.

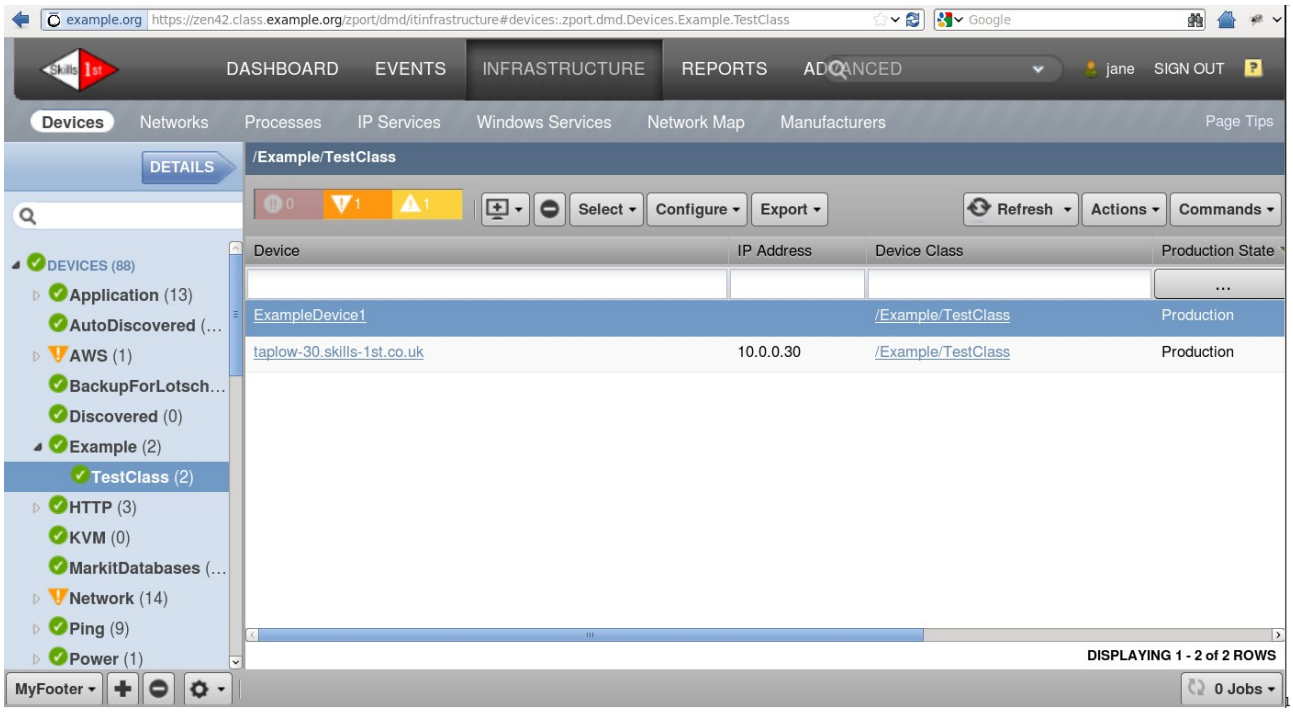


Figure 215: New Zenoss device class /Example/TestClass with ExampleDevice1 instance

The *ExampleDevice1* is defined without an IP address; for testing it may be convenient to use the device's *Action* menu to *Reset/Change IP Address* to a real, existing test device, and remodel.

15.3.2 Menu defined in __init__.py

__init__.py in the base directory of the ZenPack can be used to define a new menu.

If any use is to be made of the old-style skins files, then *__init__.py* should have the *skinsDir* directory registered. This will be *<base directory of ZenPack>/skins*. Beneath that directory will be a directory with exactly the same name as the ZenPack. *.pt* files are created under there.

```
skinsDir = os.path.join(os.path.dirname(__file__), 'skins')
from Products.CMFCore.DirectoryView import registerDirectory
if os.path.isdir(skinsDir):
  registerDirectory(skinsDir, globals())
```

It is bad practice to create new ZenPacks using skins files; however, many ZenPacks still exist with the old-style definitions.

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples
File  Edit  View  Search  Terminal  Help
import logging
log = logging.getLogger('.'.join(['zen', __name__]))

import os.path
# Register the skins directory - we don't need this now if we put page template
#   files under browser/resources in Zenoss 3 style.  It IS needed if you want to
#   find NEW pt files in the ZenPack's skins directory (all versions of Zenoss).
# Note from Zenoss 3.2 that pt files that OVERRIDE Core pt files may NOT be
#   picked up from the skins directory (even with the skinsDir registered as below).
#   pt files that override Core pt files should actively be "wired in" with
#     statements in a configure.zcml file
#
skinsDir = os.path.join(os.path.dirname(__file__), 'skins')
from Products.CMFCore.DirectoryView import registerDirectory
if os.path.isdir(skinsDir):
    registerDirectory(skinsDir, globals())

# In older-style Zenoss 2.x  ZenPacks the pt files are under skins/<ZenPack name>
# The new Zenoss 3 convention is to put them under browser/templates and use the zcml files
#  to 'wire-in' this directory

# Create a new menu item in the left-hand menu for all devices
# The label of the menu is the 'name' parameter
# The 'action' parameter refers to a page template (.pt) file  (without the .pt) if using
#   Zenoss 2 skins directory or refers to the "name" field in a page stanza
#   if using Zenoss 3 browser/configure.zcml wiring.
#
# The 'permissions' field defines the permission that a role must have for this to be valid for a user
#
from AccessControl import Permissions
from Products.ZenModel.Device import Device

myExampleMenu1 = { 'id': 'myExampleMenu1',
        'name' : 'My Example Menu 1',
        'action' : 'myExampleMenuOne',
        'permissions' : ( Permissions.view,)
        }
# Add this menu for certain object classes.  Adding to "Device" means
# it is available for all subclasses of /Device

Device.factory_type_information[0]['actions'] += (myExampleMenu1,)
"__init__.py" [Modified][readonly] 135 lines --97%--                                                    132,1
```

Figure 216: My Example Menu 1 defined in __init__.py of ZenPack

Figure 216 shows the *My Example Menu 1* menu definition in *__init__.py*, where:

- id is a unique identifier
- name is the text that will be displayed in the GUI
- action refers to a page template file (using skins) or the name field in a page stanza in a configure.zcml (Zenoss 3 and beyond)
- permissions permission that a role requires to execute the action

The last line of *__init__.py* adds this new menu to the default ***factory_type*** actions for the top-level *Device* class, which is inherited by all subclasses; hence, all devices will have a *My Example Menu 1*.

15.3.3 Old and new options for page templates for menus

The *My Example Menu 1* menu is implemented by the ***action*** statement *myExampleMenuOne*. The ZenPack actually has two different definitions of this; one under the *skins* directory using old-style "wiring" in *__init__.py*, and a second, similar file under *browser/templates* which is "wired in" with *browser/configure.zcml*. Both are called *myExampleMenuOne.pt*.

The menu *.pt* files differ slightly so they can be distinguished:

- The device attribute fields displayed include *comment* in the modern *browser/templates* version.

- The title of the first table displaying attribute information includes whether it uses skins or configure.zcml.
- Otherwise, the files are identical

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples/browser/templates
File  Edit  View  Search  Terminal  Help
<tal:block metal:use-macro="here/templates/macros/page2">
<tal:block metal:fill-slot="contentPane">

<form method="post"
    name="myExampleMenuOne" tal:attributes="action string:${here/absolute_url_path}/${template/id}">

<!-- Note the tabletitle of "My Example 1 Menu Stuff from V3 directory" which distinguishes this
     V3-style menu from the V2-style menu defined under the skins subdirectory
     This menu also differs from the V2 version by including the comments field for the device.  Other
     than these differences, the menu should be the same.
-->

<tal:block metal:define-macro="myExampleMenuOne" tal:define="tabletitle string:My Example 1 Menu Stuff from V3 directory">

<tal:block metal:use-macro="here/zenuimacros/macros/zentable">
<tal:block metal:fill-slot="zentablecontents">
<!-- BEGIN TABLE CONTENTS --
    <tr>
      <td class="tableheader" align=left>SNMP Description</td>
      <td class="tablevalues" tal:content=here/snmpDescr> </td>
      <td class="tableheader" align=left>SNMP OID</td>
      <td class="tablevalues" tal:content=here/snmpOid> </td>
    </tr>
    <tr>
      <td class="tableheader" align=left>SNMP System Contact</td>
      <td class="tablevalues" tal:content=here/snmpContact> </td>
      <td class="tableheader" align=left>SNMP System Location</td>
      <td class="tablevalues" tal:content=here/snmpLocation> </td>
    </tr>
    <tr>
      <td class="tableheader" align=left>Device comment</td>
      <td class="tablevalues" tal:content=here/comments> </td>
    </tr>
    <tr>
    </tr>
<!-- END TABLE CONTENTS -->
</tal:block>
</tal:block>
</tal:block>
<tr>
</tr>
"myExampleMenuOne.pt" [Modified][readonly] 119 lines --14%--                                  17,1          Top
```

Figure 217: myExamplemenuOne.pt page template file - modern version - part 1

Both *.pt* files define the **same** three subwindows for this menu option:

- A table showing device attribute information. The title reflects which *.pt* file is actually being used.
- The standard device Performance Graphs
- The third subwindow *My Example One Menus from Objects*, shows a table with Zenoss Group information for the device, and offers a dropdown menu to demonstrate further menus.

The first part of the .pt file:

- Defines the unique name for the first form - *myExampleMenuOne*
- Defines the table title
- Defines table headings and values for:
  - SNMP Description
  - SNMP OID
  - SNMP System Contact
  - SNMP System Location

    - Device comment (new version only)

- Each of these attributes has a single value for the device. ***here*** is the object representing this device. The values are retrieved from ZODB using syntax such as:

```
here/snmpDescr
```

  where *snmpDescr* is the device attribute in ZODB.

The second subwindow defines the standard Performance Graphs for a device:

```
<!-- Now add on the standard display of all device-level graphs -->

<tal:block metal:define-macro="objectperf"
   tal:define="tabletitle string:Performance graphs for this Device">
</tal:block>

<table metal:use-macro="here/viewPerformanceDetail/macros/objectperf" />
```

This definition is provide by Zenoss core code in *$ZENHOME/Products/ZenModel/skins/zenmodel/viewPerformanceDetail.pt*.

The third subwindow defines a table for Zenoss Group information and provides a dropdown option with two submenus.

- The title of the subwindow is *My Example One Menus from Objects*
- Column headers for the table are:
    - Link to Group
    - Group Name
    - Group Description

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples/browser/templates
File  Edit  View  Search  Terminal  Help

<!-- Add on a table to display Groups for this device
     Note the form title of "My Example One Menus from Objects"
     This form also has an extra dropdown menu called by the line
           menu_id string:ExampleOneMenuObjects_list
     ExampleOneMenuObjects_list is defined in objects/objects.xml
     with 2 menuitems, myDropDownMenu1 and myDropDownMenu2.
-->


<form method="post" name="myExampleObjectMenu"
        tal:attributes="action here/absolute_url_path">
<input type="hidden" name="zenScreenName" tal:attributes="value string:myExampleMenuOne"/>
<input type="hidden" name="redirect" value="true"/>


<tal:block tal:define="objects here/groups/objectValuesAll;
    editable python:here.checkRemotePerm('ZenCommon', here);
    tableName string:exampleOneObjectMenu;
    batch python:here.ZenTableManager.getBatch(tableName,objects);
    tabletitle string:My Example One Menus from Objects;
    menu_id string:ExampleOneMenuObjects_list">
    <input type='hidden' name='tableName' tal:attributes="value tableName" />
        <tal:block metal:use-macro="here/zenuimacros/macros/zentable">
        <tal:block metal:fill-slot="zentablecontents">

        <!-- BEGIN TABLE CONTENTS -->

    <tr tal:define="message request/message | string:">
        <td class=tableheader colspan=4 tal:content="message" />
    </tr>
    <tr>
        <th tal:replace="structure python:here.ZenTableManager.getTableHeader(
                    tableName,'id','Link to Group ')"/>
        <th tal:replace="structure python:here.ZenTableManager.getTableHeader(
                    tableName,'id','Group Name')"/>
        <th tal:replace="structure python:here.ZenTableManager.getTableHeader(
                    tableName,'id','Group Description')"/>
  </tr>

"myExampleMenuOne.pt" [Modified][readonly] 119 lines --42%--                                    51,0-1        64%
```

Figure 218: myExamplemenuOne.pt page template file - modern version - part 2

- A device can be in multiple Groups so a table mechanism must be provided to list all Groups.

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples/browser/templates
File  Edit  View  Search  Terminal  Help

  <tal:block tal:repeat="mybatch batch">
  <tr tal:define="odd repeat/mybatch/odd"
      tal:attributes="class python:odd and 'odd' or 'even'">
    <td class="tablevalues">
       <input type="hidden" name="myGroupIds"
              tal:attributes="value mybatch/id"/>
       <input type="checkbox" style="float:left" name="myDelGroupIds"
              tal:condition="editable"
              tal:attributes="value mybatch/id"/>
        <div style="float:left"
             tal:define=" link python:mybatch.getPrettyLink()"
             tal:content="structure link"/>
    </td>
    <td class="tablevalues" tal:content="mybatch/id"/>
    <td class="tablevalues" tal:content="mybatch/description"/>
  </tr>
  </tal:block>
        <!-- END TABLE CONTENTS -->

        </tal:block>
        </tal:block>
</tal:block>
</form>


</tal:block>
</tal:block>

"myExampleMenuOne.pt" [Modified][readonly] 119 lines --100%--                                   119,5         Bot
```

Figure 219: myExamplemenuOne.pt page template file - modern version - part 3

- Figure 219 shows a typical structure for a table with multiple rows where:

  `<tal:block tal:repeat="mybatch batch">`

- sets up a repeating **batch** called ***mybatch*** - this will supply multiple table values, where *mybatch* takes on the value of the Group object for each valid Zenoss Group

```
<tr tal:define="odd repeat/mybatch/odd"
   tal:attributes="class python:odd and 'odd' or 'even'">
```

- Define table rows that will gather attribute values for each row. Odd and even rows will have different highlighting to make them stand out.
- The first column in each row will have table data (td) that is a **link** to the Zenoss Group. The value will be this Group's *id*.

```
 <td class="tablevalues">
  <input type="hidden" name="myGroupIds"
     tal:attributes="value mybatch/id"/>
  <input type="checkbox" style="float:left" name="myDelGroupIds"
     tal:condition="editable"
     tal:attributes="value mybatch/id"/>
  <div style="float:left"
      tal:define=" link python:mybatch.getPrettyLink()"
      tal:content="structure link"/>
</td>
```

- The second column in each row will have table data (td) that is the Group Name. The value will also be this Group's *id*.

```
   <td class="tablevalues" tal:content="mybatch/id"/>
```

- The third column in each row will have table data (td) that is the Group Description. The value will be this Group's *description* attribute.

```
    <td class="tablevalues" tal:content="mybatch/description"/>
```

To find the correct *.pt* file to use for a device's *My Example Menu 1* menu, all *configure.zcml* files in the ZenPack are searched first. If a matching definition for the *myExampleMenuOne* **action** is found, it is used. If no definition is found then it falls back to using the old-style skins technique and searches for a matching file in the ZenPack's *skins* directory hierarchy.

In order to demonstrate both techniques, *browser/configure.zcml* has been coded with:

```
for="..ExampleDevice.ExampleDevice"
```

which ensures that this definition is **only** valid for devices of object class *ExampleDevice*. Thus, devices of any other object class will not pass this test and will default to using the old skins *.pt* file.

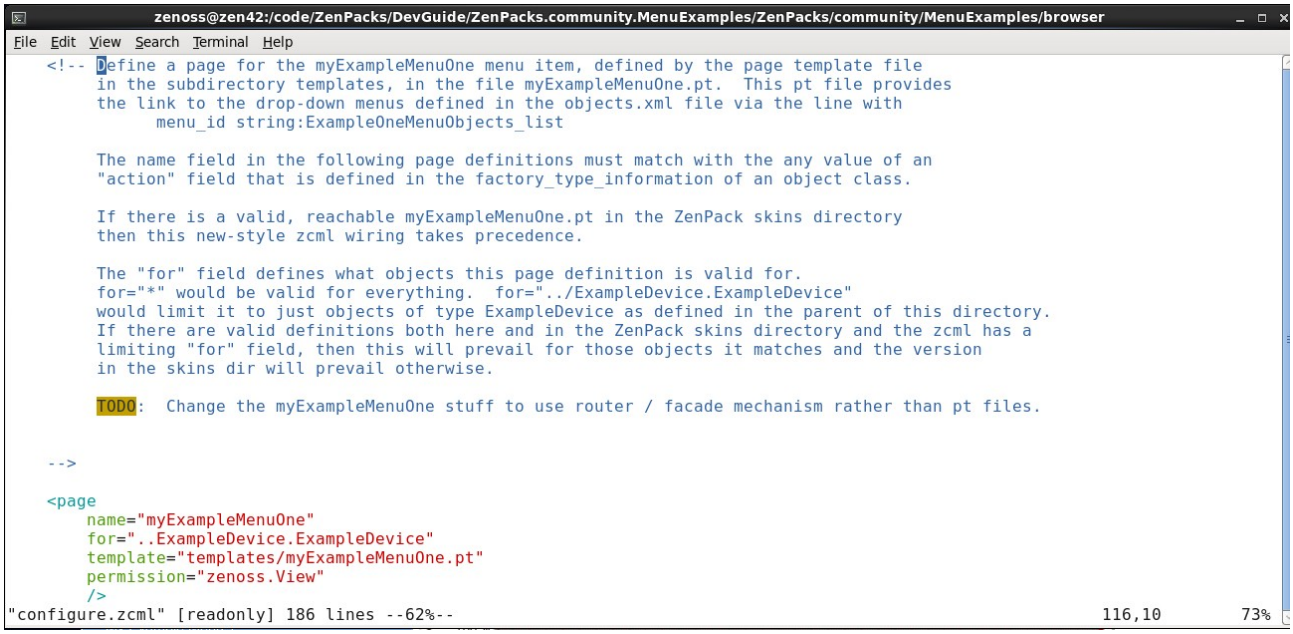


Figure 220: browser/configure.zcml for myExampleMenuOne

The *.pt* file must be specified relative to this *configure.zcml*; that is, down into the *templates* directory.

The *for* line is also specified relative to this *configure.zcml*; that is, up one directory to the base directory of the ZenPack. The object class *ExampleDevice* (the second ExampleDevice) is found in the Python module *ExampleDevice* (the first ExampleDevice).

15.3.4 New-style menus limited to specific device types

My Example Menu 2 is defined only for devices of object class *ExampleDevice*; it does not appear at all in the left-hand menu for other devices. This is a simple extension from the previous section.

browser/configure.zcml has a *page* definition for this menu, including a *for* statement.

```
<!-- myExampleMenuTwo is applicable only to ExampleDevice objects in for field -->

  <page
    name="myExampleMenuTwo"
    for="..ExampleDevice.ExampleDevice"
    template="templates/myExampleMenuTwo.pt"
    permission="zenoss.View"
    />
```

It specifies the page template file in t*emplates/myExampleMenuTwo.pt*, which is very similar to the Menu 1 example. It has two subwindows, one displaying a slightly different selection of device attributes and also includes the standard *Performance Graphs* subwindow for the device.

Importantly, there is no global definition in *__init__.py* that refers to this menu but referring back to the *ExampleDevice.py* file that defines the new object class, there is an extension of the standard *Device factory_type_information*, explicitly for this device type.

```
# The menus for this particular device class will have all the standard menus
# for a Device and then add on this new menu. The "action" field must match
# the name field in an entry in configure.zcml.
```

```
    factory_type_information = deepcopy(Device.factory_type_information)
    factory_type_information[0]['actions'] += (
        { 'id'          : 'ExampleDevice'
        , 'name'        : 'My Example Menu 2 (Example Devices only)'
        , 'action'      : 'myExampleMenuTwo'
        , 'permissions'   : ( Permissions.view,) },
        )
```

The *action* field in the *factory_type_information* **must** match the *name* field in an entry in *configure.zcml*.

If the object class of the device does **not** match the *for* statement in *configure.zcml*, there is no default to resort to, and no menu is shown.

Figure 221: My Example Menu 2 page template file

15.3.5 Dropdown menus shipped in objects.xml

My Example Menu 1 has three subwindows where the third, *My Example One Menus from Objects,* has a dropdown action with two submenus.

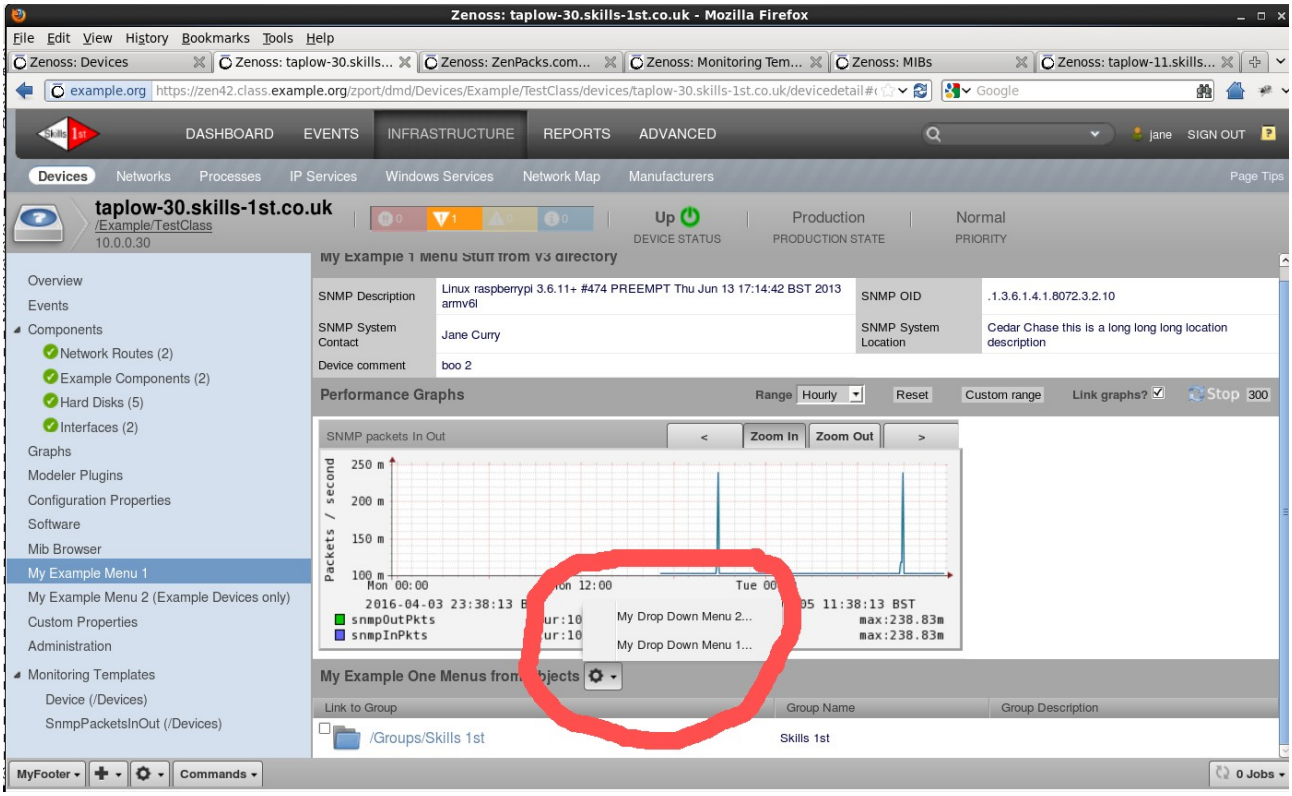


Figure 222: Dropdown menus shipped with objects.xml

Menus can be delivered as objects in *objects/objects.xml*. *mydropDownMenu1* and *mydropDownMenu2* are delivered this way. They are called from both versions of the *myExampleMenuOne.pt* file (both the skins version and the *browser/templates* version), by the line:

```
menu_id string:ExampleOneMenuObjects_list
```

The code in *objects.xml* has been built by reference to other examples.

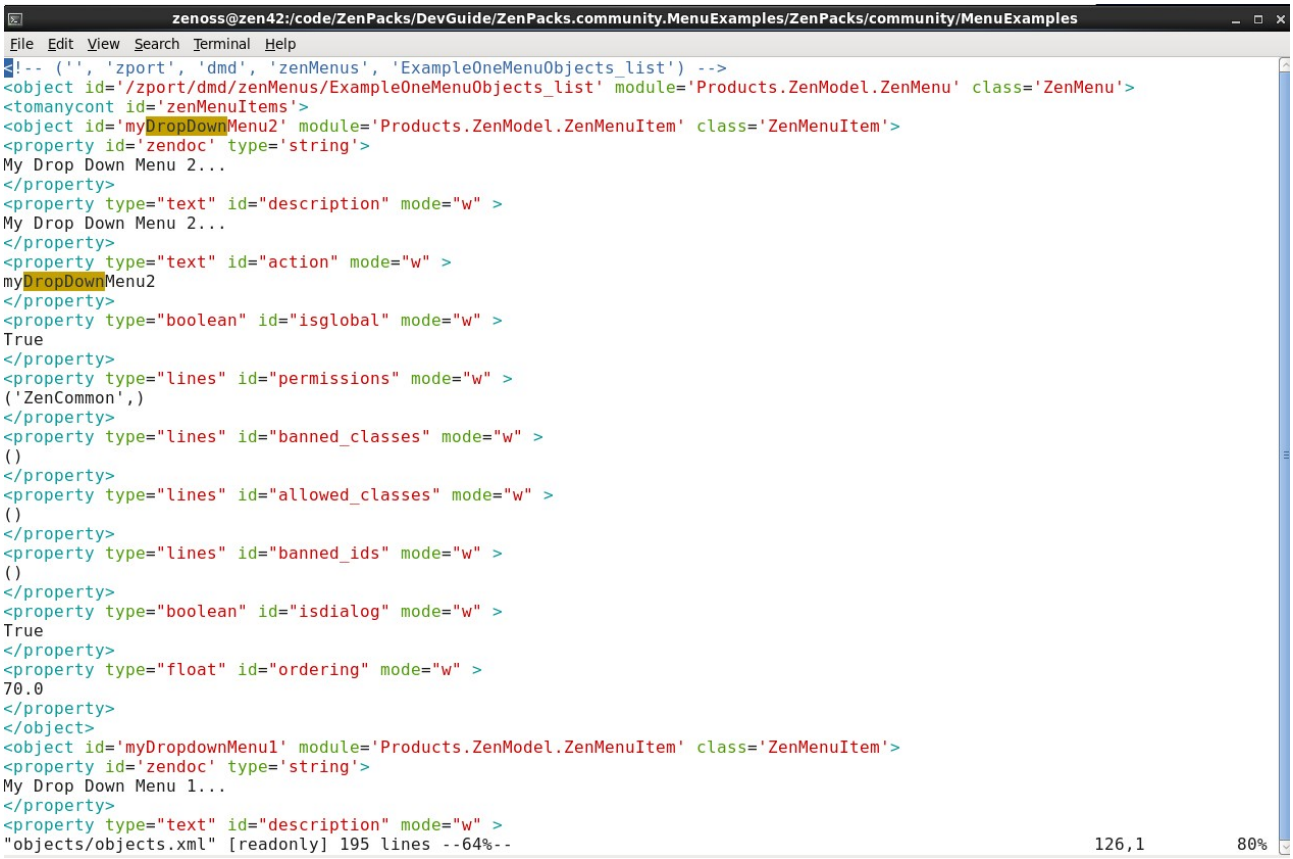


Figure 223: Definition of a menu list and its menus in objects.xml

In Figure 223, note:

- There **must** be a menu list defined, of object class ***ZenMenu***, whose name matches the *menu_id* referenced in the *.pt* file - *ExampleOneMenuObjects_list*. The menu list contains one or more ***zenMenuItems***.

```
 <object id='/zport/dmd/zenMenus/ExampleOneMenuObjects_list'
      module='Products.ZenModel.ZenMenu' class='ZenMenu'>
<tomanycont id='zenMenuItems'>
```

- Each menu item is then defined with various properties.

```
 <object id='myDropDownMenu2' module='Products.ZenModel.ZenMenuItem' class='ZenMenuItem'>
<property id='zendoc' type='string'>
My Drop Down Menu 2...
</property>
<property type="text" id="description" mode="w" >
My Drop Down Menu 2...
</property>
<property type="text" id="action" mode="w" >
myDropDownMenu2
</property>
<property type="boolean" id="isglobal" mode="w" >
True
</property>
<property type="lines" id="permissions" mode="w" >
('ZenCommon',)
.........
<property type="float" id="ordering" mode="w" >
70.0
```

```
</property>
```

- The *description* property defines the name that will be seen in the GUI.
- The *ordering* property allows control over positioning within the menu list, where lower numbers are nearer the top of the list.

- It is the *action* stanza in the menu item definition that **must** match with the *name* field of an entry in *browser/configure.zcml*.
- Both these dropdown menus are defined in Zenoss-3 style with c*onfigure.zcml* and *.pt* files in *browser/templates*.

zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples

File Edit View Search Terminal Help

```
    <!-- Next two stanzas are for the dropdown menus that are defined in
         objects.xml.  The name field must match with the action field in the MenuItem
         definition in objects.xml (ie. myDropDownMenu1 / myDropDownMenu2 ).  The layout
         for these menus is in templates/myDropDownMenu1.pt and
         templates/myDropDownMenu2.pt respectively.

         myDropDownMenu1 deliberately limits the scope to
            for="..ExampleDevice.ExampleDevice"
         as templates/myDropDownMenu1.pt uses the createComment method which is only
         defined for the ExampleDevice object class.  The result is that for devices of
         other object classes, the submit window simply hangs and can be closed harmlessly.

         For ExampleDevice devices note that after clicking the OK button, control is
         returned to the defaultdetails view as this is the default view as defined in
         the factory information for a device.
    -->

    <page
        name="myDropDownMenu1"
        for="..ExampleDevice.ExampleDevice"
        template="templates/myDropDownMenu1.pt"
        permission="zenoss.View"
        />

    <page
        name="myDropDownMenu2"
        for="*"
        template="templates/myDropDownMenu2.pt"
        permission="zenoss.View"
        />

</configure>
"browser/configure.zcml" [readonly] 186 lines --100%--                    186,10        Bot
```

Figure 224: configure.zcml "wiring" for dropdown menus shipped in object.xml

mydropDownMenu1 is restricted for use only by devices of object class *ExampleDevice* (because it uses the *createComment* method which is only defined for the ExampleDevice object class). The result is that for devices of other object classes, the submit window simply hangs and can be closed harmlessly.

myDropDownMenu2 is valid for all device object classes and produces a popup window with a few device attributes.

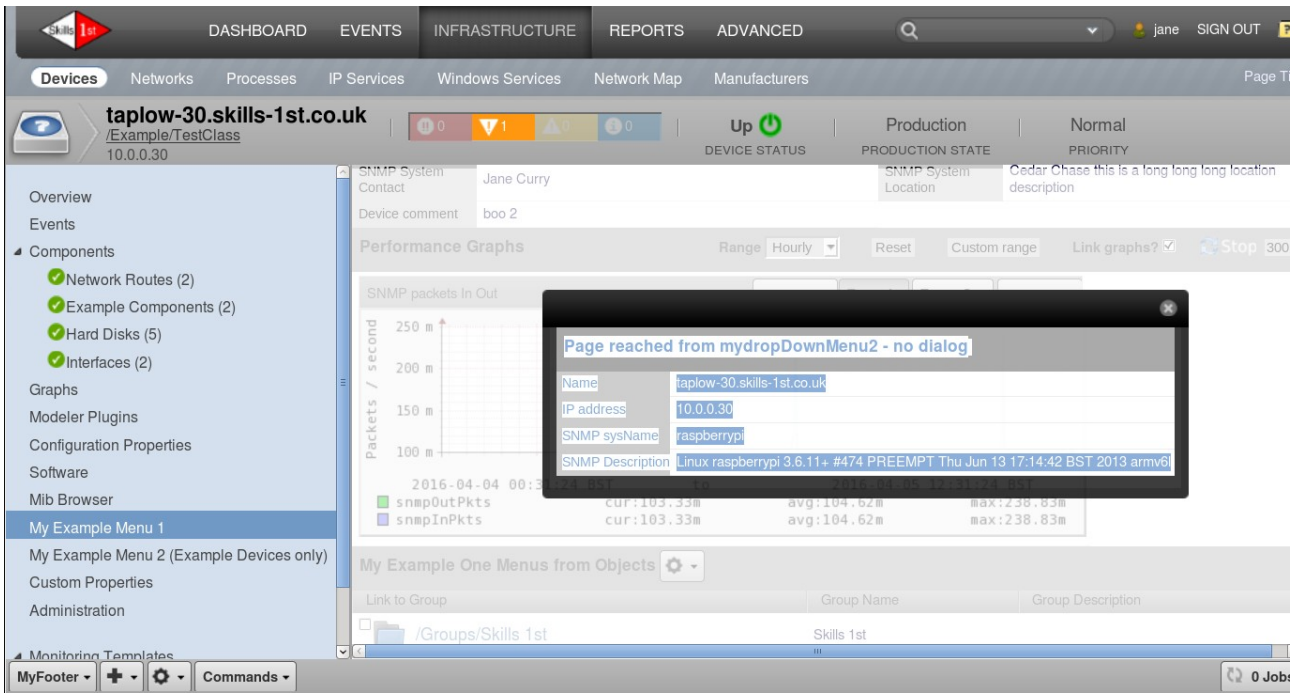


Figure 225: Popup window from myDropDownMenu2 with device attributes

The layout of this popup window is defined by *templates/myDropDownWindow2.pt*.

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples
File  Edit  View  Search  Terminal  Help
<tal:block metal:use-macro="here/templates/macros/page2">
<tal:block metal:fill-slot="contentPane">

<form method="post"
    name="myDropDownMenu2" tal:attributes="action string:${here/absolute_url_path}/${template/id}">

<tal:block metal:define-macro="myDropDownMenu2" tal:define="tabletitle string:Page reached from mydropDownMenu2 - no dialog">

<tal:block metal:use-macro="here/zenuimacros/macros/zentable">
<tal:block metal:fill-slot="zentablecontents">

<!-- BEGIN ExampleDevice TABLE CONTENTS -->
    <tr>
      <td class="tableheader" align=left>Name</td>
      <td class="tablevalues" tal:content=here/titleOrId> </td>
    </tr>
    <tr>
      <td class="tableheader" align=left>IP address</td>
      <td class="tablevalues" tal:content=here/manageIp> </td>
    </tr>
    <tr>
      <td class="tableheader" align=right>SNMP sysName</td>
      <td class="tablevalues" tal:content=here/snmpSysName> </td>
    </tr>
    <tr>
      <td class="tableheader" align=left>SNMP Description</td>
      <td class="tablevalues" tal:content=here/snmpDescr> </td>
    </tr>
<!-- END ExampleDevice TABLE CONTENTS -->
</tal:block>
</tal:block>
</tal:block>
<tr>
</tr>

</form>

</tal:block>
</tal:block>
"browser/templates/myDropDownMenu2.pt" [readonly] 40 lines --2%--                1,1           Top
```

Figure 226: myDropDownMenu2.pt in browser/templates defining the popup window for Menu 2

myDropDownMenu1 prompts for a comment for the device and uses the *createComment* method to update the comments attribute for the device.

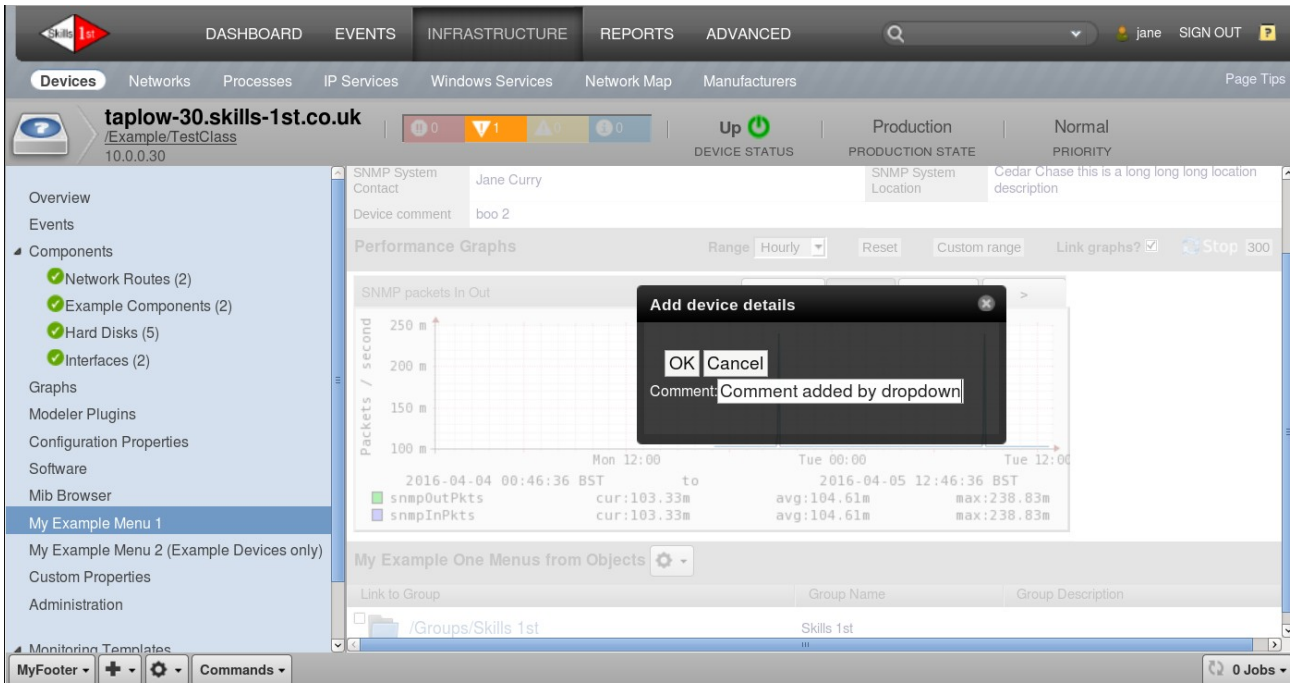


Figure 227: Popup menu from myDropDownMenu1 that prompts for input

browser/templates/myDropDownMenu1.pt defines the popup box and gathers the input value into a variable called ***comments***.

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExampl
File  Edit  View  Search  Terminal  Help
<h2>Add device details</h2>

<form method="post"
    name="createComment" tal:attributes="action context/absolute_url_path">

    <table>
    <tr><td>
    <span id="comment_label" style="color:white;">Comment: </span>
    </td><td>
        <input id="newComment" name="comments" style="width:200px">
    </td></tr>

<div id="dialog_buttons">

<!-- Note that after clicking the OK button, control is returned to the defaultdetails view
        as this is the default view as defined in the factory information for a device.
-->

<input tal:attributes="type string:submit;
                       value string:OK"
                       name="createComment:method" />

<input tal:attributes="id string:dialog_cancel;
                       type string:button;
                       value string:Cancel;
                       onclick string:$$('dialog').hide()" />

</div>
</table>
<br>
</form>
"browser/templates/myDropDownMenu1.pt" [readonly] 32 lines --3%--          1,1           Top
```

Figure 228: myDropDownMenu1.pt defining input popup and calls createComment method

It calls the *createComment* method that is defined in *ExampleDevice.py*.

Figure 229: createComment method in ExampleDevice.py

In Figure 229 note that:

- The name of the function, *createComment*, must match with the *.pt* file *dialog_buttons* section, where the function name matches the *name* field for the *OK* input.

- The function parameter *comments* must match the *name* parameter on the input line in the *.pt* file and these **both** need to match the object attribute you are changing in ZODB.

For devices of object class *ExampleDevice*, note that after clicking the *OK* button, control is returned to the *defaultdetails* view as this is the default view as defined in the factory information for a *Device*.

15.3.6 Adding items to the *Display* dropdown for a component

Any component display for a device includes a *Display* dropdown menu in the middle of the panel.

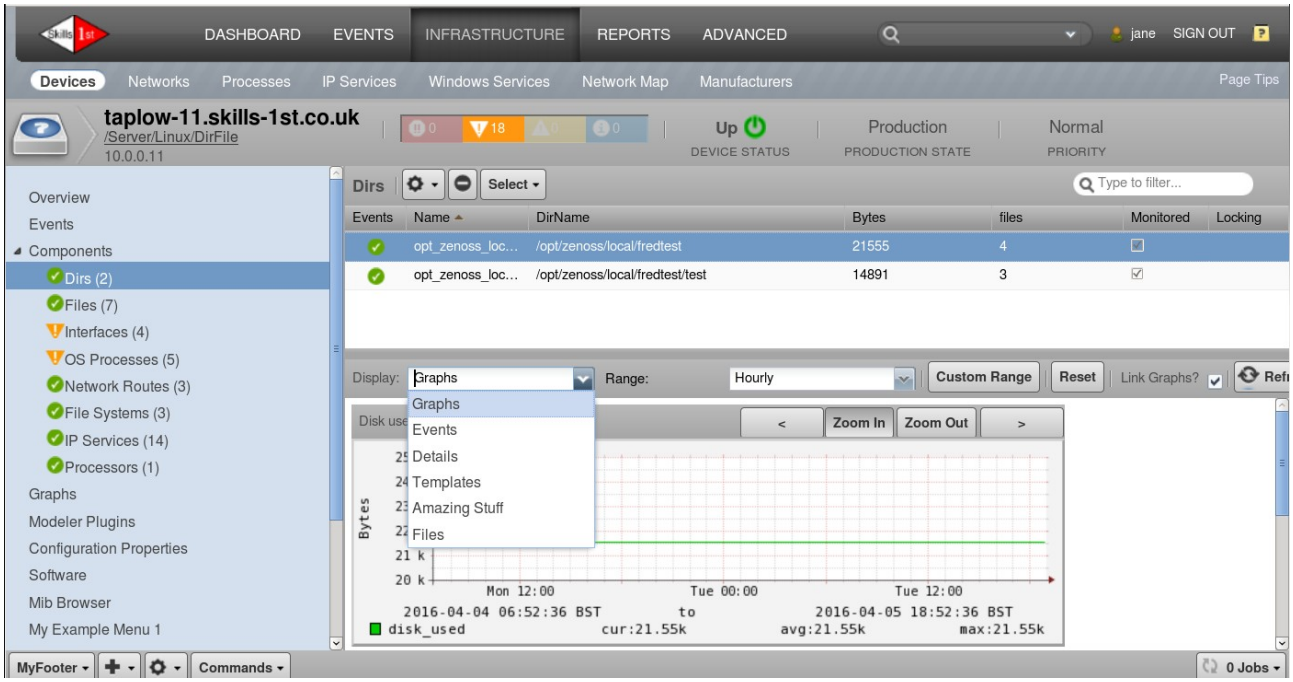


Figure 230: Display dropdown for a component

The default menu items include:

- Graphs
- Events
- Details
- Templates

Other items can be added either globally for all components or specific to a particular component.

zenpacklib is excellent for automatically creating menu links to related component objects as seen in the *Files* menu in Figure 230.

Pre zenpacklib, new menu items for the Display dropdown were coded in the object file defining the component. *ExampleComponent.py* contains:

```
# Defining the "perfConf" action here causes "Example Component
 # Template" to be available in
# the display dropdown for components of this type.
# The action "objTemplates" is a standard Zenoss page template defined
# in $ZENHOME/Products/ZenModel/skins/zenmodel/objTemplates.pt

factory_type_information = ({
  'actions': ({
    'id': 'perfConf',
    'name': 'Example Component Template',
    'action': 'objTemplates',
    'permissions': (ZEN_CHANGE_DEVICE,),
  },),
},)
```

This results in an extra *Example Component Template* option. In practise, this is a duplicate of the standard *Templates* menu.

A global *Display* item, applicable to all component types, can be added by coding a JavaScript file and linking it in with *configure.zcml* “wiring”.

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples/browser
File  Edit  View  Search  Terminal  Help
<?xml version="1.0" encoding="utf-8"?>
<configure xmlns="http://namespaces.zope.org/browser">

    <!-- A resource directory contains static web content. -->
    <!-- example is simply a unique name. The directory "resources" is directly under this directory -->
    <!-- The name "example" in the resourceDirectory stanza must match what follows ++resource++ in viewlet -->

    <resourceDirectory
        name="example"
        directory="resources"
        />

    <!-- This amazing viewlet uses a for stanza to apply to all types of devices
         Note that the manager stanza is interfaces.IHeadExtraManager
         This adds in an extra dropdown menu for Components from the Display box
    -->

    <viewlet
        name="js-amazing-componentoption"
        paths="/++resource++example/js/amazing.js"
        weight="9"
        for="Products.ZenModel.Device.Device"
        manager="Products.ZenUI3.browser.interfaces.IHeadExtraManager"
        class="Products.ZenUI3.browser.javascript.JavaScriptSrcBundleViewlet"
        permission="zope2.Public"
        />
"configure.zcml" [Modified][readonly] 167 lines --0%--                    1,5           Top
```

Figure 231: browser/configure.zcml with Display dropdown for "amazing" item

The viewlet entry in *browser/configure.zcml* has a *for* statement that ensures the option is applicable to the top-level *Device* class and all subclasses.

Note that the manager stanza is:

```
Products.ZenUI3.browser.interfaces.IHeadExtraManager
```

The JavaScript file is *amazing.js* under *browser/resources/js*.

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples/browser
File  Edit  View  Search  Terminal  Help
Zenoss.nav.appendTo('Component', [{
    id: 'amazing_stuff_bottom_panel',
    text: _t('Amazing Stuff'),
    xtype: 'componentpanel',
    action: function(node, target, combo) {
        var uid = combo.contextUid;
        alert(uid);
    }       }]);

/* Next line simply pops up a message on the screen
alert('just added amazing stuff');
*/

"resources/js/amazing.js" [readonly] 14 lines --7%--                      1,1           Top
```

Figure 232: browser/resources/js/amazing.js defines a Display menu option called Amazing Stuff

The menu simply produces an alert popup with the UID of the component.

15.3.7 Menu on INFRASTRUCTURE -> Devices to add new device type

The standard Zenoss Core menus have options to add a new device from the "+" dropdown menu at the top of the list of devices. It is possible to add an extra option to that menu that is specific for a particular device object class.

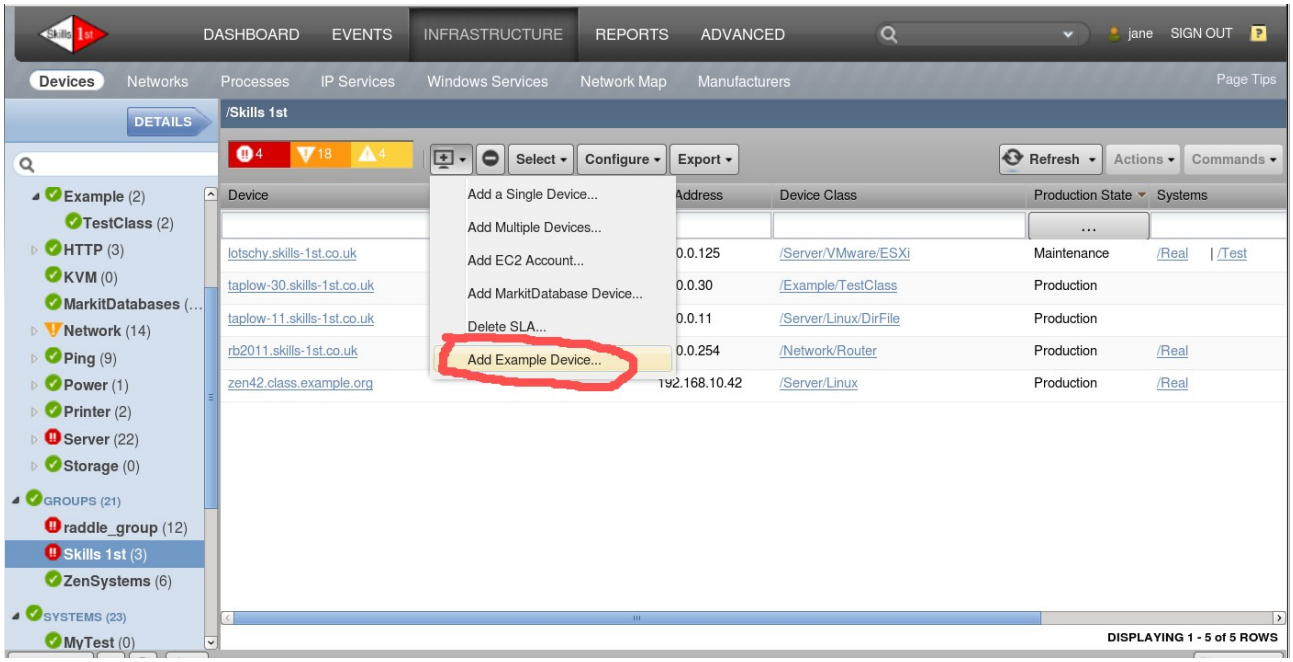


Figure 233: Add Example Device menu

This is done with a viewlet stanza in *browser/configure.zcml* that points to a JavaScript file, *add_example_device_option.js*.

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples/browser
File  Edit  View  Search  Terminal  Help
    <!-- Define an add device entry
    -->
    <viewlet
        name="js-add_example_device_option"
        paths="/++resource++example/js/add_example_device_option.js"
        weight="10"
        manager="Products.ZenUI3.browser.interfaces.IJavaScriptSrcManager"
        class="Products.ZenUI3.browser.javascript.JavaScriptSrcBundleViewlet"
        permission="zope2.Public"
        />

"configure.zcml" [readonly] 186 lines --30%--                    57,0-1        26%
```

Figure 234: configure.zcml "wiring" for Add Example Device menu

The JavaScript file creates a new Zenoss **Action** that prompts for hostname or IP address, community and comment fields and then submits a job to create the new device.

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples/browser/resources/js
File  Edit  View  Search  Terminal  Help
(function(){

// Ensure that the following var name matches the method defined in
//   interfaces.py, routers.py and facades.py
// Also ensure the var name matches with the Zenoss.extensions.adddevice.push statement
//   at the end of this file

var add_ExampleDevice = new Zenoss.Action({
    text: _t('Add Example Device') + '...',
    id: 'addExampleDevice-item',
    permission: 'Manage DMD',
    handler: function(btn, e){
        var win = new Zenoss.dialog.CloseDialog({
            width: 300,
            title: _t('Add Example Device'),
            items: [{
                xtype: 'form',
                buttonAlign: 'left',
                monitorValid: true,
                labelAlign: 'top',
                footerStyle: 'padding-left: 0',
                border: false,
                // Ensure that name field of items match the attribute names
                //   that you want to populate
                items: [{
                    xtype: 'textfield',
                    name: 'deviceIp',
                    fieldLabel: _t('Hostname or IP'),
                    id: "exampleDeviceTitleField",
                    width: 260,
                    allowBlank: false
                }, {
                    xtype: 'textfield',
                    name: 'community',
                    fieldLabel: _t('RO Community'),
                    id: "exampleDeviceRoCommunityField",
                    width: 260,
"add_example_device_option.js" [readonly] 108 lines --0%--                    1,12          Top
```

Figure 235: JavaScript for Add Example device menu - part 1 - defining the popup and input fields

In Figure 235, note:

- The *var* name must match the method defined in *interfaces.py, routers.py* and *facades.py* - ***add_exampleDevice***
- The dropdown menu text is defined as *Add Example Device...* (second highlighted line).
- A dialog window is defined whose title is *Add Example Device*
- Each item in the dialog box has a *fieldLabel*, eg *Hostname or IP*
- The *allowBlank* field for each item can mandate that an entry is specified

Figure 236: Popup dialog for adding a new Example Device

The second part of the JavaScript file defines the buttons for the user to select *ADD* or *CANCEL*.

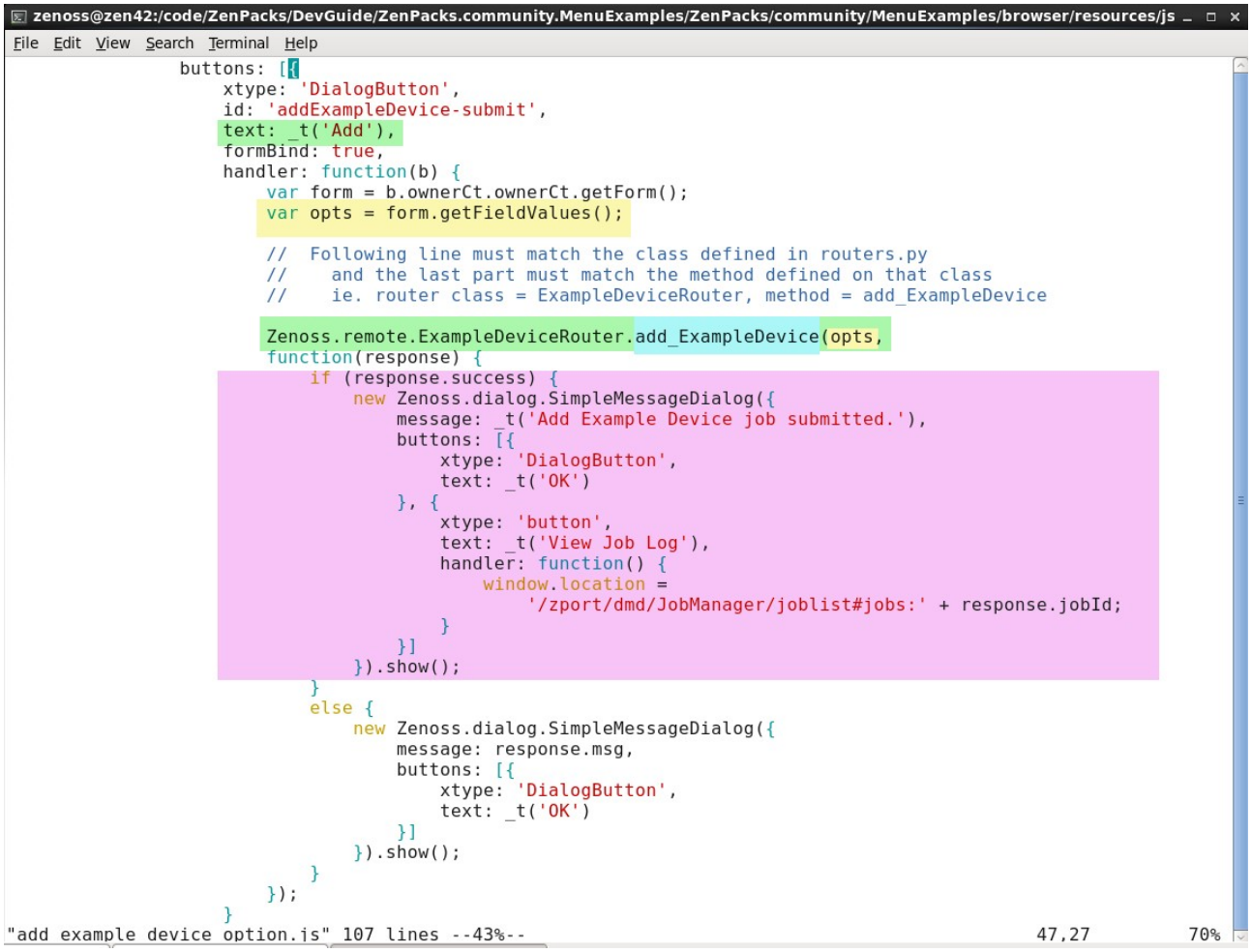


Figure 237: JavaScript for Add Example device menu - part 2 - defining the Add button

In Figure 237:

- The input fields are delivered to the *opts* variable
- *opts* is passed to a router construct, *Zenoss.remote.ExampleDeviceRouter.add_ExampleDevice(opts)*, where:
  - The remote router name must match the class defined in *routers.py* - *ExampleDeviceRouter*
  - The last element of the router construct must match the **method** defined on that class - *add_ExampleDevice*
- Different message dialogs are produced, depending on whether action was successfully taken. A successful response results in a box with two buttons:
  - *OK*
  - *View Job Log* which links to the Jobs list

- An unsuccessful response should have a box with the reason for failure and an *OK* button

The last part of the JavaScript file defines the *CANCEL* button for the A*dd Example Device* dialog.

Figure 238: JavaScript for Add Example device menu - part 3 - CANCEL and menu addition

It also arranges to “push” the new Zenoss Action onto the existing *adddevice* menu. Note that the *push* parameter must match the *var* name at the top of the JavaScript file - *add_ExampleDevice*.

15.3.7.1 Routers and facades

Routers and **facades** provide a means to handle objects. A facade is code that actually modifies objects; a router provides access to the facade, supplying the correct parameters. The router can be considered as a translation layer between the browser and the facade; thus, provided the name and parameters are maintained by the router, the underlying facade code may be changed.

routers.py in the base directory of the ZenPack, contains the definitions for routers and their functions. Typically a router calls a facade, defined in *facades.py*.

Router names, their functions and their parameters must all match up between the *routers.py* / f*acades.py* entries and the JavaScript that calls the router.

If a ZenPack provides several routers or facades then create subdirectories in the base directory of the ZenPack called *routers* and *facades* and place individual router and facade files under the appropriate directory.

add_example_device_option.js calls the *ExampleDeviceRouter*. with the *add_ExampleDevice(self, deviceIp, community, comment)* method.

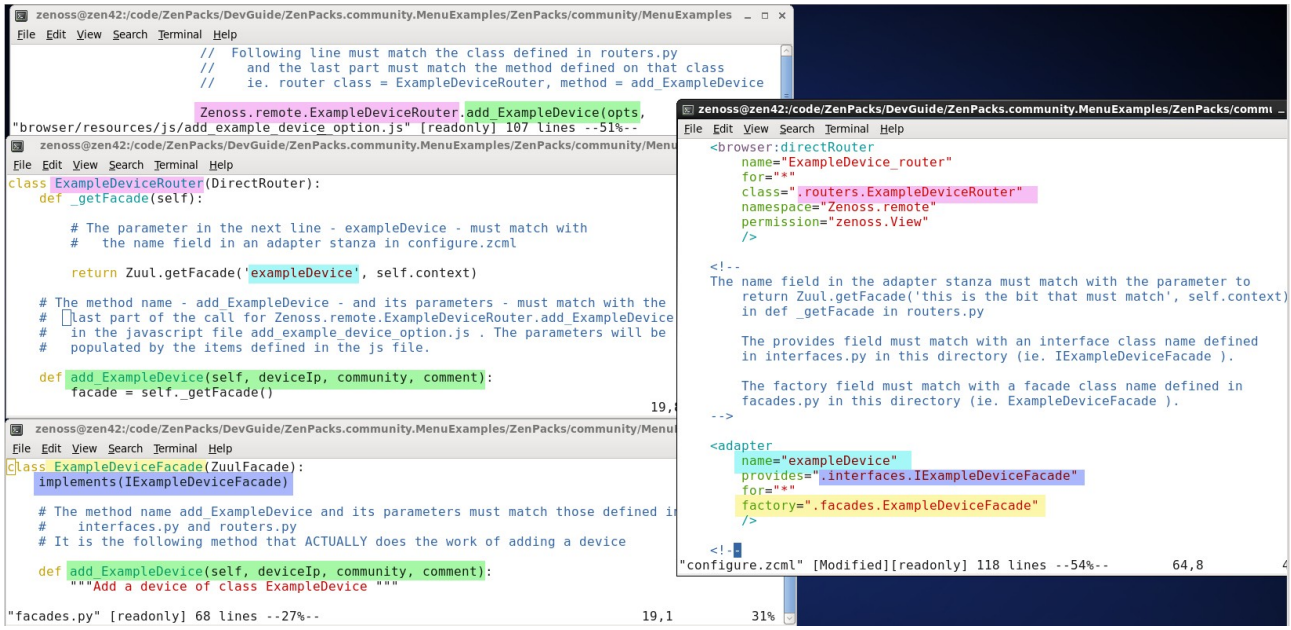


Figure 239: JavaScript, router, facade and configure.zcml for Add Example Device menu

The parameters for deviceIp, community and comment gathered by the JavaScript, are passed to the router method in the *opts* variable; *opts* must deliver the correct number and type of variables. The router passes them to the facade.

zcml "wiring" is required in the **top-level** *configure.zcml* for the router. It must also provide an adapter for the facade.

interfaces.py (in the top-level directory) must have an entry for the interface for the facade, matching any functions and their parameters.

```
# The name of the IFacade class here ( IExampleDeviceFacade ) must match
# what is defined in an adapter stanza's
# provides=".interfaces. this_is_the_bit_that_must_match"
#  ie. IExampleDeviceFacade in configure.zcml
# The method name and parameters must match those defined for
# the facade that implements
#  IExampleDeviceFacade in facades.py (ie. add_ExampleDevice )

class IExampleDeviceFacade(IFacade):
  def add_ExampleDevice(self, deviceIp, community, comment):
    """Add a device of class ExampleDevice"""
```

The facade is what **actually** implements functionality on an object. The collector for the new device is hard-coded to be *localhost*.

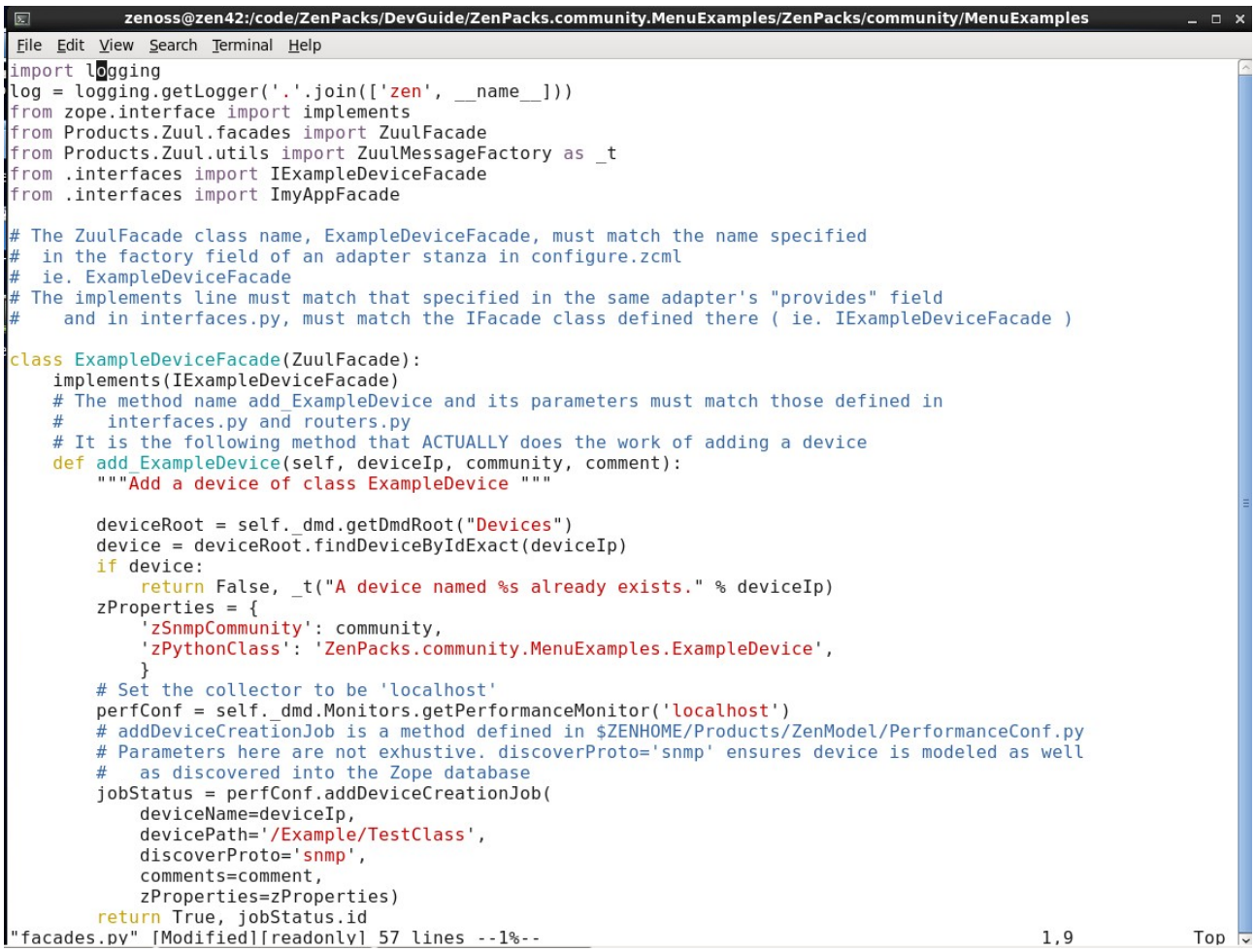


Figure 240: facades.py showing the code to Add an Example device

The *addDeviceCreationJob* method is then called on the collector object using the parameters passed to *add_ExampleDevice*. This creates a job to add the device, returning the job status id.

15.3.8 New items for left-hand DeviceClass Action menu

The *Action* (gear) icon at the bottom of the left-hand menu on *INFRASTRUCTURE -> Devices* can have new items added to it.

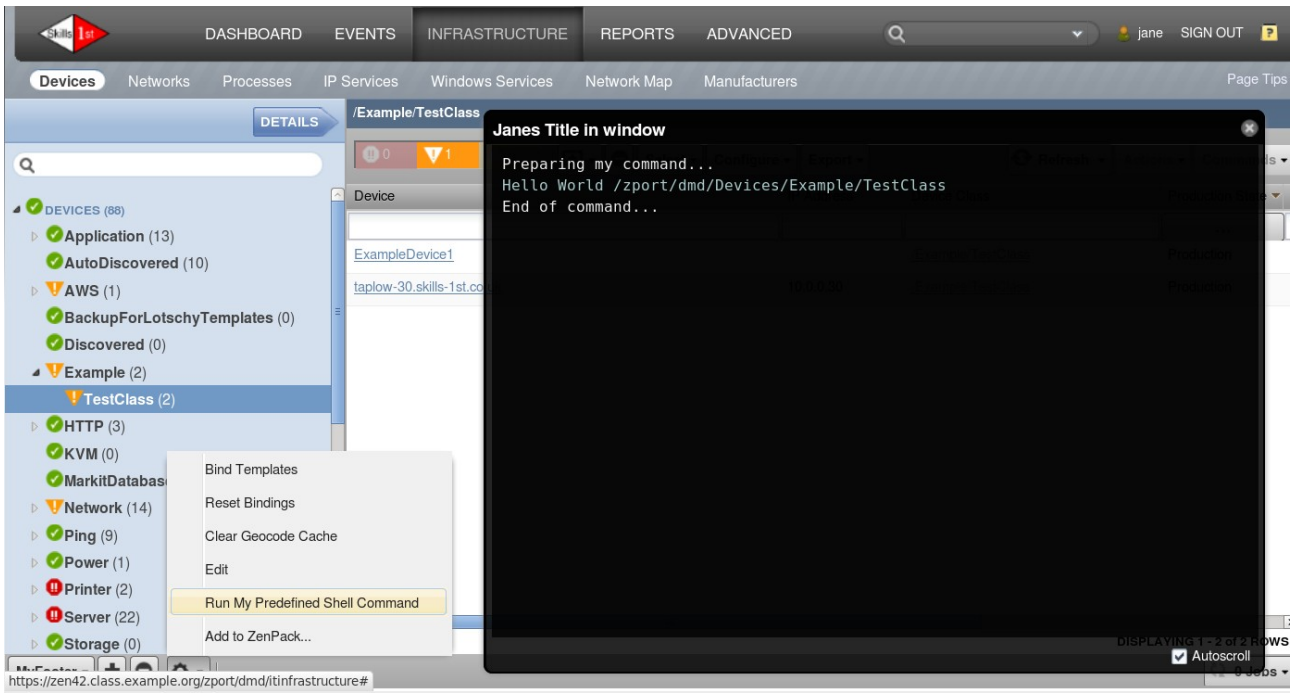


Figure 241: customized items on DeviceClass Action menu

Actions chosen apply to any selected device classes. The menu has been extended to run a predefined command that produces a popup window with the command output.

A viewlet entry is required in *browser/configure.zcml* that points to the JavaScript file *run_my_predefined_command.js*.

A page entry is also required to show the output from the command, where the class field defines an entry in *command.py* (in the ZenPack base directory) to actually run the command.

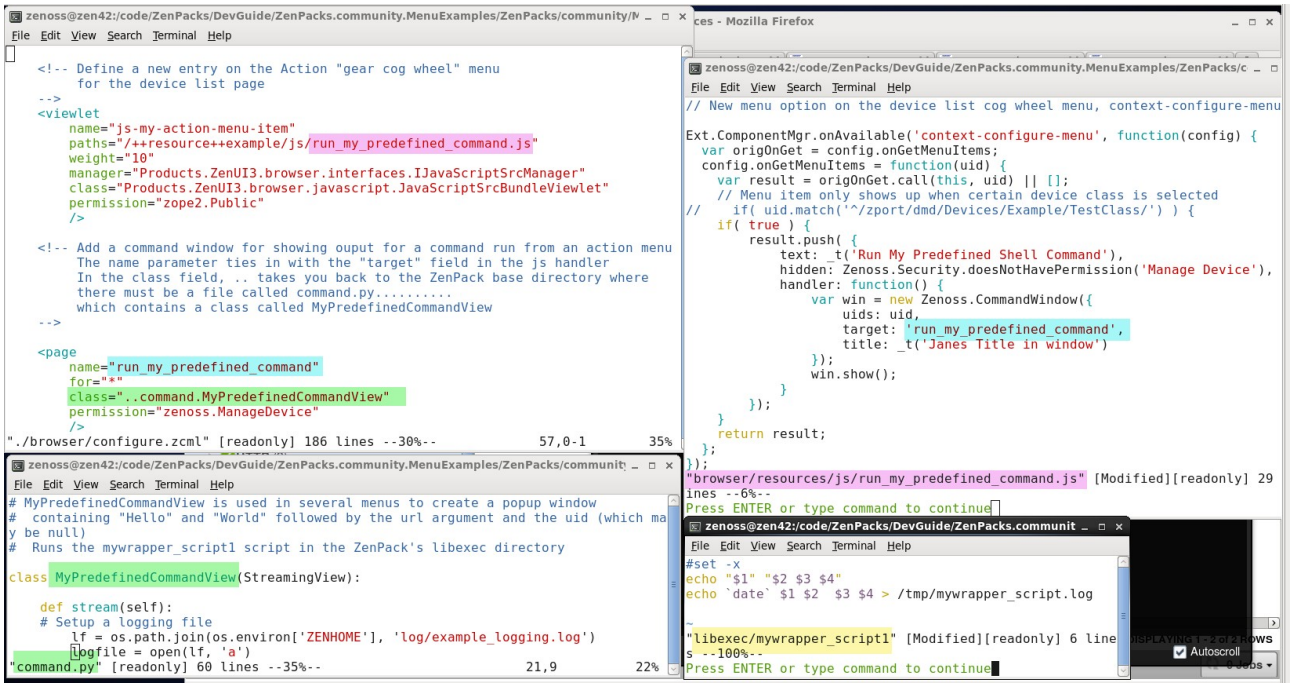


Figure 242: configure.zcml, JavaScript file, command.py and shellscript to implement device class menu item

In Figure 242, note:

- *browser/configure.zcml* references the JavaScript file in the *paths* statement of the *viewlet* stanza - *run_my_predefined_command.js*
- The JavaScript file uses the *target* statement to reference the *page* stanza in *browser/configure.zcml - run_my_predefined_command*
- *browser/configure.zcml* references the command entry in *command.py,* in the *class* statement - *MyPredefinedCommandView*
- The *MyPredefinedCommandView* command runs *mywrapper_script1* in the ZenPack's *libexec* directory

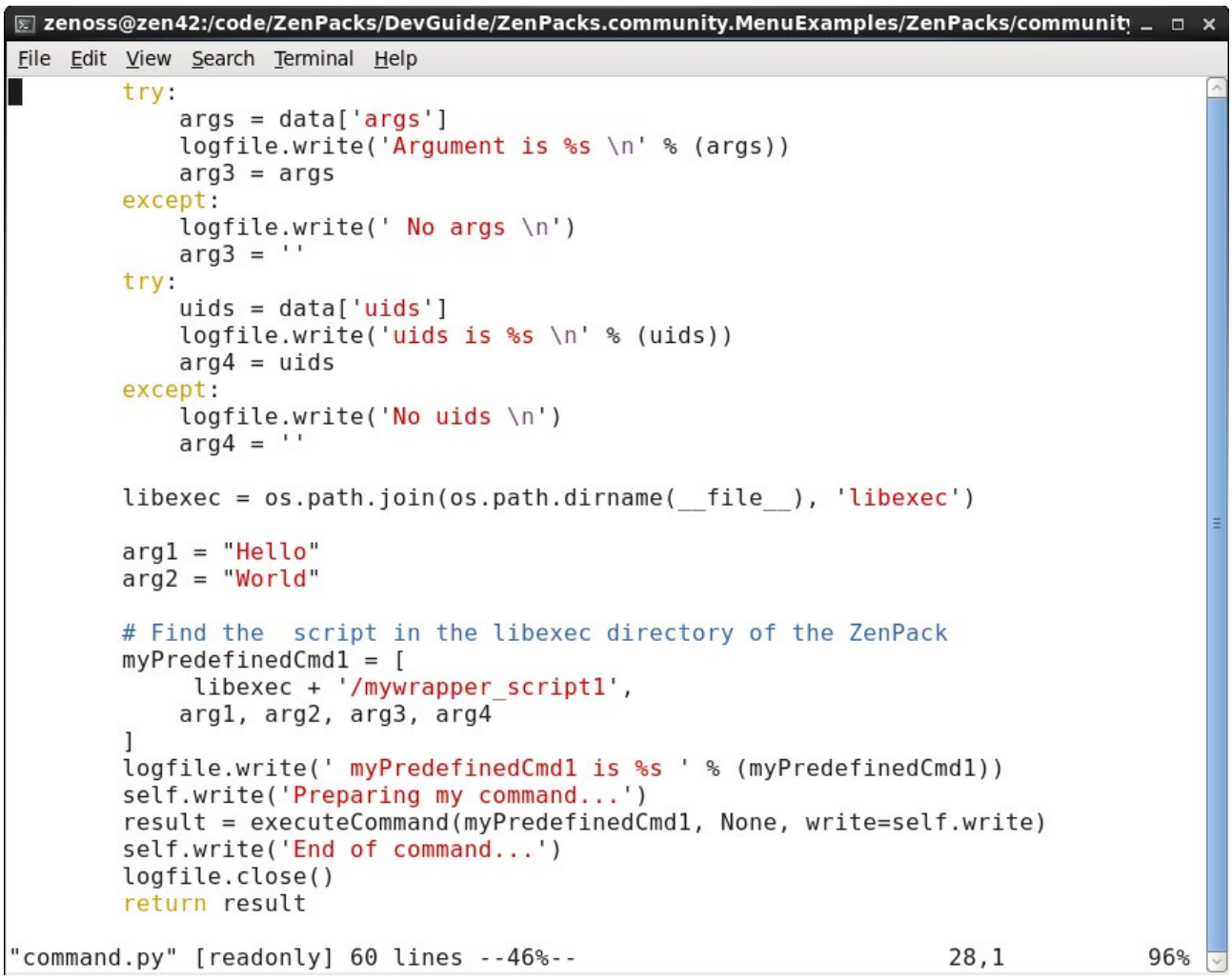


Figure 243: MyPredefinedCommandView code to call libexec/mywrapper_script1

The *MyPredefinedCommandView* class in *command.py* and uses both literal parameters (*arg1* and *arg2*) and parameters passed from the calling window. Data is delivered from the browser with the statement:

```
# data is a list that will contain 2 elements:
#  the url argument and the uid
data = unjson(self.request.get('data'))
```

where *data* is a list containing the url of the currently selected object (which becomes *arg3*) and its uid, which may be blank (which becomes *arg4*).

The actual command is in the *libexec* subdirectory as *mywrapper_script1*. It simply echos the four parameters.

15.3.9 Adding new items to a device's Action menu

It is possible to add extra items to the Action menu for a device.

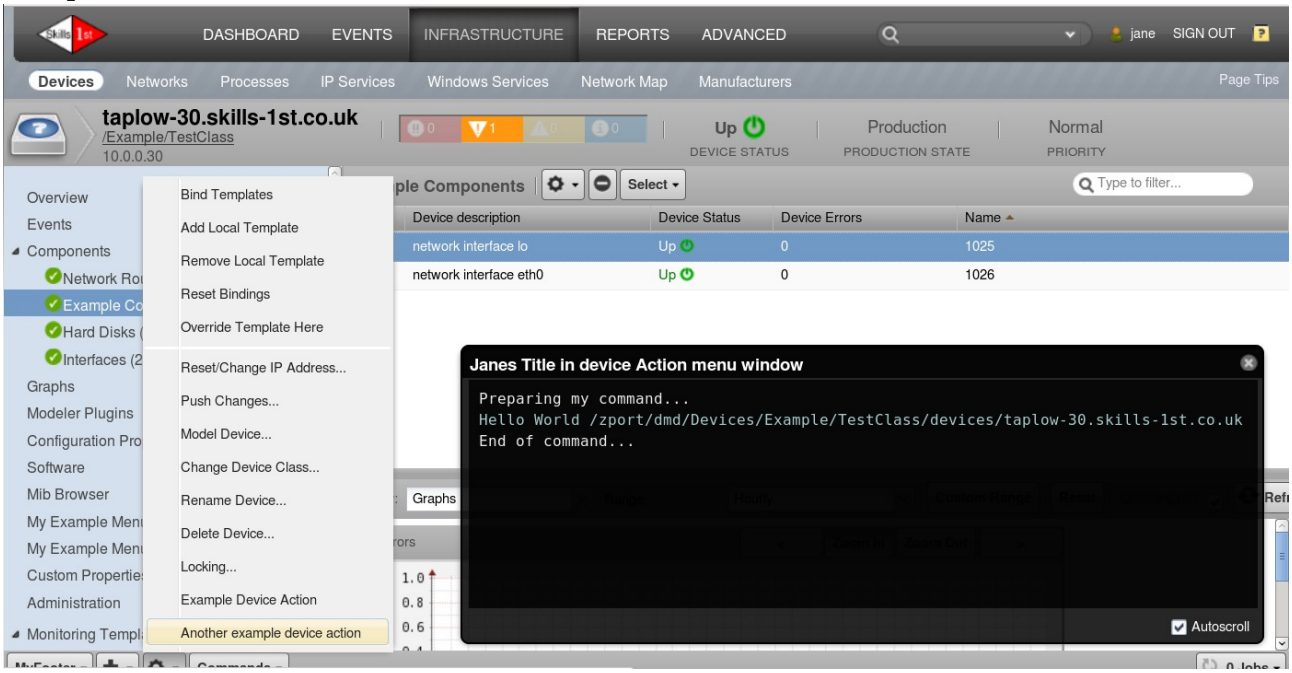


Figure 244: Adding items to a device's Action menu with Another example device action output

Two items are added by this ZenPack:

- Example Device Action — logs to a console log
- Another example device action — runs the same predefined command seen in the previous section

browser/configure.zcml requires a viewlet entry.

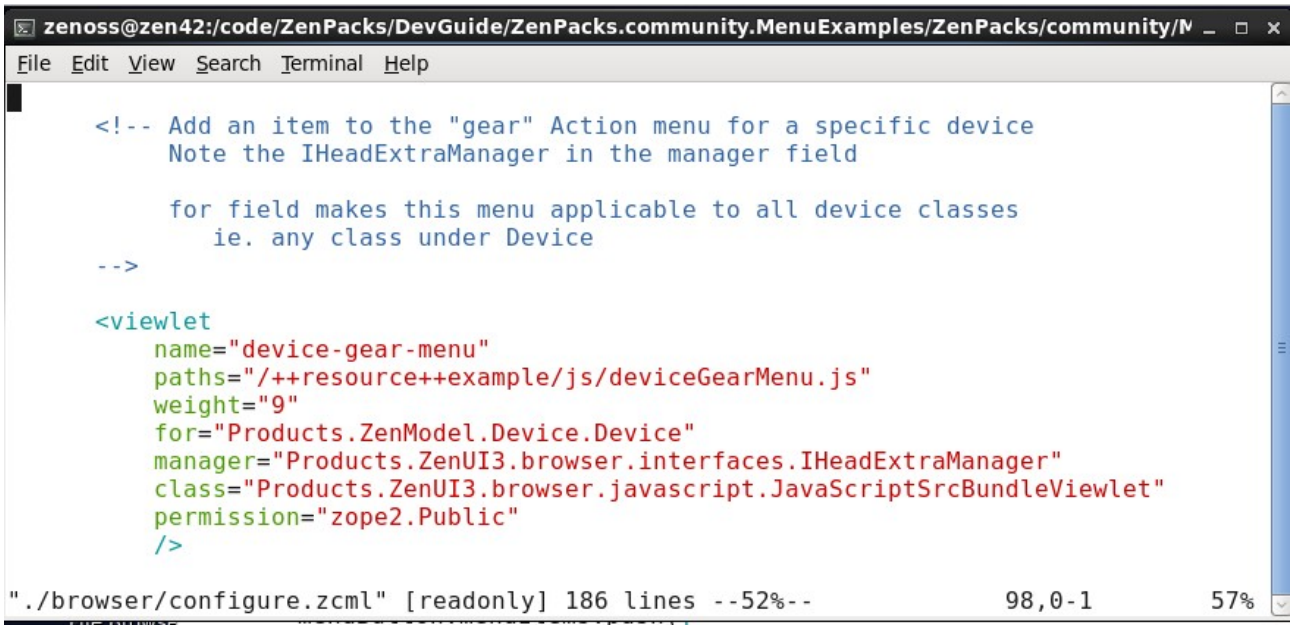


Figure 245: browser/configure.zcml to define additions to a device's Action menu

Note that:

- *paths* references the correct JavaScript file
- *weight* positions the new items towards the bottom of the menu
- *for* makes these additions valid for device class *Device* and all subclasses
- The *manager* statement **must** be *Products.ZenUI3.browser.interfaces.IHeadExtraManager*

The JavaScript file pushes extra items on to the *device_configure_menu* (which is provided by the core Zenoss code).

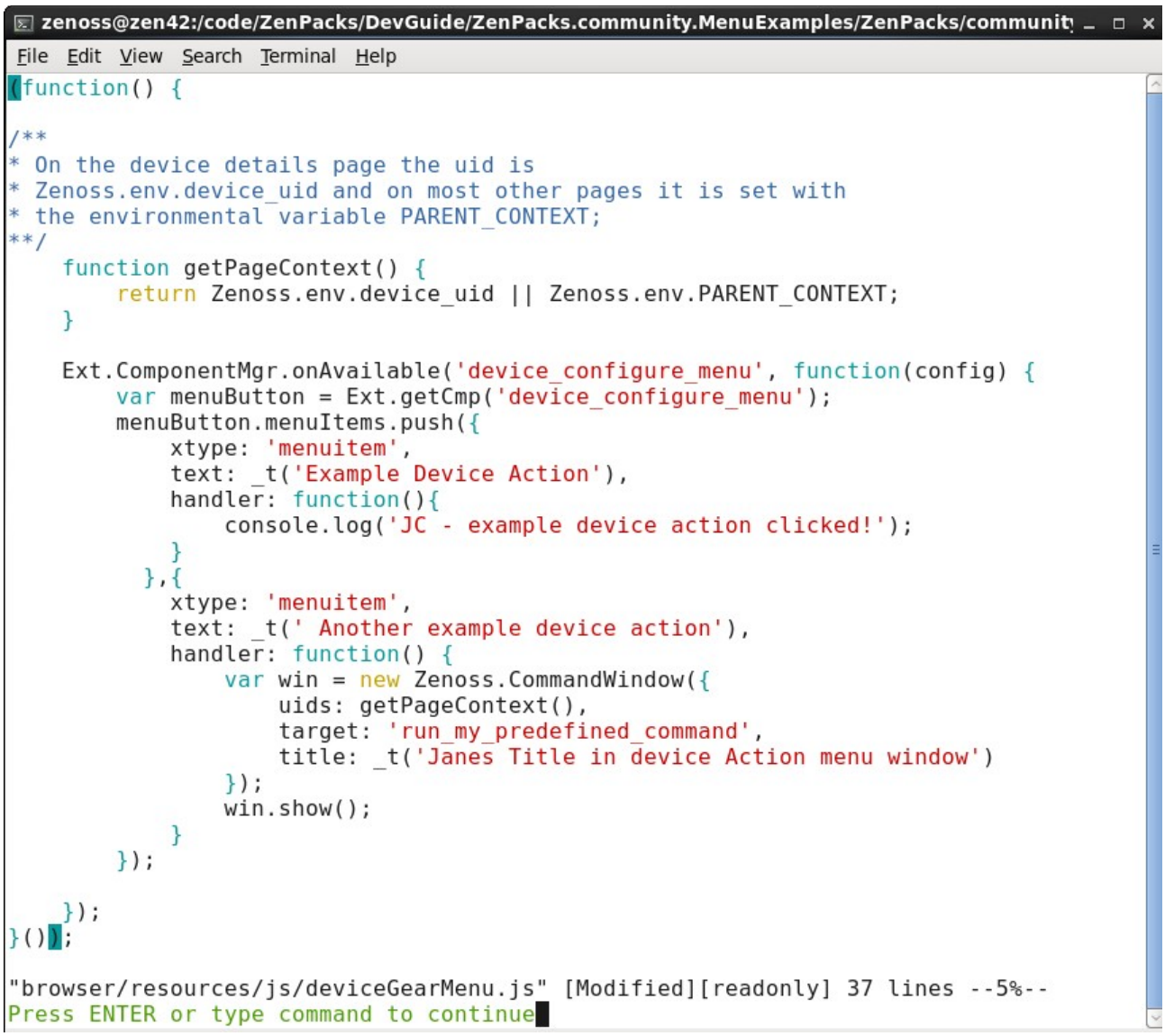


Figure 246: JavaScript file to extend a device's Action menu

Note that the second item, *Another example device action* has an identical construct to the previous example, referencing the *run_my_predefined_command* in the *target* statement; however, to distinguish it from the previous example, a different window title is configured. The output in the GUI popup will show the url to the selected device.

The first menu item demonstrates logging to the browser console log; this can be a useful debugging technique used with the browser's logging tools.

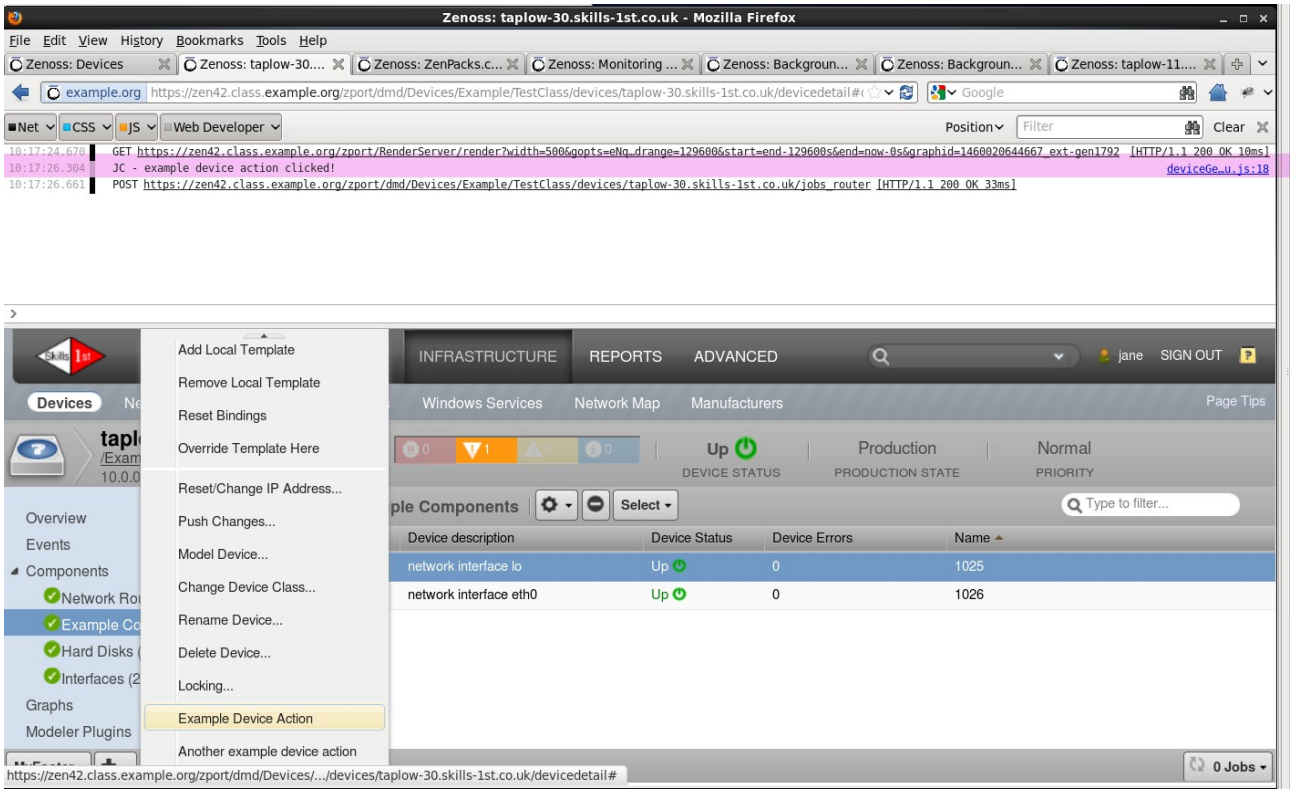


Figure 247: Firefox console log showing Example Device Action menu item output

Use Firefox menus *Tools -> Web Developer -> Web Console* to start the browser console log.

15.3.10 Adding a new menu to the Footer bar

A whole new menu can be added to the footer bar at the bottom of the navigation tree menu.

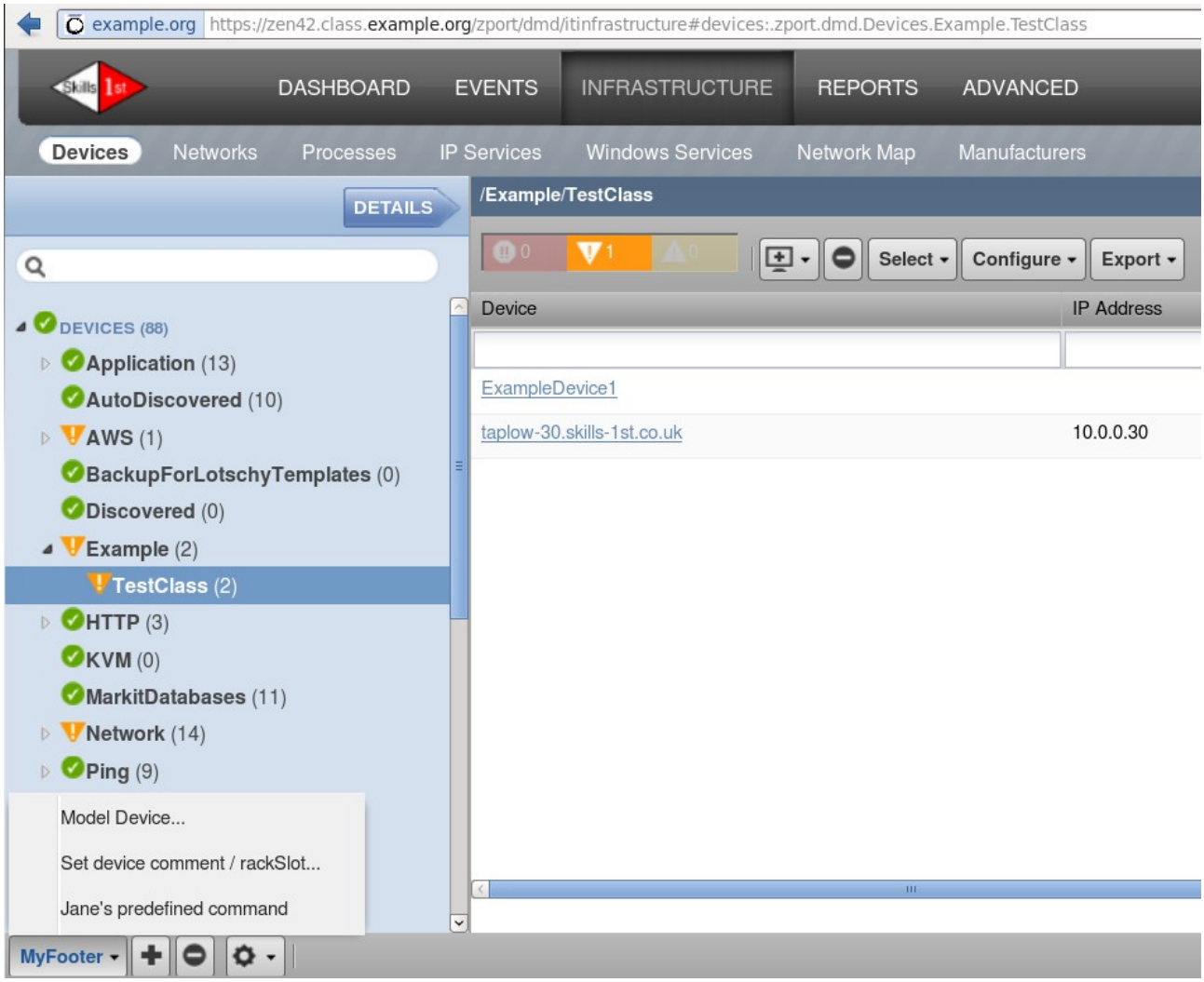


Figure 248: New menu on footer bar

The menu has:

- The standard "Model device" action
- An action to run the same predefined command discussed earlier
- An option "Set device comment / rackSlot" which prompts for these two fields and then modifies the selected device accordingly.

A *viewlet* entry is required in *browser/configure.zcml* that points to the JavaScript file *myFooterMenu.js*.

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples
File  Edit  View  Search  Terminal  Help
    <!-- Define a viewlet that puts up a new footer bar menu
         The action is defined in myFooterMenu.js under the js subdir in the resources subdirectory
         The weight field doesn't seem to change the position of this option on the
         footer bar but don't set weight="1" or it doesn't show up at all
    -->

      <viewlet
          name="js-myFooterMenu"
          paths="/++resource++example/js/myFooterMenu.js"
          weight="4"
          manager="Products.ZenUI3.browser.interfaces.IJavaScriptSrcManager"
          class="Products.ZenUI3.browser.javascript.JavaScriptSrcBundleViewlet"
          permission="zope2.Public"
          />
"./browser/configure.zcml" [readonly] 186 lines --52%--                    97,1          48%
```

Figure 249: browser/configure.zcml for footer menu

Note that the *weight* statement does not appear to have any effect but setting it to *1* results in the menu not displaying at all.

```
zenoss@zen42:/code/ZenPacks/DevGuide/ZenPacks.community.MenuExamples/ZenPacks/community/MenuExamples/
File  Edit  View  Search  Terminal  Help
    // whenever the footer bar is available we want to add a button
    Ext.ComponentMgr.onAvailable('footer_bar', function(config) {
        var footer_bar = Ext.getCmp('footer_bar');
        // the component is built but it is not ready for items to
        // be added so add a listener for after it is rendered
        footer_bar.on('render', function(){
            // add our menu items
            footer_bar.add([{
                text: 'MyFooter',
                id: 'my-footer-bar-menu',
                menu: {
                    items: [{
                        xtype: 'menuitem',
                        text: _t('Model Device') + '...',
                        hidden: Zenoss.Security.doesNotHavePermission('Manage Device'),
                        handler: function() {
                            var win = new Zenoss.CommandWindow({
                                uids: [getPageContext()],
                                target: 'run_model',
                                title: _t('Model Device')
                            });
                            win.show();
                        }
                    },{
                        xtype: 'menuitem',
                        text: 'Jane\'s predefined command',
                        handler: function() {
                            var win = new Zenoss.CommandWindow({
                                uids: getPageContext(),
                                target: 'run_my_predefined_command',
                                title: _t('Janes Title in footer_bar window')
                            });
                            win.show();
                        }
"myFooterMenu.js" [Modified] 52 lines --21%--                              11,5          55%
```

Figure 250: JavaScript file part 1 - defines standard Model Device and Predefined Command defined in previous sample

In Figure 250, note:

- The standard *footer_bar* is defined in core code. The new *MyFooter* menu will be added to it
- The first menu item has the *target* of *run_model*. This simply duplicates the *Model Device* menu item on a device's standard *Action* menu. *run_model* can be found in *$ZENHOME/Products/ZenUI3/browser/configure.zcml*.
- The second menu item has the *target* of *run_my_predefined_command*, seen in several earlier samples. Note the title reflects that it is called from the footer bar.

The third menu item is another example of using a router. A popup window is produced to supply input for the comments and rackSlot device attributes.

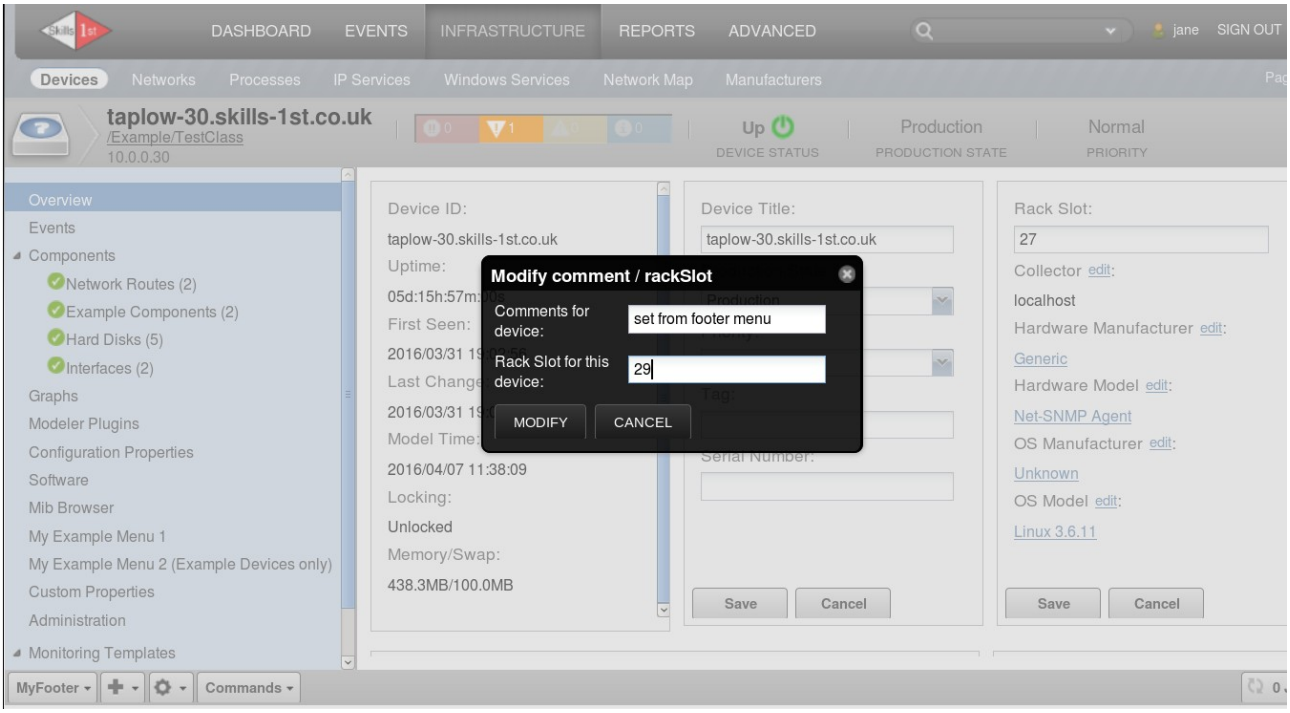


Figure 251: Popup dialog for modifying comments and rackSlot from footer menu

The popup box is defined in the JavaScript file.

Figure 252: JavaScript file part 3 - defines popup box for comments / rackSlot input for footer menu

Note that:

- The *allowBlank: False* statement mandates input before the buttons will be active.
- The *name* statements must match the object attributes to be changed - *comments* and *rackSlot*

The *Modify* button in the dialog popup calls a new router defined in the ZenPack, *Zenoss.remote.myAppRouter.myRouterFunc(opts)* to channel data from the GUI into a facade (*myAppFacade*) to actually change the attributes of the object to the values that have been input.

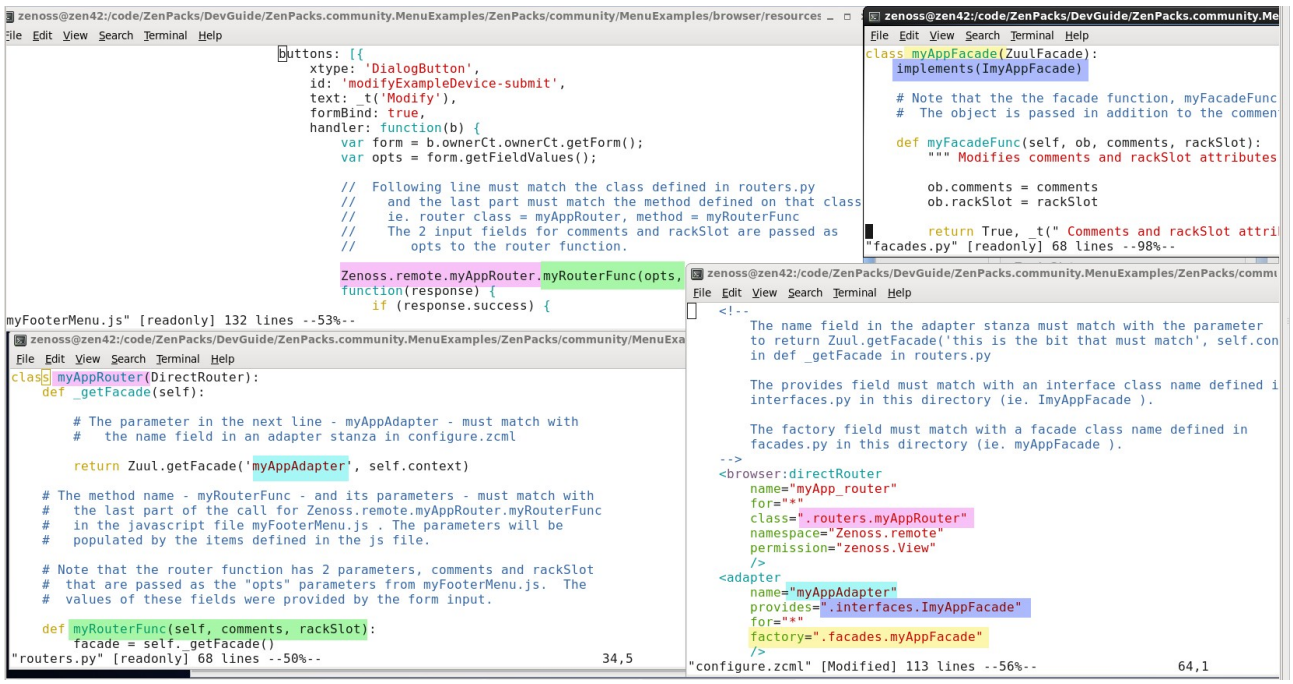


Figure 253: JavaScript, routers.py, facades.py and configure.zcml to implement the comments / rackSlot dialog

In Figure 253, note:

- Both router and facade need entries in the **top-level** *configure.zcml* and the facade also needs an entry in *interfaces.py*, as follows:

```
# The name of the IFacade class here ( ImyAppFacade ) must match what is
# defined in an adapter stanza's
# provides=".interfaces. this_is_the_bit_that_must_match"
#  ie. ImyAppFacade in configure.zcml
# The method name and parameters must match those defined for the facade
# that implements ImyAppFacade in facades.py (ie. myFacadeFunc )

class ImyAppFacade(IFacade):
  def myFacadeFunc(self, ob, comments, rackSlot):
    """ Modify comments / rackSlot attributes for a device object"""
```

- The JavaScript file calls the router *myAppRouter* with the method *myRouterFunc*(*opts*) , passing in *opts* the values received from the GUI for *comments* and *rackSlot*.

- The *myAppRouter* class in *routers.py* defines the facade as *myAppAdapter* which is found by referring to the *configure.zcml* wiring. The *adapter* stanza points to *facades.myAppFacade*.

- The *interfaces.py* file defines the *myFacadeFunc* method with parameters of (*ob, comments, rackSlot*) where *ob* is the device object that is passed by the *myRouterFunc* method from *routers.py*, as follows:

```
def myRouterFunc(self, comments, rackSlot):
   facade = self._getFacade()
   # The object that is being operated on is in self.context
   ob = self.context
   # The facade name in the next line & its parameters must match
   # with a method defined in facades.py
   # (ie. myFacadeFunc(ob, comments, rackSlot) )
```

```
        # Note that facade.myFacadeFunc needs 3 parameters as we
        # need to pass the object as well as the comments and
        # rackSlot attribute values.

        success, message = facade.myFacadeFunc(ob, comments, rackSlot)
```

- The facade method, *myFacadeFunc*, is the code that actually updates the object and returns a success message.

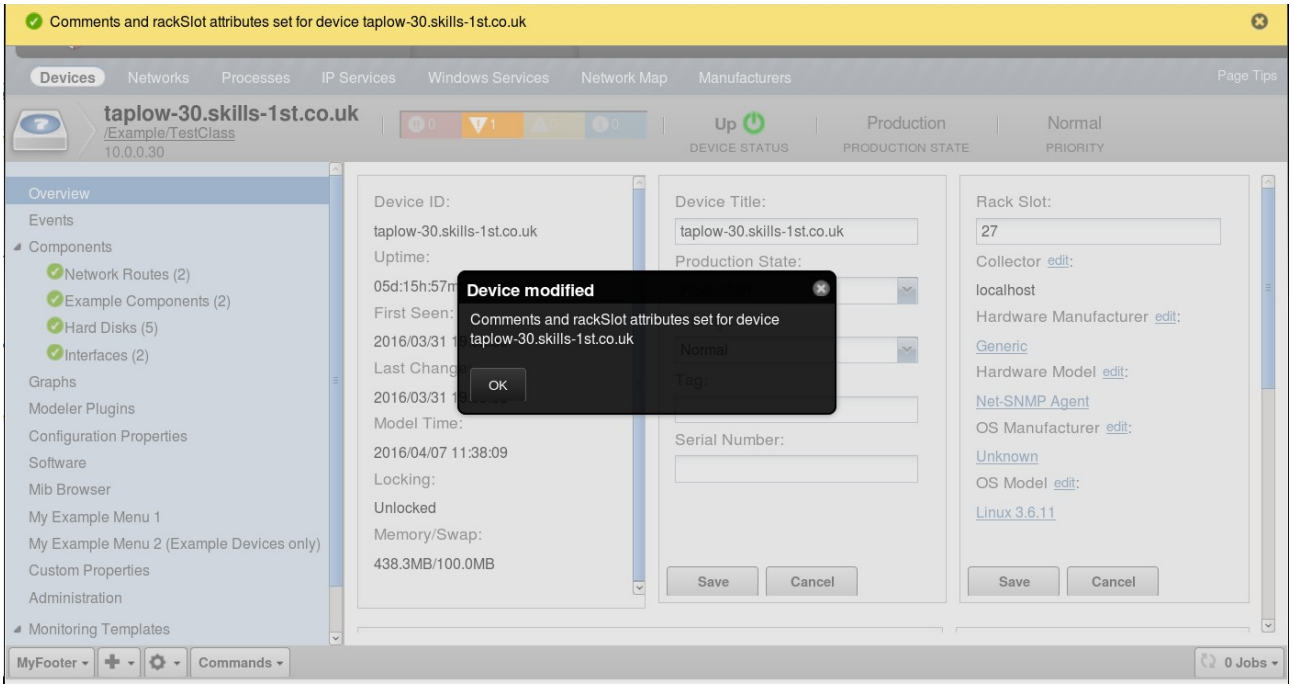


Figure 254: Popup reporting success setting comments and rackSlot from footer menu

16.0 Testing and debugging ZenPacks

The chances of getting a ZenPack with new source code, correct first time, is not high. This section offers some testing and debugging hints. At this stage, the information only addresses Zenoss 4 though many comments are pertinent to earlier Zenoss versions and some will be relevant for Zenoss 5.

Some of the points raised in this section have been discussed earlier in the text but they are repeated here to provide a problem-driven reference chapter.

TODO: This chapter need lots of Zenoss 5 specific details.

16.1 Log files and logging

Zenoss logfiles are all held under *$ZENHOME/log; i*n Zenoss 5, one must attach to the relevant container first to see the active logfile. By default logfiles have an *Info* level (*logseverity* = 20) of logging but this can be increased to *Debug* to provide lots more data (*logseverity* = 10). When the problem is resolved, the original logging level should be restored.

Daemon log files and their configuration can be inspected from the *ADVANCED -> Settings -> Daemons* menu. To increase the debug level, change the *logseverity* to *Debug*. If you check the configuration file for this daemon in *$ZENHOME/etc* you will see a line:

```
logseverity  10
```

Any changes to a daemon's configuration file requires a restart of the daemon, either through the GUI or using *<daemon> restart* from a command line.

In addition to checking specific Zenoss daemon files like *zenmodeler.log* or *zenperfsnmp.log*, it is always worth also checking **zenhub.log** and **event.log.**

Most daemons will respond to the debug parameter, which is a toggle switch.

```
zencommand debug
```

Turns on debug level on the running daemon. A second identical command toggles debugging off again. This debug state will **not** persist past a daemon restart.

Make use of log statements in any code that you write. Modeler plugins automatically provide for logging from the standard *CollectorPlugin* code. Make use of the different log severities:

```
log.debug('logMatchTable is %s ' % (logMatchTable))
log.info('Modeler %s processing data for device %s',self.name(), device.id)
log.warn( 'No SNMP response from %s for the %s plugin ', device.id, self.name() )
log.error('%s: %s', device.id, e)
```

Run daemons in the foreground with the logging set to debug (*-v 10*) and output both stdout and stderr to a file for further inspection. For example:

```
zenmodeler run -v 10 -d taplow-11.skills-1st.co.uk --collect DirFilePythonMap > /tmp/fred1 2>&1
```

16.1.1 Log messages and their likely meanings

1. In *zenpython.log*:

   ```
   2015-12-15 10:26:01,268 DEBUG zen.ZenPacks.community.DirFile.dsplugins: In
   OnError - result is [Failure instance: Traceback: <type
   'exceptions.TypeError'>: 'str' object is not callable
   ```

   a. The "'str' object is not callable" is often a log or print statement where % has been missed. For example

   ```
   log.debug('In datapoint loop datapoint_id is %s ' (datapoint_id))
   ```

   should be:

   ```
   log.debug('In datapoint loop datapoint_id is %s ' % (datapoint_id))
   ```

2. In *zenmodeler.log*:

   ```
   No decoder for oid 1.3.6.1.4.1.2021.16.100.3.0 type ASN_BIT8 - returning None
   ```

   a. Suggests that an SNMP modeler plugin is requesting the wrong OID or that the target device does not support that MIB value.

3. In *zenhub.log*:

   ```
   2015-11-15 17:17:21,973 WARNING zen.ApplyDataMap: The attribute
   logMatchIndex was not found on object fred2_daily from device taplow-
   11.skills-1st.co.uk
   ```

a. Indicates that a modeler plugin is trying to populate an object attribute that does not exist in the object class file or *zenpack.yaml*. The error is only a WARNING and other aspects of the modeler should be unaffected.

4. In any collector daemon log:

```
2016-01-20 10:44:52,329 ERROR zen.collector.config: Configuration for
taplow-11.skills-1st.co.uk unavailable -- is that the correct name?
```

a. Typically there is no attempt to collect any data whatsoever for this device.

b. The issue is in the ***config_key*** or ***params*** methods of a performance *DataSourcePlugin* run by *zenhub*, resulting in a null configuration for the device.

c. An error in one plugin will prevent collection of **all** data by that daemon for that device.

d. Matching errors should also appear in **zenhub.log**.

16.2 General hints and tips

1. A frustrating common error is that most directories in a ZenPack need an *__init__.py* file, even though it may be empty. It is particularly easy to omit this from a ZenPack's **browser** directory. It is perfectly acceptable to create a null-length file using:

```
touch __init__.py
```

2. A common error is to omit to import required classes before using them. pyflakes in *vi* should help highlight these as the classes will be undefined.
3. A classic error to make in Python files is to get white space indentation wrong. Python uses indentation to structure *if*, *while*, *for* and other constructs; you **must** be consistent with the number of spaces used at each level of indentation.
4. A common issue with some environments and browsers is to see a blank screen in the Zenoss GUI. This is usually resolved simply by resizing fonts in the browser using Ctrl - .
5. Use ***zendmd*** as a general debugging tool. It is an excellent “sandpit” to test bits of Python and to query Zenoss objects and their attributes and methods. Several examples have already been demonstrated throughout this document.
6. When weird things happen that really make no sense at all, try recycling the whole Zenoss system with:

```
zenoss stop
zenoss start
```

7. Using the Python *try ... except* construct allows error conditions to fail in a much more controllable manner.

16.3 Testing and debugging new object class files

If you have created or changed object class files, you should always delete any discovered instances that use those files and rediscover them to ensure that any relationship changes are established correctly. You should certainly recycle **zenhub** and **zopectl** with:

- zenhub restart
- zopectl restart

When a new device object class has been added or a relationship has been added to a device class, then you need to recycle the whole of Zenoss with:

- zenoss stop
- zenoss start

Typically you will be doing initial testing with a single device so delete the device and use the *Add Device* menu to re-add it, ensuring that you specify your new device class in the *Device Class* dropdown. Adding the device runs *zendisc* which calls *zenmodeler*. You may see error messages in the discovery GUI. Usually they are quite good at pinpointing the problem to a particular line in a particular file. Watch out especially for syntax errors in your code such as missing closing brackets, missing quotes or missing colons (:).

Another way to start testing object class files is to use the Zope ZMI interface. For example, navigate to *http://zen42.class.example.org/zport/dmd/manage* and then navigate down *Devices/Server/Linux/SimpleTest/devices/<a specific device>* and check that the expected attributes and relationships exist.

16.3.1 New components do not appear in left-hand menu

1. If you don't recycle everything then a new component that is declared in the new relationship, doesn't appear in the left-hand menu of the GUI.
2. Check the spelling and capitalisation on names, especially relationship names - at least 3 times!
3. Use the ZMI to check whether the objects exist. If attributes and relations are correct then the issue is probably with the GUI code.

16.4 Testing and debugging modeler plugins

If you have created or changed a modeler plugin, you need to restart **zenhub** and **zopectl**; typically you do **not** need to delete your test device and re-add it. It should be sufficient to simply use the *Model Device* menu from the *Action* icon and watch the output.

Note the dialogue particularly to ensure that your modeler does at least attempt to run – the output will show what plugins are to be run. You may need to uncheck the *Autoscroll* box at the bottom-right to be able to scroll back up the window.

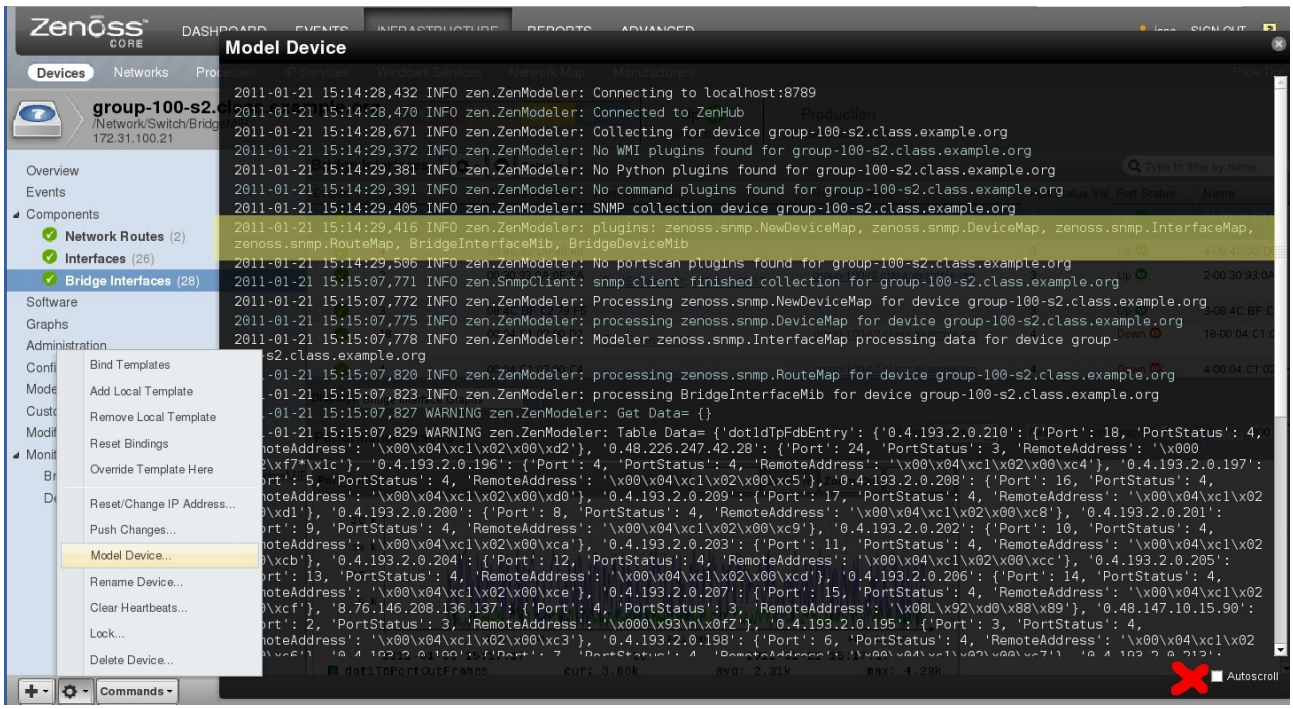


Figure 255: Output from Model Device highlighting the plugins to be run

16.4.1 Compilation errors

If your modeler doesn't appear on the plugins list it is probably a compilation error. Remember that you have created python source files (ending in *.py*); Zenoss will compile-on-demand to generate *.pyc* files. A good check is always to inspect the base Zenoss directory and the *modeler/plugins* directory hierarchy to ensure that you have matching *.pyc* files for each of your *.py* files.

A good way to test for compilation errors is to use the *zendmd* utility to import the file in question.

```
zenoss@zen241: > zendmd
Welcome to the Zenoss dmd command shell!
'dmd' is bound to the DataRoot. 'zhelp()' to get a list of commands.
>>> from ZenPacks.skills1st.bridge.modeler.plugins import BridgeDeviceMib
>>> from ZenPacks.skills1st.bridge.modeler.plugins import BridgeDeviceMib
>>>
zenoss@zen241:~> zendmd
Welcome to the Zenoss dmd command shell!
'dmd' is bound to the DataRoot. 'zhelp()' to get a list of commands.
>>> from ZenPacks.skills1st.bridge.modeler.plugins import BridgeDeviceMib
Traceback (most recent call last):
  File "<console>", line 1, in ?
  File "/usr/local/zenoss/zenoss/local/jane/ZenPacks.skills1st.bridge/ZenPacks/skills1st/bridge/m
odeler/plugins/BridgeDeviceMib.py", line 32
    )
    ^
SyntaxError: invalid syntax
>>>
```

Figure 256: zendmd dialogue showing successful compilation and unsuccessful compilation

The figure above shows a successful import (in fact, two of them!) – you simply receive a command prompt back. Note that you need to specify an **object** path to the Python source

file, not a **file** path. The second *zendmd* dialogue shows a failed compilation (I removed a closing bracket from line 32).

1. Watch for yellow highlighted messages when using a *Modeler Plugin* menu
2. Check *$ZENHOME/log/event.log* for error messages. For example:

```
  File
"/code/ZenPacks/DevGuide/ZenPacks.community.LogMatch/ZenPacks/community/Log
Match/modeler/plugins/community/snmp/LogMatchMap.py", line 54
  def process(self, device, results, log)
                           ^
SyntaxError: invalid syntax
```

3. Do not leave test / half-finished plugin files under *modeler/plugins* if it has a *.py* suffix. *zenhub* checks **all** python files under such directories and will attempt to use them as modeler plugins. Errors will appear in *zenhub.log*:

```
2016-01-19 17:19:32,213 INFO zen.ZenHub: Worker (1699) reports 2016-01-19
17:19:32,213 WARNING zen.ModelerService: The following error occurred while
loading
ZenPacks.community.DirFile.modeler.plugins.community.python.DirFilePythonMa
pBad plugin:
 ('Traceback (most recent call last):', ' File
"/opt/zenoss/Products/DataCollector/Plugins.py", line 100, in create', '
self.modPath)', ' File "/opt/zenoss/Products/DataCollector/Plugins.py",
line 192, in importPlugin', '  return importClass(modPath)', ' File
"/opt/zenoss/Products/ZenUtils/Utils.py", line 536, in importClass', '
__import__(modulePath, globals(), locals(), classname)', ' File
"/code/ZenPacks/DevGuide/ZenPacks.community.DirFile/ZenPacks/community/DirF
ile/modeler/plugins/community/python/DirFilePythonMapBad.py", line 24', '
def create_dirRegex(self, device, log)', '                          ^',
'SyntaxError: invalid syntax')
```

16.4.2 General modeler debugging hints

1. If the modeler runs but fails then hopefully you get a message in the GUI showing the modeler output. If there are insufficient clues here, try running *zenmodeler* standalone with full debugging turned on (*-v 10*) and send the output to a file:

```
zenmodeler run -v10 -d taplow-11.skills-1st.co.uk --collect LogMatchMap \
        > /tmp/fred 2>&1
```

2. Note that the device parameter **must** be the device **id** if you use a name; alternatively, the IP address can be used.
3. Using the *--collect* parameter to specify the modeler to run, can significantly reduce the output.
4. If you still can't see the problem, try putting log statements in the modeler plugin code to output intermediate data stages. Figure 257 highlights *log.warn* statements that output the results of the SNMP *getdata* and *tabledata* structures.

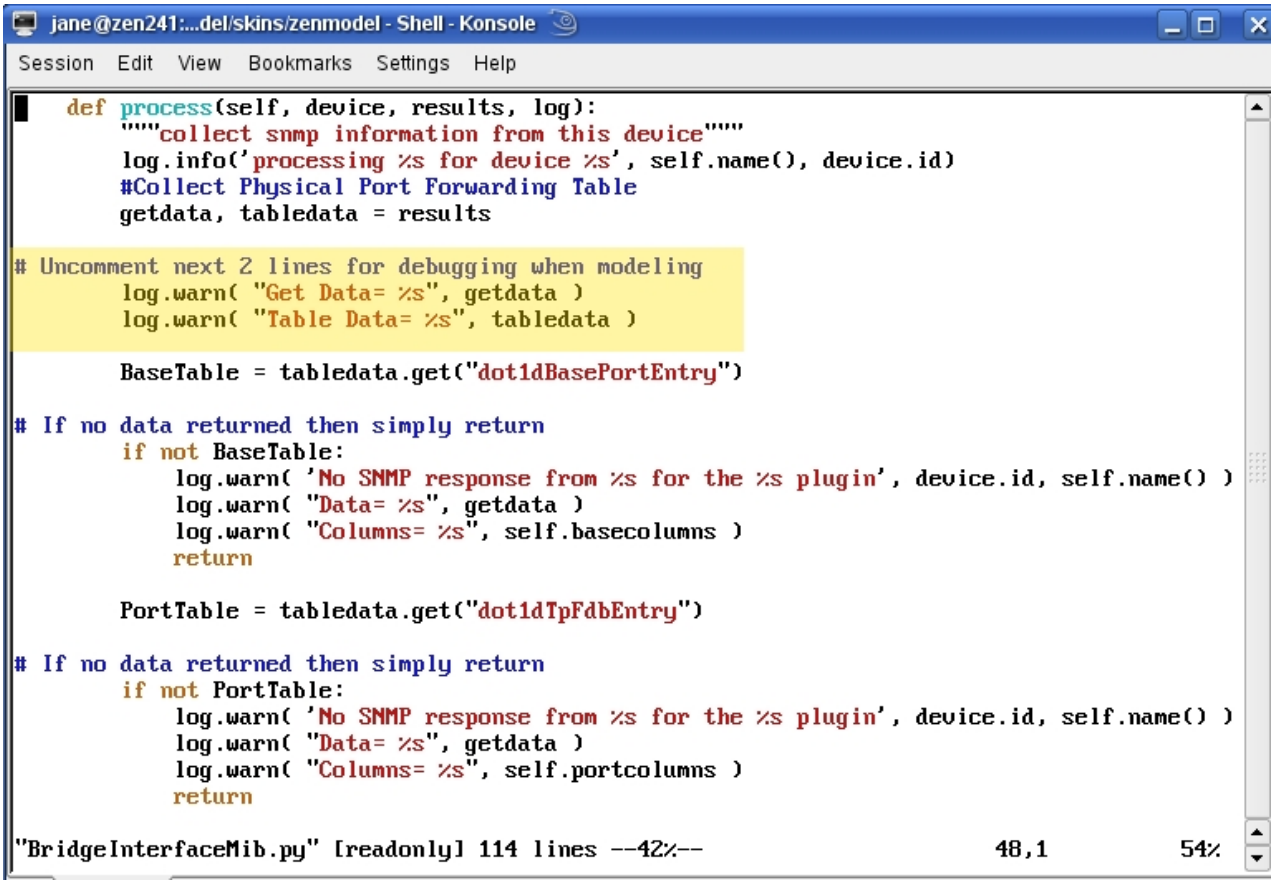


Figure 257: BridgeInterfaceMib.py code highlighting debugging logging

5. A modeler filename **must** be the same as the plugin classname.
6. It is perfectly normal and not an error condition to get a line in *zenmodeler.log* saying:

```
2015-11-30 13:16:46,421 DEBUG zen.Classifier: No classifier defined
```

16.4.3 Attributes or relationships are not populated

1. Check the spelling and capitalisation on names, especially relationship names - at least 3 times!
2. If a relationship is not created then check relationship names and object files for both device and component.
3. If relationship **instance(s)** are not created, check:
   i. *relname* and *modname* statements in modeler plugin exists
   ii. *relname* and *modname* are correct (especially case-sensitivity)
4. If one or more attributes do not have values:
   i. Check spelling of attributes in modeler plugin table column names
   ii. If SNMP modeler, check OID is correct and that data is collected
   iii. Check attribute names in object class files do match with (i)
   iv. Check type of attributes for errors eg. *string* data received for *int* defined attribute

5. For SNMP modelers, in particular:

   a. Check that output is actually received from SNMP and that it matches what you expect. The following message in *zenperfsnmp.log* would be an indication that the OID you are requesting is incorrect or not supported on the agent:

```
 No decoder for oid 1.3.6.1.4.1.2021.16.100.3.0 type ASN_BIT8 - returning
None
```

   b. Use the standalone *snmpwalk* utility as a test tool to check SNMP access, authentication parameters and OIDs. For example:

```
snmpwalk -v 2c -c fraclmye taplow-11.skills-1st.co.uk .1.3.6.1.4.1.2021.16.2.1

snmpwalk -v 3 -a MD5 -A fraclmyea -l authNoPriv -u jane zenny .1.3.6.1.4.1.2021.16.2.1
```

   c. Check that Zenoss really does have the same parameters configured in the various SNMP zProperties for the device.

   d. Note that the leading dot "." on the OID is optional (it confirms a "fully-qualified" OID starting at the root of the MIB tree).

      i. Note that some earlier versions of Zenoss had issues if this leading dot was omitted.

6. For modelers that use ssh, it is important to test ssh, as the zenoss user, to each target **directly**, as the first communication will generate the host fingerprint entry and ask whether to add it to the *known_hosts* file in the *zenoss* user's *.ssh* directory. The target name must be **identical** to that used by Zenoss. Zenoss modelers will probably fail if asked this question, typically with a *Host key verification failed* error message.

   a. Note on Zenoss 5 the test must be performed from the *zencommand* container as the *.ssh/known_hosts* inside the container is not the same as that for the *zenoss* user on the base host. If the *zenpython* daemon also uses ssh then it is **not** necessary to repeat this for the *zenpython* container.

```
serviced service attach zencommand su zenoss -l
ssh -l zenplug zenny2.class.example.org
cat .ssh/known_hosts
```

7. Insert extra *log.debug* statements in the modeler code and rerun the zenmodeler command in debug. For Zenoss Core 4 and earlier, you only need to recycle *zenhub* and *zopectl* daemons if you insert extra log statements. No reinstall of the ZenPack is necessary. For example:

```
log.debug('logMatchTable is %s ' % (logMatchTable))
```

8. Beware coding mappings for attributes that have not been defined in either an object class file or in *zenpack.yaml*. This will result in messages in *zenhub.log* like:

```
2015-11-15 17:17:21,973 WARNING zen.ApplyDataMap: The attribute
logMatchIndex was not found on object fred2_daily from device taplow-
11.skills-1st.co.uk
```

9. The other explanation for such messages may be mis-typed names in either the object class or in the modeler.

10. If such attributes have been coded into the modeler for future use, you could preface the name with an underscore temporarily, to avoid error messages.
11. Where multiple relationships are created, eg. *Device -> Component -> Sub-component*, great care is needed in returning legal data maps. If more than one relationship map or object map is returned by a modeler plugin, then the returned value must be a **list** of relationship maps and/or object maps.

    a. A typical algorithm might be:

```
maps = []
crm = self.relMap()
for each component:
  create component object map (com)
  crm.append(com)
  call function to deliver sub-component relationship map(srm)
maps.append(crm)
maps.extend(srm)
```

    b. Note the use of ***append*** and ***extend*** when creating the list of maps:

        i. maps starts as an empty list

        ii. Component object maps are **appended** to it - each component object map is a single item added to a list.

        iii. A function delivers a **list** of sub-component object maps

        iv. The srm list of maps **extends** maps

    c. The argument of **append** is inserted to the next element of the list.

    d. With **extend**, each element of the iterable argument, is appended as the next element of the list

    e. If crm is:

```
[ com1, com2, com3 ]        then maps becomes
[ com1, com2, com3 ]
```

    f. If *srm* is:

```
[ som1, som2, som3 ]
```

    appending *srm* to maps would give:

```
[ com1, com2, com3, [ som1, som2, som3 ] ]     not what we want!
```

    whereas **extend**ing *srm* to maps would give:

```
[ com1, com2, com3, som1, som2, som3 ]         much better!
```

12.

16.4.4 Modeler issues related to using zenpacklib

1. Beware letting relation names default in *zenpack.yaml*. It is better practice to name the relationships, in both device and component object classes, and to use the names in

the class_relationships stanza. It is essential to get the relation names correct when using them in modeler plugin code.

2. The modeler code also needs to know the module path for object classes for the *modname* variable. Before zenpacklib, this was the path to the object class file; with zenpacklib, it is the ZenPack name concatenated with the **name** of the object. For example:

```
modname = "ZenPacks.community.LogMatch.LogMatch"
```

3. If a ZenPack is created with zenpacklib then none of the *modeler*/*plugin* directory hierarchy is created. This must be done manually, remembering to create *__init__.py* files at each level.

16.5 Testing and debugging problems with performance data

16.5.1 General performance issues

Each performance collector daemon has its own logfile in *$ZENHOME*/*log; zenperfsnmp.log*, *zencommand.log*, *zenpython.log* etc.

1. Be aware which ZenPacks and datasources use which daemon and hence which log file to check for problems. For example:

   a. Many of the original Zenoss ZenPacks, like ftp, dns, dig and Apache use Nagios plugins that are fundamentally driven by zencommand. Check a ZenPack's documentation for information on daemons utilised.

   b. ZenPacks.zenoss.Microsoft.Windows uses zenpython.

   c. Check prerequisites for ZenPacks; any that has the PythonCollector ZenPack as a prerequisite is likely to use zenpython.

2. Examine logfiles for *Missed_Runs* and *Queued_Tasks* messages. This is an indication that the daemon cannot keep up with the load demanded of it. You may also see timeout messages.

16.5.2 Configuration issues

There are several ways that performance data collection can fail because configuration has not been completed or is incorrect:

1. A template is created but not **bound** to a device. In this case, no attempt will be made to collect data. Go to the device's details page and check the *Monitoring Templates* listed there.

2. Component templates, matching the component object class name, do **not** need binding – this happens automatically – and the component template will not be listed for the device. Check component templates from the middle drop-down menu on the Component page and select *Templates*.

3. *zenpack.yaml* may specify other component templates for automatic binding to a component object:

```
Dir:
 label: Dir # NB It is label, with spaces removed, that is used to match a component template
 meta_type: Dir  # Will default to this but in for completeness
 label_width: 150  # This controls the column width for Dir in the Files component display
 order: 60  # before file
 auto_expand_column: dirName
 monitoring_templates: [Dir, DirPythonXml] # will default to Dir but explicit for clarity
```

4. You could also check the *zDeviceTemplates* property from the *Configuration Properties* menu to ensure the correct device templates are bound.
5. In SNMP datasources, scalar MIB values need the trailing *.0*; otherwise no data will be collected.
6. If SNMP community names configured in Zenoss do not match those in the target agents then you will get no SNMP data. Test with a simple *snmpwalk* command from a command line; for example:

   ```
   snmpwalk -v 1 -c public switch.skills-1st.co.uk system
   ```

7. Similarly for templates performing ssh commands, check the *zCommand* configuration properties match the device, and test from the command line.
8. If a template is correctly configured and bound but there are only one or two data values collected (counter values need at least two values before a point can be plotted as it is a rate-of-change measurement), you will see a graph with no data and the *cur, avg* and *max* values will have the value **nan**. This simply means graph points are not yet available; another polling interval usually fixes this issue.
9. If you see no graphs, do check that they have been configured - with the correct datapoints.
10. For component device templates collecting tables of SNMP data, the instance may be the issue. Increasing the logging level for *zenperfsnmp* may help diagnose this.
11. Note that *zCommandSearchPath* appears to be ignored by the *zencommand* mechanism.
12. If specifying commands in a *COMMAND* template, either use a fully-qualified path name to the command, or ensure that a local standard exists to implement the ***zCommandPath*** property.
13. When specifying attributes to be substituted into datasource templates, take care with names, syntax and open / closing braces. For example:

    ```
    /usr/bin/du -P -b -s ${here/dirName}
    ```

16.5.3 Checking for collected performance data

In Zenoss 4 and earlier, templates collect data into Round Robin Database (rrd) files, held under *$ZENHOME/perf/Devices* with a separate subdirectory for each device and each device may have subdirectories for components such as *os* or *logMatchs* (that is, the **relationship** name of the contained component); there may be further subdirectories for each instance where the subdirectory name is the id of the sub-component; for example:

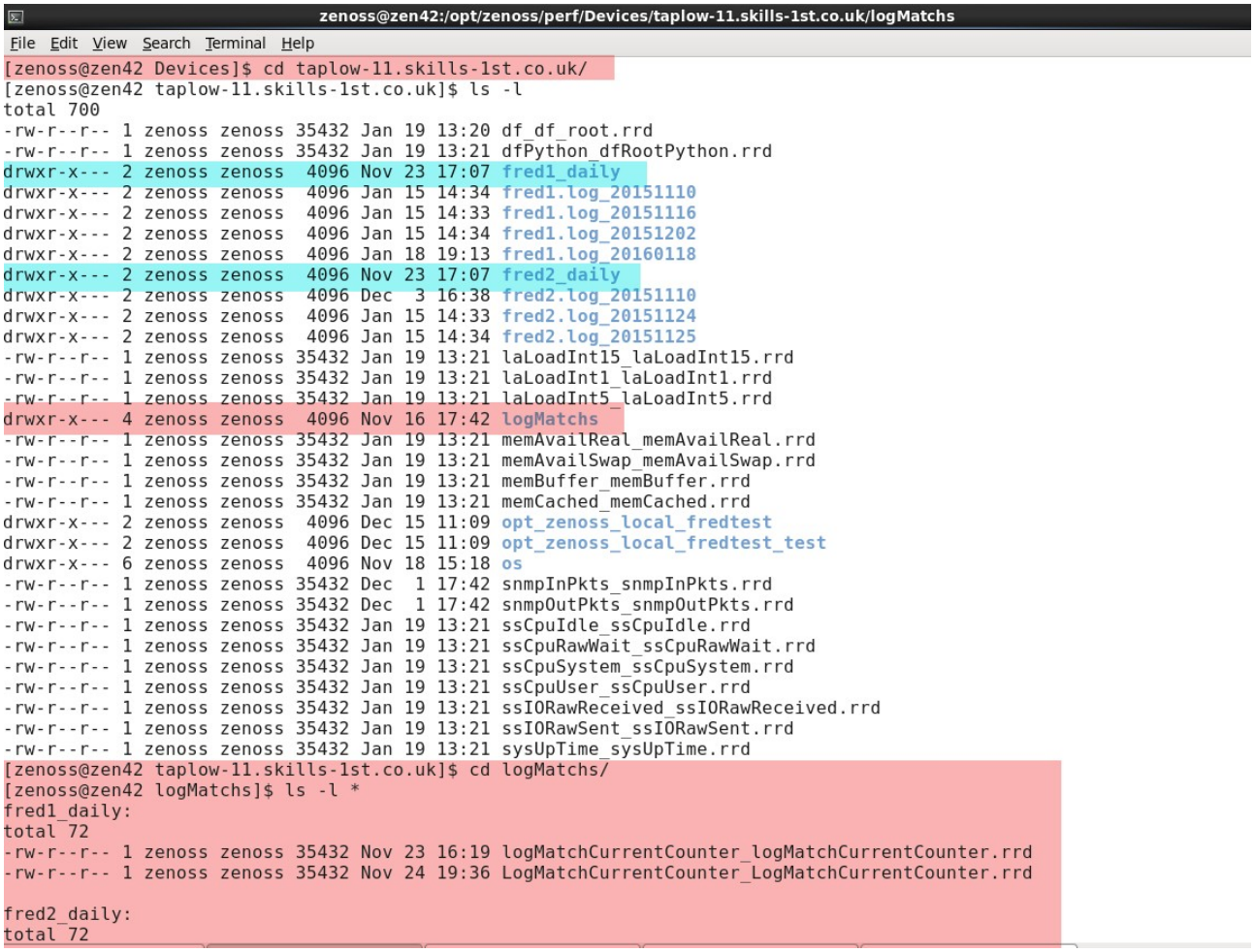


Figure 258: Directories for performance files for devices and components

1. Always check that rrd files exist and are non-zero length. Template files have the format:

   ```
   <datasource name>_<datapoint name>.rrd
   ```

2. Note that components created with zenpacklib will gather data structured slightly differently. There will be no <relationship> level of subdirectory. Subdirectories will be created directly for component instances with data files beneath that. The red-highlights in Figure 258 are for **non-zenpacklib** components. The *fred1_daily* and *fred2_daily* directories highlighted in blue have been created for components developed **with zenpacklib**.

3. Particular care must be taken when converting a ZenPack to use zenpacklib in order not to lose performance data. See the README for https://github.com/jcurry/ZenPacks.community.VMwareESXiMonitorPython for a description of this problem and the workaround which involves modifying the *rrdPath* method in the *zenpacklib.py* in the ZenPack's base directory. See also section 10.3.5.

4. If you see graphs that have no data at all, this generally means that a template is bound but there is no rrd file.

5. rrd files can be inspected with the *rrdtool* command

   a. *rrdtool info <rrd file> | more*

   - Shows outline information for the RRD archives and data in the file

b. rrdtool dump <rrd file> | more

- Shows all the RRD archives and data in the file

6. In Zenoss 4, if the cycle time of any datasource is changed, it will **stop the collection** of rrd data for this datasource. This is because the ***step*** value in the rrd file is setup when the datasource rrd file is created and it cannot be subsequently changed (at least, not easily without rrd export / import tools). If the rrd file is deleted, it will be automatically recreated on the next cycle, with the new step value. Typically it will require two cycle intervals before data values appear. With Zenoss 5, there is no issue and the new cycle time will be honoured on the next sample interval.

16.5.4 Test buttons in datasources

Note that when you configure some data sources in a template, there is a test button that you can use to specify a device known to Zenoss. This not always trustworthy. Double-click a datasource line to bring up the *Edit Data Source* dialogue.

1. For an SNMP datasource, the test that is run, strictly, is an *snmpwalk* whereas the *zenperfsnmp* daemon is more likely to issue an *snmpget,* so the test button can disguise problems with instances.

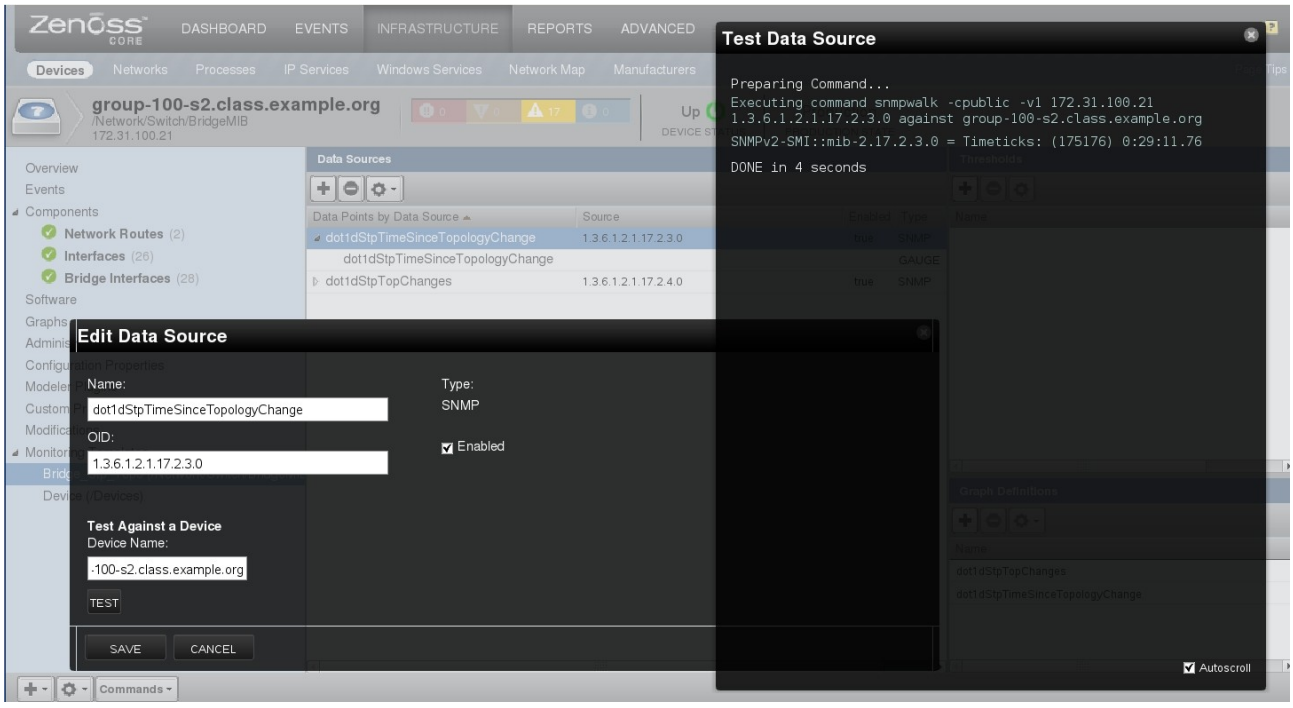


Figure 259: Using the TEST button from the Edit Data Source configuration dialogue

2. Tests against component datasources are not trustworthy - sometimes they may work; other times they don't. Tests against **local devices** should work.
3. The Test button does **not** work when running commands against remote hosts over ssh.

16.5.5 Issues with datasource plugins

1. Datasource plugins can either be in *dsplugins.py* in the base directory of the ZenPack or in a *dsplugins* subdirectory. If a *dsplugins* directory is used, ensure it has an *__init__.py* file. Unlike most such files, this must have entries in it, one per *PythonDataSourcePlugin*. For example:

```
from DirDiskUsedPythonDeviceData import DirDiskUsedPythonDeviceData
```

2. When writing *PythonDataSourcePlugins*, beware creating Twisted responses that **block** rather than returning an immediate **Deferred**. The PythonCollector ZenPack from 1.7.3 onwards will permanently disable any datasource, **for all devices**, if it runs for longer than the *blockingtimeout* parameter (default 5 seconds). Once a plugin is blocked, it will remain **permanently blocked** until its name is removed from either ***/var/zenoss/zenpython.blocked*** on Zenoss 5, or ***/opt/zenoss/var/zenpython.blocked*** on Zenoss 4. The *zenpython* service must be restarted after manual modifications to this file.
3. Remember that **fundamentally, collector daemons do *not* have access to the Zope database (ZODB)**. If a collector daemon needs access to attributes or methods of a device instance or component instance, they have to be gathered by *zenhub* and passed to the collector daemon in a configuration phase using the *config_key* and *params* methods of a *PythonDataSourcePlugin*.
4. If a *zenpython* debug log apparently skips over processing datasources for a device, with the message shown below, then suspect issues in *config_key* or *params* methods. Also check *zenhub.log*.

```
2016-01-20 10:44:52,329 ERROR zen.collector.config: Configuration for
taplow-11.skills-1st.co.uk unavailable -- is that the correct name?
```

5. When creating the *config_key* method, ensure that *datasource.getCycleTime(context)* is always the second element. See section 13.3.3.2.

16.5.6 Issues with datasources

1. Ensure that any script used by a *COMMAND* datasource or called by a Python datasource does have the correct permissions to be executable.
2. Always check any ssh communications, as the *zenoss* user, using the command line before using with Zenoss. The first time communication is established a dialogue will be entered to add the server to the *known_hosts* file.
   a. Note on Zenoss 5 the test must be performed from the *zencommand* **container** as the *.ssh/known_hosts* inside the container is **not** the same as that for the *zenoss* user on the base host.

```
serviced service attach zencommand su zenoss -l
ssh -l zenplug zenny2.class.example.org
cat .ssh/known_hosts
```

3. Check the relevant daemon's debug log file for authentication or authorisation error messages. This also applies for community names with SNMP datasources.

4. Avoid underscore characters in datasource names. Underscore separates the datasource from the datapoint in a fully-qualified datapoint name. Using underscore as part of the datasource can cause confusions, especially with earlier versions of Zenoss.
5. If an *eventClass* is defined in a datasource file, that class must already exist in the ZODB database when an instance of this datasource is created; otherwise the *eventClass* field will be blank in the GUI.
6. Remember that datasource instances need to be unique because the RRD data for each device is saved in a filename that concatenates **datasource_datapoint** eg. *withoutPython_matches.rrd*.

16.5.7 Performance collection issues related to using zenpacklib

1. The ***label*** field for a component object class **not** the object class name is used, by default, to find component performance template to automatically apply.
2. If *zenpack.yaml* specifies the *zDeviceTemplates* zProperty (which is a list), to bind templates to a Zenoss device class, note that this property must contain **all** device-level templates to be bound; specifying a single new template in the *zDeviceTemplates* field **will remove any existing templates**.
3. If using zenpacklib to export templates, beware that a template description field tends to have single quotes around it; *zenpack.yaml* requires double-quotes, otherwise subsequent lines are all interpreted as comment.

16.6 Testing skins files and JavaScript files

If skins or JavaScript files have been created or changed, you generally only need to restart **zopectl** and then refresh the web page in the Zenoss GUI.

16.6.1 General failure errors

If the code is incorrect a standard error page is shown and you can get more information by clicking the *View Error Details* link.

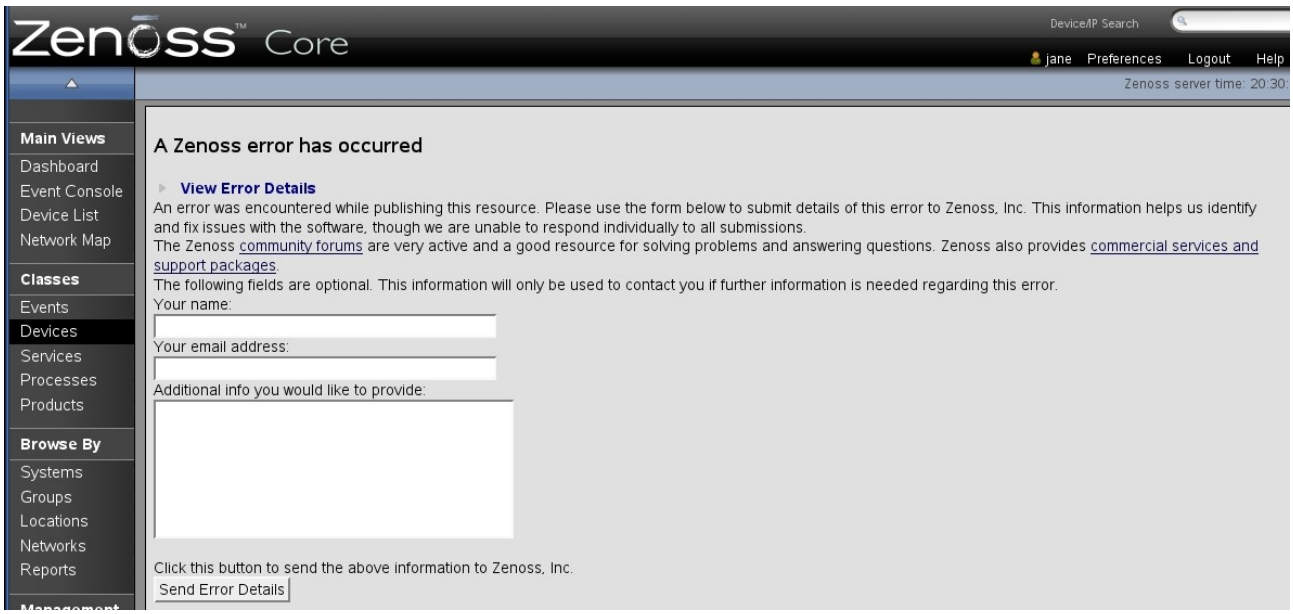


Figure 260: Standard error message for a faulty web page definition

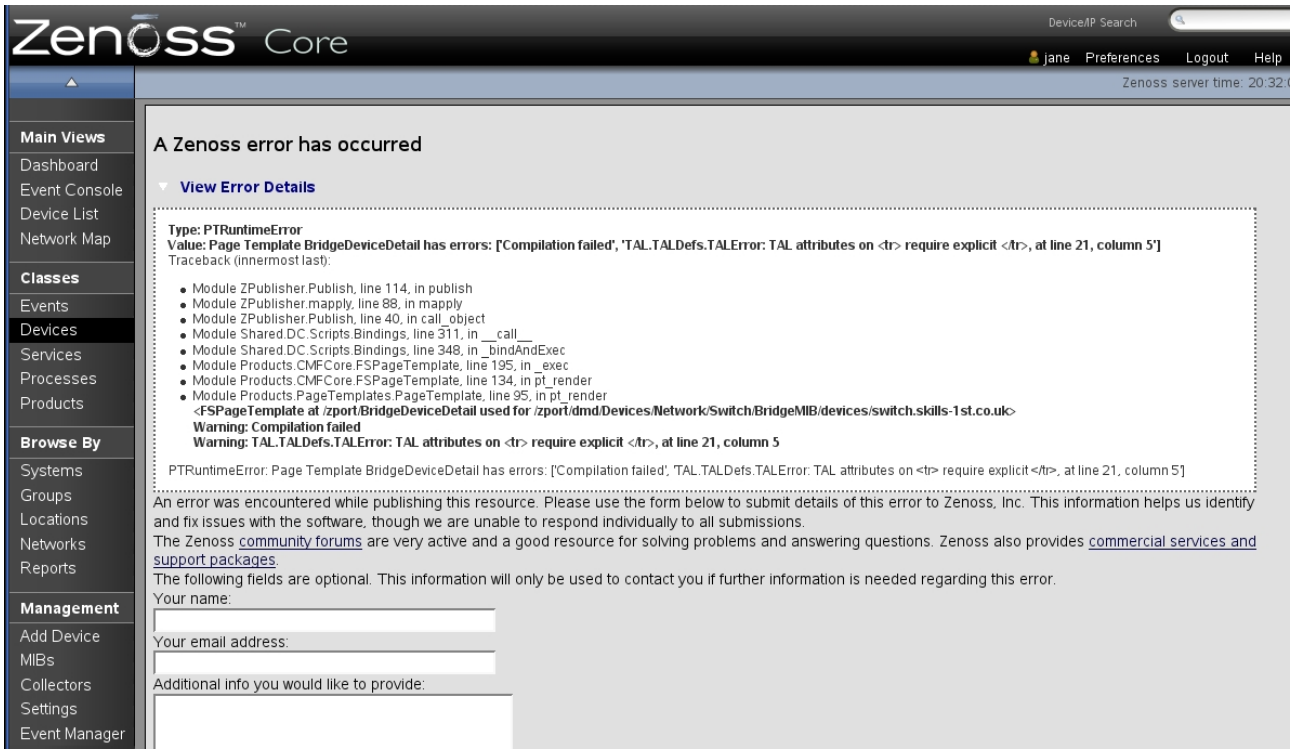


Figure 261: View Error Details for a faulty web page definition

Figure 261 shows the detailed error output. The file and line number at fault are documented (I had indeed commented out a closing </tr> at line 21 of the BridgeDeviceDetail.pt file). Simply fix the file, issue *zopectl restart* and refresh the web page.

It is always worth checking with a *zenoss status* command to see all daemons are running. Sometimes JavaScript errors are not immediately apparent in the GUI but debugging output appears in response to this command.

Sometimes the web *View Error Details* page suggests something is wrong that is nowhere near anything you have recently changed. If this happens, try restarting the whole Zenoss system with:

```
zenoss stop
zenoss start
```

If you suspect issues with JavaScript you could run a Firefox Web Console to see the JavaScript errors - “There will be tons of CSS issues coming from different CSS pages (it's annoying, but not fatal), and you can safely ignore them”, says the debugging section 13.8 of the original Zenoss Developer's Guide! If you filter out Warning severity messages from the Firefox Web Console, it may help you spot real issues quickly.

You may also find it helpful to view the page source of a page – for Firefox, use the *View -> Page Source* option.

Section 13.8 of the original Developer's Guide also has the following advice with regard to JavaScript:

> “The Firefox Error Console will not tell you if Firefox wasn't able to find or load a JavaScript file (if the path you've specified in your Web page to get to the JavaScript file is incorrect). In order to determine if Zope was given a path to a filename that it couldn't find, you'll need to go into Zope's ZMI, go to the error log (*http://yourzenossserver:<your port>/error_log/manage_main)* and remove all of the error log filters” - the default filters are *Unauthorized*, *NotFound* and *Redirect*.

You will need to be logged in as *admin* to do this or an error message will result. All Zenoss-configured users live in the */zport/acl_users* user folder; this means they can only ever hope to access items under */zport*. Only the *admin* user who lives in the *Zope /acl_users* user folder can access items at the root.

After changing the filters, retry the operation and you can see what files Zope wasn't able to find and fix the paths in your page. This technique is also helpful for debugging authentication issues – make sure that *Unauthorized* exceptions are not filtered out.

16.6.2 Problems displaying components

1. If you are not using zenpacklib and are writing JavaScript files, note that the new component panel **must** be called <object component name>Panel eg. *LogMatchPanel*. **There should be no *Grid* in this name**.
2. In JavaScript files, beware the common error of adding a column definition without adding the new definition into the *fields* stanza.
3. Brackets mismatch is extremely easy, especially in JavaScript files. The *vi %* command to match brackets is enormously helpful.

16.6.3 Issues with Info and Interface definitions and configure.zcml

The ***info.py*** file abstracts object attribute information saved in the Zope Object Database (ZODB), that will be displayed to the user. Note that the file must have this exact name. It describes **what** will be displayed not how something will be displayed.

interfaces.py describes **how** the data is displayed (and again, this filename is prescribed).

Something needs to ultimately tie together the different display elements. That is the role of ***configure.zcml*** which provides the "glue" between interfaces and JavaScript display code and this exact name will be searched for by the Zope mechanisms.

Forgetting to write / update *info.py*, *interfaces.py* and/or *configure.zcml* is common.

Errors in a *configure.zcml* are scary as they will prevent *zenhub* from starting, with an error message. Fortunately it is quite good at pinpointing where the error is.

```
File "/opt/zenoss/lib/python/zope/configuration/fields.py", line 229, in fromUnicode
  raise InvalidToken("%s in %s" % (v, u))
zope.configuration.xmlconfig.ZopeXMLConfigurationError: File
"/opt/zenoss/etc/site.zcml", line 16.2-16.23
  ZopeXMLConfigurationError: File
"/code/ZenPacks/DevGuide/ZenPacks.community.LogMatch/ZenPacks/community/LogMatch
/configure.zcml", line 15.4-18.11
  ConfigurationError: ('Invalid value for', 'for', 'ImportError: Module
ZenPacks.community.LogMatch.LogMatch has no global LogMatchDevice in
.LogMatch.LogMatchDevice')
```

1. Beware typos!
2. In *interfaces.py*, beware that some constructs require *schema* and other don't. It is easy to copy a line with say:

   ```
   logMatchRegEx = SingleLineText(title=_t(u"LogMatch RegEx"))
   ```

   and modify it with a construct such as

   ```
   logMatchCycle = Int(title=_t(u"LogMatch Cycle"))
   ```

   forgetting that *Int* needs to be *schema.Int*.
3. Ensure that *IComponentInfo* classes in *interfaces.py* specify suitable display types. Using a *schema.Int* definition when the object data is actually a string, will cause issues.
4. Edit *configure.zcml* with an editor that is XML-aware. color coding can help to prevent missed quotes or angle brackets.
5. If a component or effect is not displayed, check that the *for* statement in *configure.zcml* does not prevent display for the object..

   ```
   <viewlet
     name="js-custom-overview-device"
     paths="/++resource++LogMatchJavascript/js/custom-overview-device.js"
     for="..LogMatchDevice.LogMatchDevice"
     weight="10"
     manager="Products.ZenUI3.browser.interfaces.IJavaScriptSrcManager"
     class="Products.ZenUI3.browser.javascript.JavaScriptSrcBundleViewlet"
     permission="zope2.Public"
     />
   ```

6. Field names in JavaScript files must match **ComponentInfo** class attributes, **not** the *DeviceComponent* class attributes.

7. If there is an interface entry but no matching info for an attribute then the keyword will appear in the GUI dialogue but will not show default values and changes in the GUI will not be honoured.
8. If there is an entry for the info but not the interface then the keyword will not appear in the GUI but default values will be honoured. The Zope Management Interface (ZMI) is a good tool to check what values exist on the object.

16.6.4 GUI issues when using zenpacklib

1. When defining *column_width* fields in *zenpack.yaml,* if the total exceeds 750 pixels then all the *column_width* directives are ignored and revert to the default.

16.7 Testing and debugging problems with event elements

1. If a datasource template defines an Event Class field, ensure that Event Class is either already defined or is shipped as part of the ZenPack; otherwise the datasource customisation will show a blank Event Class.
2. Remember that any Event Classes defined in a ZenPack will be removed if the ZenPack is removed.
3. If the ZenPack installation fails after adding Event Class definitions, suspect syntax issues.
4. If a trigger or notification cannot be opened in the GUI suspect a specification error such as bad fields or typos.
5. Ensure that double quotes as part of a value string in a trigger or notification definition, must be escaped with backslash.
6. Ensure there are no comments in either *zep.json* or *actions.json* .

16.8 Problems with installing / removing ZenPacks

1. Ensure that a ZenPack has any GUI-created templates added to a ZenPack and the ZenPack is exported before reinstalling a ZenPack. Otherwise there is a danger of losing or overwriting required customisation.
2. Only ***remove*** a ZenPack if you really need to. Any objects that are part of the ZenPack (Events, MIBs, templates, etc) will be removed from the ZODB. Performing a ***reinstall*** of a ZenPack, although it performs a remove and install, does **not** remove any items from the ZODB database.
3. If a ZenPack includes a Zenoss device class (and zenpacklib encourages such an inclusion), then removing the ZenPack deletes any instances of devices in that class. For a zenpacklib-created device class, this is only true if:

   ```
   remove: True
   ```

   is specified as part of the device class definition in *zenpack.yaml.* The default is ***False***.
4. Removing a ZenPack that uses zenpacklib to create a device class with

```
remove: False
```

can cause issues. The device class is preserved but any device **instances** in that class remain in the ZODB and are inconsistent. The result is that the *INFRASTRUCTURE -> Devices* menu will be broken for the removed device class **and all its parents up to the root Devices level**. The symptom is a yellow message at the top of the GUI saying:

```
Type error ('Could not adapt ' ,,)
```

The resolution is to reinstall the ZenPack.

Best practice should either be to set the *remove* parameter to *True*, understanding that device instances will be deleted, or ensure a manual process exists to ensure that a device class is empty before ZenPack removal, if *remove* is set to *False*.

5. When exporting a ZenPack, note that there is a bug whereby the *build/lib* subdirectory is **not** cleaned out before export. This means that existing files **will** be updated, new files **will** be added but if old files have been deleted then they will **still exist** in the *build/lib* subdirectory and hence, in the new egg file.

17.0 Developing a ZenPack and making it publicly available

A ZenPack for private use can be developed using any techniques; a ZenPack to be shared with other users within an organization or with the wider Zenoss community, should probably be developed within a **git** framework.

git is a tool for tracking changes to a set of files over time, a task traditionally known as “version control”. It has become one of the leading standards for package development and is available for many different platforms (https://git-scm.com/).

GitHub (https://github.com/) is the public, free, open source site where users and organizations can create, store and share repositories. This is where Zenoss maintains their repository of ZenPacks (https://github.com/zenoss).

A good starting point for documentation is the git website at https://git-scm.com/doc ; there is also a useful O'Reilly Git Pocket Guide.

17.1 Simple procedure for git development

git can be a very powerful and complex collaborative tool; however, this is not a git tutorial. There is a useful article on the Zenoss wiki about releasing ZenPacks - http://wiki.zenoss.org/Releasing_your_ZenPack .

The first step is to install git on your development system. Instructions and code for most environments can be found at https://git-scm.com/downloads .

Once git is installed some global configuration should take place to identify a git user. The following will set a global username and email for all repositories:

```
git config --global user.email "jane.curry@skills-1st.co.uk"
git config --global user.name "Jane Curry"
```

Note that git commit statements will fail unless a git user is identified thus.

A simple procedure for creating a ZenPack with git would be:

1. Create a development directory.

   a. It is good practice for this **not** to be under *$ZENHOME/ZenPacks*. The examples here will be under */code/ZenPacks/DevGuide* .

```
mkdir /code/ZenPacks/DevGuide/ZenPacks.community.DirFile
cd /code/ZenPacks/DevGuide/ZenPacks.community.DirFile
```

2. Initialise the current directory as a git repository

```
git init
```

   a. This creates a *.git* directory for the metadata and history involved with this repository. Don't change anything in this directory unless you really know what you are doing!

   b. The default **branch** will be created, called *master*.

3. Create a *.gitignor*e file in */code/ZenPacks/DevGuide/ZenPacks.community.DirFile*

   a. The *.gitignore* file defines file patterns to exclude from the repository. For example, you should not include compiled Python files ending in *.pyc* or *.pyo*. A typical ZenPack *.gitignore* would be:

```
*.pyc
*.pyo
build/
setuptools*
*.egg-info/
*EGG-INFO*
```

   b. Note that the *dist* directory (where the *.egg* file will be saved) is **not** ignored, though the *build* directory **is**.

4. If this is a brand-new ZenPack, start development as usual in */code/ZenPacks/DevGuide/ZenPacks.community.DirFile.*

5. If the ZenPack has already been partially developed in a different directory hierarchy, (say, */tmp/fred/ZenPacks.community.DirFile*), perform a recursive copy:

```
cd /tmp/fred/ZenPacks.community.DirFile
cp -R * /code/ZenPacks/DevGuide/ZenPacks.community.DirFile
```

6. Ensure that the ZenPack has a *README .rst* documentation file in the top-level directory.

7. From the top-level git development directory, */code/ZenPacks/DevGuide/ZenPacks.community.DirFile,* run

```
git status
```

   a. This shows the current status of all files in the repository with respect to git. At this stage, all files will be untracked.

```
# On branch master
#
```

```
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
#       MANIFEST.in
#       README.rst
#       ZenPacks/
#       dist/
#       setup.py
nothing added to commit but untracked files present (use "git add" to track)
```

8. To add all files in the current directory and subdirectories to the git repository, use:

```
git add -A
```

   a. Note that the *-A* flag ensures **all** changes, additions and deletions are incorporated into the git index.

   b. A second `git status` should now show all the files as "new" and as changes to be "committed".

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   .gitignore
#       new file:   MANIFEST.in
#       new file:   README.rst
#       new file:   ZenPacks/__init__.py
#       new file:   ZenPacks/community/DirFile/DirFileTemp.yaml
#       new file:   ZenPacks/community/DirFile/DirFile_templates.yaml
#       new file:   ZenPacks/community/DirFile/LICENSE.txt
#       new file:   ZenPacks/community/DirFile/__init__.py
#       new file:   ZenPacks/community/DirFile/configure.zcml
#       new file:
ZenPacks/community/DirFile/datasources/DirFileDataSource.py
#       new file:
ZenPacks/community/DirFile/datasources/DirFilePythonDataSource.py
#       new file:   ZenPacks/community/DirFile/datasources/__init__.py
#       new file:   ZenPacks/community/DirFile/dsplugins.py_works
```

9. Perform an initial git ***commit***. The commit is the fundamental unit of change in git. It is a snapshot of the entire repository content, together with identifying information, and the relationship of this historical repository state to other recorded states, as the content has evolved over time.

   a. The *-m* parameter should **always** be unique.

```
 git commit -m 'First commit'
```

   b. A further `git status` should show:

```
# On branch master
nothing to commit (working directory clean)
```

10. Development of the ZenPack can now continue. Install the ZenPack in development mode with the link flag:

```
zenpack --link --install ZenPacks.community.DirFile
zenoss restart
```

11. As content is added, removed and updated, repeat the cycle of:

```
git add -A
git commit -m 'With updated readme 1'
git status
```

    a. Don't forget to repeat these steps after adding items to a ZenPack from the GUI and exporting the ZenPack (thus updating *objects.xml* and the ZenPack *.egg* file).

17.2 Working with GitHub

Once the ZenPack has been developed and tested it can be uploaded to GitHub to be shared with other people (it is also a good way to effect a remote backup of your development).

A free GitHub account can be created by visiting https://github.com/join .

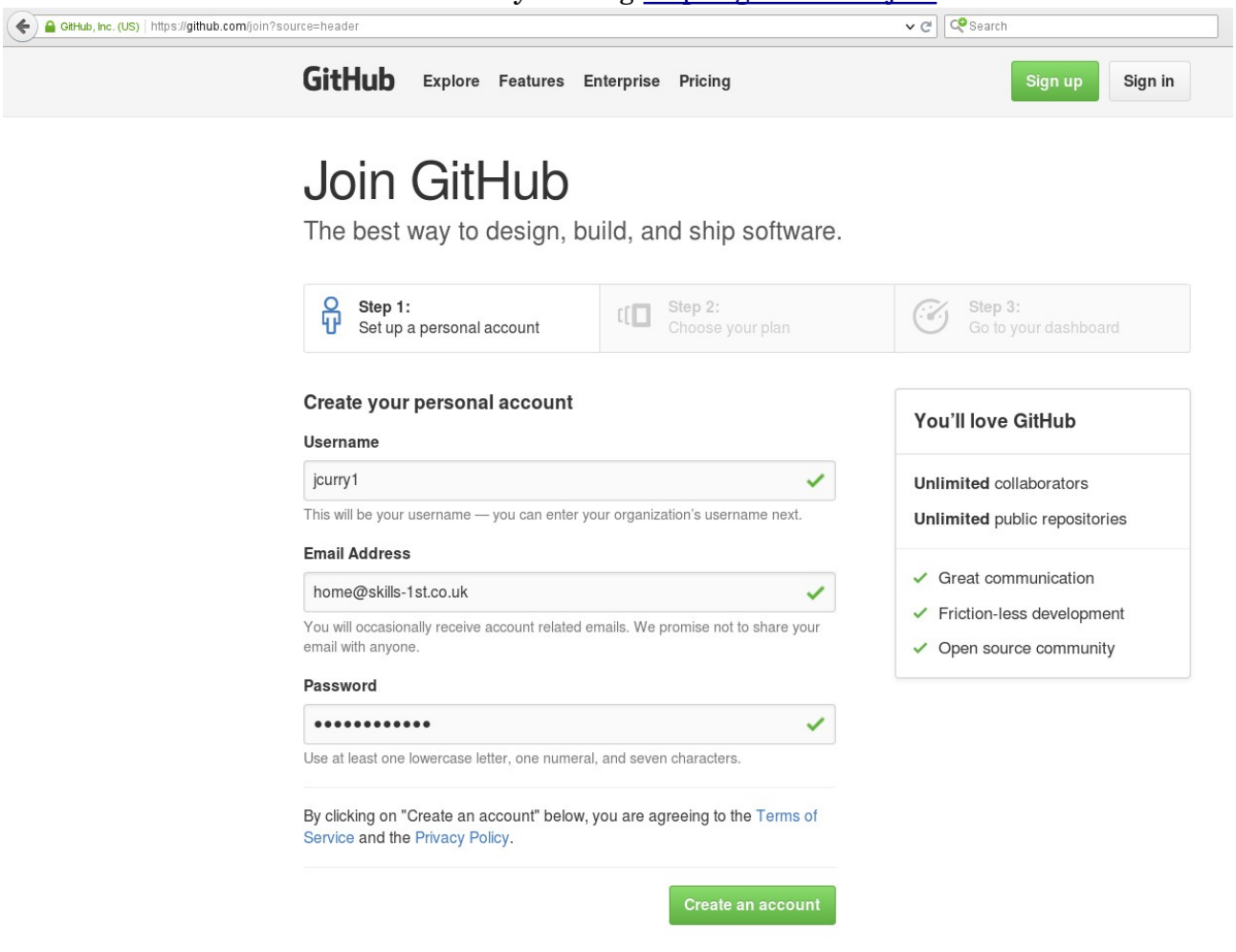


Figure 262: Sign up for a GitHub account

Choose the *Free* account option.

17.2.1 Using ssh authentication with GitHub

ssh is the usual method for transferring repositories between a local system and GitHub - note that this requires OpenSSH to be installed on the local system. There is useful help on GitHub for setting up ssh key authentication - https://help.github.com/articles/generating-an-ssh-key/ .

ZenPack development will generally be done as the *zenoss* user so check this user's home directory for a *.ssh* directory (note the leading dot). If this directory contains *id_rsa.pub* then this will be used with GitHub; otherwise, use the following to generate a new public-key pair:

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

The contents of *.ssh/id_rsa.pub* should be copied to the clipboard for pasting into GitHub. Ensure that no extra whitespace or linefeeds are added.

ssh keys are added to a GitHub account through the user's *Settings* menu.

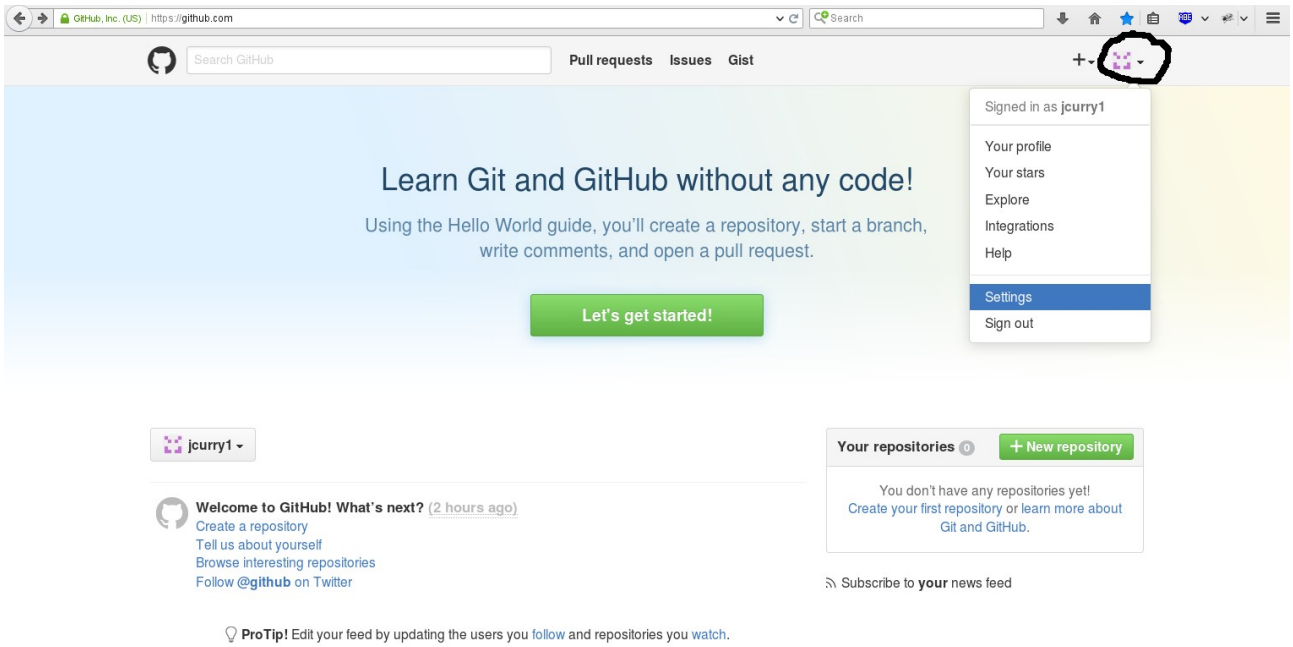


Figure 263: Settings menus for GitHub

Select *SSH Keys* from the left-hand menu, and use the *New SSH key* button. The *Title* field should provide identification for you as to what the key is for; the contents of *id_rsa.pub* should be copied from the clipboard into the *Key* window.

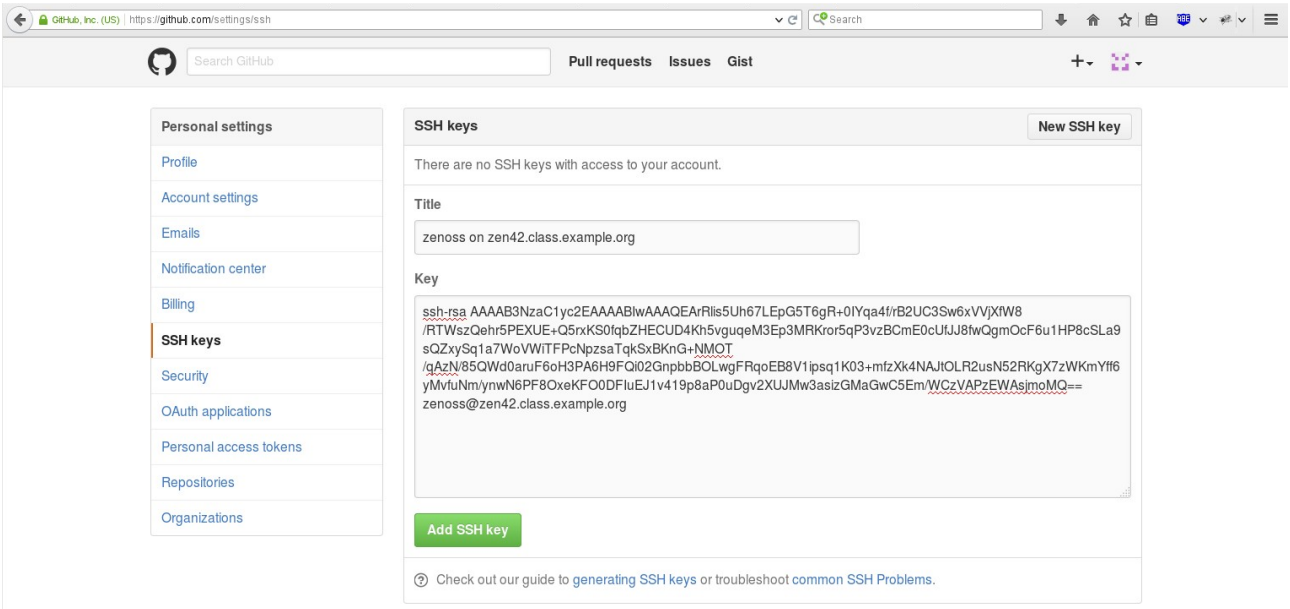


Figure 264: Configuring SSH keys for a GitHub user

17.2.2 Creating the GitHub repository

A new repository can be created on GitHub by using the + icon (top-right) once the user has logged in. The name should be identical to the repository name on the local development system; that is, the name of the ZenPack.

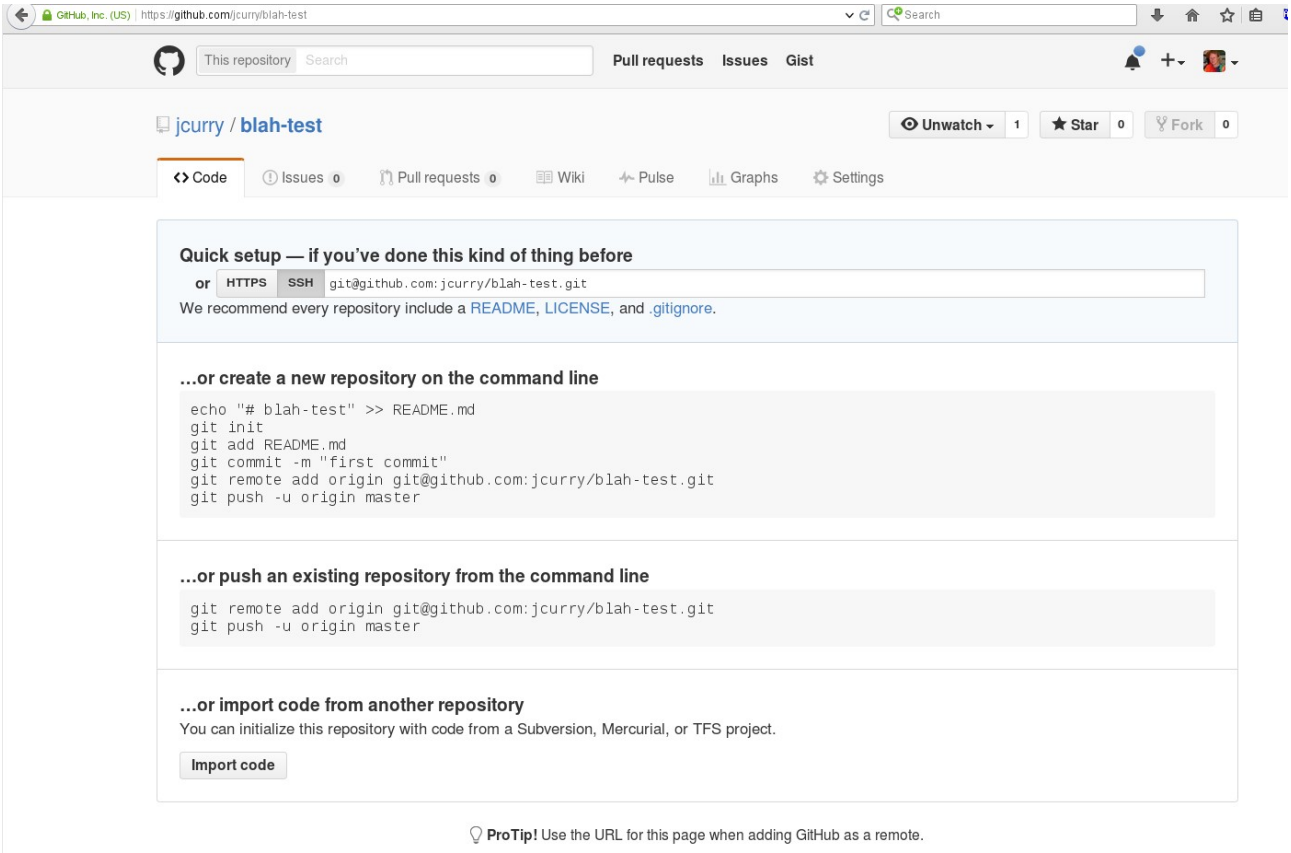


Figure 265: Creating a new repository on GitHub

Once created, GitHub provides the correct commands to setup a **remote** on your local system. Any local repository may be able to communicate with one or more remote repositories, typically on GitHub. By convention, the remote on GitHub is called *origin*. Note carefully the format of the remote; the user should be git@github.com; the GitHub user name then follows after a colon.

```
git remote add origin git@github.com:jcurry/ZenPacks.community.DirFile.git
```

git remotes can be shown on the local system with:

```
git remote -v
```

The final stage is to push the repository from the local system to GitHub:

```
git push -u origin master
```

The *-u* parameter will only be used on the initial push to create the *master* branch on the remote GitHub repository; *origin* is the remote target repository; *master* is the remote branch. The local repository to be pushed is whatever is currently active (ie the directory you are in) and the branch is whatever is currently checked out (at this stage, only the *master* branch exists locally).

17.3 git branches

git has the useful capability of supporting lots of different **branches**. Once a basic working repository exists, a new development can be tested in a separate branch and then merged into the *master* branch, if required, once tested.

```
git checkout -b events
```

will create a new branch called *events* and check it out (that is, make it the active branch). The original *master* branch will be unaffected by any subsequent changes until it is made active with:

```
git checkout master
```

To see all branches, including the currently active one, use:

```
[zenoss@zen42 ZenPacks.community.DirFile]$ git branch -a
* events
 master
 python
 remotes/origin/events
 remotes/origin/master
 remotes/origin/python
```

The asterisk against *events* indicates that is currently the active branch.

17.4 Cloning from GitHub to a local machine

If a repository exists on GitHub that you wish to use, it can be **cloned**. Assuming that *ZenPacks.community.UserGroup* does not exist on the local machine, change to the ZenPack development directory and use:

```
git clone ssh://git@github.com/ZenossDevGuide/ZenPacks.community.UserGroup
```

The code can then be modified locally and pushed back to GitHub. If, in the meantime, someone else has updated the GitHub code, the push will fail. It is necessary to pull the GitHub code and merge the changes on the local system, before pushing the combined package back. A “pull” is actually a combined *fetch* and *merge*:

```
git fetch origin
git merge origin/master -m 'test merge message'
```

17.5 Other ways to use GitHub

If you cannot install git in your environment, repositories can still be retrieved from GitHub. From a repository on GitHub, use the *Download ZIP* button to get a zipped archive. The archive then needs to be manually unpacked.

17.6 ZenPacks on the Zenoss wiki

All publicly-available ZenPacks are documented on the Zenoss wiki at http://wiki.zenoss.org/ZenPack_Catalog with various filters to retrieve different subsets such as:

- ZenPacks by Zenoss - both free and commercial varieties
- Community ZenPacks

There is a wiki item with good advice for releasing ZenPacks - http://wiki.zenoss.org/Releasing_your_ZenPack .

It is good practice to contribute any new, tested ZenPack into the community and a link to a form is provided at the bottom of the ZenPack Catalog wiki page - http://wiki.zenoss.org/Form:ZenPack .

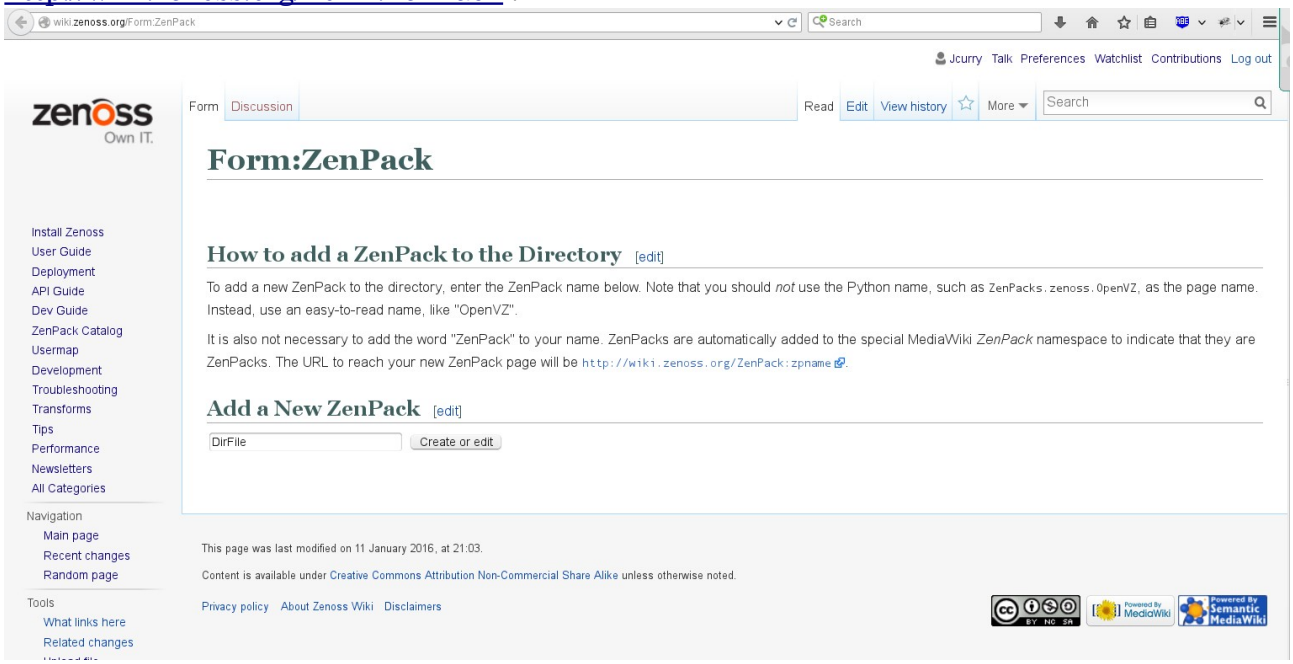


Figure 266: Form for submitting a new ZenPack to the Zenoss wiki

Note that the new ZenPack name should be a short name, not the full ZenPack name eg. *DirFile*.

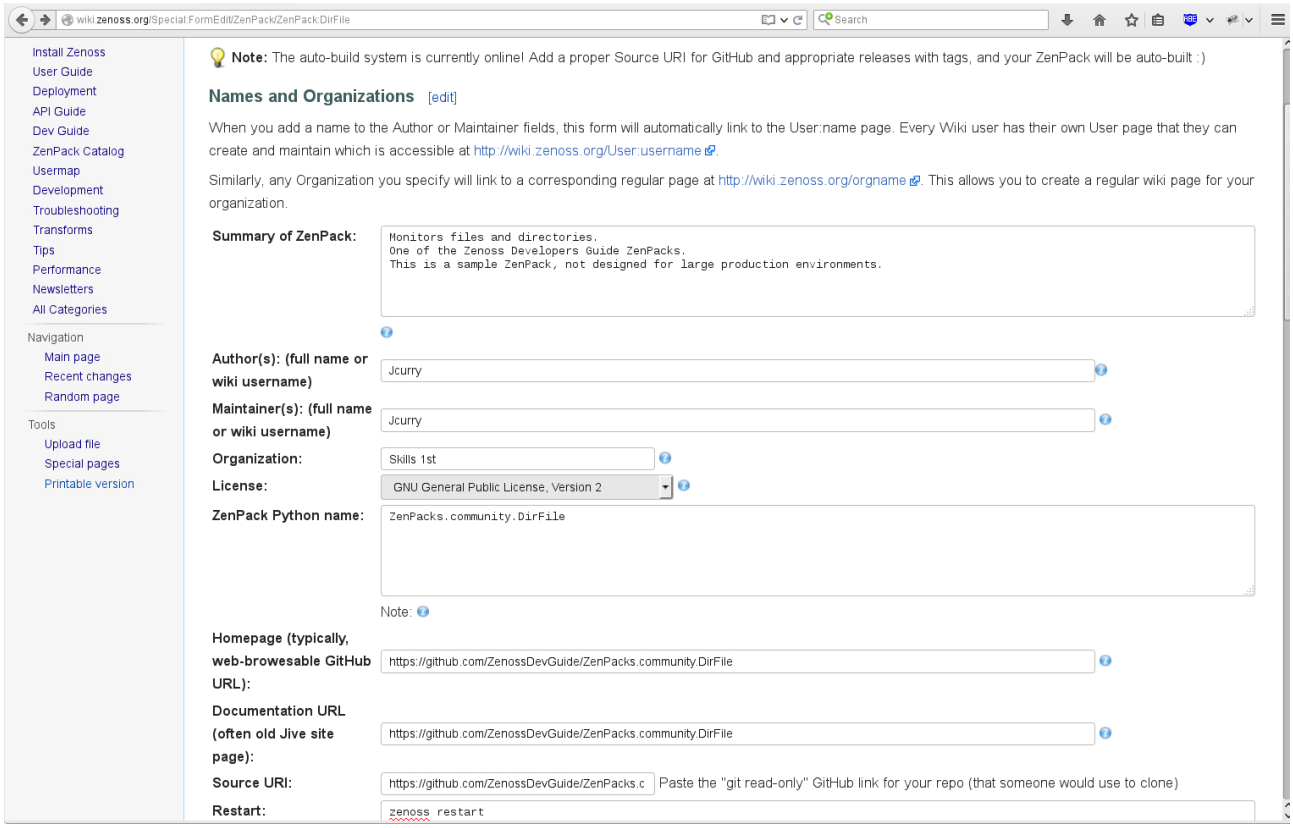


Figure 267: Submitting a new ZenPack to the Zenoss wiki

Provided the ZenPack includes a *README.rst* file documenting the ZenPack, supply the URL to the ZenPack main page on GitHub for each URL/URI request.

The *Flavor* entry should be *Free*.

Add a Release form for each separate release of the ZenPack.

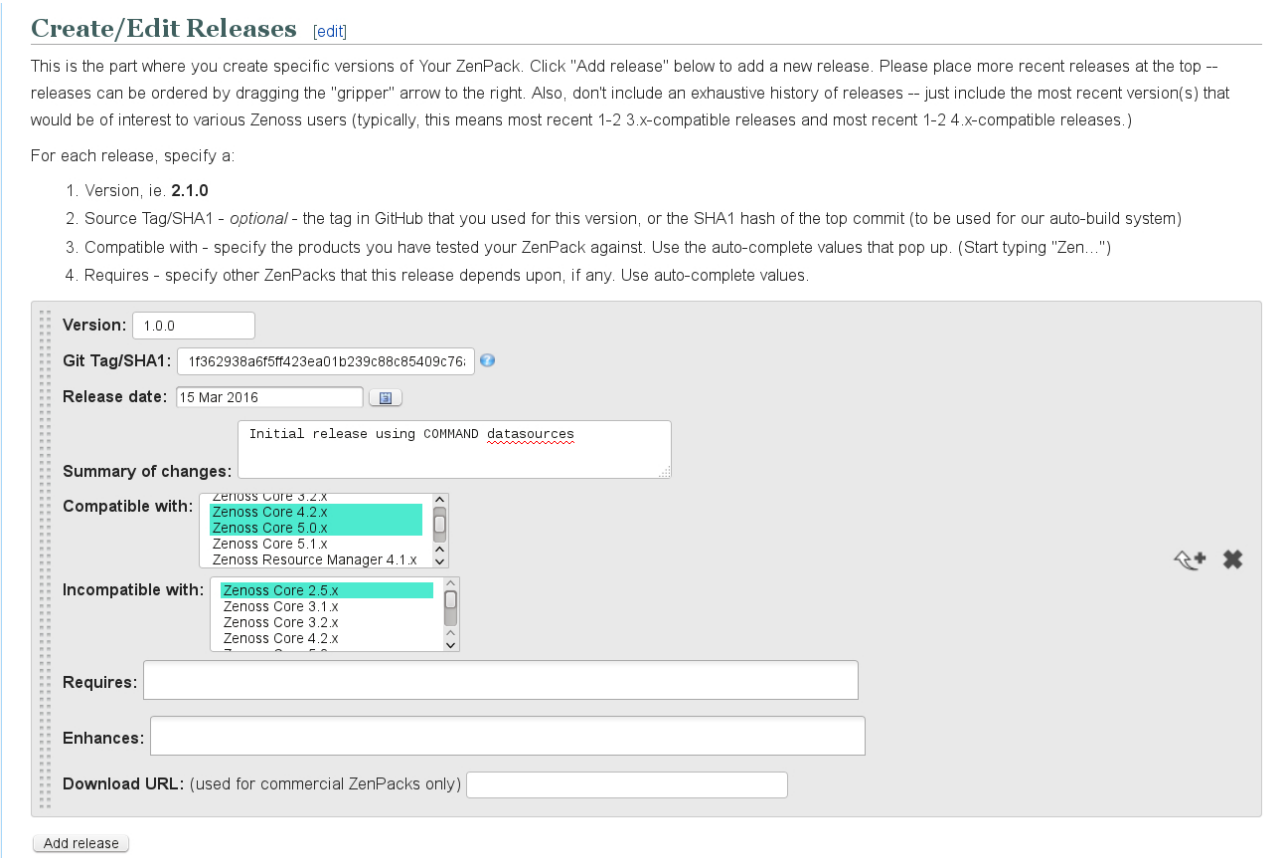


Figure 268: Add a release form for each separate release

Note in Figure 268 that the Git Tag/SHA1 field is found on the repository page in GitHub.

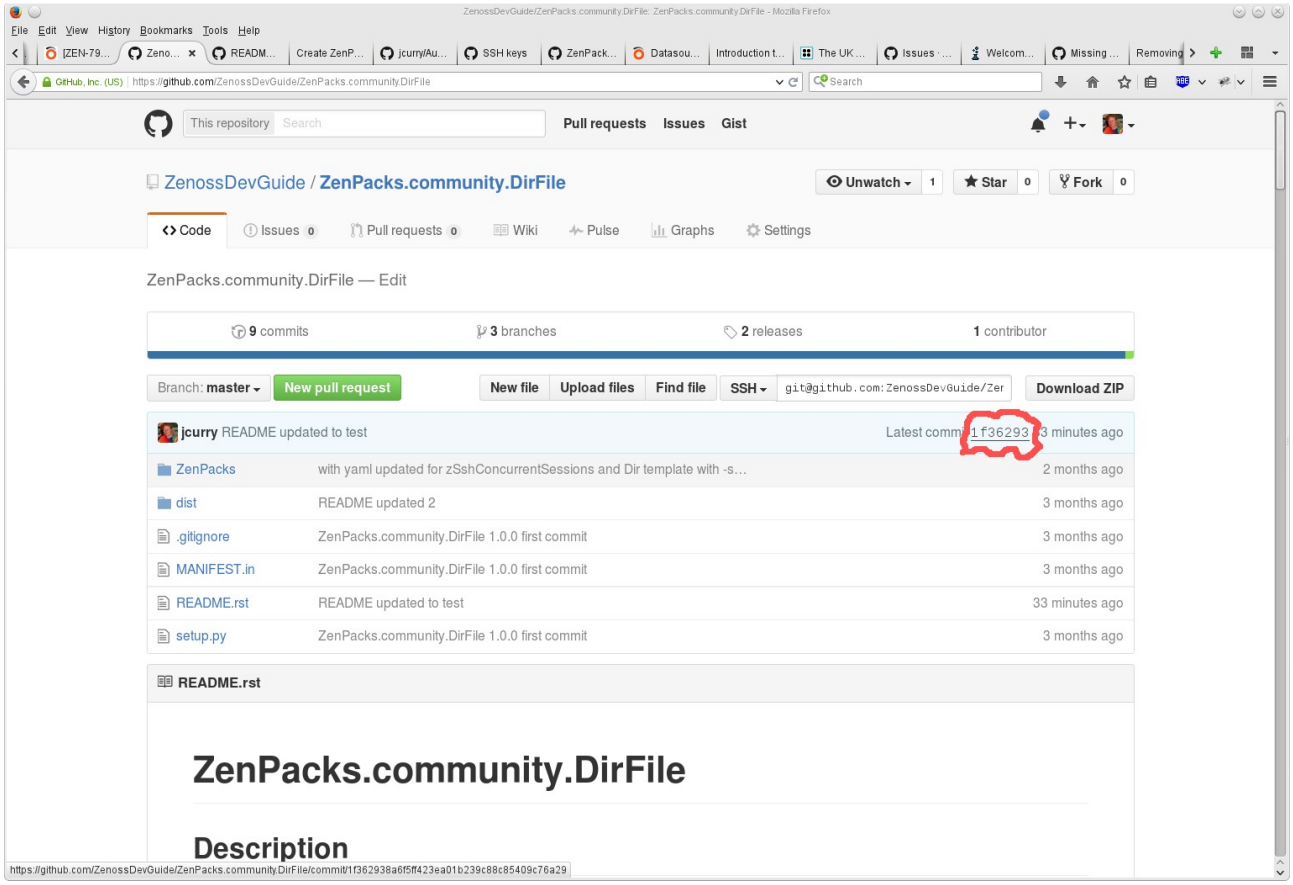


Figure 269: Finding the SHA1 field for the ZenPack release form

Click on the Latest commit reference to see the entire identification and cut-and-paste it to the release form.

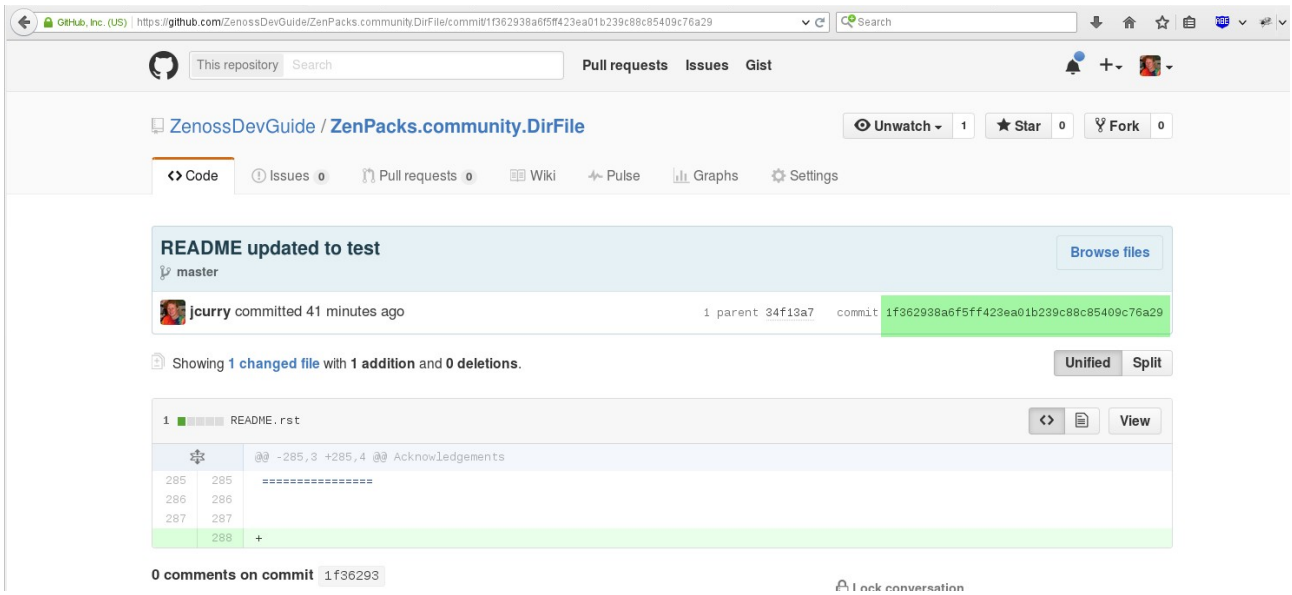


Figure 270: Cut and paste the full SHA1 reference to the Release form

Provided this SHA1 identification is correct, the wiki will automatically find the code, build an egg version of the ZenPack and make it available through the wiki ZenPack Catalog. The auto-build process runs every 5 minutes.

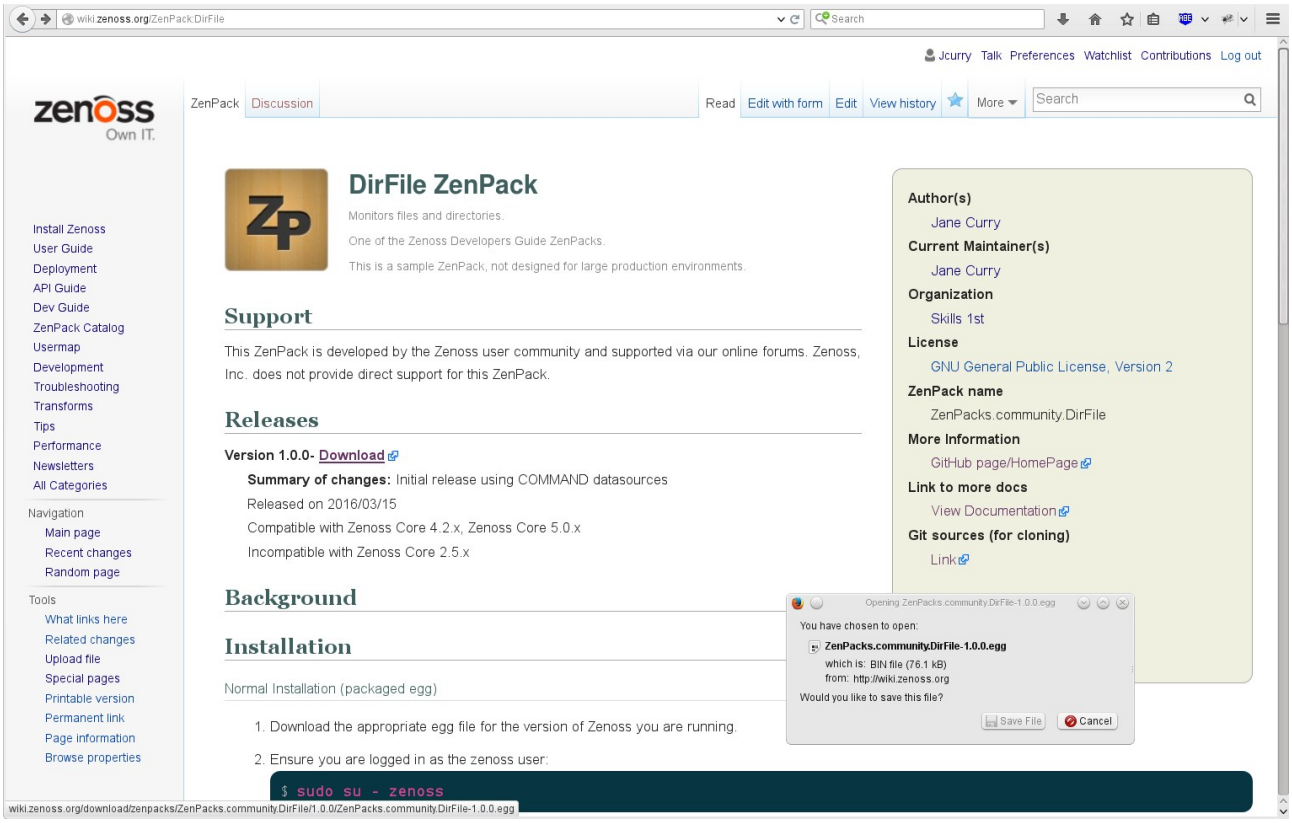


Figure 271: Zenoss wiki ZenPack Catalog page for new ZenPack with automatically created download link

References

1. Official Zenoss documentation - https://www.zenoss.com/resources/documentation?field_zsd_core_value_selective=Core
2. Zenoss wiki ZenPacks home page - http://wiki.zenoss.org/ZenPack_Catalog
3. Free Zenoss provided ZenPacks on the wiki - http://wiki.zenoss.org/Free_ZenPacks_by_Zenoss
4. Zenoss community ZenPacks on the wiki - http://wiki.zenoss.org/Community_ZenPacks
5. Chargeable Zenoss ZenPacks available only to Service Dynamics customers - http://wiki.zenoss.org/Commercial_ZenPacks_by_Zenoss
6. GitHub site for Zenoss ZenPacks - https://github.com/zenoss
7. Zenoss documentation and taxonomy guidance at https://zenosslabs.readthedocs.org/en/latest/ or the pdf version at http://media.readthedocs.org/pdf/zenosslabs/latest/zenosslabs.pdf
8. ZenPacks standards guide and best practices - https://zenosslabs.readthedocs.org/en/latest/zenpack_standards_guide.html
9. ZenPackers documentation website - http://zenpackers.readthedocs.io/en/latest/ . This is maintained on github at https://github.com/zenoss/ZenPackers/tree/master
10. reStructuredText reference for README files - http://docutils.sourceforge.net/rst.html
11. zenpacklib documentation at http://zenpacklib.zenoss.com/en/latest/
12. Zenoss wiki page with links to sample ZenPack examples and web-based training - http://wiki.zenoss.org/ZenPack_Development_Guide
13. Zenoss wiki tips for creating a ZenPack development environment - http://wiki.zenoss.org/ZenPack_Development_Guide/Development_Environment
14. Zenoss Developer's Guide 3 - http://docs.huihoo.com/zenoss/3/Zenoss_Developers_Guide_08-102010-3.0-v01.pdf
15. “Creating Zenoss ZenPacks” paper from http://www.skills-1st.co.uk/papers/jane/zenpacks/
16. “Creating Zenoss ZenPacks for Zenoss 3” paper from http://www.skills-1st.co.uk/papers/jane/zenpacks/zenpacks3.pdf
17. “Zenoss Datasources through the eyes of the Python Collector ZenPack” paper from http://www.skills-1st.co.uk/papers/jane/PythonZenPacks.pdf
18. “ZenPack Development Procedures” document for working with ZenPacks on Zenoss's ZenPack site, written by David Buler (“phonegi”) - http://community.zenoss.org/docs/DOC-10223
19. Wiki item about Devices, Device Components and relationships - http://wiki.zenoss.org/ZenPack_Development_Guide/Background_Information
20. Wiki item for *zendmd* tips - http://wiki.zenoss.org/ZenDMD_Tips
21. Wiki Tips category - http://wiki.zenoss.org/Category:Tips

22. For advice on running a ZenPack development environment for Zenoss 5, see http://zenpacklib.zenoss.com/en/latest/development-environment-5.html
23. Zenoss community developer site wiki at http://monitoringartist.github.io/community.zenoss.org/docs/DOC-2350.html, "Diving into the device model".
24. "Custom ZenPacks rough guide" contributed by ***blacks*** to the Zenoss forum at http://community.zenoss.org/docs/DOC-2358
25. Brief description of Zenoss 4 daemons - https://support.zenoss.com/hc/en-us/articles/202250769-An-Overview-of-Resource-Manager-4-x-Daemons
26. netSnmp SNMP agent from http://www.net-snmp.org/
27. BRIDGE MIB, RFC 1493 - http://www.ietf.org/rfc/rfc1493.txt
28. http://www.net-snmp.org/docs/mibs/UCD-SNMP-MIB.txt for the UCD MIB
29. oidview online website for viewing MIBs such as the BRIDGE MIB - http://www.oidview.com/mibs/0/BRIDGE-MIB.html
30. ZenPacks.community.mib_browser from http://wiki.zenoss.org/ZenPack:MIB_Browser
31. "Learning Python" by Mark Lutz, published by O'Reilly
32. "Python Pocket Reference" by Mark Lutz, published by O'Reilly
33. "Twisted - Network programming Essentials" by Jessica McKellar & Abe Fettig,https://github.com/zenoss/zenpacklib/issues published by O'Reilly
34. https://docs.python.org/2/library/index.html for online Python documentation
35. pyflakes Python syntax checker obtainable from https://pypi.python.org/pypi/pyflakes and pyflakes-vim plugin for vi at http://www.vim.org/scripts/script.php?script_id=2441
36. pep8 Python style guide checker - https://pypi.python.org/pypi/pep8
37. Zope web application server information from http://www.zope.org/WhatIsZope
38. "The Zope2 Book" from http://docs.zope.org/zope2/zope2book/
39. Zope Page Templates Reference - http://docs.zope.org/zope2/zope2book/AppendixC.html
40. Zope Configuration Markup Language (ZCML) reference - http://docs.zope.org/zopetoolkit/codingstyle/zcml-style.html
41. Zope 2 Interfaces reference - http://docs.zope.org/zope2/zdgbook/ComponentsAndInterfaces.html
42. Modifying the Overview panel for a device - http://wiki.zenoss.org/Device_Overview_Panels and https://github.com/cluther/ZenPacks.example.CustomOverview
43. Differences in ZenPacks between ExtJs3 and ExtJs4 for custom Component Grid panels - https://github.com/zenoss/Zenoss-User-Interface-API-Docs/tree/master/guides/component_grid_upgrade
44. YAML website - http://yaml.org/

45. zenpacklib command reference - http://zenpacklib.zenoss.com/en/latest/command-line-reference.html
46. zenpacklib YAML reference - http://zenpacklib.zenoss.com/en/latest/yaml-reference.html
47. zenpacklib on GitHub - outstanding issues and requests - https://github.com/zenoss/zenpacklib/issues
48. Sample command-based ZenPack, ZenPacks.zenoss.RabbitMQ - https://github.com/zenoss/ZenPacks.zenoss.RabbitMQ
49. General comments on modeler process method output in the ZenPacks.zenoss.OpenVZ ZenPack - https://github.com/zenoss/ZenPacks.zenoss.OpenVZ/blob/develop/ZenPacks/zenoss/OpenVZ/modeler/plugins/zenoss/cmd/linux/OpenVZ.py - see comments at the end
50. Good example modeler handling components, sub-components and dsplugins - ZenPacks.zenoss.AWS - https://github.com/zenoss/ZenPacks.zenoss.AWS/blob/develop/ZenPacks/zenoss/AWS/modeler/plugins/aws/EC2.py
51. PythonCollector ZenPack - http://wiki.zenoss.org/ZenPack:PythonCollector
52. See https://github.com/cluther/ZenPacks.example.EvaluatedCommandModeler for an example to permit the use of zProperties in the command definition of a CommandPlugin modeler.
53. Look at https://gist.github.com/James-Newman/9609c84688a0b9a4fee842878b9a5b00 which adds device classes, templates to device classes and event classes through the __init__.py of a ZenPack.
54. See http://zenpacklib.zenoss.com/en/latest/tutorial-http-api/datasource-plugin-datapoints.html for examples of Python datasources with explanations.
55. Good example ZenPack demonstrating *dsplugins.py* for Python datasources - http://wiki.zenoss.org/ZenPack:Hadoop and https://github.com/zenoss/ZenPacks.zenoss.Hadoop
56. Good example of using *dsplugins* directory - http://wiki.zenoss.org/ZenPack:HBase , https://github.com/zenoss/ZenPacks.zenoss.HBase , https://github.com/jcurry/ZenPacks.community.zplib.twemproxy and https://github.com/jcurry/ZenPacks.community.zplib.Redis
57. Helpful items regarding Python Twisted and the yield function - http://stackoverflow.com/questions/3894278/twisted-deferred-addcallback-vs-yield-and-inlinedeferred , https://confluence.oceanobservatories.org/display/CIDev/Gotchas+with+inlineCallbacks,+yield+and+returnValue
58. Discussion on blocking and non-blocking Python plugins on Zenoss forum at http://www.zenoss.org/forum/136876
59. Dave Peticolas' Twisted Introduction - http://krondo.com/?page_id=1327
60. Nagios format output guide - https://nagios-plugins.org/doc/guidelines.html#PLUGOUTPUT

61. Python regex check application - http://www.pyregex.com/
62. For a wiki tip on providing Event Details, Triggers and Notifications in a ZenPack, see http://wiki.zenoss.org/Providing_Triggers_Notifications_and_Event_Details_in_ZenPack
63. Audit package on GitHub for dumping many parts of the ZODB database, including triggers and notifications - https://github.com/jcurry/Audit
64. git website - https://git-scm.com/ and git documentation - https://git-scm.com/doc
65. GitHub site - https://github.com/
66. Zenoss area on GitHub - https://github.com/zenoss
67. “Git Pocket Guide” by Richard E. Silverman, published by)'Reilly
68. Wiki item for releasing ZenPacks - http://wiki.zenoss.org/Releasing_your_ZenPack
69. Discussing on versions of the JavaScript Library ExtJs - https://github.com/zenoss/Zenoss-User-Interface-API-Docs/tree/master/guides/component_grid_upgrade
70. Some useful Knowledge Base articles are emerging which provide assistance with managing the Zenoss 5 environment:
    a) “Virtualization and Docker Containerization for Poets” - https://support.zenoss.com/hc/en-us/articles/202254069-Virtualization-and-Docker-Containerization-for-Poets
    b) “How to troubleshoot Resource Manager 5.x services that fail to start” - https://support.zenoss.com/hc/en-us/articles/207348996-How-to-troubleshoot-Resource-Manager-5-x-services-that-fail-to-start
    c) “How to Recover Control Center from Hardware Failure” - https://support.zenoss.com/hc/en-us/articles/204643769-How-to-Recover-Control-Center-from-Hardware-Failure
    d) “How to add tools or scripts into a Resource Manager 5.x Docker Container” - https://support.zenoss.com/hc/en-us/articles/207610516-How-to-add-tools-or-scripts-into-a-Resource-Manager-5-x-Docker-Container
    e) “Introduction to Zenoss Control Center” - https://support.zenoss.com/hc/en-us/articles/206278353-Introduction-to-Zenoss-Control-Center

ZenPack Reference

All ZenPacks created for this paper are available on GitHub under:

https://github.com/ZenossDevGuide/

1. ZenPacks.community.dummy
   a. This ZenPack simply has the default files and directories after initial ZenPack creation.
2. ZenPacks.community.IpServices
   a. This ZenPack is built entirely through the GUI to demonstrate adding IP services to a ZenPack.
3. ZenPacks.community.WinServices
   a. This ZenPack is built entirely through the GUI to demonstrate adding Windows services to a ZenPack.
4. ZenPacks.community.Processes
   a. This ZenPack is built entirely through the GUI to demonstrate adding processes to a ZenPack.
5. ZenPacks.community.simple1
   a. This ZenPack is created purely using the GUI. It provides SNMP and COMMAND templates, an SNMP MIB, an event with an event mapping instance, and a device class.
6. ZenPacks.community.LogMatch
   a. Version 1.0, git branch *master*
      i. This ZenPack monitors logfiles using SNMP capabilities from the netSnmp UCD agent. It also modifies the Overview panel in V1.0.1.
   b. Version 1.0.2, git branch *device*
      i. The 1.0.2 version of this ZenPack ignores the LogMatchDevice definition and modifies the __init__.py to make the LogMatch a component of the os component of the Device class. The versionTag and versionDate attributes are added directly to the Device class attributes.
   c. Version 1.0.3, git branch *zenpacklib*
      i. This version of the ZenPack uses zenpacklib, starting from the 1.0.1 version.
   d. Version 1.0.4, git branch *zpl_and_datapoint*
      i. This version of the ZenPack uses zenpacklib, starting from the 1.0.3 version, and adds a component configuration attribute that is populated by performance data.
7. ZenPacks.community.DirFile
   a. Version 1.0, git branch *master*

i. This ZenPack is designed to be very easy to setup and test, rather than a production-strength monitoring solution. It uses several different methods to:

- Gather directory and file data based on new ZenPack-delivered zProperties
- Monitor disk space used by files and directories
- Monitor disk space used by the root filesystem
- Monitor counts of specific strings in the discovered files

ii. Version 1.0 uses COMMAND-based modeling and datasources

b. Version 1.0.1, git branch *python*

i. Converts all functionality in V1.0 to use the PythonCollector ZenPack

c. Version 1.0.2, git branch *events*

i. Adds event class, transforms, custom event field display, triggers and notifications to 1.0.1 version

d. Version 1.0.3, git branch *evalCommand*

i. Starts from *master* branch and 1.0 version and modifies *__init__.py* and the command modeler to make use of zProperties in the command.

8. ZenPacks.community.MenuExamples

a. Demonstrates several examples of extending Zenoss GUI menus:

9. ZenPacks.community.UserGroup

a. zenpacklib example that gathers Unix user and group information

10. ZenPacks.skills1st.bridge

a. Example used to demonstrate older GUI techniques

Acknowledgements

Several people have contributed either actively or passively to this paper:

- Jason Stanley for proof reading and code help
- James Newman for proof-reading and code help
- Bryan Irvine for proof-reading, comments and contributions
- Steve Paras-Charlton for proof-reading and comments
- Simon Nasrallah for proof-reading and comments
- Helena Patching for proof reading and comments
- Frederic Ma for proof-reading and comments
- Kevin Smith for proof-reading and comments
- Luckie Ford for proof-reading and comments
- Ken Jenkins for proof-reading and comments
- Andrew Kirch for proof-reading and comments
- Chet Luther for his awesome knowledge of Zenoss and his willingness to share that knowledge
- Andrew Findlay of Skills 1st for help with typesetting
- Georges Reichs for the original "amazing event architecture" diagram
- “blacks” on the Zenoss forum for his Custom ZenPack Rough Guide that got me started. The original work for this was submitted by Zach Davis.
- George Fakhri for his blog post on “How to create a ZenPack..”
- “bigegor” on the Zenoss forum for his excellent ZenPacks used extensively as examples, and for his responses to questions
- David Buler (“phonegi”) contributed hugely by doing the “detective work” on the mechanics of the component panel code
- Alison Guzzio for managing this project through the Zenoss corporation

About the author

Jane Curry has been a network and systems management technical consultant and trainer for 30 years. During her 11 years working for IBM she fulfilled both pre-sales and consultancy roles spanning the full range of IBM's SystemView products prior to 1996 and then, when IBM bought Tivoli, she specialized in the systems management products of Distributed Monitoring & IBM Tivoli Monitoring (ITM), the network management product, Tivoli NetView and the problem management product Tivoli Enterprise Console (TEC), all based around the Tivoli Framework architecture.

Since 1997 Jane has been an independent businesswoman working with many companies, both large and small, commercial and public sector, initially delivering Tivoli consultancy and training. Over the last 8 years her work has been more involved with Open Source offerings, especially Zenoss.

She has developed a number of ZenPack add-ons to Zenoss Core and has a large number of local and remote consultancy clients for Zenoss customisation and development. She has also created several workshop offerings to augment Zenoss's own educational offerings. She is a frequent contributor to the Zenoss forums, wiki and IRC chat conversations and was made a Zenoss Master by Zenoss in February 2009.