Event Management for Zenoss Core 4

January 2013

Jane Curry

Skills 1st Ltd

www.skills-1st.co.uk

Jane Curry
Skills 1st Ltd
2 Cedar Chase
Taplow
Maidenhead
SL6 0EU
01628 782565

jane.curry@skills-1st.co.uk

www.skills-1st.co.uk

Synopsis

This paper is intended as an intermediate-level discussion of the Zenoss event system in Zenoss Core 4. The event architecture has changed dramatically in Zenoss 4 from previous versions.

It is assumed that the reader is already familiar with the Zenoss Event Console and with basic navigation around the Zenoss Graphical User Interface (GUI). It looks in some detail at the architecture behind the Zenoss event system – the daemons and how they are inter-related – and it looks at the structure of a Zenoss event and the event life cycle.

Zenoss can receive events from many sources in addition to Zenoss itself. Events from Windows, Unix syslogs and Simple Networks Management Protocol (SNMP) TRAPs are all examined in detail.

The process by which an incoming event is converted into a particular Zenoss event is known as event mapping and there are a number of different possible techniques for performing that conversion. These will all be explored along with the creation of new event classes.

Once an event has been received, classified and stored by Zenoss, automation may be required. Alerting to users by email and page is discussed, as are background actions to run commands or generate TRAPs.

Logging and debugging techniques are discussed in some details as is the JSON API for extracting data out of Zenoss.

This paper was written using Zenoss Core 4.2.3

The paper is a companion text to the Zenoss 4 Event Management Workshop.

Notations

Throughout this paper, text to by typed, file names and menu options to be selected, are highlighted by *italics;* important points to take note of are shown in bold.

Points of particular note are highlighted by an icon.

Table of Contents

1 Introduction....6
2 Zenoss event architecture....6
  2.1 Event Console....6
  2.2 Event Manager settings....10
  2.3 Event database tables....11
    2.3.1 Zenoss 2.x and 3.x....11
    2.3.2 Zenoss 4....14
  2.4 New event daemons....20
    2.4.1 RabbitMQ....20
    2.4.2 zeneventserver....22
    2.4.3 zeneventd....22
    2.4.4 zenactiond....23
    2.4.5 memcached....23
  2.5 Other database-related changes in Zenoss 4....24
  2.6 Event life cycle....25
    2.6.1 Event generation....27
    2.6.2 Application of device context....29
    2.6.3 Event class mapping....29
    2.6.4 Application of event context....30
    2.6.5 Event transforms....30
    2.6.6 Database insertions and de-duplication....31
    2.6.7 Resolution....32
    2.6.8 Ageing and archiving....34
3 Events generated by Zenoss....34
  3.1 zenping....35
  3.2 zenstatus....36
  3.3 zenprocess....36
  3.4 zenwin....37
  3.5 zenwinperf....37
  3.6 zenperfsnmp....37
  3.7 zencommand....38
4 Syslog events....38
  4.1 Configuring syslog.conf....39
  4.2 Zenoss processing of syslog messages....40
5 Zenoss processing of Windows event logs....48
  5.1 Management using the WMI protocol....48
  5.2 Management of Windows systems using syslog....51
6 Event Mapping....51
  6.1 Working with event classes and event mappings....52
    6.1.1 Generating test events....54
  6.2 Regex in event mappings....55

6.3 Rules in event mappings....57
6.4 Other elements of event mappings....58
7 Event transforms....58
7.1 Different ways to apply transforms....59
7.2 Understanding fields available for event processing....60
7.2.1 Event Proxies....63
7.2.2 Event Details....66
7.3 Transform examples....68
7.3.1 Combining user defined fields from Regex with transform....68
7.3.2 Applying event and device context in relation to transforms....69
8 Testing and debugging aids....71
8.1 Log files....71
8.1.1 zeneventd.log....71
8.1.2 zeneventserver.log....72
8.1.3 Other log files....75
8.2 Using zendmd to run Python commands....75
8.2.1 Referencing an existing Zenoss event for use in zendmd....75
8.2.2 Using zendmd to understand attributes for an EventSummaryProxy....79
8.3 Using the Python debugger in transforms....83
9 Zenoss and SNMP....87
9.1 SNMP introduction....87
9.2 SNMP on Linux systems....88
9.3 Zenoss SNMP architecture....91
9.3.1 The zentrap daemon....91
9.4 Interpreting MIBs....93
9.4.1 zenmib example....94
9.4.2 A few comments on importing MIBs with Zenoss....99
9.5 The MIB Browser ZenPack....100
9.5.1 Modifying Zenoss Core 4.2 to make the MIB Browser ZenPack work....102
9.6 Mapping SNMP events....103
9.6.1 SNMP event mapping example....103
10 Event Triggers and Notifications....108
10.1 Zenoss prior to V4....108
10.2 Zenoss 4 architecture....109
10.3 Triggers....110
10.4 Notifications....111
10.4.1 email Notifications....113
10.4.2 Page Notifications....118
10.4.3 Command Notifications....118
10.4.4 TRAP Notifications....120
10.5 Notification Schedules....122
10.6 Using zenactiond.log....123
10.7 The effect of device Production State....125
11 Accessing events with the JSON API....126

11.1 Definitions.....126
11.2 Understanding the JSON API.....127
11.3 Using the JSON API.....130
11.3.1 Bash examples.....130
11.3.2 Python examples.....134
12 Conclusions.....139
13 Appendix A.....143
13.1 getevents.py.....143
13.2 zensendevent.....148
14 References.....152

1 Introduction

Zenoss is an Open Source, multi-function systems and network management tool. There is a free, Core offering (which has most things you need), and a chargeable offering, Zenoss Resource Manager, which has extra add-on goodies such as high availability configurations, distributed management servers, service management and event correlation; it also includes a support contract.

Zenoss offers configuration discovery, including layer 3 topology maps, availability monitoring, problem management and performance management. It is designed around the ITIL concept of a Configuration Management Database (CMDB), "the Zenoss Standard Model". Zenoss is built using the Python-based Zope web application server and uses the object-oriented Zope Object Database (ZODB) as the CMDB, used to store Python objects and their states. Zenoss 3 used ZEO, as a layer between Zope and the ZODB; in Zenoss 4 the ZODB data is stored in a MySQL database.

The relational MySQL database is also used to hold current and historical events. Performance data is held in Round Robin Database (RRD) files.

The default protocols for monitoring are typically "agentless" - the Simple Network Management protocol (SNMP), Windows Management Instrumentation (WMI) and collecting events from syslogs. It is also possible to monitor devices using telnet, ssh and to use Nagios plugins.

Zenoss provides documentation at http://community.zenoss.org/community/documentation. There is also a wealth of information on the Zenoss website in various forums, FAQs, and the Wiki. A useful book is available from PACKT Publishing, "Zenoss Core 3.x Network and System Monitoring" by Michael Badger, which provides much of the same information as the Zenoss Administration Guide but in a much clearer format with plenty of screenshots. Although this is a Zenoss 3 text, it still provides good basic information.

This paper is an attempt to expand on the event information in the Zenoss Core 4 Administration Guide by drawing on my own experience and the collected wisdom of several Zenoss employees and contributors from the community.

2 Zenoss event architecture

2.1 Event Console

When an event arrives at Zenoss, it is parsed, associated with an event classification and then typically (but not always), it is inserted into the **event_summary** table of the **zenoss_zep** database. Events can then be viewed by users using the Event Console of the Zenoss Graphical User Interface (GUI).

There are a number ways to access the Event Console. The main Event Console is reached from the top *EVENTS -> Event Console* menu. The default is to show events with a severity of Info or higher, sorted first by severity and then by time (most recent first). Events are assigned different **severities**:

Name	Number	Colour
Critical	5	Red
Error	4	Orange
Warning	3	Yellow
Info	2	Blue
Debug	1	Grey
Cleared	0	Green

All events also have an **eventState** field. Zenoss 3 eventState had three possible values - New, Acknowledged and Suppressed. Zenoss 4 has enhanced these definitions so we now have:

Name	Number	Description
New	0	New event - no previous "similar" event
Acknowledged	1	Acknowledged by user or rule
Suppressed	2	Typically from beyond a single point of failure
Closed	3	Closed by a user
Cleared	4	Closed by a rule
Dropped	5	Discarded - not saved in the database
Aged	6	Auto-closed due to age / severity

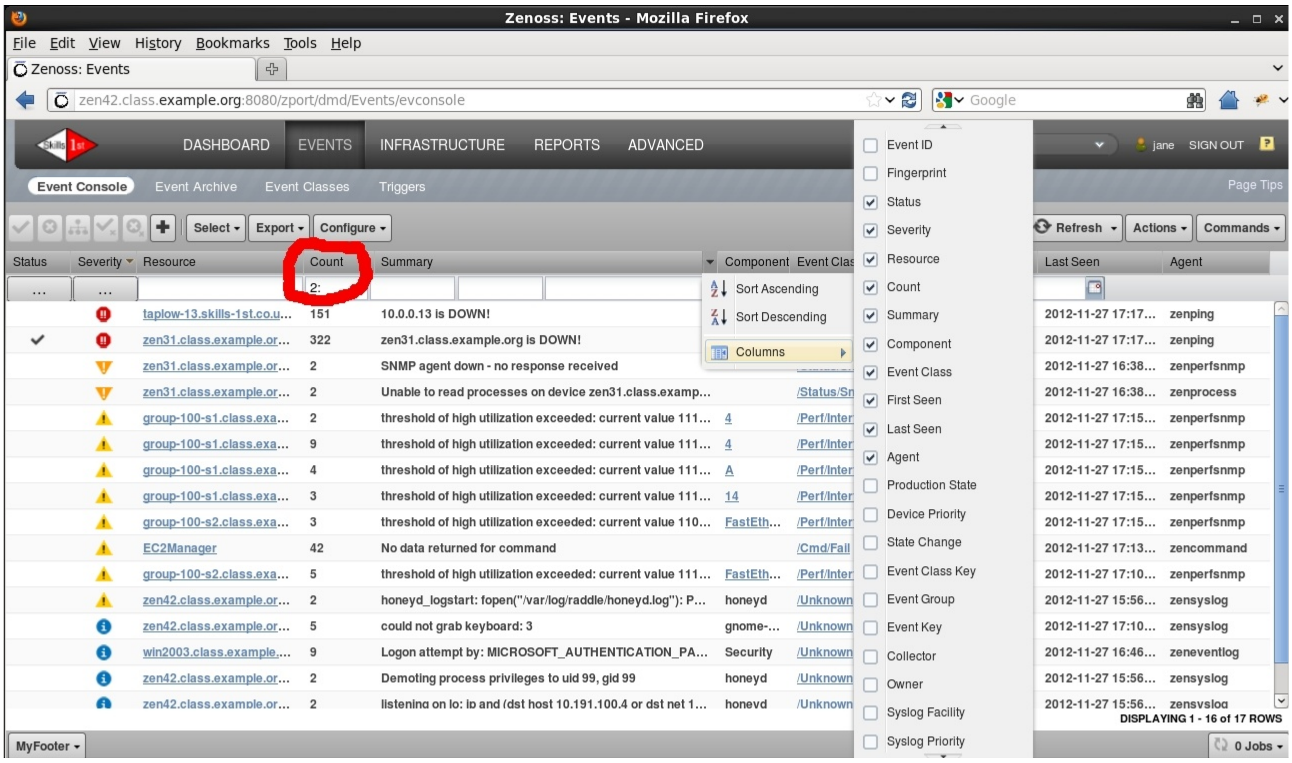
Note that Closed, Cleared and Aged events all have the same status icon in the Event Console.

By default, New and Acknowledged events are shown in the Event Console. Any event which has been Acknowledged has a tick in its status column. A Suppressed event is not shown by default but can be filtered in if desired; it has a "snowflake" icon. Zenoss builds an internal topology of the network it is managing (using nmap). If an event is received for a device that the topology map knows is unreachable, the event is automatically suppressed. Thus Zenoss has a built-in mechanism for pinpointing failure devices and suppressing the flood of events from behind such failure points.

Events can be sorted by clicking on a desired column header; clicking again sorts in the reverse order. To change the order of columns, simply drag a column header.

There is a filter box above each column header to help select relevant events. Most filters are a match for a partial text string (you don't need to supply wild cards). Date fields provide a calendar icon to select an earliest date. The count field permits you to enter a range, for example to show events with count > 10, use *10:* (if you type something illegal in the count filter it will supply help for the required syntax).

To select fields to display, hover the mouse at the end of a header to see the down-arrow for sorting; the third option on the dropdown menu is to configure the fields to display.



- [ ] Event ID
- [ ] Fingerprint
- [x] Status
- [x] Severity
- [x] Resource
- [x] Count
- [x] Summary
- [x] Component
- [x] Event Class
- [x] First Seen
- [x] Last Seen
- [x] Agent
- [ ] Production State
- [ ] Device Priority
- [ ] State Change
- [ ] Event Class Key
- [ ] Event Group
- [ ] Event Key
- [ ] Collector
- [ ] Owner
- [ ] Syslog Facility
- [ ] Syslog Priority

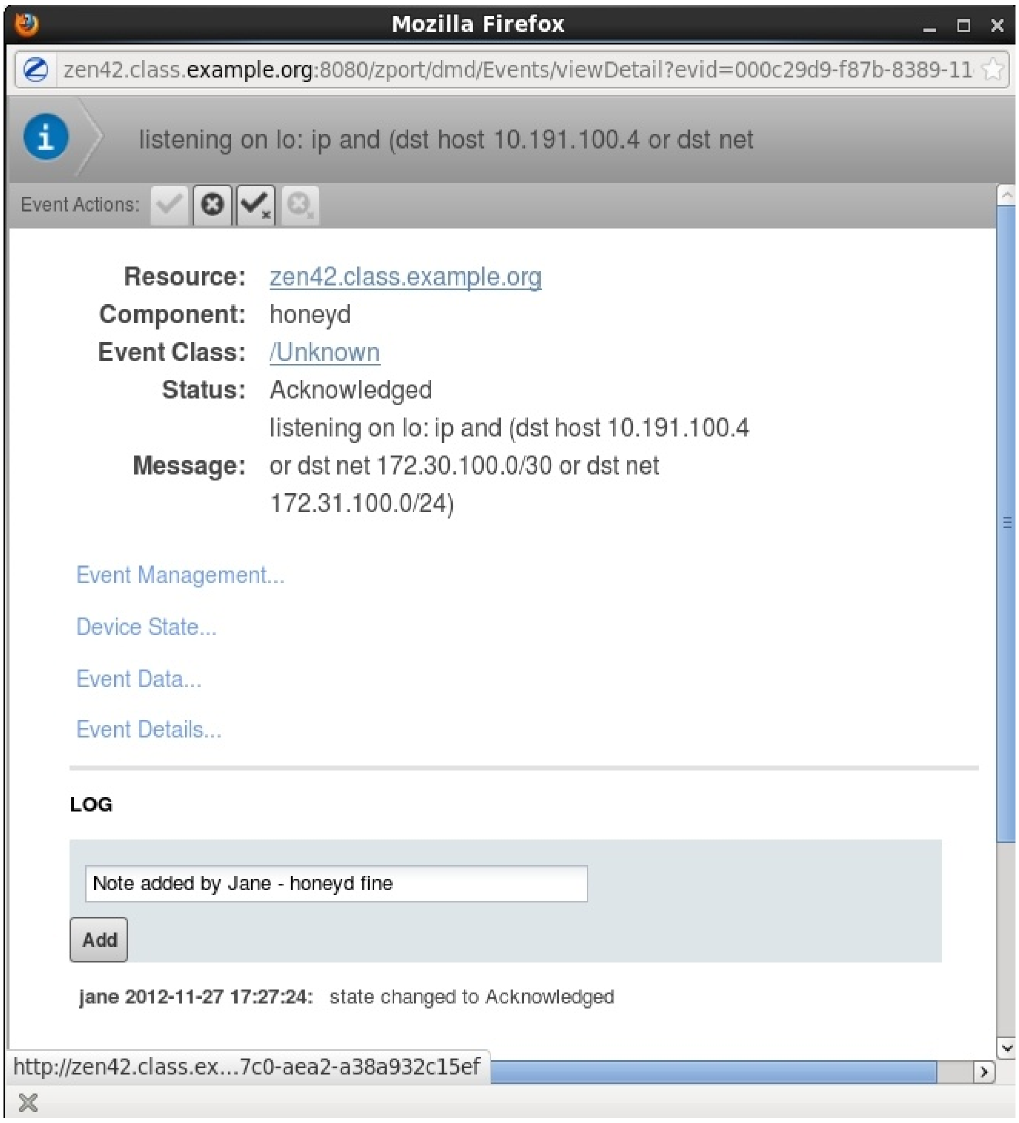
Status	Severity	Resource	Count	Summary	Component	Event Clas	Last Seen	Agent
...	...		2:					
	❗	taplow-13.skills-1st.co.u...	151	10.0.0.13 is DOWN!			2012-11-27 17:17...	zenping
✓	❗	zen31.class.example.or...	322	zen31.class.example.org is DOWN!			2012-11-27 17:17...	zenping
	▼	zen31.class.example.or...	2	SNMP agent down - no response received			2012-11-27 16:38...	zenperfsnmp
	▼	zen31.class.example.or...	2	Unable to read processes on device zen31.class.examp...		/Status/Sn	2012-11-27 16:38...	zenprocess
	⚠	group-100-s1.class.exa...	2	threshold of high utilization exceeded: current value 111...	4	/Perf/Inter	2012-11-27 17:15...	zenperfsnmp
	⚠	group-100-s1.class.exa...	9	threshold of high utilization exceeded: current value 111...	4	/Perf/Inter	2012-11-27 17:15...	zenperfsnmp
	⚠	group-100-s1.class.exa...	4	threshold of high utilization exceeded: current value 111...	A	/Perf/Inter	2012-11-27 17:15...	zenperfsnmp
	⚠	group-100-s1.class.exa...	3	threshold of high utilization exceeded: current value 111...	14	/Perf/Inter	2012-11-27 17:15...	zenperfsnmp
	⚠	group-100-s2.class.exa...	3	threshold of high utilization exceeded: current value 110...	FastEth...	/Perf/Inter	2012-11-27 17:15...	zenperfsnmp
	⚠	EC2Manager	42	No data returned for command		/Cmd/Fail	2012-11-27 17:13...	zencommand
	⚠	group-100-s2.class.exa...	5	threshold of high utilization exceeded: current value 111...	FastEth...	/Perf/Inter	2012-11-27 17:10...	zenperfsnmp
	⚠	zen42.class.example.or...	2	honeyd_logstart: fopen("/var/log/raddle/honeyd.log"): P...	honeyd	/Unknown	2012-11-27 15:56...	zensyslog
	ℹ	zen42.class.example.or...	5	could not grab keyboard: 3	gnome-...	/Unknown	2012-11-27 17:10...	zensyslog
	ℹ	win2003.class.example....	9	Logon attempt by: MICROSOFT_AUTHENTICATION_PA...	Security	/Unknown	2012-11-27 16:46...	zeneventlog
	ℹ	zen42.class.example.or...	2	Demoting process privileges to uid 99, gid 99	honeyd	/Unknown	2012-11-27 15:56...	zensyslog
	ℹ	zen42.class.example.or...	2	listening on lo: ip and (dst host 10.191.100.4 or dst net 1...	honeyd	/Unknown	2012-11-27 15:56...	zensyslog

DISPLAYING 1 - 16 of 17 ROWS

Figure 1: Zenoss Event Console

From the Event Console, one or more events can be selected by clicking on the line - be careful not to click something that is a link (like the device name or event class). The icons at the top-left can be used to Acknowledge, Close, Map to an Event Class, Unacknowledge or ReOpen. The "+" icon at the end of this row of icons can be used to generate test events.

Double-click an event to show the details of an event. This shows both standard fields and any user-defined fields organised under several groupings which can be expanded and contracted. Any Acknowledge, Close or ReOpen will be shown at the bottom, including who performed the action. Free-form notes can also be logged here.


Mozilla Firefox

zen42.class.example.org:8080/zport/dmd/Events/viewDetail?evid=000c29d9-f87b-8389-11

listening on lo: ip and (dst host 10.191.100.4 or dst net

Event Actions:

Resource: zen42.class.example.org
Component: honeyd
Event Class: /Unknown
Status: Acknowledged
Message: listening on lo: ip and (dst host 10.191.100.4 or dst net 172.30.100.0/30 or dst net 172.31.100.0/24)

Event Management...

Device State...

Event Data...

Event Details...

LOG

Note added by Jane - honeyd fine

Add

jane 2012-11-27 17:27:24: state changed to Acknowledged

http://zen42.class.ex...7c0-aea2-a38a932c15ef

Figure 2: Event details showing Acknowledgement and added note

The summary and message fields are free-form text fields. The summary field allows up to 255 characters; the message field allows up to 4096 characters. These fields usually contain similar data. For details of other fields, see section 7.1.2 of the Zenoss Core 4 Administration guide.

By default, the Event Console is refreshed every minute. The dropdown beside the Refresh button allows you to change the interval or to refresh manually.

Event Consoles are also available at various places in the GUI which have filters already applied:

- From a **device's** detail page, select *Events* in the lefthand menu
- For a **device class,** click the *DETAILS* link and then *Events* in the lefthand menu
- For a **Location**, **Group** or **System**, click the *DETAILS* link and then *Events* in the lefthand menu
- From an **Event Class**, select *Events* in the lefthand menu

Prior to V4, Zenoss events were either “Open” or “Closed”. Open events were stored in the MySQL **events** database in the **status** table. When an event was closed, it was moved to the **history** table of the events database.

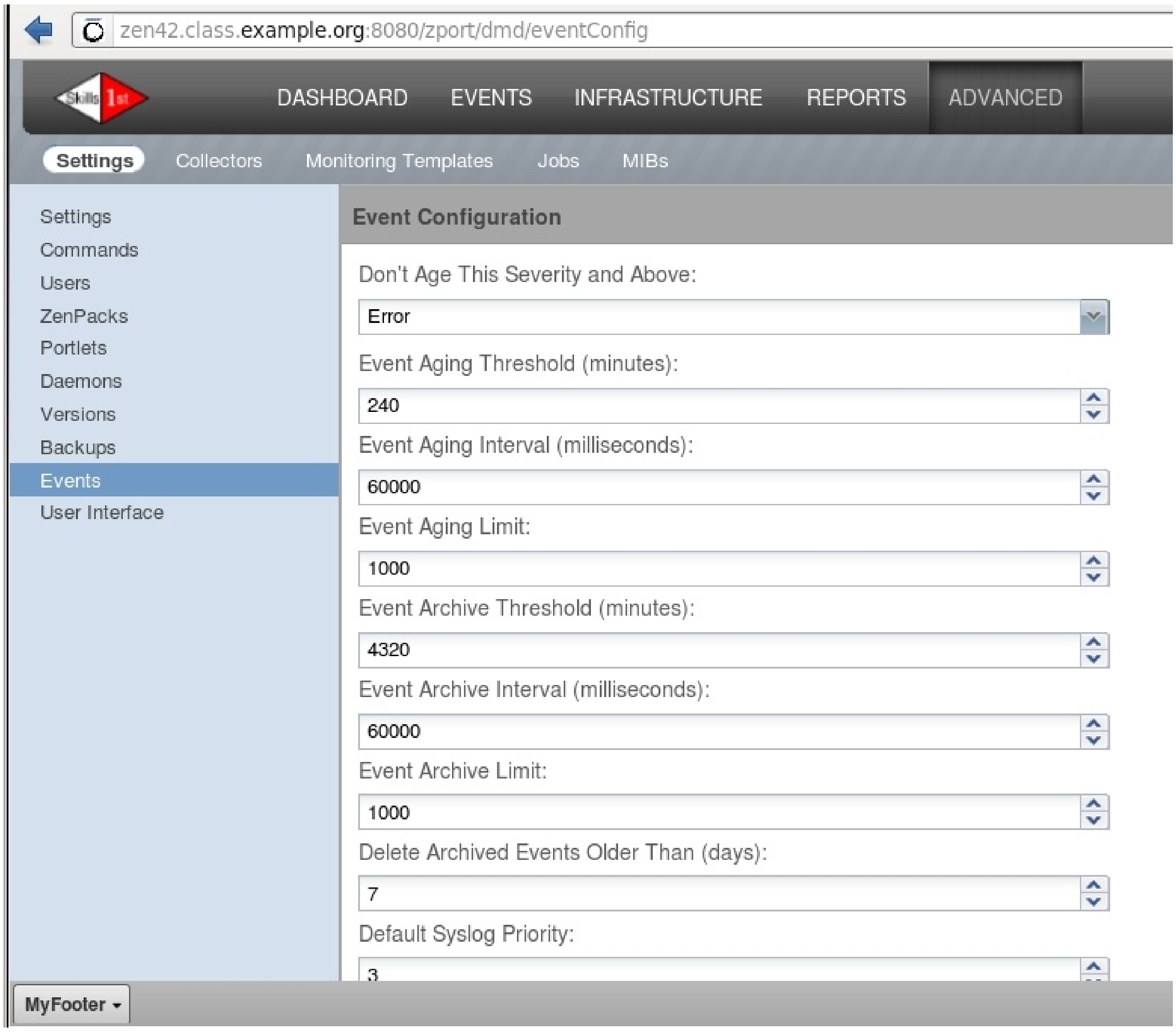
With Zenoss 4 there is a significant change. The MySQL database for events is called **zenoss_zep** and it has far more tables, including **event_summary** and **event_archive**. Open events will be stored in the events_summary table. **Be aware** that the events_summary table will also hold closed, cleared and aged events - this catches out many people migrating from older versions of Zenoss to Zenoss 4. Check the Status filter in the Event Console to show Closed, Cleared and Aged events (they all have the same status icon). Closed, Cleared and Aged events may be automatically moved to the event_archive table based on age (after 3 days, by default).

2.2 Event Manager settings

From the *ADVANCED -> Settings* menu, choose *Events* in the lefthand menu to setup various parameters that control the events subsystem, including how events are aged and finally purged.

Figure 3 on page 11 shows largely default settings. Events of severity Warning and below will be Aged after 240 minutes (4 hours). After 4320 minutes (3 days) events with status of Closed, Cleared or Aged will be Archived (moved to the events_archive table). After 7 days Archived events will be deleted entirely (note this last setting is 90 days by default and can result in a **very large database**).

See chapter 7 of the Zenoss Core 4 Administrators Guide for more information.



Event Configuration

Don't Age This Severity and Above: Error

Event Aging Threshold (minutes): 240

Event Aging Interval (milliseconds): 60000

Event Aging Limit: 1000

Event Archive Threshold (minutes): 4320

Event Archive Interval (milliseconds): 60000

Event Archive Limit: 1000

Delete Archived Events Older Than (days): 7

Default Syslog Priority: 3

Figure 3: Event Manager parameters for ageing and archiving

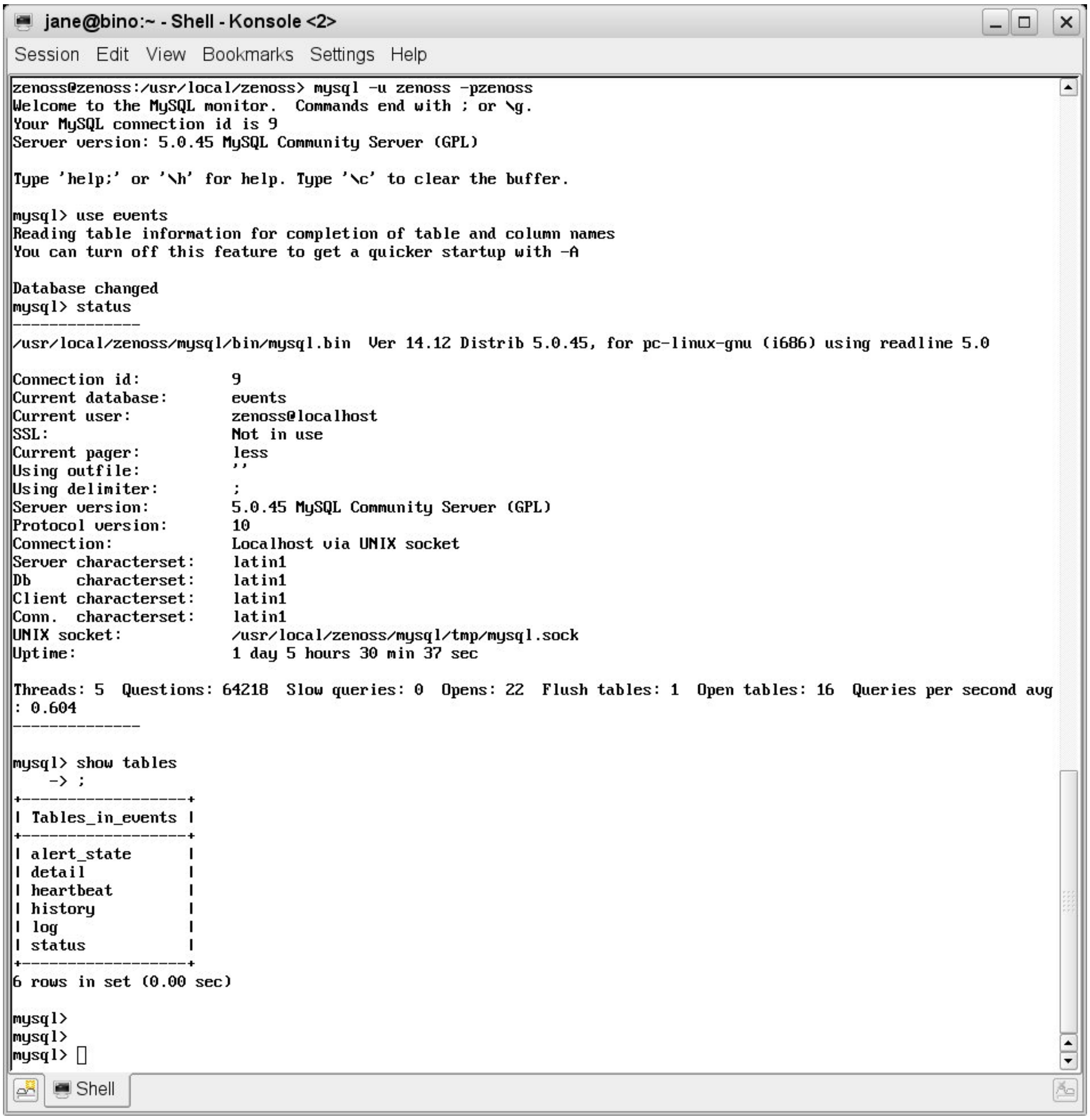
2.3 Event database tables

2.3.1 Zenoss 2.x and 3.x

The events architecture was the same for versions 2 and 3 and was relatively simple. Events were generated from “somewhere”. The **zenhub** daemon processed them and usually then saved them either in the **status** table of the MySQL events database or could send them to the **history** table.

The database fields of the status and history tables matched the details seen in an Event Console and if you wrote rules and transforms to process events, they were based on these same field names.

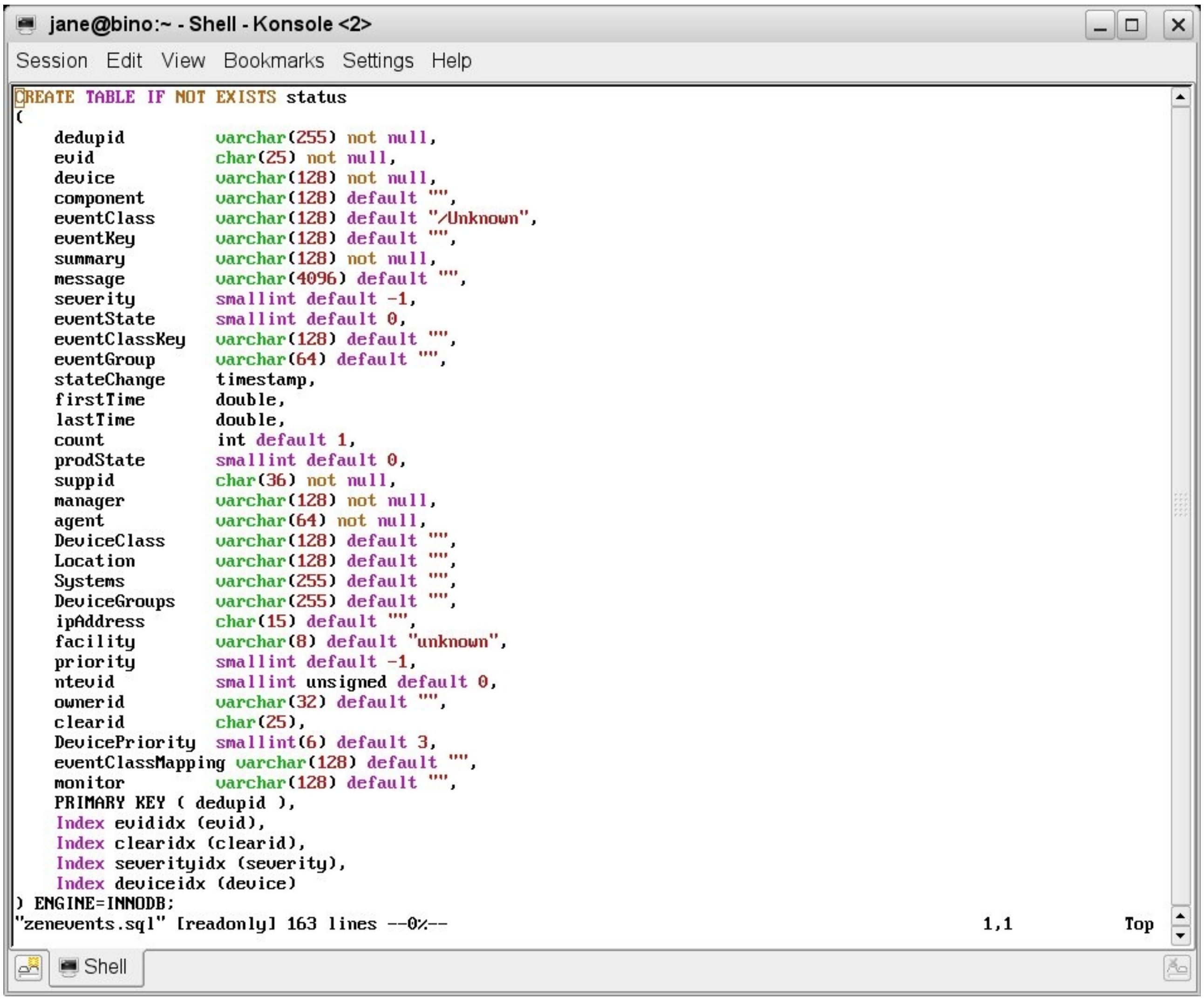
The events database is created automatically when Zenoss is installed and can typically be accessed by the *zenoss* user with a password of *zenoss* - see Figure 4.


jane@bino:~ - Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell

```
zenoss@zenoss:/usr/local/zenoss> mysql -u zenoss -pzenoss
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 5.0.45 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use events
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> status
--------------
/usr/local/zenoss/mysql/bin/mysql.bin  Ver 14.12 Distrib 5.0.45, for pc-linux-gnu (i686) using readline 5.0

Connection id:          9
Current database:       events
Current user:           zenoss@localhost
SSL:                    Not in use
Current pager:          less
Using outfile:          ''
Using delimiter:        ;
Server version:         5.0.45 MySQL Community Server (GPL)
Protocol version:       10
Connection:             Localhost via UNIX socket
Server characterset:    latin1
Db     characterset:    latin1
Client characterset:    latin1
Conn.  characterset:    latin1
UNIX socket:            /usr/local/zenoss/mysql/tmp/mysql.sock
Uptime:                 1 day 5 hours 30 min 37 sec

Threads: 5  Questions: 64218  Slow queries: 0  Opens: 22  Flush tables: 1  Open tables: 16  Queries per second avg
: 0.604
--------------

mysql> show tables
    -> ;
+------------------+
| Tables_in_events |
+------------------+
| alert_state      |
| detail           |
| heartbeat        |
| history          |
| log              |
| status           |
+------------------+
6 rows in set (0.00 sec)

mysql>
mysql>
mysql> 
```

Figure 4: Zenoss events database prior to Zenoss 4

The format of each of these tables and the valid fields for a Zenoss event can be seen by examining the Zenoss database setup file in *$ZENHOME/Products/ZenEvents/db/zenevents.sql* , where **$ZENHOME** will be */opt/zenoss* for a Core 4.2 Zenoss on RedHat / CentOS (the only currently supported platform).

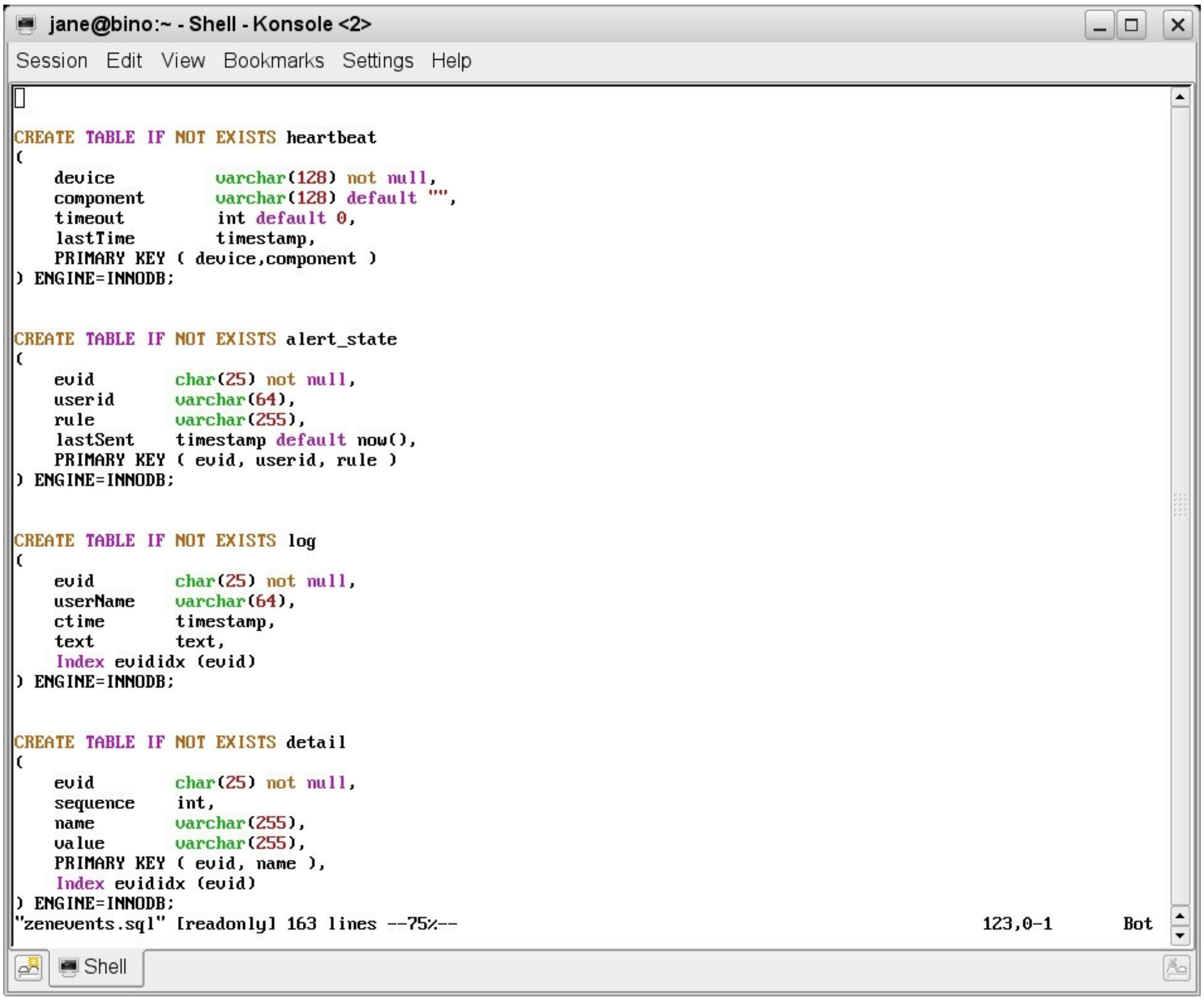

jane@bino:~ - Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell

```
CREATE TABLE IF NOT EXISTS status
(
    dedupid         varchar(255) not null,
    evid            char(25) not null,
    device          varchar(128) not null,
    component       varchar(128) default "",
    eventClass      varchar(128) default "/Unknown",
    eventKey        varchar(128) default "",
    summary         varchar(128) not null,
    message         varchar(4096) default "",
    severity        smallint default -1,
    eventState      smallint default 0,
    eventClassKey   varchar(128) default "",
    eventGroup      varchar(64) default "",
    stateChange     timestamp,
    firstTime       double,
    lastTime        double,
    count           int default 1,
    prodState       smallint default 0,
    suppid          char(36) not null,
    manager         varchar(128) not null,
    agent           varchar(64) not null,
    DeviceClass     varchar(128) default "",
    Location        varchar(128) default "",
    Systems         varchar(255) default "",
    DeviceGroups    varchar(255) default "",
    ipAddress       char(15) default "",
    facility        varchar(8) default "unknown",
    priority        smallint default -1,
    ntevid          smallint unsigned default 0,
    ownerid         varchar(32) default "",
    clearid         char(25),
    DevicePriority  smallint(6) default 3,
    eventClassMapping varchar(128) default "",
    monitor         varchar(128) default "",
    PRIMARY KEY ( dedupid ),
    Index evididx (evid),
    Index clearidx (clearid),
    Index severityidx (severity),
    Index deviceidx (device)
) ENGINE=INNODB;
"zenevents.sql" [readonly] 163 lines --0%--                    1,1           Top
```

Figure 5: Definition of status event fields in zenevents.sql prior to Zenoss 4

zenevents.sql also defines the **history** table in a similar fashion.

A further four tables are defined for **heartbeat, alert_state, log** and **detail** . The detail table can be used to extend the default event fields to include any information that the Zenoss administrator requires for an event.


jane@bino:~ - Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell

```
CREATE TABLE IF NOT EXISTS heartbeat
(
    device          varchar(128) not null,
    component       varchar(128) default "",
    timeout         int default 0,
    lastTime        timestamp,
    PRIMARY KEY ( device,component )
) ENGINE=INNODB;

CREATE TABLE IF NOT EXISTS alert_state
(
    evid        char(25) not null,
    userid      varchar(64),
    rule        varchar(255),
    lastSent    timestamp default now(),
    PRIMARY KEY ( evid, userid, rule )
) ENGINE=INNODB;

CREATE TABLE IF NOT EXISTS log
(
    evid        char(25) not null,
    userName    varchar(64),
    ctime       timestamp,
    text        text,
    Index evididx (evid)
) ENGINE=INNODB;

CREATE TABLE IF NOT EXISTS detail
(
    evid        char(25) not null,
    sequence    int,
    name        varchar(255),
    value       varchar(255),
    PRIMARY KEY ( evid, name ),
    Index evididx (evid)
) ENGINE=INNODB;
"zenevents.sql" [readonly] 163 lines --75%--                                          123,0-1       Bot
```

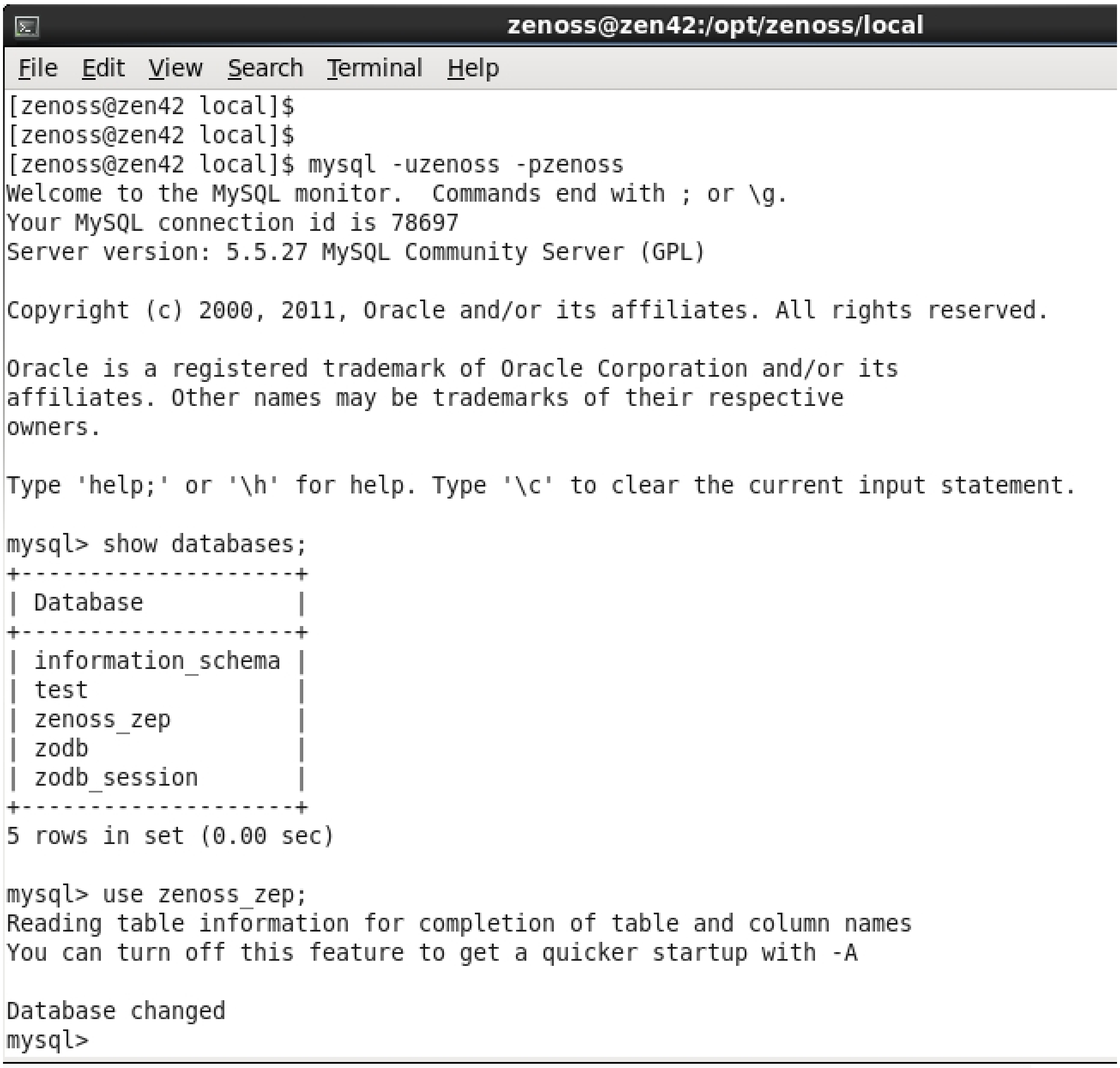
Figure 6: zenevents.sql showing heartbeat, alert_state, log and detail tables - zenoss 2 and 3 only

If you are using Zenoss prior to version 4, get the older version of this Zenoss Event Management paper from http://www.skills-1st.co.uk/papers/jane/zenoss_event_management_paper.pdf .

2.3.2 Zenoss 4

With Zenoss 4 events are still held in a MySQL database which is now called **zenoss_zep** and it is created when Zenoss is installed. As with earlier versions, the *zenoss* user can access this database with a password of *zenoss*.

Note that with Zenoss 4.2.3, if installed with the core-autodeploy script, then the password for the MySQL *zenoss* user is changed to a robust, random password that is then saved in *$ZENHOME/etc/global.conf*. Permissions for *$ZENHOME/etc* and its contents are all set to full access for the *zenoss* user and no access for anyone else.

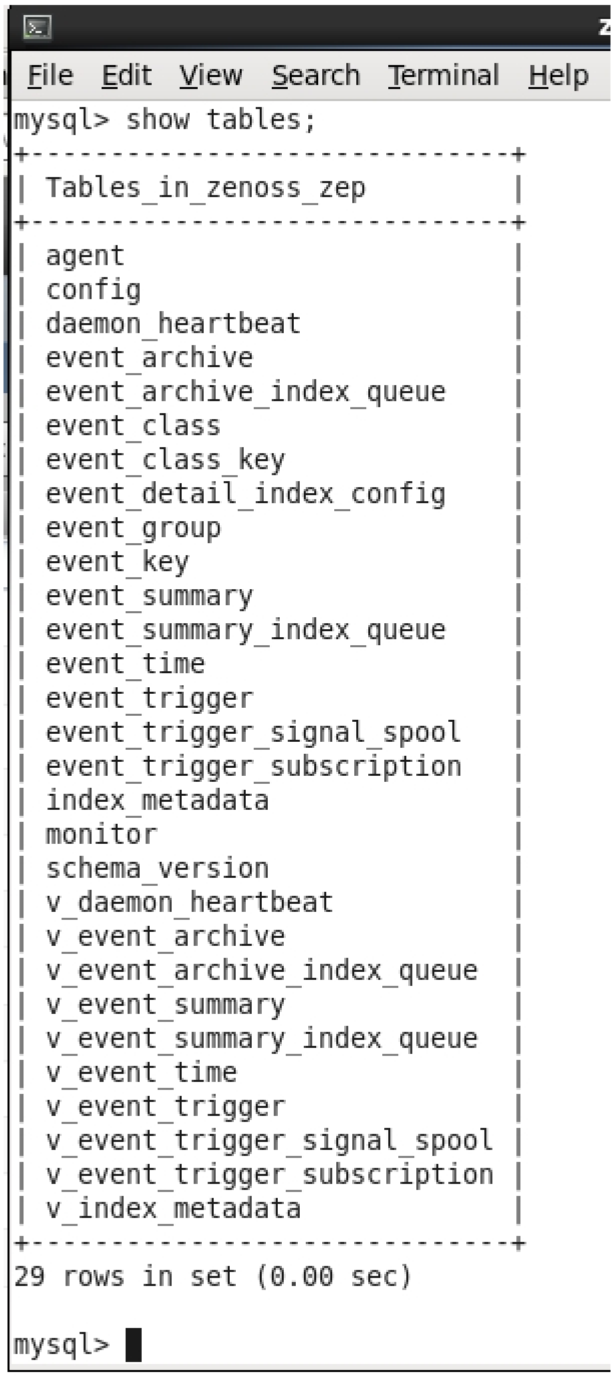

zenoss@zen42:/opt/zenoss/local
File Edit View Search Terminal Help

```
[zenoss@zen42 local]$
[zenoss@zen42 local]$
[zenoss@zen42 local]$ mysql -uzenoss -pzenoss
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 78697
Server version: 5.5.27 MySQL Community Server (GPL)

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| test               |
| zenoss_zep         |
| zodb               |
| zodb_session       |
+--------------------+
5 rows in set (0.00 sec)

mysql> use zenoss_zep;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql>
```

Figure 7: Accessing MySQL databases with Zenoss 4

In passing, note that in addition to the zenoss_zep database, their is also a **zodb** and a **zodb_session** database. The Zope database (ZODB) that stores all the objects (devices, device classes, processes, networks, etc) is now in MySQL.

Examining the tables of the zenoss_zep database is where things diverge significantly from previous versions.



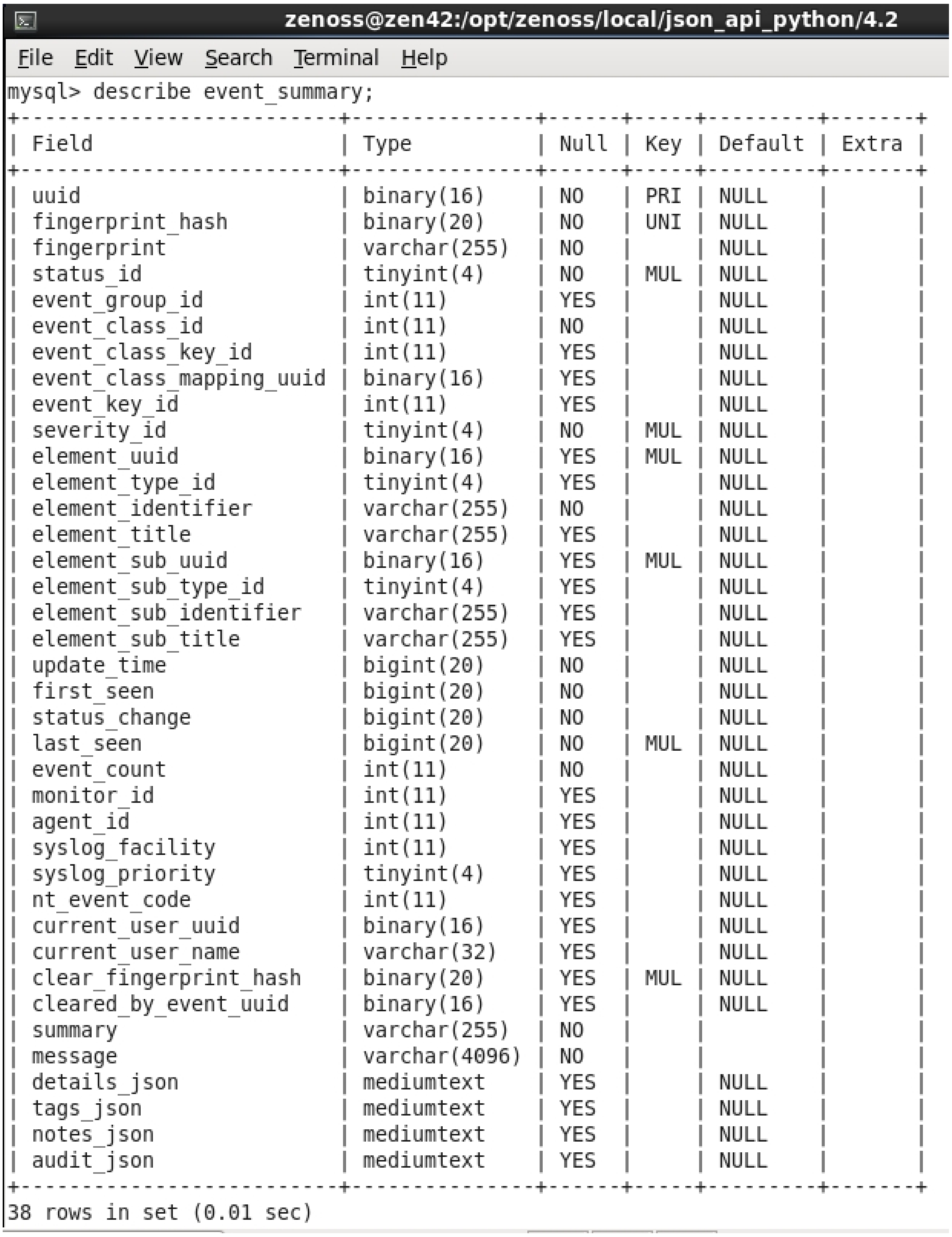
```
mysql> show tables;
+-----------------------------+
| Tables_in_zenoss_zep        |
+-----------------------------+
| agent                       |
| config                      |
| daemon_heartbeat            |
| event_archive               |
| event_archive_index_queue   |
| event_class                 |
| event_class_key             |
| event_detail_index_config   |
| event_group                 |
| event_key                   |
| event_summary               |
| event_summary_index_queue   |
| event_time                  |
| event_trigger               |
| event_trigger_signal_spool  |
| event_trigger_subscription  |
| index_metadata              |
| monitor                     |
| schema_version              |
| v_daemon_heartbeat          |
| v_event_archive             |
| v_event_archive_index_queue |
| v_event_summary             |
| v_event_summary_index_queue |
| v_event_time                |
| v_event_trigger             |
| v_event_trigger_signal_spool |
| v_event_trigger_subscription |
| v_index_metadata            |
+-----------------------------+
29 rows in set (0.00 sec)

mysql> 
```

Figure 8: Tables in the Zenoss 4 zenoss_zep database

The main tables are now **event_summary** and **event_archive** but the structure is more complicated. Some of the data is held in separate tables with pointers to them from the main tables. These include:

- agent
- event_class
- event_class_key
- event_group
- event_key
- monitor

The details of the event_summary table is shown below. The event archive table is very similar with just the two fingerprint_hash fields omitted.



zenoss@zen42:/opt/zenoss/local/json_api_python/4.2
File Edit View Search Terminal Help

```
mysql> describe event_summary;
+--------------------------+---------------+------+-----+---------+-------+
| Field                    | Type          | Null | Key | Default | Extra |
+--------------------------+---------------+------+-----+---------+-------+
| uuid                     | binary(16)    | NO   | PRI | NULL    |       |
| fingerprint_hash         | binary(20)    | NO   | UNI | NULL    |       |
| fingerprint              | varchar(255)  | NO   |     | NULL    |       |
| status_id                | tinyint(4)    | NO   | MUL | NULL    |       |
| event_group_id           | int(11)       | YES  |     | NULL    |       |
| event_class_id           | int(11)       | NO   |     | NULL    |       |
| event_class_key_id       | int(11)       | YES  |     | NULL    |       |
| event_class_mapping_uuid | binary(16)    | YES  |     | NULL    |       |
| event_key_id             | int(11)       | YES  |     | NULL    |       |
| severity_id              | tinyint(4)    | NO   | MUL | NULL    |       |
| element_uuid             | binary(16)    | YES  | MUL | NULL    |       |
| element_type_id          | tinyint(4)    | YES  |     | NULL    |       |
| element_identifier       | varchar(255)  | NO   |     | NULL    |       |
| element_title            | varchar(255)  | YES  |     | NULL    |       |
| element_sub_uuid         | binary(16)    | YES  | MUL | NULL    |       |
| element_sub_type_id      | tinyint(4)    | YES  |     | NULL    |       |
| element_sub_identifier   | varchar(255)  | YES  |     | NULL    |       |
| element_sub_title        | varchar(255)  | YES  |     | NULL    |       |
| update_time              | bigint(20)    | NO   |     | NULL    |       |
| first_seen               | bigint(20)    | NO   |     | NULL    |       |
| status_change            | bigint(20)    | NO   |     | NULL    |       |
| last_seen                | bigint(20)    | NO   | MUL | NULL    |       |
| event_count              | int(11)       | NO   |     | NULL    |       |
| monitor_id               | int(11)       | YES  |     | NULL    |       |
| agent_id                 | int(11)       | YES  |     | NULL    |       |
| syslog_facility          | int(11)       | YES  |     | NULL    |       |
| syslog_priority          | tinyint(4)    | YES  |     | NULL    |       |
| nt_event_code            | int(11)       | YES  |     | NULL    |       |
| current_user_uuid        | binary(16)    | YES  |     | NULL    |       |
| current_user_name        | varchar(32)   | YES  |     | NULL    |       |
| clear_fingerprint_hash   | binary(20)    | YES  | MUL | NULL    |       |
| cleared_by_event_uuid    | binary(16)    | YES  |     | NULL    |       |
| summary                  | varchar(255)  | NO   |     |         |       |
| message                  | varchar(4096) | NO   |     |         |       |
| details_json             | mediumtext    | YES  |     | NULL    |       |
| tags_json                | mediumtext    | YES  |     | NULL    |       |
| notes_json               | mediumtext    | YES  |     | NULL    |       |
| audit_json               | mediumtext    | YES  |     | NULL    |       |
+--------------------------+---------------+------+-----+---------+-------+
38 rows in set (0.01 sec)
```

Figure 9: Fields in the event_summary table in Zenoss 4

The eagle-eyed will also spot that some of the field names have changed from those in Figure 5. eventClass in the old version becomes event_class in V4; firstTime in Figure 5 becomes first_seen in the later version - and there are a number of other similar, subtle changes.

As mentioned above, some of the data is held in separate tables so agent_id, event_class_id, event_class_key_id, event_group_id, event_key_id and monitor_key are links to separate tables with the corresponding data.

Some data has changed fairly subtly:

Old	New
evid	uuid
eventState	status_id
eventClassMapping	event_class_mapping_uuid
severity	severity_id
stateChange	status_change
firstTime	first_seen
lastTime	last_seen
count	event_count
facility	syslog_facility
priority	syslog_priority
ntevid	nt_event_code
ownerid	current_user_uuid / current_user_name
clearid	clear_fingerprint_hash / cleared_by_event_uuid

All references to the device have changed significantly. **device** is replaced by the four fields, **element_uuid, element_type_id, element identifier** and **element_title** whilst the **component** field is replaced by **element_sub_uuid, element_sub_type_id, element_sub_identifier** and **element_sub_title**.

dedupid has become **fingerprint** and **fingerprint_hash**.

Other fields with device context such as **prodState, DeviceClass, Location, Systems, DeviceGroups, ipAddress, monitor** and **DevicePriority** will now be found from the **tags_json** field; they are also available in the event details.

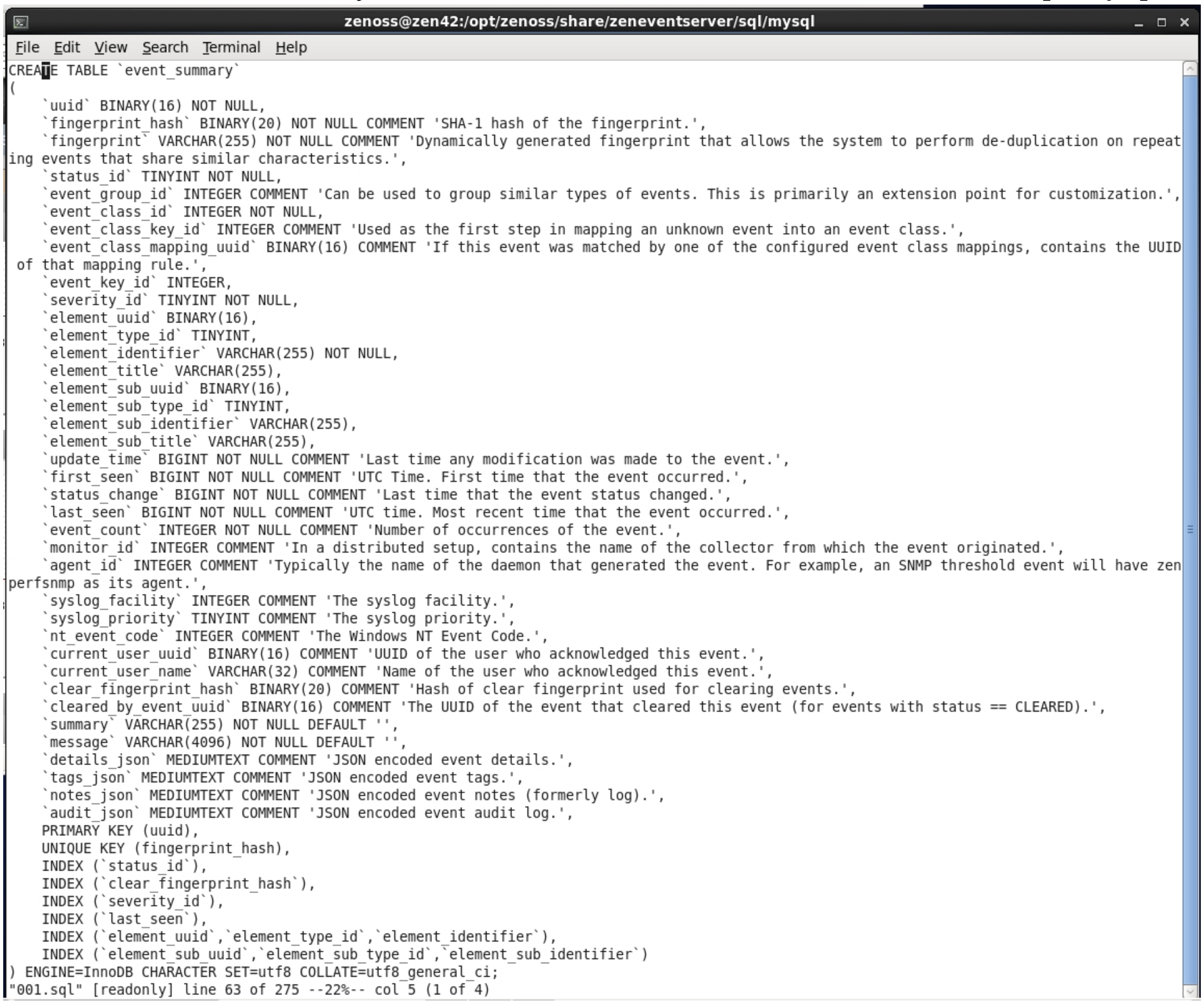
Prior to Zenoss 4 there was a separate **log** table whose role is now taken by the **notes_json** field of the event_summary table.

Event details rather than being in a separate table, are now reached from **details_json**.

update_time has been added - the last time an event was updated.

suppid (which was never used) has disappeared in the Zenoss 4 schema. **manager** has also disappeared from Zenoss 4.

These tables are created by the files in *$ZENHOME/share/zeneventserver/sql/mysql*.



```
CREATE TABLE `event_summary`
(
    `uuid` BINARY(16) NOT NULL,
    `fingerprint_hash` BINARY(20) NOT NULL COMMENT 'SHA-1 hash of the fingerprint.',
    `fingerprint` VARCHAR(255) NOT NULL COMMENT 'Dynamically generated fingerprint that allows the system to perform de-duplication on repeat
ing events that share similar characteristics.',
    `status_id` TINYINT NOT NULL,
    `event_group_id` INTEGER COMMENT 'Can be used to group similar types of events. This is primarily an extension point for customization.',
    `event_class_id` INTEGER NOT NULL,
    `event_class_key_id` INTEGER COMMENT 'Used as the first step in mapping an unknown event into an event class.',
    `event_class_mapping_uuid` BINARY(16) COMMENT 'If this event was matched by one of the configured event class mappings, contains the UUID
 of that mapping rule.',
    `event_key_id` INTEGER,
    `severity_id` TINYINT NOT NULL,
    `element_uuid` BINARY(16),
    `element_type_id` TINYINT,
    `element_identifier` VARCHAR(255) NOT NULL,
    `element_title` VARCHAR(255),
    `element_sub_uuid` BINARY(16),
    `element_sub_type_id` TINYINT,
    `element_sub_identifier` VARCHAR(255),
    `element_sub_title` VARCHAR(255),
    `update_time` BIGINT NOT NULL COMMENT 'Last time any modification was made to the event.',
    `first_seen` BIGINT NOT NULL COMMENT 'UTC Time. First time that the event occurred.',
    `status_change` BIGINT NOT NULL COMMENT 'Last time that the event status changed.',
    `last_seen` BIGINT NOT NULL COMMENT 'UTC time. Most recent time that the event occurred.',
    `event_count` INTEGER NOT NULL COMMENT 'Number of occurrences of the event.',
    `monitor_id` INTEGER COMMENT 'In a distributed setup, contains the name of the collector from which the event originated.',
    `agent_id` INTEGER COMMENT 'Typically the name of the daemon that generated the event. For example, an SNMP threshold event will have zen
perfsnmp as its agent.',
    `syslog_facility` INTEGER COMMENT 'The syslog facility.',
    `syslog_priority` TINYINT COMMENT 'The syslog priority.',
    `nt_event_code` INTEGER COMMENT 'The Windows NT Event Code.',
    `current_user_uuid` BINARY(16) COMMENT 'UUID of the user who acknowledged this event.',
    `current_user_name` VARCHAR(32) COMMENT 'Name of the user who acknowledged this event.',
    `clear_fingerprint_hash` BINARY(20) COMMENT 'Hash of clear fingerprint used for clearing events.',
    `cleared_by_event_uuid` BINARY(16) COMMENT 'The UUID of the event that cleared this event (for events with status == CLEARED).',
    `summary` VARCHAR(255) NOT NULL DEFAULT '',
    `message` VARCHAR(4096) NOT NULL DEFAULT '',
    `details_json` MEDIUMTEXT COMMENT 'JSON encoded event details.',
    `tags_json` MEDIUMTEXT COMMENT 'JSON encoded event tags.',
    `notes_json` MEDIUMTEXT COMMENT 'JSON encoded event notes (formerly log).',
    `audit_json` MEDIUMTEXT COMMENT 'JSON encoded event audit log.',
    PRIMARY KEY (uuid),
    UNIQUE KEY (fingerprint_hash),
    INDEX (`status_id`),
    INDEX (`clear_fingerprint_hash`),
    INDEX (`severity_id`),
    INDEX (`last_seen`),
    INDEX (`element_uuid`,`element_type_id`,`element_identifier`),
    INDEX (`element_sub_uuid`,`element_sub_type_id`,`element_sub_identifier`)
) ENGINE=InnoDB CHARACTER SET=utf8 COLLATE=utf8_general_ci;
"001.sql" [readonly] line 63 of 275 --22%-- col 5 (1 of 4)
```

Figure 10: Part of the 001.sql file that defines MySQL tables in the zenoss_zep database for Zenoss 4

Some of these event fields are particularly pertinent depending on how the event was generated:

- Syslog events populate the **facility** and **priority** fields
- Windows events populate the **ntevid** field
- SNMP TRAPs populate at least **community** and **oid** fields in the event detail. They also use the event detail to provide any variables passed by an SNMP TRAP.
- The **agent** field denotes which Zenoss daemon generated or processed the incoming event; for example, zentrap, zeneventlog, zenping .

Fundamentally Zenoss administrators should **not** be accessing the zenoss_zep database directly. Zenoss have provided an internal event mapping so that, largely, administrators can continue to use the same event attribute names as have been used previously. This **event proxy** mapping will be discussed in more detail later. In general, this paper will use the old names unless explicitly stated otherwise.

If you do need to access event data in the database tables, perhaps for reporting on events, it is possible with the JSON API (also more on this later).

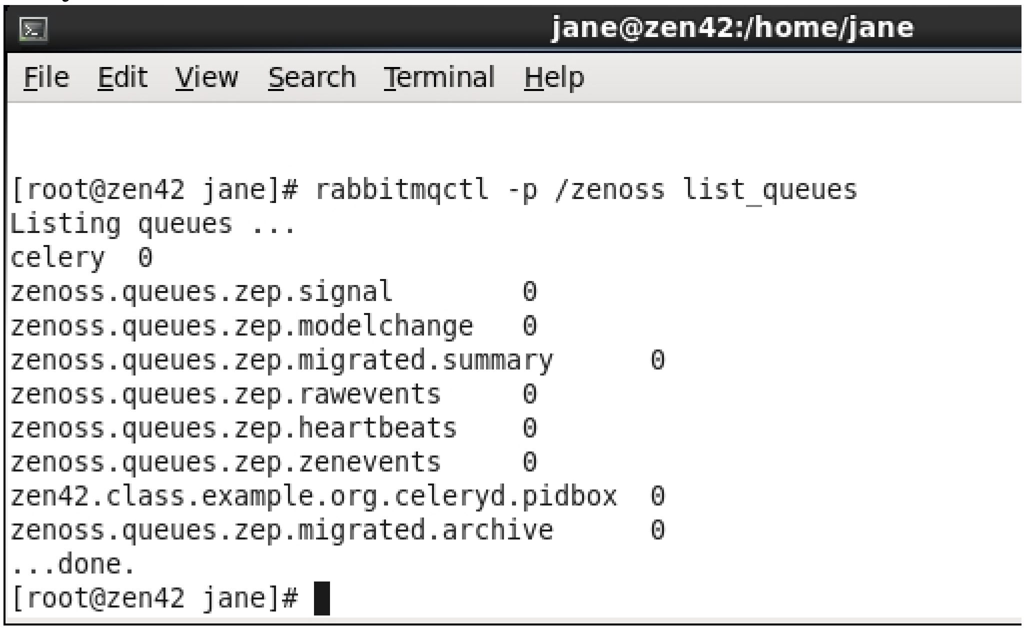
2.4 New event daemons

Prior to Zenoss 4 most of the work of processing an event was performed by the **zenhub** daemon which also has lots of other roles to fulfil. Event processing could become a severe bottleneck. Zenoss 4 has introduced several new subsystems and daemons to dramatically improve the throughput of event processing.

2.4.1 RabbitMQ

A Message Queueing architecture has been implemented to speed up processing and to offer an API so that Zenoss and other application providers can interact with events. It is also used by the new Job architecture. It uses the Advanced Message Queueing Protocol (AMQP) standard, and the open source RabbitMQ implementation in particular, for the event pipeline.

When Zenoss is installed the RabbitMQ subsystem is also installed and configured with a vhost of *zenoss*, user *zenoss*, password *zenoss*. The *rabbitmqctl* utility can provide information about the state of the MQ environment; note that *rabbitmqctl* commands must be run by the root user.



```
[root@zen42 jane]# rabbitmqctl -p /zenoss list_queues
Listing queues ...
celery  0
zenoss.queues.zep.signal        0
zenoss.queues.zep.modelchange   0
zenoss.queues.zep.migrated.summary      0
zenoss.queues.zep.rawevents     0
zenoss.queues.zep.heartbeats    0
zenoss.queues.zep.zenevents     0
zen42.class.example.org.celeryd.pidbox  0
zenoss.queues.zep.migrated.archive      0
...done.
[root@zen42 jane]#
```

Figure 11: Using the rabbitmqctl utility to show queues for the /zenoss vhost

An easy way to see queues building up is to temporarily stop zeneventd and the rawevents queue will then build rapidly.

rabbitmqctl on its own or with insufficient arguments provides the usage help. *rabbitmqctl report* gives a good overall view of the subsystem.

If the Zenoss server is renamed then you must clear and rebuild queues before the zenhub and zenjobs daemons will restart. To resolve this, issue the following commands as the root user (although any data queued at restart time will be lost):

```
export VHOST="/zenoss"
export USER="zenoss"
export PASS="zenoss"
rabbitmqctl stop_app
rabbitmqctl reset
rabbitmqctl start_app
rabbitmqctl add_vhost "$VHOST"
rabbitmqctl add_user "$USER" "$PASS"
rabbitmqctl set_permissions -p "$VHOST" "$USER" '.*' '.*' '.*'
```

See section 14.8 of the Zenoss Core 4 Administrators Guide for this information.

Note that with Zenoss Core 4.2.3 installed using the auto-deploy script, or if the **secure_zenoss.sh** script has been run standalone, then the password in the third line above will have been changed. Examine *$ZENHOME/etc/global.conf* for the **amqppassword** and substitue that value, rather than using *zenoss* as the password.

Provided the RabbitMQ subsystem is running, any missing queue will automatically be recreated when Zenoss is restarted.

To simply have the queues recreated, start as the zenoss user:

```
zenoss stop
su              (to become root user)
rabbitmqctl delete_vhost /zenoss
rabbitmqctl add_vhost /zenoss
rabbitmqctl add_user zenoss zenoss # might create an error
zenoss rabbitmqctl set_permissions -p /zenoss zenoss '.*' '.*' '.*'
rabbitmqctl list_vhosts                     (should have zenoss again)
rabbitmqctl -p /zenoss list_queues           (should be none)
exit                 (back to zenoss user)
zenoss start
su
rabbitmqctl -p /zenoss list_queues           (should be several)
```

There is a further script available at gist, written by cluther, to reset RabbitMQ - https://gist.github.com/4192854 .

Two utilities are available for the zenoss user to get RabbitMQ information:

```
zenqdump <queue name>
```

dumps the events in a queue, converting the binary "blobs" (which is how the events are actually stored) into human-readable text.

Note that the zenqdump utility has parameters for user and password for authentication, that default to **zenoss** / **zenoss** (you can find this code in *$ZENHOME/lib/python/zenoss/protocols/amqpconfig.py*). In Zenoss 4.2.3, passwords are likely to have been improved on installation so the simple command shown above will fail. Examine *$ZENHOME/etc/global.conf* for the parameters **amqpuser** and **amqppassword** and supply those values. For example:

```
zenqdump -u zenoss -p uy+680bEubHgdPow8Tfh zenoss.queues.zep.rawevents
```

The zenq utility has three different options to manage a queue:

```
zenq count <queue name>
zenq purge <queue name>
zenq delete <queue name>
```

The count parameter gives a continual output of timestamp and queue length.

The purge parameter purges events from a queue. This command is safe when Zenoss is running.

The delete parameter deletes the queue and should **not** be used when Zenoss is running.

zenq does not have authentication parameters.

2.4.2 zeneventserver

A new Java daemon, zeneventserver (also known as **zep**), has been created. Its role is to present events to the user interface and other clients, and to manage the flow of data between the RabbitMQ queues and the MySQL database. Data is presented to clients via JSON calls.

2.4.3 zeneventd

zeneventd is a new Python daemon whose responsibility is to take data from the incoming raw event queue, classify it (if the event does not already have a class), add device context and event context, and perform any transforms. It then outputs to the zenevents queue so that the zeneventserver daemon can manage its progress to the MySQL database, to the user interface and for alerting action.

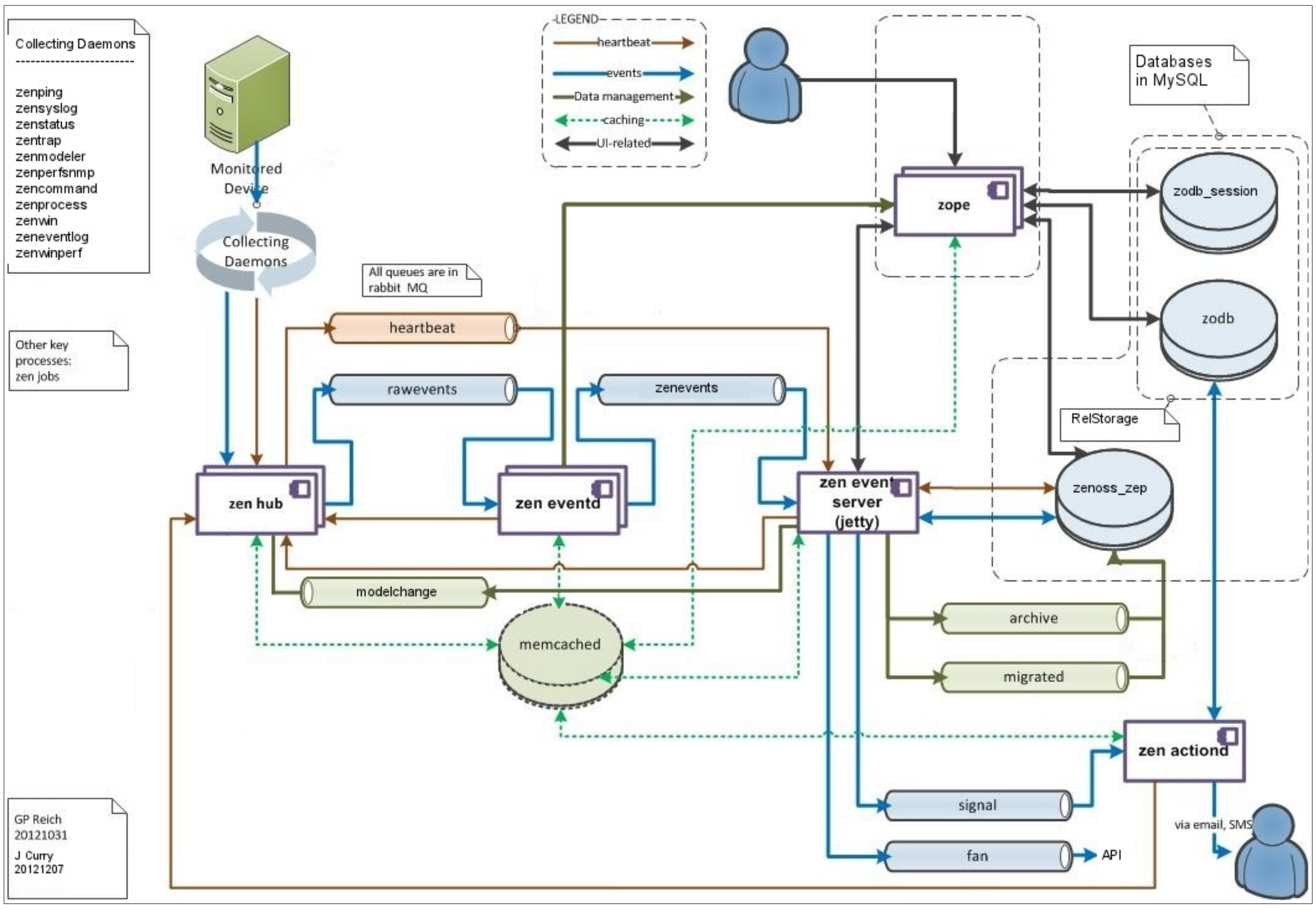


Figure 12: Zenoss 4 event architecture

2.4.4 zenactiond

zenactiond has been completely rewritten for Zenoss 4. It is responsible for executing actions associated with notifications such as paging, email, executing background commands and raising notification TRAPs. zenactiond will periodically inspect the **signal** queue for signal messages, dump them in to its share of memcached and subsequently act on the messages as instructed in the associated notification.

2.4.5 memcached

Prior to Zenoss 4 each of the daemons had its own cache. This could be a wasteful allocation of memory. With Zenoss 4, a memcached subsystem is introduced which provides shared L2 memory cache for all daemons, offering much better performance.

memcached is configured in */etc/sysconfig/memcached*. The default is to configure 64Mb for memcached (which is not pre-allocated; it is only used as necessary). This should be increased to at least 1Gb on production systems with more than 100 devices (and run */etc/init.d/memcached restart*) . Also ensure that memcached is enabled in *$ZENHOME/etc/zope.conf*.

2.5 Other database-related changes in Zenoss 4

Not directly related to the events subsystem, but the Zope database (ZODB) that used to be held in **$ZENHOME/var/Data.fs** and accessed by the zeoctl daemon, is now stored in the same MySQL instance as zenoss_zep (and ZEO has gone).

The **zodb** database is the main Zope database and there is also a **zodb_session** database which holds user preferences - think of zodb_session as an expanded set of user's cookies; if necessary, it can be deleted and it will be recreated automatically.

ZODB is where all the object data is stored relating to devices, components, processes, services, networks, MIBs, etc. The event processing daemons need access to the zodb database to enrich events with device and component information.

Zope objects are known as **pickles**, typically a string representation of encoded data (a **blob**) - in other words, treat the ZODB database as a "black box" (just as Data.fs was). A JSON interface is provided to access data in the ZODB and the zendmd tool still works in exactly the same way as in previous versions of Zenoss, despite the ZODB now being in MySQL.

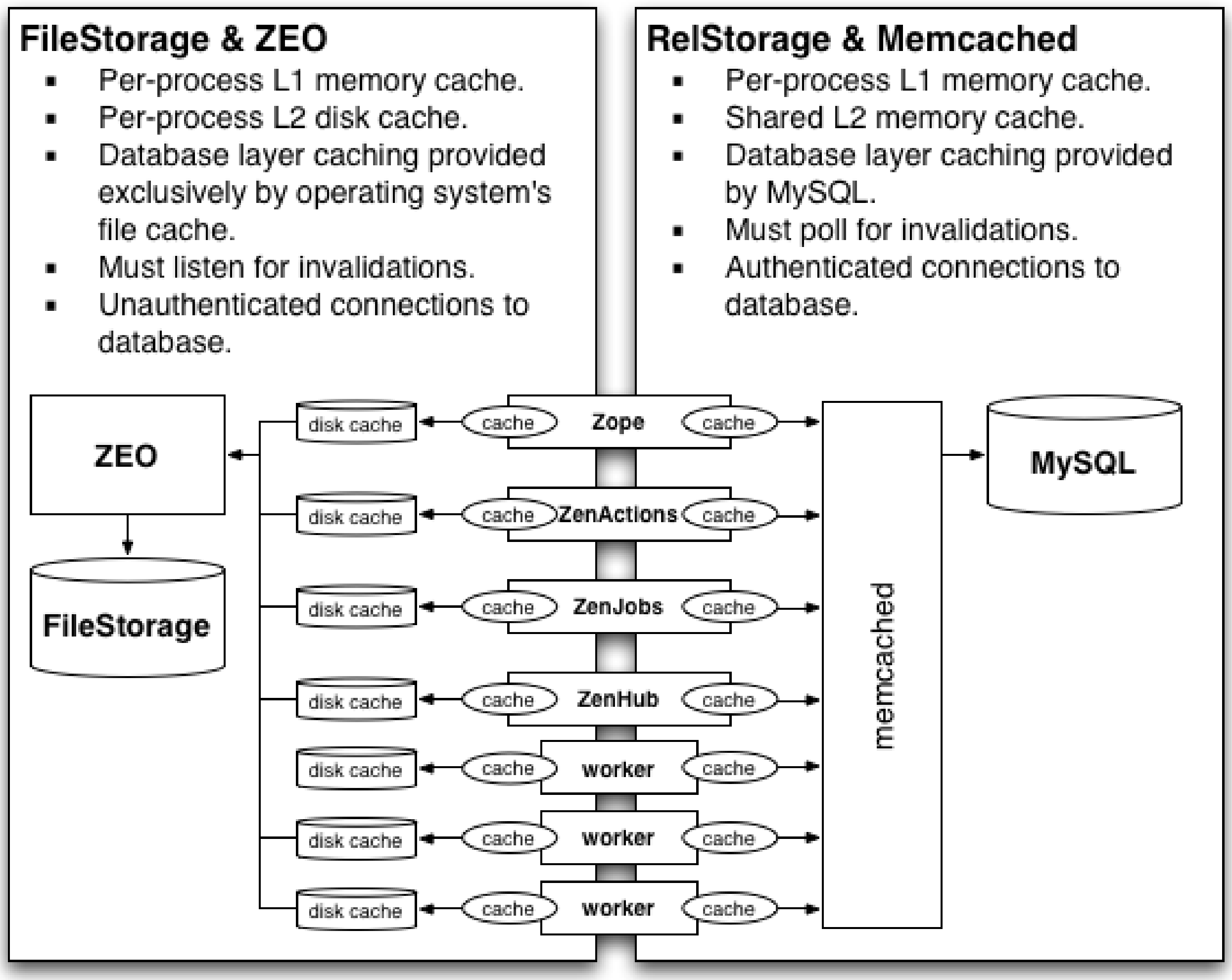

FileStorage & ZEO
Per-process L1 memory cache.
Per-process L2 disk cache.
Database layer caching provided exclusively by operating system's file cache.
Must listen for invalidations.
Unauthenticated connections to database.
RelStorage & Memcached
Per-process L1 memory cache.
Shared L2 memory cache.
Database layer caching provided by MySQL.
Must poll for invalidations.
Authenticated connections to database.
ZEO
FileStorage
disk cache
cache
Zope
ZenActions
ZenJobs
ZenHub
worker
memcached
MySQL

Figure 13: Comparison of old and new technologies to hold Zope ZODB database

To provide access to the the zodb MySQL database, a **RelStorage** subsystem is used as a high performance backend to ZODB. RelStorage may also use memcached to further enhance performance.

The older versions of Zenoss did not do much by way of indexing the events database. With Zenoss 4 holding ZODB data as well as events data in MySQL, an effective indexing mechanism was required so the **Lucene** package is used from Apache. Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is used to hold indexes for both zodb and zenoss_zep.

2.6 Event life cycle

The life cycle of an event has eight phases:

- Event generation
- Device context – additional information about the device that generated the event
- Event class mapping – to distinguish one type (class) of event from another
- Event context - additional information pertinent to a class of event

- Event transform – manipulation of event fields
- Database insertion and de-duplication
- Resolution
- Ageing and archiving

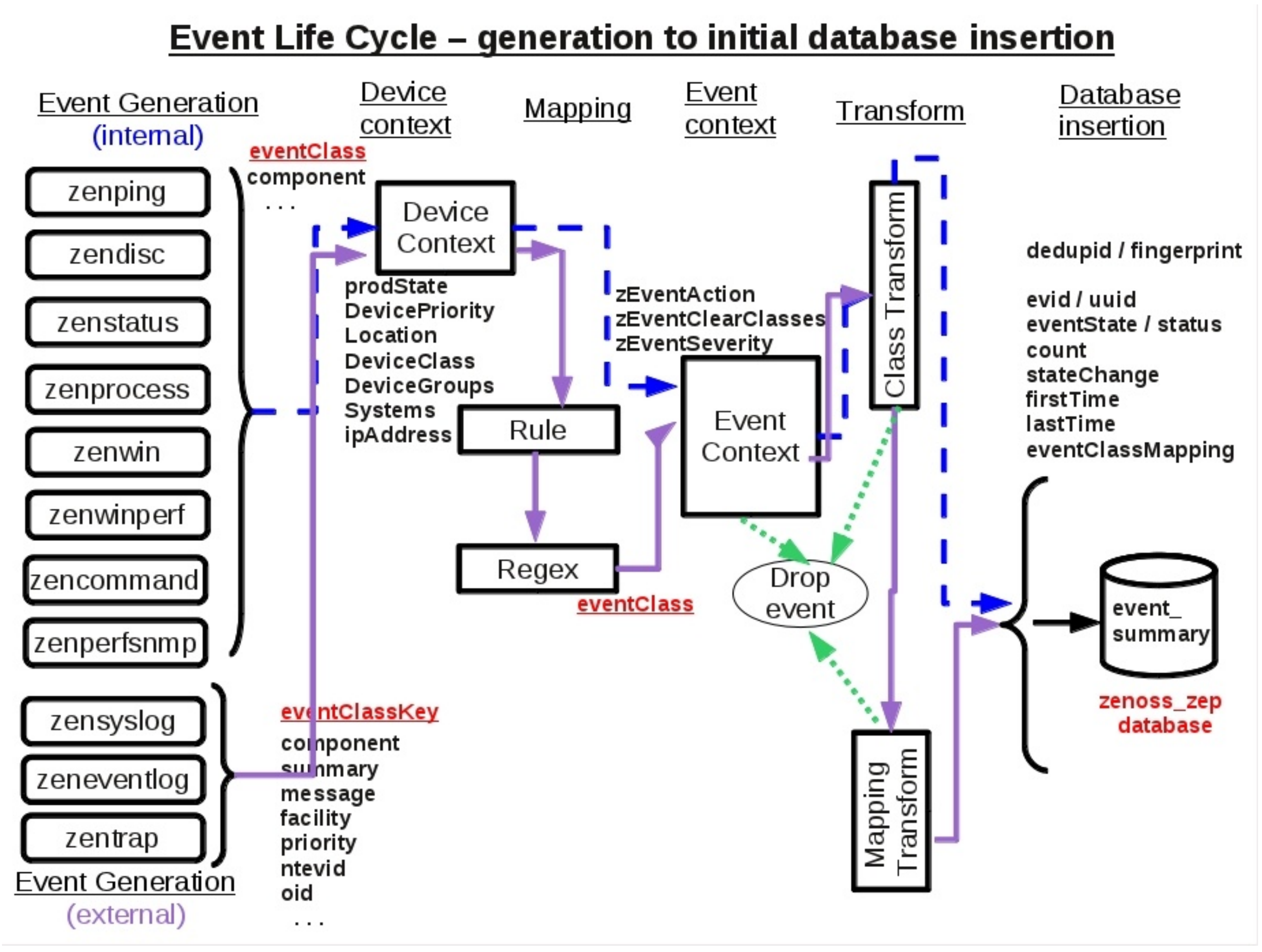


Figure 14: Event life cycle, generation to database insertion

Processing of an event depends on the **event class** that an event is assigned to – the value of its **eventClass** field. A description of each of these phases will be given here: subsequent sections of the paper provide more details of some areas.

In Figure 14, the first six phases of the event life cycle are shown. The blue, dashed path shows the progress of an internally generated Zenoss event, which does not pass through an event mapping phase. An **eventClass** field is produced by the daemon that generated the event. Its only way to apply a transform is as a class transform.

The purple path shows the progress of an event that is generated externally to Zenoss. The initial parsing daemon must provide an **eventClassKey** field which is then used, along with other fields, in an event class mapping Rule and/or Regex, which in turn provides an **eventClass** field. After mapping, the event may pass through both an event class transform and an event mapping transform.

An area that has changed fairly significantly in Zenoss 4 is the mechanism for resolving and ageing events. Prior to Version 4, an event was fundamentally open (which also encompassed eventState of Acknowledged and Suppressed as well as New) and such an event resided in the **status** table of the **events** database; alternatively, an event was Closed, in which case it was moved to the **history** table of the events database.

With Zenoss 4, the possible values of eventState have been expanded to include:

Name	Number	Description
New	0	A new event
Acknowledged	1	Acknowledged by user or transform
Suppressed	2	Event typically beyond a single point of failure
Closed	3	Event resolved by a user
Cleared	4	Event resolved by an automatic rule
Dropped	5	Would never reach the MySQL database
Aged	6	Event automatically closed according to the severity and last seen time of the event.

These are well described in chapter 7 of the Zenoss Core 4 Administration Guide. The huge difference here is that the new event_summary table in the MySQL database will probably have Closed / Cleared / Aged events in it. The event_archive table has events that have been automatically aged-out based on their severity and age.

2.6.1 Event generation

Fundamentally, events will either be generated by Zenoss itself in the process of discovery, availability and performance checking, or events will be generated outside Zenoss and captured by specialised Zenoss daemons.

Zenoss daemon	Example of when event generated
zenping	ping failure on interface
zendisc	new device discovered
zenstatus	TCP / UDP service unavailable
zenprocess	process unavailable
zenwin	Windows service failed
zenwinperf	WMI performance data collection failure / threshold
zencommand	ssh performance data collection failure / threshold
zenperfsnmp	SNMP performance data collection failure / threshold
zenmodeler	Configuration data changed on zenmodeler poll

Table 2.1.: Events generated by Zenoss itself

Zenoss daemon	Example of when event generated
zensyslog	processes syslog events received on UDP/514 (default)
zeneventlog	processes Windows events received using WMI
zentrap	processes SNMP TRAPs received on UDP/162

Table 2.2.: External events captured by specialised Zenoss daemons

Events generated internally by Zenoss need no further processing to interpret the event. The daemon that generates the event parses the native information and assigns a value to the **eventClass** field and any other relevant fields such as **component, summary, message** and **agent**. Typically the **eventClassKey** field will be blank. Some Zenoss daemons populate the **eventKey** field (for example an Interface discovery event will populate the eventKey field with the IP address of the discovered interface).

Events that are initially generated outside Zenoss are captured by **zensyslog**, **zeneventlog** or **zentrap**. These daemons each have a parsing mechanism to interpret the native event into the Zenoss event format. The Python code for the zensyslog and zentrap parsing is in *$ZENHOME/Products/ZenEvents*. (By default, $ZENHOME will be */opt/zenoss*). *SyslogProcessing.py* decodes syslog events; *zentrap.py* decodes SNMP TRAPs.

The daemons for processing Windows WMI data used to be a standard part of the Core code but with Zenoss 4 this has moved to a Zenoss-supplied ZenPack - **ZenPacks.zenoss.WindowsMonitor**. zenwin, zenwinperf and zeneventlog can all be found under that ZenPack's base directory.

Typically, the external event parsing mechanisms do not deliver a value for **eventClass**; rather they deliver a value for the **eventClassKey** field, along with values for some other fields such as component, summary, message and agent. It is then the job of the event mapping phase to distinguish the event class.

2.6.2 Application of device context

Early in the event processing life cycle, the zeneventd daemon applies **device context** to the event. This means that seven fields of the event are populated by determining the device that generated the event and then looking up the following values for the device in the ZODB database:

- prodState
- DevicePriority
- Location
- DeviceClass
- DeviceGroups
- Systems
- ipAddress (may have already been assigned)

2.6.3 Event class mapping

Event class mapping tends only to be applicable to events that originate outside the Zenoss system. It is the process by which an event is assigned a value for its **eventClass** field and, potentially, other fields.

Typically, the event generation phase will deliver an event with a few fields populated; generally this does **not** include the eventClass field but **does** include the eventClassKey field. Often the Zenoss parsing daemon (such as zensyslog), will use the **same** eventClassKey for several different native events. For example, an eventClassKey of *dropbear* is used for several login security events. The component, summary, message and agent fields may also be populated.

The event class mapping phase examines the event (such as it is, so far) and then uses a number of tests to determine the eventClass to assign to this event:

1. An eventClassKey field **must** exist for mapping to be successful.
2. A Python **Rule** can be written to test any available field of the event or any available attribute of the device from which the event came. Such rules can be complex Python expressions, including logical ANDs and ORs. If the rule is satisfied, the incoming event's eventClass field will be given the class associated with that mapping. If the rule is not satisfied, this mapping is discarded, the class is not associated, and the next mapping will be tested for a match. A Rule does **not** have to exist in a mapping instance.
3. If the Rule is satisfied (or does not exist), the mapping can then use a **Regex** Python regular expression to parse the event's summary field, checking for particular strings. The Regex can also assign parts of the summary field to new,

user-defined detail fields of the event. If a Rule exists and is satisfied, the class mapping **will** apply, even if the Regex is **not** satisfied; any user-defined fields in the Regex will **not** be created if the Regex does not match. If a Rule does **not** exist then the Regex **must** be satisfied for the mapping (and any transform) to apply.

4. The GUI dialogue that defines the mapping specifies the eventClassKey, the Rule, the Regex and any Transform. A **sequence number** is also available so that if multiple incoming events have the same eventClassKey then the sequence number defines the order in which the various mappings will be applied, lowest number first. The first Rule / Regex mapping combination that matches will be applied.

Event class mapping is executed by the zeneventd daemon.

2.6.4 Application of event context

Event context is defined by the Configuration Properties (zProperties) of an event. Event context can be defined at the event class level, for an event subclass, or at the event mapping level. As with all object-oriented attributes, the values are inherited by child objects so applying event context to a class automatically sets it for any subclasses and subclass mappings. The three event context attributes are:

● zEventAction	status | history | drop	default is status
● zEventClearClasses	by default this is an empty Python list of strings	
● zEventSeverity	*Original* by default	

Event context is applied in the event life cycle, after Rule and Regex processing but before any event transforms. Thus, the zEventAction zProperty can specify history but an event transform could override that action by setting the evt._action value to "status".

Note that the *status* and *history* values reflect the old database tables prior to Zenoss 4. status now maps to an eventState of New and history maps to an eventState of Closed; both will be stored in the event_summary database table.

Event context is applied by the zeneventd daemon.

2.6.5 Event transforms

Event transforms can be specified for an event class mapping or for an event class (or subclass). A transform is written in Python and can be used to modify any available fields of either the event or the device that generated the event. It can also create user-defined fields.

From Zenoss 2.4, cascading event transforms mean that class transforms are applied from **every** level in the appropriate class hierarchy, followed by any transform for an applied event mapping. Prior to Zenoss 2.4, **either** a mapping transform was applied, **or** a class transform, but not both. Class transforms were only applied to the exact class, not from the event class hierarchy.

A transform in an event mapping will only be executed once the eventClassKey has been matched, and the Rule has been satisfied (if it exists). If a Rule does not exist, any Regex has to be satisfied for the transform to be executed.

Event transforms are executed by the zeneventd daemon.

2.6.6 Database insertions and de-duplication

Zenoss events are now stored in a MySQL database called **zenoss_zep** (used to be **events**). The main tables for the event life cycle are the **event_summary** table for recent events, the **event_archive** table for old events.

Some fields of the event are only assigned at database insertion time – they are not available at event mapping or event transform time. These include:

- count
- eventState
- evid
- stateChange
- dedupid
- eventClassMapping
- firstTime
- lastTime

It is the Java zeneventserver daemon that is responsible for getting events into the database.

Zenoss automatically applies a duplication detection rule so that if a “duplicate” event arrives, then the repeat count of an existing event will be incremented. “duplicate” is defined as having the following fields the same:

- device
- component
- eventClass
- eventKey
- severity

If the event does not populate the eventKey field, then the summary field must also match. The **dedupid** field is created by concatenating the above fields together, separated by the pipe (vertical bar) symbol. Thus an example dedupid might be:

```
zenoss.skills-1st.co.uk|su|/Security/Su||5|FAILED SU (to root)jane on /dev/pts/1
```

where the device is *zenoss.skills-1st.co.uk*, component is *su*, eventClass is */Security/Su* , the eventKey is unset, severity is *5* (Critical), and the summary is *FAILED SU (to root) jane on /dev/pts/1* .

In Zenoss 4, the dedupid field is also known as the **fingerprint**.

When a new event is received by the system, the dedupid is constructed by the zeneventd daemon. Transforms may modify either component fields of the fingerprint or may directly modify the dedupid field.

When zeneventserver comes to insert the event in the database, if it matches the dedupid for any active event, the existing event is updated with properties of the new event occurrence, the event's count is incremented by one, and the lastTime field is updated to be the created time of the new event occurrence.

Note that this is a subtle but significant change from prior versions of Zenoss as the existing event is updated with properties of the new event; older versions of Zenoss simply updated the count and lastTime fields. For example, if the fingerprint includes an eventKey so does not include the summary, the resulting event will now show the summary of the latest received duplicate event.

If the incoming event does not match the dedupid of any active events, then it is inserted into the active event table with a count of 1, and the firstTime and lastTime fields are set to the created time of the new event.

2.6.7 Resolution

Resolution of a problem represented by an event can happen in several ways:

- A user closes the event (eventState = Closed)
- The event context zEventAction zProperty for an event class is drop (the event is discarded). For example, event class /Ignore.
- The event context zEventAction zProperty for an event class is history (eventState=Closed). For example, event class /Archive.
- A transform sets evt._action to 'drop' (the event is discarded)
- A transform sets evt._action to 'history' (eventState=Closed)
- Another clearing event arrives that clears the initial event (eventState=Cleared)
- The Event Manager settings have severity and lastSeen parameters that denote which events will be automatically aged (eventState=Aged)

All the above events will still be in the event_summary table of the MySQL database. The Event Manager parameter for Event Archive Threshold is the only automatic action that moves events from event_summary to event_archive and it will move all events with eventState of Closed, Cleared and Aged.

The more interesting forms of event resolution involve correlation of events; there are two different mechanisms. The basic principle is that “good news” clears “bad news”.

The first clearing mechanism is that any event with a severity of **Clear** will search the event_summary table for “similar” active events and set their eventState to **Cleared** (not **Closed**).

The Zenoss Core 4 Administrators Guide defines this **auto-clear fingerprint** as:

- If component UUID exists:
    - component UUID
    - eventClass
    - eventKey (can be blank)
- If component UUID does not exist:
    - device
    - component (can be blank)
    - eventClass
    - eventKey (can be blank)

This can be a little confusing. The Event Console shows a “component” field. It does not show a component UUID field. Strictly the component field in the Event Console shows the **element_sub_identifier** field from the MySQL database table - the name of the component. Some events generate a component UUID (Universally Unique Identifier) and some do not. Inspecting the event in the database or using the JSON interface is the only way to determine whether this unique component id field exists or not. If it does exist then it should also, by implication, denote the device that the component belongs to, hence the device field is unnecessary. (Versions of Zenoss prior to 4 did not have a component UUID; “similar” was defined as having the same **eventClass**, **device** and **component** fields.)

Either way in Core 4, the eventClass and the eventKey fields are significant. If the component UUID does not exist then it is the element_sub_identifier (component name) that must match, along with the device name (element_identifier in the MySQL table).

The second automatic clearing mechanism extends the auto-clear fingerprint definition of eventClass. The event context of an event class includes zEventClearClasses which is a list of other event classes that this “good news” event will clear, in addition to its own class. The other conditions of the auto-clear fingerprint remain the same.

Note that the same effect can be achieved in a transform by assigning a list of class names to *evt._clearClasses* .

All events with the same auto-clear fingerprint are cleared, not just the most recent.

The clearing event will automatically have its eventState set to **Closed**, provided it matches one or more “bad news” events. If it does not match any events then the clearing event is dropped and will not be persisted to the zenoss_zep database. This is to avoid filling up the database with redundant “good news” events.

When correlation takes place some of the existing “bad news” event fields are updated; **stateChange** becomes the time when the event was resolved; **clearid** is populated with the **evid** field of the clearing, “good news” event.

This automatic resolution of events is performed by the zeneventserver daemon.

2.6.8 Ageing and archiving

Maintenance is required on the tables of the zenoss_zep database or the disk will simply fill up eventually. Three mechanisms are provided by the Event Manager:

- By default, events with severity less than Error will be Aged after an *Event **Ageing** Threshold* of 4 hours; that is, the eventState will be set to Aged (strictly the value 6).
- By default, the *Event **Archive** Threshold* is 4320 minutes (3 days). This means any event with eventState of Closed, Cleared or Aged will be moved from the event_summary table to the event_archive table of the zenoss_zep database.
- The *Delete Archived Events Older Than (days)* parameter is 90 by default. This is the only parameter that automatically deletes data. It is not possible to fine-tune this to delete, say, lower severity events after different intervals.

Zenoss prior to version 4 provided a utility,
$ZENHOME/Products/ZenUtils/ZenDeleteHistory.py
which could delete events selectively based on age and severity. This utility is not shipped with Zenoss 4 and currently has no equivalent function.

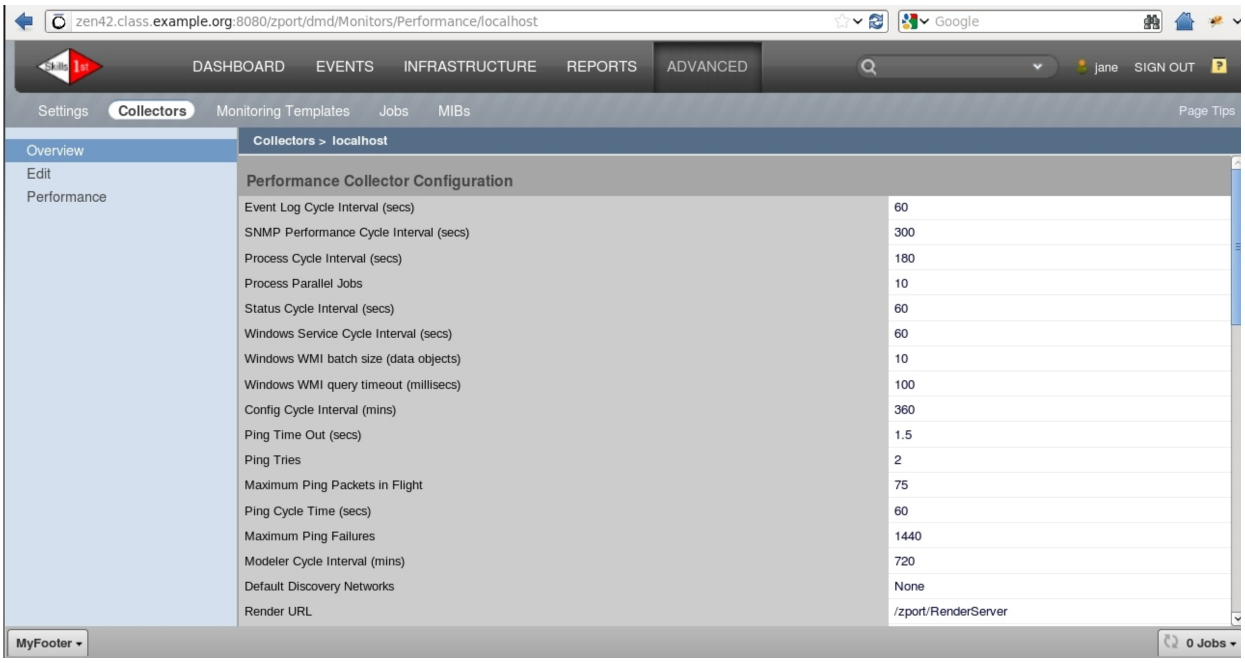
Deleting data from the old history table in Zenoss 3 used to be very slow. In Zenoss 4, the event_archive table is **partitioned**, by day, rather than being one huge file. This means that deleting data is simply a matter of dropping partition files. This can be seen from the mysql interface with:

```
show create table event_archive;
```

3 Events generated by Zenoss

In the course of discovery, availability monitoring and performance monitoring, Zenoss may generate events to represent a change in the current status. Although many events are “bad news” it should be recognised that events can also be “good news” - Interface Up, Threshold no longer breached, etc.

Events generated by Zenoss are dependent on the various polling intervals configured. To examine the default parameters, use the *ADVANCED -> Collectors* menu. Click on *localhost* (the collector on the Zenoss system). **Note** that early versions of Zenoss used the term and menu-option **Monitors** rather than **Collectors**.



Performance Collector Configuration

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

Figure 15: Default parameters for localhost Collector

Parameters to note particularly are:

• SNMP Performance Cycle Interval	300 secs (5 mins)
• Process Cycle Interval	180 secs (3 mins)
• Status Cycle Interval	60 secs (1 min)
• Windows Service Cycle Interval	60 secs (1 min)
• Ping Cycle Time	60 secs (1 min)
• Modeler Cycle Interval	420 mins (12 hours)

3.1 zenping

The most basic level of availability checking is to ping-poll. The **zenping** daemon will, by default, ping-poll each interface, every minute. An interface down event is generated when the ping fails to get a response. This event is automatically cleared when a similar ping is successful; meantime, while an interface remains down, the count field of the event is increased.

The zenping daemon can detect when the network path to a device is broken, for example if a single-point-of-failure router is down. With Zenoss 4 this is achieved using **nmap**; with earlier versions, Zenoss built an internal topology based on querying routing tables with SNMP.

If an event is received for an isolated element, an event is generated with an eventState field of *Suppressed* and the summary field reports not only the interface for which the ping failed, but also the causal device; for example:

ip 10.191.101.1 is down, failed at bino.skills-1st.co.uk

All other device availability monitoring is dependent on ping access. Once a ping has failed, SNMP, process, TCP/UDP service and windows service monitoring will all be suspended until ping access is restored. The count field of the higher level monitoring events will not increase until ping access is resumed.

Also note that if there is no ping access, no performance information will be collected. If a device really does not support ping, perhaps because of firewall restrictions, then ensure that the zProperty *zPingMonitorIgnore* is set to *True;* this will permit SNMP and ssh availability monitoring and performance data collection.

The logfile for zenping is *zenping.log* in *$ZENHOME/log.*

3.2 zenstatus

The **zenstatus** daemon can be configured to check for access to various TCP and/or UDP ports on both Windows and Unix architectures. By default, it checks every minute. Zenoss comes with a huge number of services pre-configured; these can be examined from the *INFRASTRUCTURE -> Ip Services* menu. By default, the only service monitors that are active are for *smtp* and *http*; the rest are set with monitoring disabled.

As with ping polling, a "good news" service event for a device automatically clears a similar "bad news" event and the count field of the event increases whilst the service remains down.

The logfile for zenstatus is *zenstatus.log* in *$ZENHOME/log.*

3.3 zenprocess

zenprocess monitors Windows and Unix systems for the presence of processes. In a Unix context, this would be whether the process appears in a *ps -ef* listing; in a Windows context, the process must appear in the Windows Task Manager (and note that this check **is** case sensitive on both architectures). Monitoring is every 3 minutes, by default.

Configuration of process monitoring for a device is similar as for services – the *INFRASTRUCTURE -> Processes* menu provides a way to configure processes to be monitored. Zenoss 4 comes with definitions preconfigured for all the Zenoss processes.

Process monitoring is actually achieved using the Host Resources Management Information Base (MIB) of SNMP, by retrieving the **hrSWRun** table. This means that if SNMP access to a device is broken, there will be no process information.

As with the other availability daemons, "good news" events clear "bad news" events and the count field increases on subsequent failed polls.

The logfile for zenprocess is *zenprocess.log* in *$ZENHOME/log.*

3.4 zenwin

The zenwin daemon ships with the **ZenPacks.zenoss.WindowsMonitor** ZenPack with Zenoss 4 (it was a standard part of the Core code in earlier versions). It monitors Windows services (not TCP / UDP services). These can be examined from the *INFRASTRUCTURE -> Windows Services*. By default, none of these monitors are active.

zenwin uses the Windows Management Instrumentation (WMI) interface to access services on the remote system every minute, by default. The zProperties for a device (or device class) must be configured to allow access to WMI before windows service polling can be successful.

As with ping polling, a "good news" windows service event for a device automatically clears a similar "bad news" event and the count field increases on subsequent failed polls.

The logfile for zenwin is *zenwin.log* in *$ZENHOME/log*.

3.5 zenwinperf

zenwinperf is a new daemon for Zenoss 4 which is also part of the ZenPacks.zenoss.WindowsMonitor ZenPack. With earlier versions of Zenoss, many users deployed the excellent community WMI Data Source and WMI Windows Performance ZenPacks to achieve something very similar to this new daemon.

zenwinperf provides performance monitoring of interfaces, filesystems, memory, CPU and paging using the WMI protocol. Default thresholds are configured for some metrics which then generate events when exceeded. It can be extended by the user to monitor other perfmon metrics using the WMI protocol.

Data is gathered every 5 minutes.

The logfile for zenwinperf is *zenwinperf.log* in *$ZENHOME/log*.

3.6 zenperfsnmp

zenperfsnmp polls each device every 5 minutes, by default. It can collect both SNMP performance information and status information for processes. Even if SNMP performance monitoring is not configured, zenperfsnmp checks that the SNMP agent is available.

Within 5 minutes of an SNMP poll failure, an "snmp agent down" event should be generated. Within a further 3 minutes there should be an "Unable to read processes on device .." event, if process monitoring is configured. Note also that the count field for individual missing process events should stop increasing. While SNMP access to the device remains broken, the count field for the "Unable to read processes on device .." event will increase every 3 minutes.

The logfile for zenperfsnmp is *zenperfsnmp.log* in *$ZENHOME/log*.

3.7 zencommand

The zencommand daemon performs monitoring based on running commands, typically over an **ssh** connection. Like zenperfsnmp and zenwinperf it uses performance templates to monitor metrics and can generate an event if a threshold is breached.

The logfile for zencommand is *zencommand.log* in *$ZENHOME/log*.

4 Syslog events

The Unix syslog mechanism is pervasive throughout all versions of Unix / Linux although slightly different versions and formats exist. There are also open source implementations of syslog for Windows systems and many networking devices also support the syslog concept.

Typically system messages are output to one or more log files such as */var/log/messages*. The syslog subsystem can also be configured to send syslog messages to a central syslog rather than holding files on each system. The well-known default port for forwarding syslog messages is **UDP/514**.

A standard syslog system is configured by the *syslog.conf* file, typically in */etc* . A newer version of syslog is implemented on some systems, **syslog-ng**, which has greater filtering capabilities. The syslog-ng configuration file is typically */etc/syslog-ng/syslog-ng.conf.*

Another variation is **rsyslogd** which is typically shipped with newer RedHat / CentOS SuSE systems, configured through **/etc/rsyslog.conf**.

A syslog message includes a **priority** and a **facility**. The priorities are:

0	emerg
1	alert
2	crit
3	err
4	warning
5	notice
6	info
7	debug

Facilities include:

auth (4)	authpriv (10)
cron (9)	daemon (3)
ftp (11)	kern (0)
lpr (6)	mail (2)

news (7)	syslog (5)
user (1)	uucp (8)

These definitions can be found in *syslog.h* (typically in */usr/include/sys*). Both priority and facility are encoded in a single 32-bit integer where the bottom 3 bits represent priority and the remaining 28 bits are used to represent facilities.

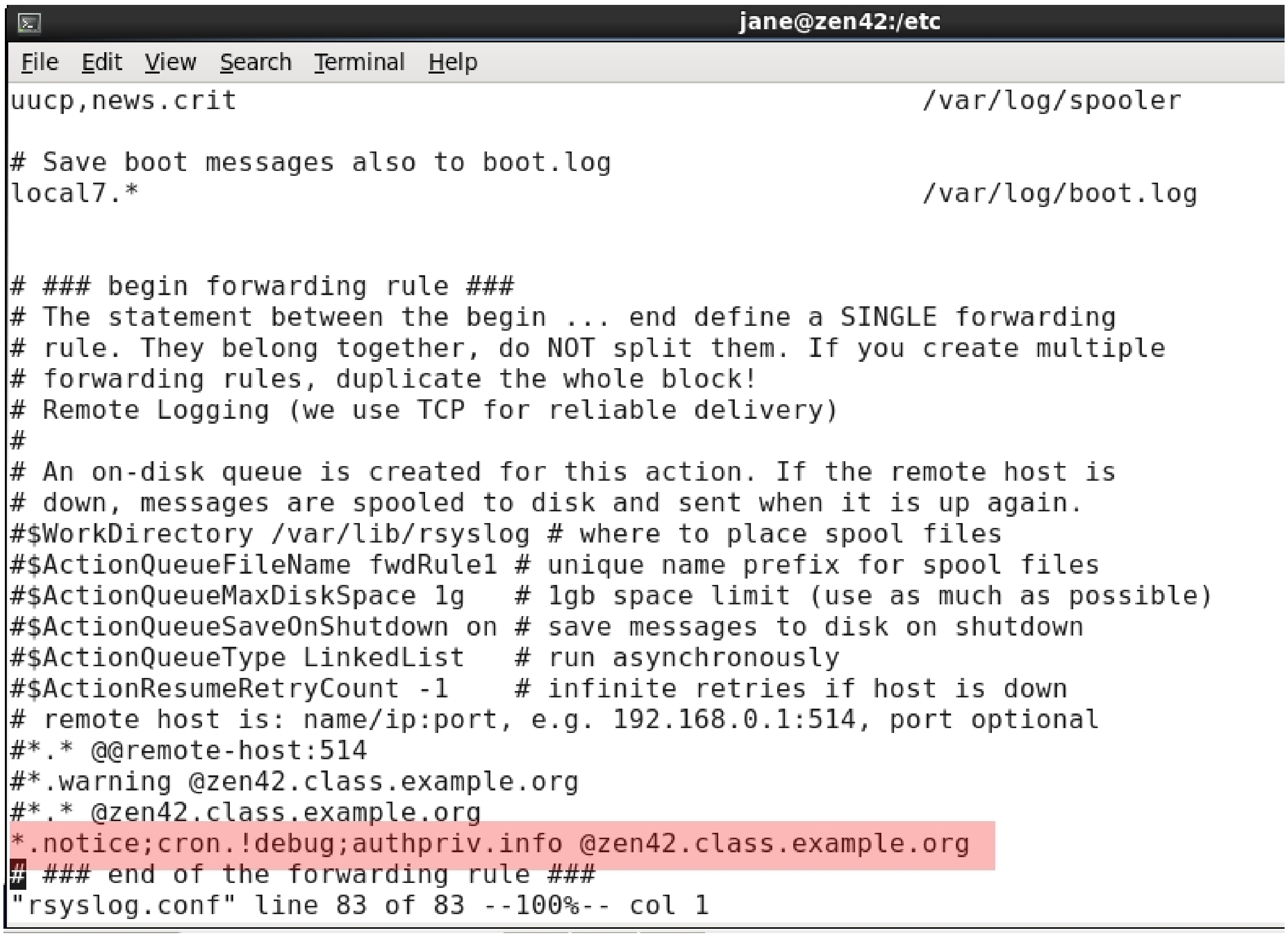
For example, if the facility/priority tag is <22>, this would be 00010110 in binary, where the bottom 110 represents a priority of 6 (info) and the top 00010 represents a facility of 2 = mail.

4.1 Configuring syslog.conf

Any device that is going to report syslog events to Zenoss must have its syslog.conf file configured with the destination address of the Zenoss system. The original syslog.conf permits filtering based on priority and facility so, a catch-all statement to send all events to the Zenoss system, would be:

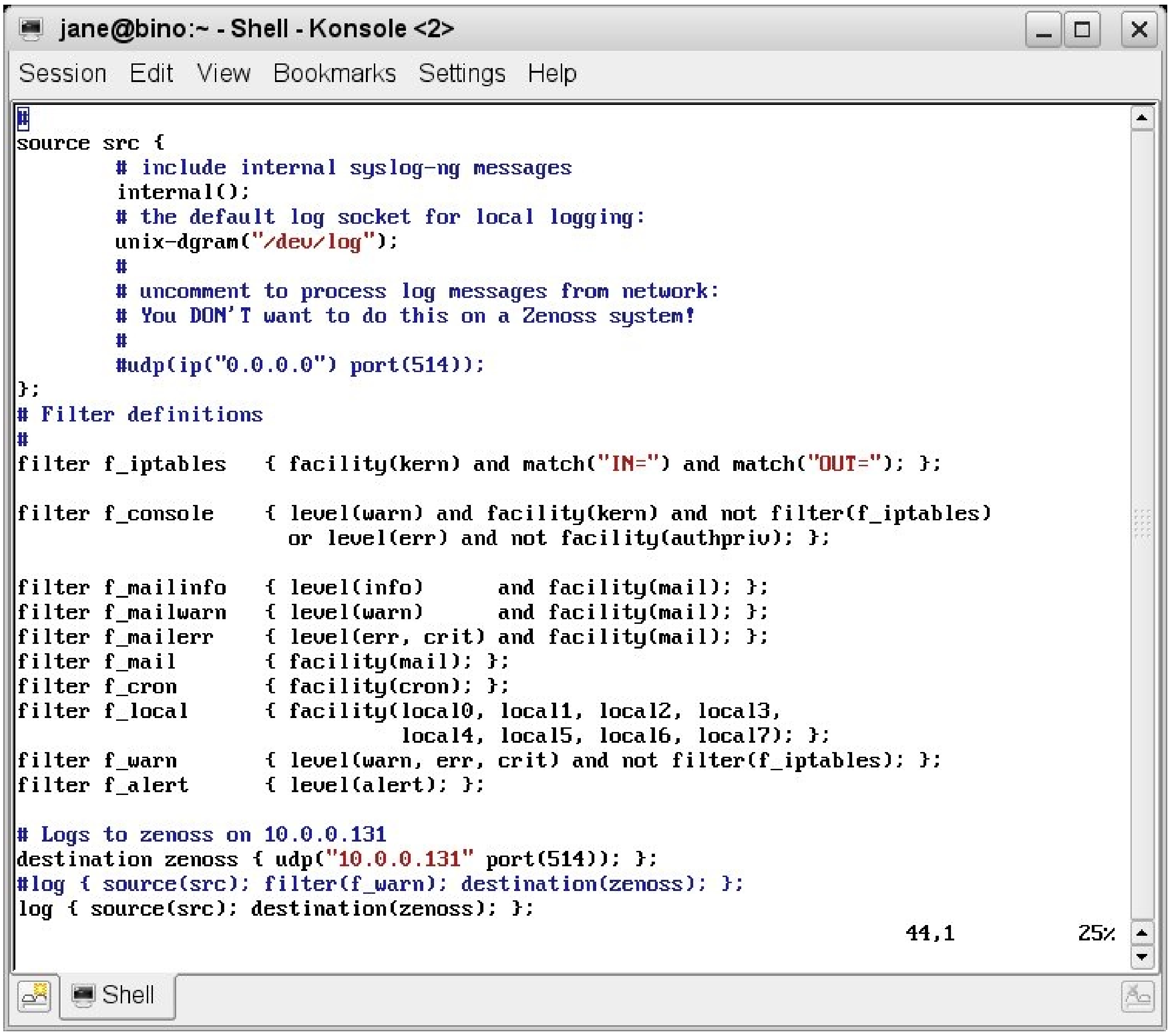
```
    *.debug       @<IP address of your Zenoss system>
```

This also works for rsyslogd. See Figure 16 for an rsyslog / syslog example that forwards to zen42.class.example.org all facilities with priority of notice and above but all cron messages are filtered out; authpriv messages will be forwarded with severity info and above.



```
uucp,news.crit                                          /var/log/spooler

# Save boot messages also to boot.log
local7.*                                                /var/log/boot.log


# ### begin forwarding rule ###
# The statement between the begin ... end define a SINGLE forwarding
# rule. They belong together, do NOT split them. If you create multiple
# forwarding rules, duplicate the whole block!
# Remote Logging (we use TCP for reliable delivery)
#
# An on-disk queue is created for this action. If the remote host is
# down, messages are spooled to disk and sent when it is up again.
#$WorkDirectory /var/lib/rsyslog # where to place spool files
#$ActionQueueFileName fwdRule1 # unique name prefix for spool files
#$ActionQueueMaxDiskSpace 1g   # 1gb space limit (use as much as possible)
#$ActionQueueSaveOnShutdown on # save messages to disk on shutdown
#$ActionQueueType LinkedList   # run asynchronously
#$ActionResumeRetryCount -1    # infinite retries if host is down
# remote host is: name/ip:port, e.g. 192.168.0.1:514, port optional
#*.* @@remote-host:514
#*.warning @zen42.class.example.org
#*.* @zen42.class.example.org
*.notice;cron.!debug;authpriv.info @zen42.class.example.org
# ### end of the forwarding rule ###
"rsyslog.conf" line 83 of 83 --100%-- col 1
```

Figure 16Configuration file for rsyslog sending selected events to Zenoss server syslog-ng.conf requires at least a **source**, a **destination** and a **log** statement. syslog-ng offers superior filtering over the original syslog so one or more **filter** statements may also be present.


jane@bino:~ - Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell

```
#
source src {
        # include internal syslog-ng messages
        internal();
        # the default log socket for local logging:
        unix-dgram("/dev/log");
        #
        # uncomment to process log messages from network:
        # You DON'T want to do this on a Zenoss system!
        #
        #udp(ip("0.0.0.0") port(514));
};
# Filter definitions
#
filter f_iptables   { facility(kern) and match("IN=") and match("OUT="); };

filter f_console    { level(warn) and facility(kern) and not filter(f_iptables)
                      or level(err) and not facility(authpriv); };

filter f_mailinfo   { level(info)      and facility(mail); };
filter f_mailwarn   { level(warn)      and facility(mail); };
filter f_mailerr    { level(err, crit) and facility(mail); };
filter f_mail       { facility(mail); };
filter f_cron       { facility(cron); };
filter f_local      { facility(local0, local1, local2, local3,
                               local4, local5, local6, local7); };
filter f_warn       { level(warn, err, crit) and not filter(f_iptables); };
filter f_alert      { level(alert); };

# Logs to zenoss on 10.0.0.131
destination zenoss { udp("10.0.0.131" port(514)); };
#log { source(src); filter(f_warn); destination(zenoss); };
log { source(src); destination(zenoss); };
                                                                    44,1          25%
```

Figure 17: syslog-ng.conf to send all events to Zenoss system at 10.0.0.131 (no filtering active)

4.2 Zenoss processing of syslog messages

To collect syslog messages with Zenoss, the **zensyslog** process automatically starts on port UDP/514 and collects any syslog messages directed from other systems. zensyslog then parses these messages into Zenoss events. You must ensure that the syslog.conf file on the Zenoss system does **not** enable collecting remote syslogs or the syslogd and zensyslog processes will clash over who gets UDP/514 (it is possible to reconfigure either daemon, if required).

To examine the incoming syslog messages and the parsing that zensyslog performs, the level of zensyslog logging can be increased.

1. Use the *INFRASTRUCTURE -> Settings -> Daemons* menu.
2. Click the *edit config* link for the zensyslog daemon.
3. Change the following parameters and click *Save*:

logorig	select this
logseverity	*Debug*

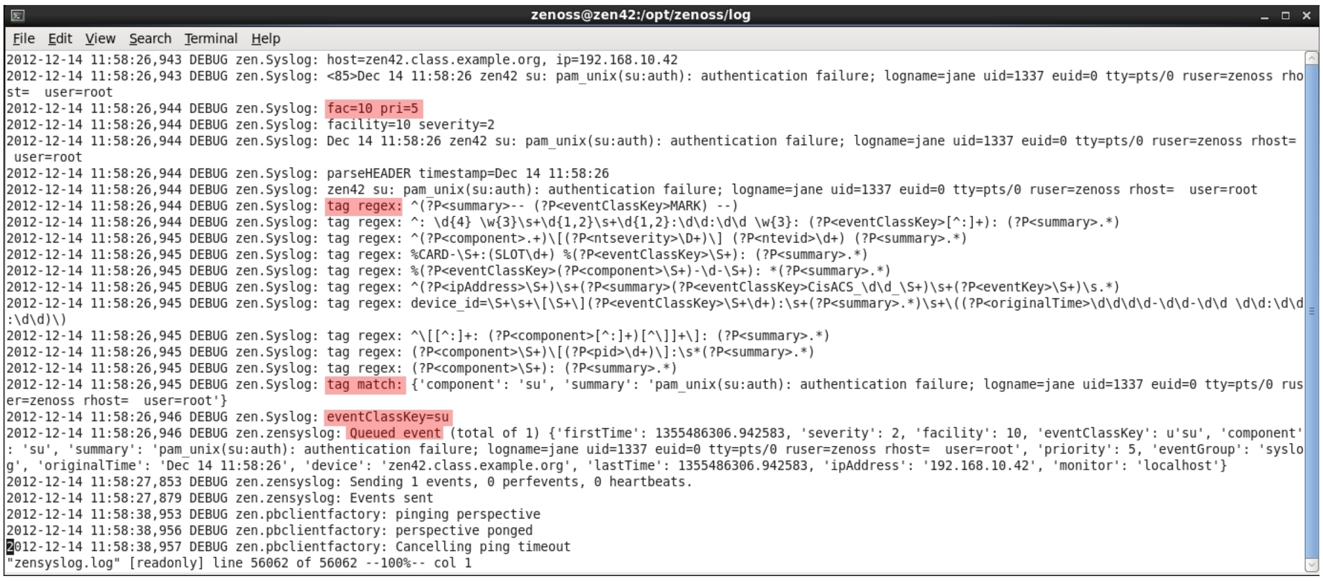
4. Inspect the underlying configuration file in *$ZENHOME/etc/zensyslog.conf*.
5. The **logorig** line says to log the original incoming syslog message; it will be in *$ZENHOME/log/origsyslog.log*. Note that this parameter is unique to zensyslog and is useful for debugging.
6. The **logseverity** line is a generic Zenoss daemon parameter; a value of *10* is the maximum *Debug* level.
7. Don't forget to Save this change
8. Use the *Restart* link to recycle zensyslog. Alternatively, as the *zenoss* user, issue the command:

```
zensyslog restart
```

9. Examine the zensyslog log file in *$ZENHOME/log/zensyslog.log*
10. A new incoming event starts with a line showing hostname and ip address, eg.

```
host=zen241.class.example.org, ip=172.16.222.241
```

11. The next 2 lines show the raw message and the decoding for facility and priority.
12. Lines starting with *tag* show the zensyslog parsing process as it tests the incoming line against various Python regular expressions, hopefully ending with a *tag match* line.
13. If a match is successful, an eventClassKey may be determined
14. The last line for a parsed event should be a *Queueing event* .



zenoss@zen42:/opt/zenoss/log
File Edit View Search Terminal Help

```
2012-12-14 11:58:26,943 DEBUG zen.Syslog: host=zen42.class.example.org, ip=192.168.10.42
2012-12-14 11:58:26,943 DEBUG zen.Syslog: <85>Dec 14 11:58:26 zen42 su: pam_unix(su:auth): authentication failure; logname=jane uid=1337 euid=0 tty=pts/0 ruser=zenoss rho
st=  user=root
2012-12-14 11:58:26,944 DEBUG zen.Syslog: fac=10 pri=5
2012-12-14 11:58:26,944 DEBUG zen.Syslog: facility=10 severity=2
2012-12-14 11:58:26,944 DEBUG zen.Syslog: Dec 14 11:58:26 zen42 su: pam_unix(su:auth): authentication failure; logname=jane uid=1337 euid=0 tty=pts/0 ruser=zenoss rhost=
 user=root
2012-12-14 11:58:26,944 DEBUG zen.Syslog: parseHEADER timestamp=Dec 14 11:58:26
2012-12-14 11:58:26,944 DEBUG zen.Syslog: zen42 su: pam_unix(su:auth): authentication failure; logname=jane uid=1337 euid=0 tty=pts/0 ruser=zenoss rhost=  user=root
2012-12-14 11:58:26,944 DEBUG zen.Syslog: tag regex: ^(?P<summary>-- (?P<eventClassKey>MARK) --)
2012-12-14 11:58:26,944 DEBUG zen.Syslog: tag regex: ^: \d{4} \w{3}\s+\d{1,2}\s+\d{1,2}:\d\d:\d\d \w{3}: (?P<eventClassKey>[^:]+): (?P<summary>.*)
2012-12-14 11:58:26,945 DEBUG zen.Syslog: tag regex: ^(?P<component>.+)\[(?P<ntseverity>\D+)\] (?P<ntevid>\d+) (?P<summary>.*)
2012-12-14 11:58:26,945 DEBUG zen.Syslog: tag regex: %CARD-\S+:(SLOT\d+) %(?P<eventClassKey>\S+): (?P<summary>.*)
2012-12-14 11:58:26,945 DEBUG zen.Syslog: tag regex: %(?P<eventClassKey>(?P<component>\S+)-\d-\S+): *(?P<summary>.*)
2012-12-14 11:58:26,945 DEBUG zen.Syslog: tag regex: ^(?P<ipAddress>\S+)\s+(?P<summary>(?P<eventClassKey>CisACS_\d\d_\S+)\s+(?P<eventKey>\S+)\s.*)
2012-12-14 11:58:26,945 DEBUG zen.Syslog: tag regex: device_id=\S+\s+\[\S+\](?P<eventClassKey>\S+\d+):\s+(?P<summary>.*)\s+\((?P<originalTime>\d\d\d\d-\d\d-\d\d \d\d:\d\d
:\d\d)\)
2012-12-14 11:58:26,945 DEBUG zen.Syslog: tag regex: ^\[[^:]+: (?P<component>[^:]+)[^\]]+\]: (?P<summary>.*)
2012-12-14 11:58:26,945 DEBUG zen.Syslog: tag regex: (?P<component>\S+)\[(?P<pid>\d+)\]:\s*(?P<summary>.*)
2012-12-14 11:58:26,945 DEBUG zen.Syslog: tag regex: (?P<component>\S+): (?P<summary>.*)
2012-12-14 11:58:26,945 DEBUG zen.Syslog: tag match: {'component': 'su', 'summary': 'pam_unix(su:auth): authentication failure; logname=jane uid=1337 euid=0 tty=pts/0 rus
er=zenoss rhost=  user=root'}
2012-12-14 11:58:26,946 DEBUG zen.Syslog: eventClassKey=su
2012-12-14 11:58:26,946 DEBUG zen.zensyslog: Queued event (total of 1) {'firstTime': 1355486306.942583, 'severity': 2, 'facility': 10, 'eventClassKey': u'su', 'component'
: 'su', 'summary': 'pam_unix(su:auth): authentication failure; logname=jane uid=1337 euid=0 tty=pts/0 ruser=zenoss rhost=  user=root', 'priority': 5, 'eventGroup': 'syslo
g', 'originalTime': 'Dec 14 11:58:26', 'device': 'zen42.class.example.org', 'lastTime': 1355486306.942583, 'ipAddress': '192.168.10.42', 'monitor': 'localhost'}
2012-12-14 11:58:27,853 DEBUG zen.zensyslog: Sending 1 events, 0 perfevents, 0 heartbeats.
2012-12-14 11:58:27,879 DEBUG zen.zensyslog: Events sent
2012-12-14 11:58:38,953 DEBUG zen.pbclientfactory: pinging perspective
2012-12-14 11:58:38,956 DEBUG zen.pbclientfactory: perspective ponged
2012-12-14 11:58:38,957 DEBUG zen.pbclientfactory: Cancelling ping timeout
"zensyslog.log" [readonly] line 56062 of 56062 --100%-- col 1
```

Figure 18: zensyslog.log showing parsing process

Whenever different native event log systems are integrated there is almost inevitably a mismatch of severities. The following table demonstrates this.

Zenoss	**syslog priority**	**Windows**
Critical (red) (5)	emerg (0)	Error (1)
Error (orange) (4)	alert (1)	Warning (2)
Warning (yellow) (3)	crit (2)	Informational (3)
Info (blue) (2)	err (3)	Security audit success (4)
Debug (grey) (1)	warning (4)	Security audit failure (5)
Clear (green) (0)	notice (5)	
	info (6)	
	debug (7)	

Table 4.1.: Event severities for Zenoss, syslog and Windows

Note that the numeric value of Zenoss event severity **decreases** as events get less critical but that the priority of syslog events **increases** as events get less critical.

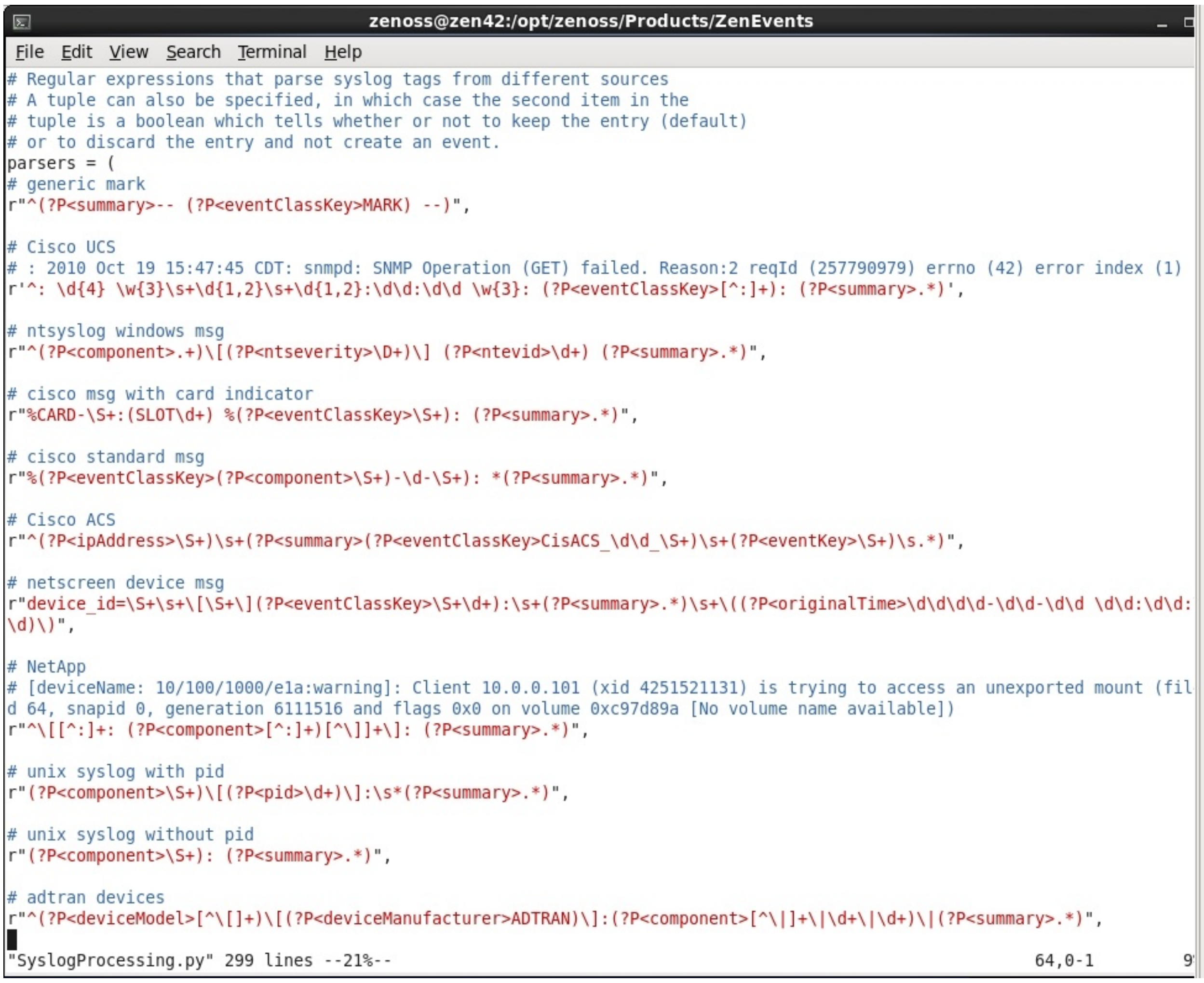
Default mapping from syslog priority to Zenoss event severity, is performed by *$ZENHOME/Products/ZenEvents/SyslogProcessing.py* – search for *defaultSeverityMap* around line 187 in Core 4.2. The result is that:

- syslog priority < 3 (emerg, alert, crit) map to Zenoss severity 5 (Critical)
- syslog priority 3 (err) maps to Zenoss severity 4 (Error)
- syslog priority 4 (warning) maps to Zenoss severity 3 (Warning)
- syslog priority 5 or 6 (notice , info) map to Zenoss severity 2 (Info)

Out-of-the-box, all syslog events map to the Zenoss event class of **/Unknown** .

SyslogProcessing.py is the code that parses any incoming syslog message and generates a Zenoss event.

The first section has a series of Python regular expressions to match against the incoming syslog line. Each expression is checked in turn until a match is found. If no match is found then an entry goes to *$ZENHOME/log/zensyslog.log* with *parseTag failed* .



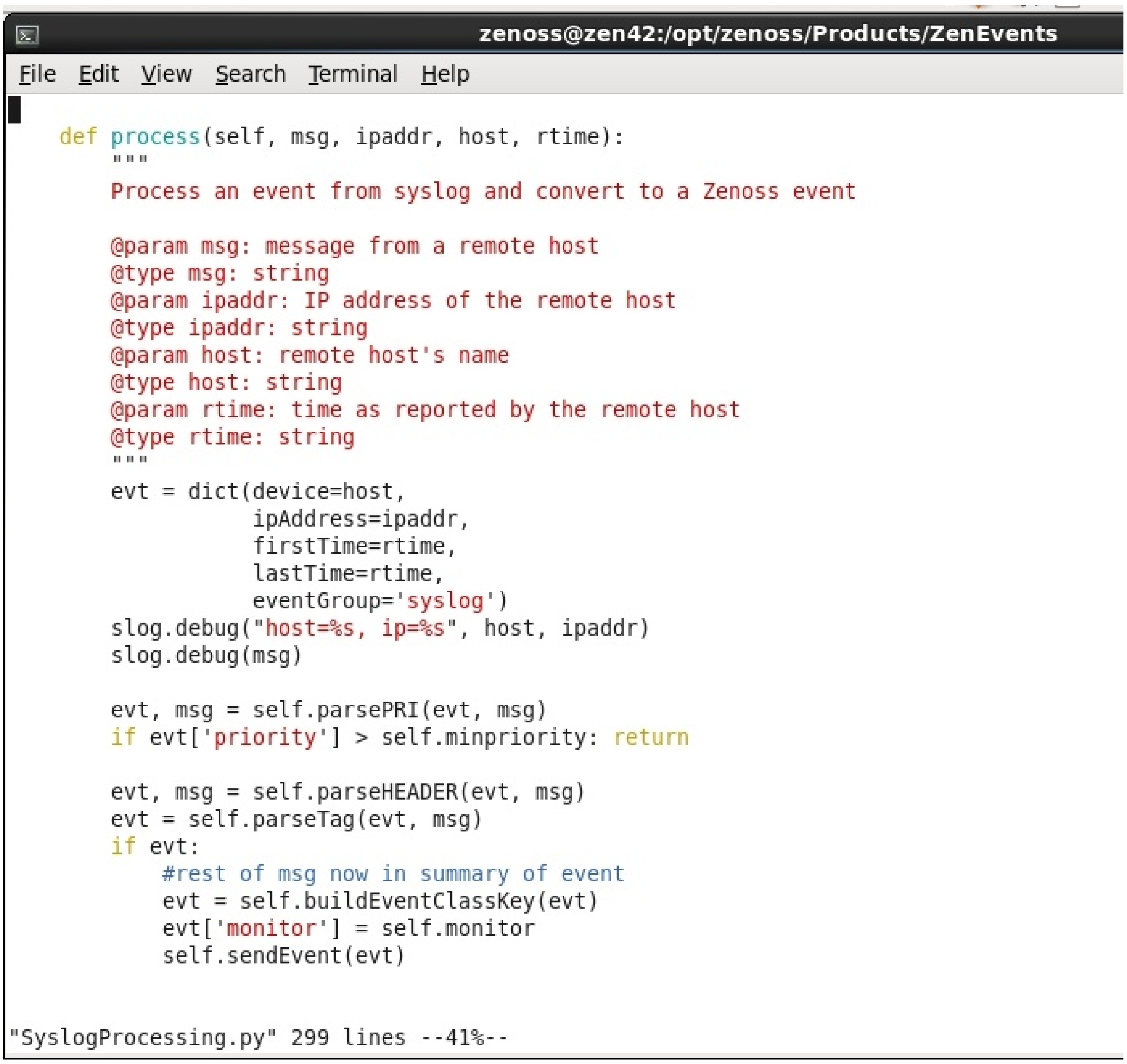
zenoss@zen42:/opt/zenoss/Products/ZenEvents
File Edit View Search Terminal Help

```
# Regular expressions that parse syslog tags from different sources
# A tuple can also be specified, in which case the second item in the
# tuple is a boolean which tells whether or not to keep the entry (default)
# or to discard the entry and not create an event.
parsers = (
# generic mark
r"^(?P<summary>-- (?P<eventClassKey>MARK) --)",

# Cisco UCS
# : 2010 Oct 19 15:47:45 CDT: snmpd: SNMP Operation (GET) failed. Reason:2 reqId (257790979) errno (42) error index (1)
r'^: \d{4} \w{3}\s+\d{1,2}\s+\d{1,2}:\d\d:\d\d \w{3}: (?P<eventClassKey>[^:]+): (?P<summary>.*)',

# ntsyslog windows msg
r"^(?P<component>.+)\[(?P<ntseverity>\D+)\] (?P<ntevid>\d+) (?P<summary>.*)",

# cisco msg with card indicator
r"%CARD-\S+:(SLOT\d+) %(?P<eventClassKey>\S+): (?P<summary>.*)",

# cisco standard msg
r"%(?P<eventClassKey>(?P<component>\S+)-\d-\S+): *(?P<summary>.*)",

# Cisco ACS
r"^(?P<ipAddress>\S+)\s+(?P<summary>(?P<eventClassKey>CisACS_\d\d_\S+)\s+(?P<eventKey>\S+)\s.*)",

# netscreen device msg
r"device_id=\S+\s+\[\S+\](?P<eventClassKey>\S+\d+):\s+(?P<summary>.*)\s+\((?P<originalTime>\d\d\d\d-\d\d-\d\d \d\d:\d\d:
\d)\)",

# NetApp
# [deviceName: 10/100/1000/e1a:warning]: Client 10.0.0.101 (xid 4251521131) is trying to access an unexported mount (fil
d 64, snapid 0, generation 6111516 and flags 0x0 on volume 0xc97d89a [No volume name available])
r"^\[[^:]+: (?P<component>[^:]+)[^\]]+\]: (?P<summary>.*)",

# unix syslog with pid
r"(?P<component>\S+)\[(?P<pid>\d+)\]:\s*(?P<summary>.*)",

# unix syslog without pid
r"(?P<component>\S+): (?P<summary>.*)",

# adtran devices
r"^(?P<deviceModel>[^\[]+)\[(?P<deviceManufacturer>ADTRAN)\]:(?P<component>[^\|]+\|\d+\|\d+)\|(?P<summary>.*)",

"SyslogProcessing.py" 299 lines --21%--                                                        64,0-1          9
```

Figure 19: SyslogProcessing.py regular expressions to match syslog tags

The main body of SyslogProcessing.py starts by assigning values from the incoming event to Zenoss event class fields, as follows:

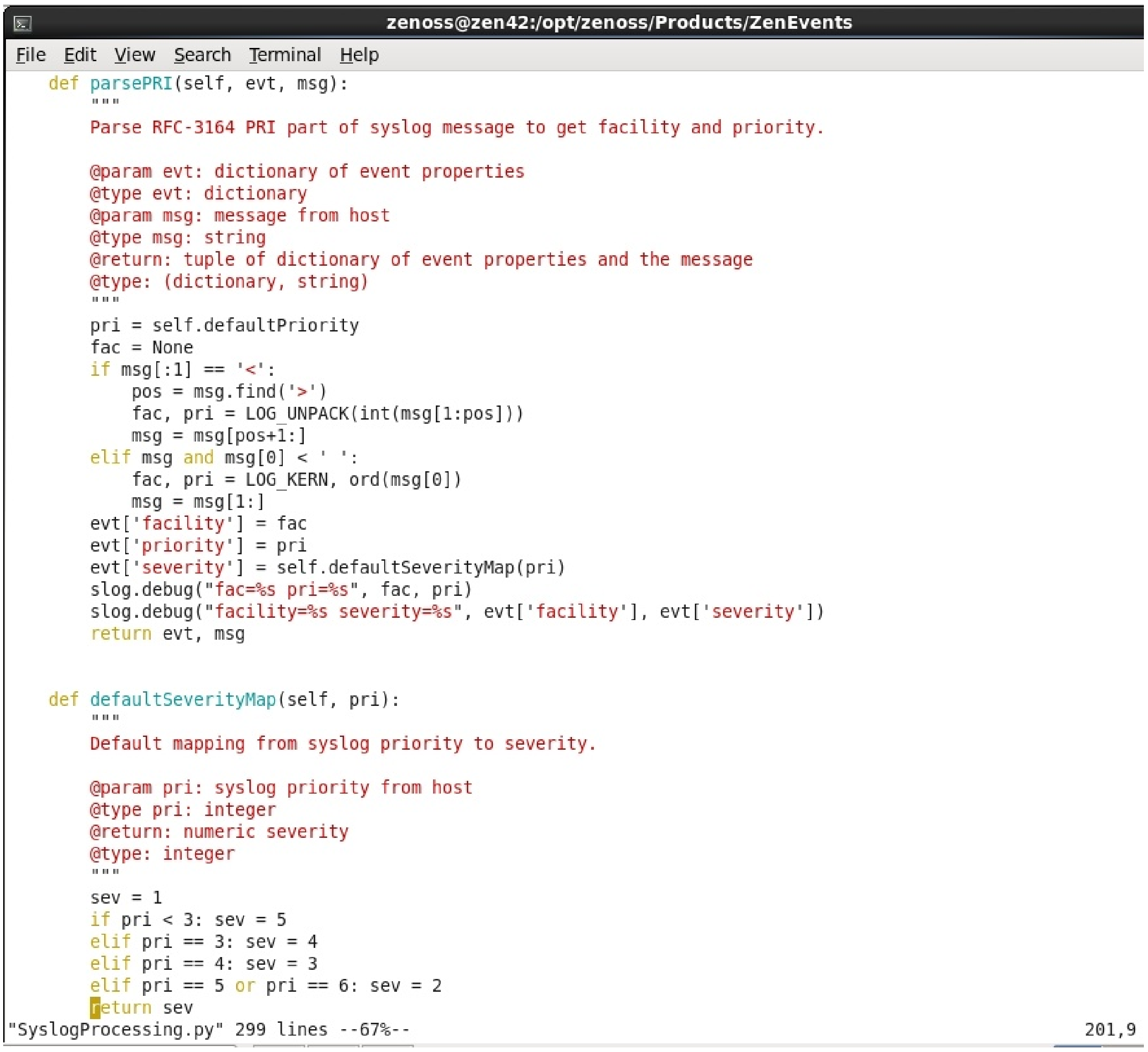
```
    def process(self, msg, ipaddr, host, rtime):
        evt = dict(device=host,
                   ipAddress=ipaddr,
                   firstTime=rtime,
                   lastTime=rtime,
                   eventGroup='syslog')
```
At this stage, no account of duplicates is taken so the firstTime and lastTime fields are both set to the timestamp on the incoming event. Note that the Zenoss eventGroup field is hardcoded at this stage to *syslog* .


zenoss@zen42:/opt/zenoss/Products/ZenEvents
File Edit View Search Terminal Help

```
    def process(self, msg, ipaddr, host, rtime):
        """
        Process an event from syslog and convert to a Zenoss event

        @param msg: message from a remote host
        @type msg: string
        @param ipaddr: IP address of the remote host
        @type ipaddr: string
        @param host: remote host's name
        @type host: string
        @param rtime: time as reported by the remote host
        @type rtime: string
        """
        evt = dict(device=host,
                   ipAddress=ipaddr,
                   firstTime=rtime,
                   lastTime=rtime,
                   eventGroup='syslog')
        slog.debug("host=%s, ip=%s", host, ipaddr)
        slog.debug(msg)

        evt, msg = self.parsePRI(evt, msg)
        if evt['priority'] > self.minpriority: return

        evt, msg = self.parseHEADER(evt, msg)
        evt = self.parseTag(evt, msg)
        if evt:
            #rest of msg now in summary of event
            evt = self.buildEventClassKey(evt)
            evt['monitor'] = self.monitor
            self.sendEvent(evt)


"SyslogProcessing.py" 299 lines --41%--
```
Figure 20: SyslogProcessing.py process main routine

parsePRI is the Python function called to parse out the syslog priority and facility.

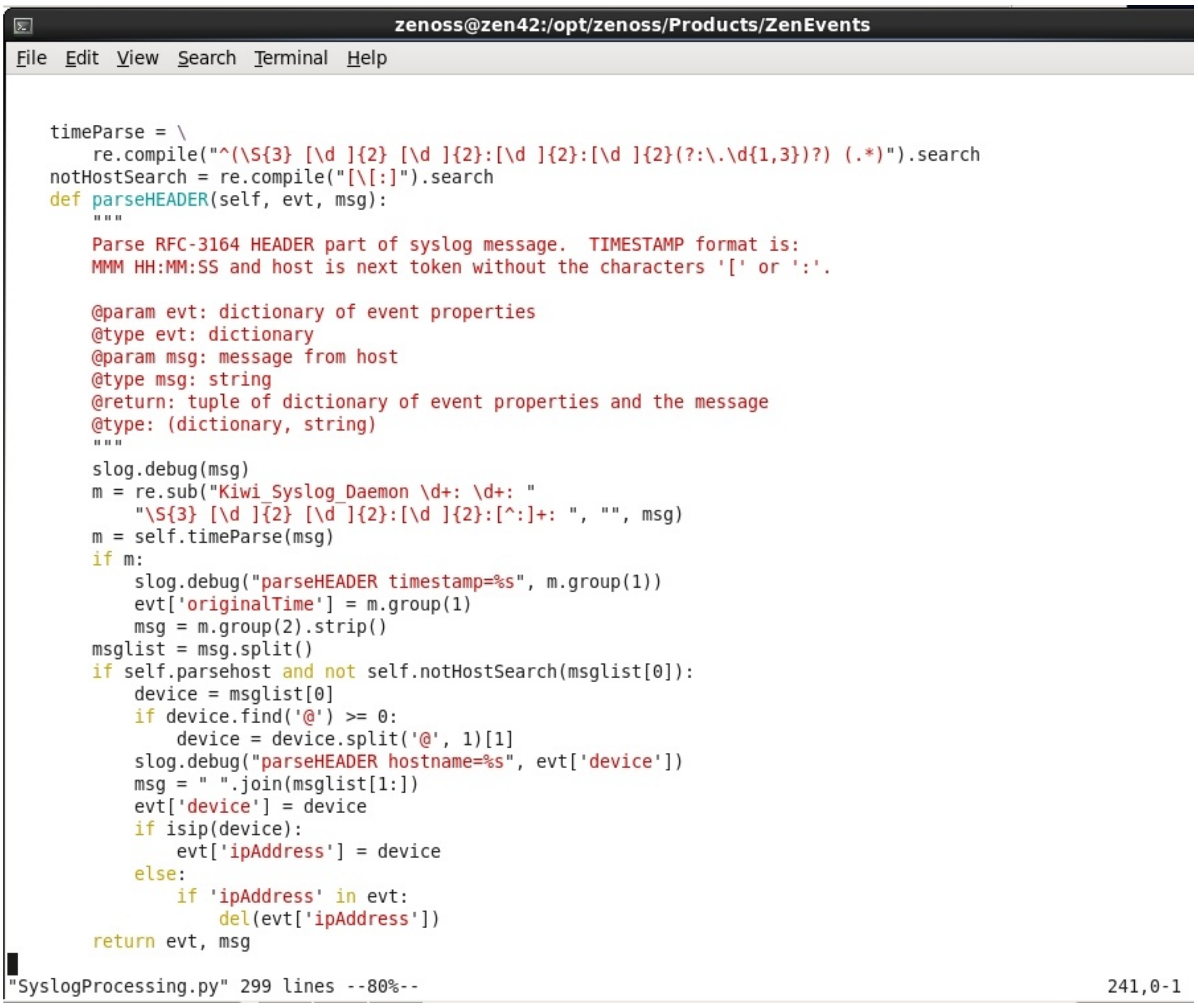
The *defaultSeverityMap* function is called from within the parsePRI function to set the severity field of the Zenoss event.


zenoss@zen42:/opt/zenoss/Products/ZenEvents
File Edit View Search Terminal Help

```
    def parsePRI(self, evt, msg):
        """
        Parse RFC-3164 PRI part of syslog message to get facility and priority.

        @param evt: dictionary of event properties
        @type evt: dictionary
        @param msg: message from host
        @type msg: string
        @return: tuple of dictionary of event properties and the message
        @type: (dictionary, string)
        """
        pri = self.defaultPriority
        fac = None
        if msg[:1] == '<':
            pos = msg.find('>')
            fac, pri = LOG_UNPACK(int(msg[1:pos]))
            msg = msg[pos+1:]
        elif msg and msg[0] < ' ':
            fac, pri = LOG_KERN, ord(msg[0])
            msg = msg[1:]
        evt['facility'] = fac
        evt['priority'] = pri
        evt['severity'] = self.defaultSeverityMap(pri)
        slog.debug("fac=%s pri=%s", fac, pri)
        slog.debug("facility=%s severity=%s", evt['facility'], evt['severity'])
        return evt, msg


    def defaultSeverityMap(self, pri):
        """
        Default mapping from syslog priority to severity.

        @param pri: syslog priority from host
        @type pri: integer
        @return: numeric severity
        @type: integer
        """
        sev = 1
        if pri < 3: sev = 5
        elif pri == 3: sev = 4
        elif pri == 4: sev = 3
        elif pri == 5 or pri == 6: sev = 2
        return sev
"SyslogProcessing.py" 299 lines --67%--                                 201,9
```

Figure 21: SyslogProcessing.py parsing of priority, facility and severity

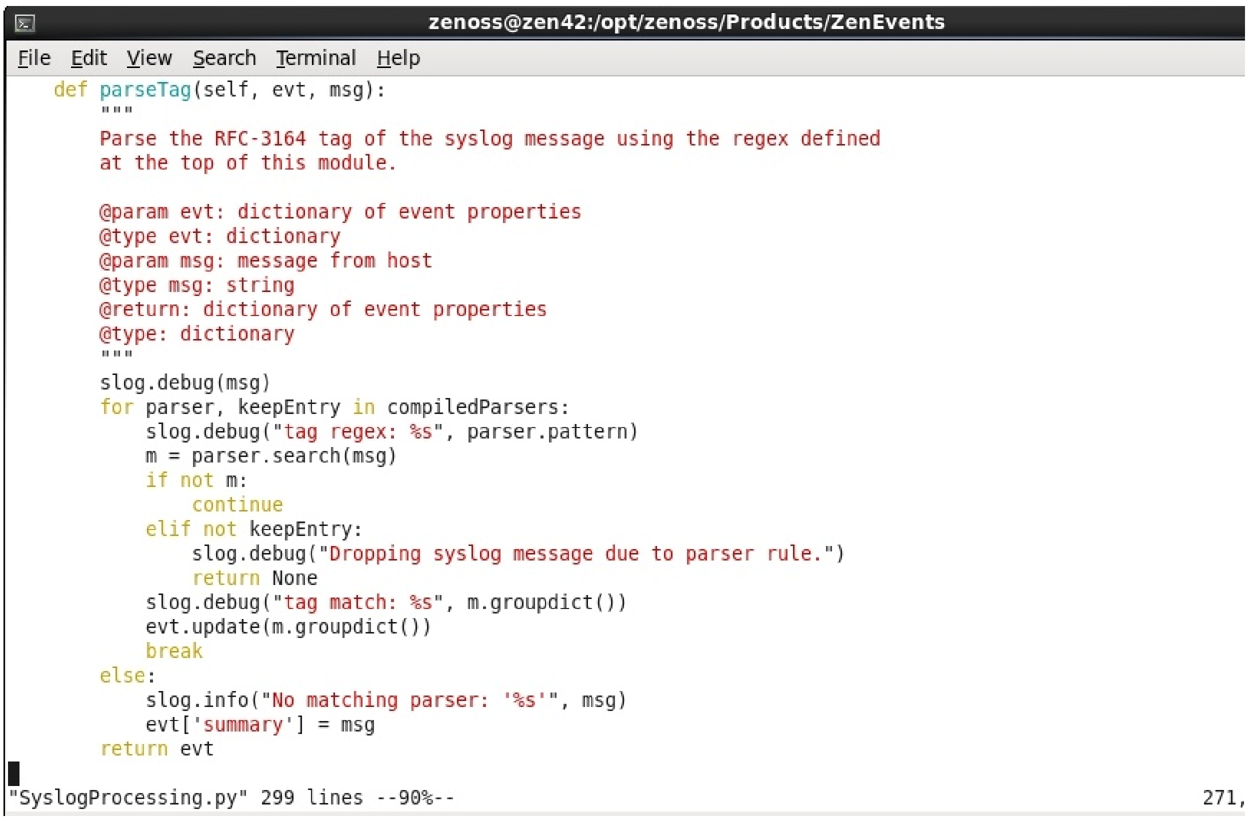
Next, the *parseHEADER* function is called to extract the timestamp and host name from the incoming event. The device and ipAddress fields of the Zenoss event are set at the end of this function.


zenoss@zen42:/opt/zenoss/Products/ZenEvents
File Edit View Search Terminal Help

```
    timeParse = \
        re.compile("^(\S{3} [\d ]{2} [\d ]{2}:[\d ]{2}:[\d ]{2}(?:\.\d{1,3})?) (.*)").search
    notHostSearch = re.compile("[\[:]").search
    def parseHEADER(self, evt, msg):
        """
        Parse RFC-3164 HEADER part of syslog message.  TIMESTAMP format is:
        MMM HH:MM:SS and host is next token without the characters '[' or ':'.

        @param evt: dictionary of event properties
        @type evt: dictionary
        @param msg: message from host
        @type msg: string
        @return: tuple of dictionary of event properties and the message
        @type: (dictionary, string)
        """
        slog.debug(msg)
        m = re.sub("Kiwi_Syslog_Daemon \d+: \d+: "
            "\S{3} [\d ]{2} [\d ]{2}:[\d ]{2}:[^:]+: ", "", msg)
        m = self.timeParse(msg)
        if m:
            slog.debug("parseHEADER timestamp=%s", m.group(1))
            evt['originalTime'] = m.group(1)
            msg = m.group(2).strip()
        msglist = msg.split()
        if self.parsehost and not self.notHostSearch(msglist[0]):
            device = msglist[0]
            if device.find('@') >= 0:
                device = device.split('@', 1)[1]
            slog.debug("parseHEADER hostname=%s", evt['device'])
            msg = " ".join(msglist[1:])
            evt['device'] = device
            if isip(device):
                evt['ipAddress'] = device
            else:
                if 'ipAddress' in evt:
                    del(evt['ipAddress'])
        return evt, msg

"SyslogProcessing.py" 299 lines --80%--                                    241,0-1
```

Figure 22: SyslogProcessing.py processing the header information

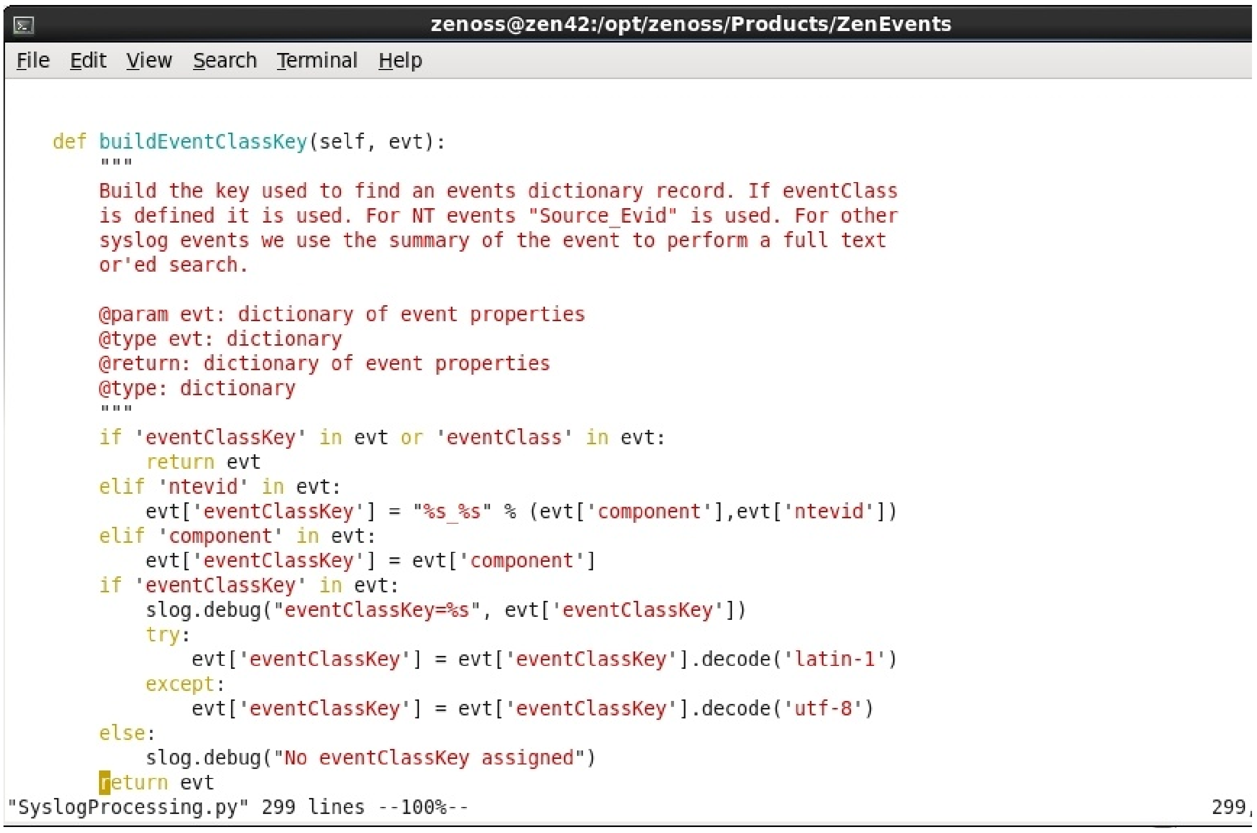
The *parseTag* function is called to parse out the syslog tag, using the regex expressions at the beginning of the file. If no match exists then a *parseTag failed* message is logged. The end of the function returns the remainder of the incoming message in the Zenoss event summary field.


zenoss@zen42:/opt/zenoss/Products/ZenEvents
File Edit View Search Terminal Help

```
    def parseTag(self, evt, msg):
        """
        Parse the RFC-3164 tag of the syslog message using the regex defined
        at the top of this module.

        @param evt: dictionary of event properties
        @type evt: dictionary
        @param msg: message from host
        @type msg: string
        @return: dictionary of event properties
        @type: dictionary
        """
        slog.debug(msg)
        for parser, keepEntry in compiledParsers:
            slog.debug("tag regex: %s", parser.pattern)
            m = parser.search(msg)
            if not m:
                continue
            elif not keepEntry:
                slog.debug("Dropping syslog message due to parser rule.")
                return None
            slog.debug("tag match: %s", m.groupdict())
            evt.update(m.groupdict())
            break
        else:
            slog.info("No matching parser: '%s'", msg)
            evt['summary'] = msg
        return evt

"SyslogProcessing.py" 299 lines --90%--                                    271,
```

Figure 23: SyslogProcessing.py parsing the syslog tag

The crux of event processing in Zenoss is to derive an **eventClassKey** – this is done with the *buildEventClassKey* function.


zenoss@zen42:/opt/zenoss/Products/ZenEvents
File Edit View Search Terminal Help

```
    def buildEventClassKey(self, evt):
        """
        Build the key used to find an events dictionary record. If eventClass
        is defined it is used. For NT events "Source_Evid" is used. For other
        syslog events we use the summary of the event to perform a full text
        or'ed search.

        @param evt: dictionary of event properties
        @type evt: dictionary
        @return: dictionary of event properties
        @type: dictionary
        """
        if 'eventClassKey' in evt or 'eventClass' in evt:
            return evt
        elif 'ntevid' in evt:
            evt['eventClassKey'] = "%s_%s" % (evt['component'],evt['ntevid'])
        elif 'component' in evt:
            evt['eventClassKey'] = evt['component']
        if 'eventClassKey' in evt:
            slog.debug("eventClassKey=%s", evt['eventClassKey'])
            try:
                evt['eventClassKey'] = evt['eventClassKey'].decode('latin-1')
            except:
                evt['eventClassKey'] = evt['eventClassKey'].decode('utf-8')
        else:
            slog.debug("No eventClassKey assigned")
        return evt
"SyslogProcessing.py" 299 lines --100%--                                     299,
```

Figure 24: SyslogProcessing.py determining the EventClassKey

Note that if the event has the **component** field populated then that is used as the eventClassKey after checking for a pre-existing **eventClassKey** and for an **ntevid** field.

5 Zenoss processing of Windows event logs

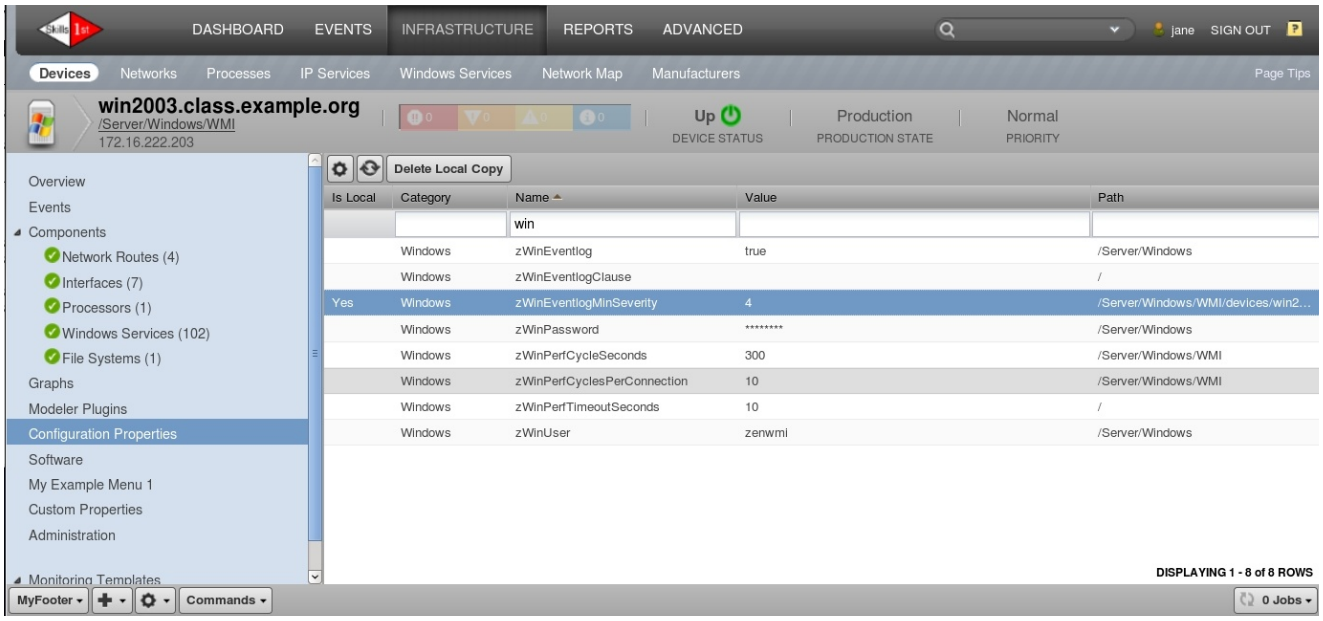
5.1 Management using the WMI protocol

Zenoss prior to version 4 shipped Windows monitoring as part of the Core code. Zenoss 4 ships Windows support with the ZenPacks.zenoss.WindowsMonitor ZenPack which has a prerequisite of ZenPacks.zenoss.PySamba. These are Zenoss-provided Core ZenPacks.

If a Windows device supports SNMP then it is perfectly possible to use that protocol, especially as most Windows SNMP agents also support the Host Resources MIB so some system information is available in addition to the standard MIB-2 network type information.

The Zenoss Windows ZenPacks introduce the /Server/Windows/WMI device class which has both WMI modeler plugins and WMI performance templates associated with it. Target devices should be added to this class or subclasses thereof. This allows monitoring using the Windows Management Instrumentation (WMI) protocol. A userid and password need to be configured on target hosts to permit WMI access from the

Zenoss server; it also means that firewalls both on the Windows devices and any intervening network firewalls, must be configured to permit WMI access. The Zenoss Server must then be configured with matching Windows zProperties (zWinUser and zWinPassword) for the target devices / device classes. There are a few other Windows-specific Configuration Properties - see Figure 25. These zProperties can be changed for a device class or for a specific device.



Is Local	Category	Name	Value	Path
		win		
	Windows	zWinEventlog	true	/Server/Windows
	Windows	zWinEventlogClause		/
Yes	Windows	zWinEventlogMinSeverity	4	/Server/Windows/WMI/devices/win2...
	Windows	zWinPassword	********	/Server/Windows
	Windows	zWinPerfCycleSeconds	300	/Server/Windows/WMI
	Windows	zWinPerfCyclesPerConnection	10	/Server/Windows/WMI
	Windows	zWinPerfTimeoutSeconds	10	/
	Windows	zWinUser	zenwmi	/Server/Windows

Figure 25 zProperties for Windows targets

ZenPacks.zenoss.WindowsMonitor provides three new daemons:

● zenwin	monitors windows services using WMI
● zenwinperf	collects performance data using the WMI protocol
● zeneventlog	retrieves Windows event log information using WMI

The three *zWinPerf...* zProperties fine-tune the configuration of the zenwinperf daemon; the *zWinEventlog* parameter must be *True* to collect Windows events from a target device.

The *zWinEventlogMinSeverity* property defines the least serious severity events that will be forwarded from Windows to Zenoss. Note that the numeric denotation of windows event severities and their names and support currency, have changed over the life of Zenoss. See Table 4.1 on page 42 for current valid severities. Also note that if you change this parameter you are presented with a list of Zenoss severities, not Windows-style severities; again refer to the earlier table for a translation. If you want to include all Windows severities, including security audit failure (5), you need to select the *Clear* severity in the dropdown menu when changing zWinEventlogMinSeverity.

The *zWinEventlogClause* was introduced during the lifetime of Zenoss 3 to help filter events from Windows devices. Consult the Zenoss Core 4 Administrators Guide, chapter

6.6.6 for documentation and examples. This parameter is rather obtuse. Fundamentally a Windows Query Language (WQL) query is constructed to be run by zeneventlog:

```
SELECT * FROM __InstanceCreationEvent
WHERE TargetInstance ISA 'Win32_NTLogEvent'
   AND TargetInstance.EventType <= zWinEventlogMinSeverity
```

Any zWinEventlogClause is logically AND'ed with this WQL; thus if you want to ONLY see events with event id of 528 and 529 (Successful logon and Logon failure), configure zWinEventlogClause to be:

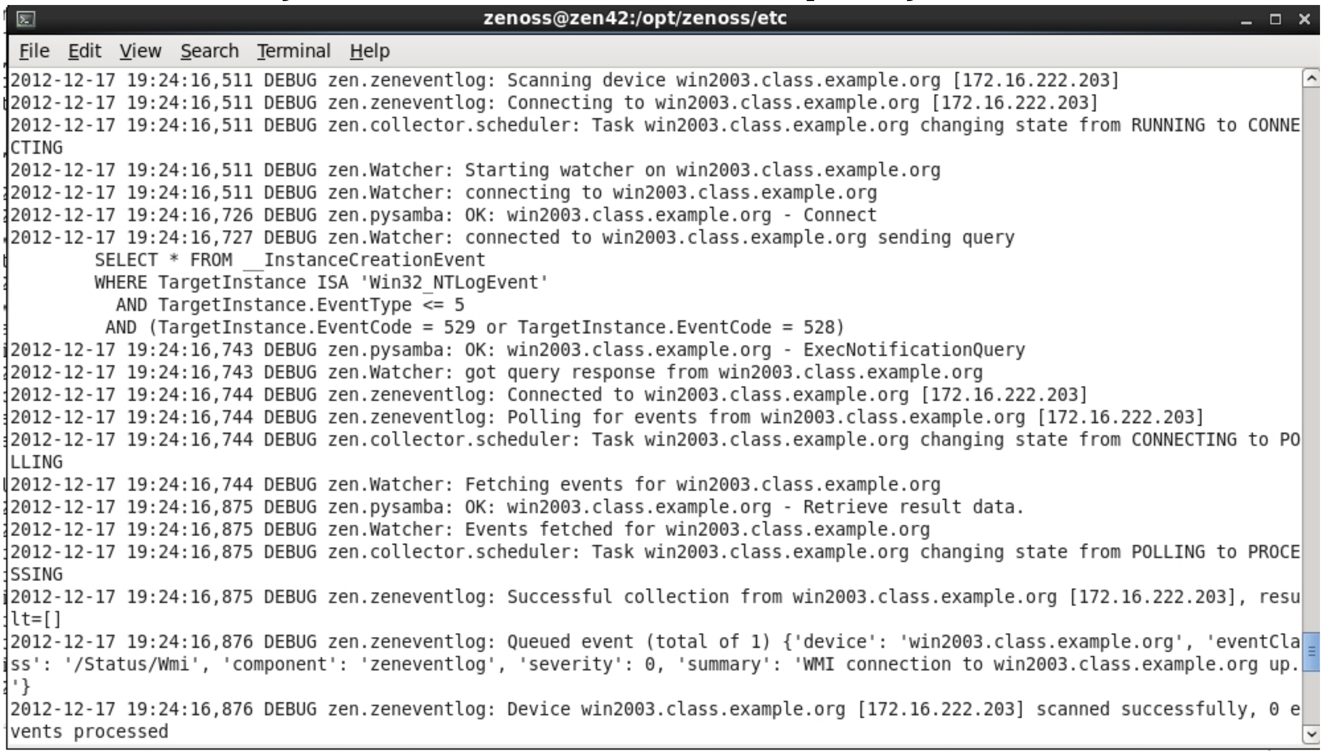
```
(TargetInstance.EventCode = 529 or TargetInstance.EventCode = 528)
```

Strictly, the zeneventlog daemon polls target Windows systems for events and parses them in to Zenoss-style events. Typically, the Source field on the Windows event maps to the component field in the Zenoss event; the Zenoss eventClassKey is composed of the Windows <*Source*>_<*Event ID*> (eg. Perflib_2003); the Zenoss eventGroup becomes the Windows log file name (Application, Security, etc) and the Windows Event ID is mapped to the Zenoss ntevid field.

To see the workings of zeneventlog, change the logging level to Debug (10), restart the daemon and inspect *$ZENHOME/log/zeneventlog.log*.

A good way to see the WQL statement being used is to run zeneventlog as a one-off command in the foreground:

```
zeneventlog run -v 10 -d win2003.class.example.org
```


zenoss@zen42:/opt/zenoss/etc
File Edit View Search Terminal Help

```
2012-12-17 19:24:16,511 DEBUG zen.zeneventlog: Scanning device win2003.class.example.org [172.16.222.203]
2012-12-17 19:24:16,511 DEBUG zen.zeneventlog: Connecting to win2003.class.example.org [172.16.222.203]
2012-12-17 19:24:16,511 DEBUG zen.collector.scheduler: Task win2003.class.example.org changing state from RUNNING to CONNE
CTING
2012-12-17 19:24:16,511 DEBUG zen.Watcher: Starting watcher on win2003.class.example.org
2012-12-17 19:24:16,511 DEBUG zen.Watcher: connecting to win2003.class.example.org
2012-12-17 19:24:16,726 DEBUG zen.pysamba: OK: win2003.class.example.org - Connect
2012-12-17 19:24:16,727 DEBUG zen.Watcher: connected to win2003.class.example.org sending query
        SELECT * FROM __InstanceCreationEvent
        WHERE TargetInstance ISA 'Win32_NTLogEvent'
          AND TargetInstance.EventType <= 5
         AND (TargetInstance.EventCode = 529 or TargetInstance.EventCode = 528)
2012-12-17 19:24:16,743 DEBUG zen.pysamba: OK: win2003.class.example.org - ExecNotificationQuery
2012-12-17 19:24:16,743 DEBUG zen.Watcher: got query response from win2003.class.example.org
2012-12-17 19:24:16,744 DEBUG zen.zeneventlog: Connected to win2003.class.example.org [172.16.222.203]
2012-12-17 19:24:16,744 DEBUG zen.zeneventlog: Polling for events from win2003.class.example.org [172.16.222.203]
2012-12-17 19:24:16,744 DEBUG zen.collector.scheduler: Task win2003.class.example.org changing state from CONNECTING to PO
LLING
2012-12-17 19:24:16,744 DEBUG zen.Watcher: Fetching events for win2003.class.example.org
2012-12-17 19:24:16,875 DEBUG zen.pysamba: OK: win2003.class.example.org - Retrieve result data.
2012-12-17 19:24:16,875 DEBUG zen.Watcher: Events fetched for win2003.class.example.org
2012-12-17 19:24:16,875 DEBUG zen.collector.scheduler: Task win2003.class.example.org changing state from POLLING to PROCE
SSING
2012-12-17 19:24:16,875 DEBUG zen.zeneventlog: Successful collection from win2003.class.example.org [172.16.222.203], resu
lt=[]
2012-12-17 19:24:16,876 DEBUG zen.zeneventlog: Queued event (total of 1) {'device': 'win2003.class.example.org', 'eventCla
ss': '/Status/Wmi', 'component': 'zeneventlog', 'severity': 0, 'summary': 'WMI connection to win2003.class.example.org up.
'}
2012-12-17 19:24:16,876 DEBUG zen.zeneventlog: Device win2003.class.example.org [172.16.222.203] scanned successfully, 0 e
vents processed
```

Figure 26Partial output from zeneventlog run -v 10 -d win2003.class.example.org showing WQL statement

Many Windows event log events are automatically mapped to event classes but they may have a low severity (such as *Debug*) and they may have their zEventAction event zProperty set to *history* so that they do not appear in the status table of the events database.

5.2 Management of Windows systems using syslog

There is also a syslog utility available for Windows systems from Datagram Consulting at http://syslogserver.com . The client utility is SyslogAgent and is made available under the GNU license. Syslog server utilities for Windows are also available as chargeable products. This means that Windows event logs can also be collected with the zensyslog daemon.

Note that the Syslog agent is capable of being configured to monitor Windows application log files, in addition to the standard Windows event logs. When monitoring the standard event logs, there are better filtering capabilities with Syslog then with zeneventlog.

6 Event Mapping

Zenoss events are categorised into a hierarchy of event **Classes**, many of which are defined out-of-the-box but which can easily be modified or augmented. The process of **Event Class Mapping** is about associating an incoming event with a particular Zenoss Event Class (setting its *eventClass* field) and, potentially, modifying other fields of that event by using an event **transform**.

Event classes and subclasses are treated identically from the point-of-view of event class mapping. The class hierarchy can be useful in that event **context,** as implemented by event zProperties (zEventSeverity, zEventAction, zEventClearClasses), follows the normal rules for object inheritance – if zEventAction is set to *drop* on the event class */Ignore* , then any subclasses of */Ignore* will also inherit that property.

Notable out-of-the-box event zProperties are that /Ignore classes and subclasses drop incoming events totally; /Archive classes and subclasses automatically set the eventState field to Closed.

Most event classes have one or more mappings associated with them – these are known as **instances**. Note that an event does **not have** to have any mappings associated, in which case an event of that class will only appear in an Event Console if the daemon that generates the event, assigns the event class at that time (*/Perf* events may well come into this category, for example). Out-of-the-box event class mappings are defined in *$ZENHOME/Products/ZenModel/data/events.xml* . They can be inspected from the Zenoss GUI by selecting the *EVENTS -> Event Classes* menu.

Most out-of-the-box event class mappings simply match on the **eventClassKey** field which is populated by the native event parsing mechanism (such as zensyslog, zeneventlog, zentrap). These mechanisms may generate several different events with the same eventClassKey field; thus other techniques are needed to distinguish between such events and potentially to separate them into different event classes.

The sequence number in an event mapping gives the order in which mappings are tested against the incoming event - lowest numbers are tested first. Depending on which mapping actually matches (if any) will determine the resulting eventClass of the event.

6.1 Working with event classes and event mappings

Events are organised in an object-oriented hierarchy; thus attributes assigned to a “parent” event class are inherited by a “child” event subclass.

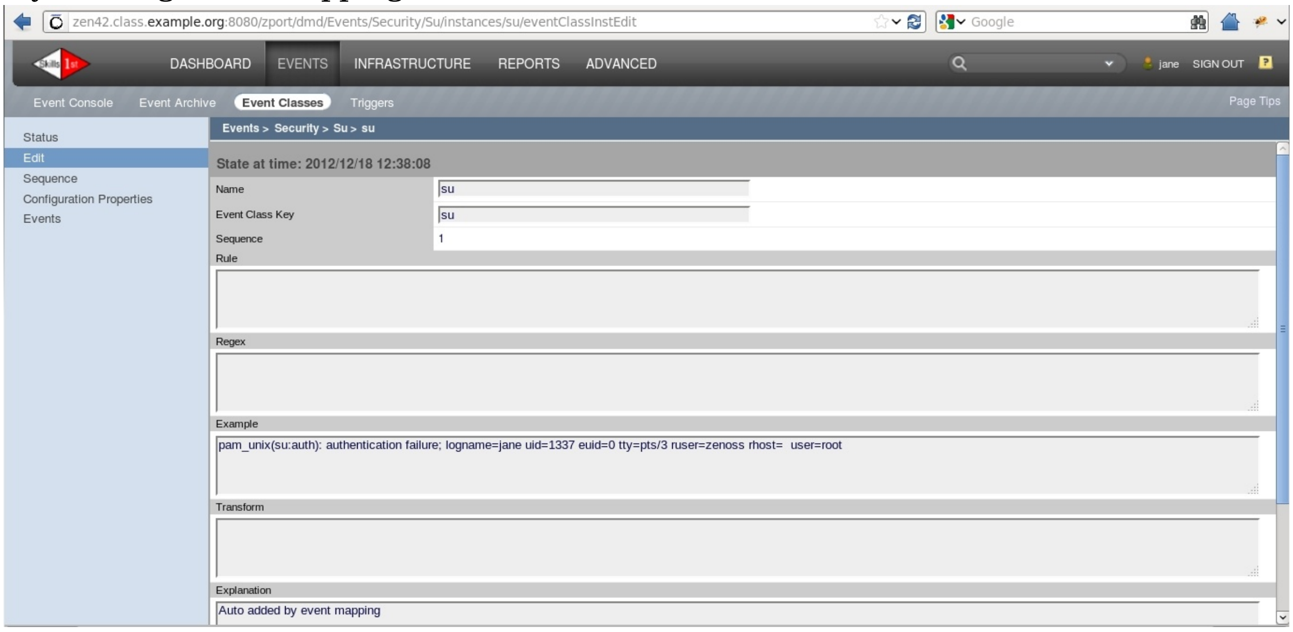
New event classes can be defined by navigating to an event class and using the dropdown menu alongside *SubClasses* to *Add New Organizer*. The name supplied is the name of the new event class. For example, drill down to the */Security* event class and create a new subclass called *Su*.

Any event which does not map to an event class is the given the class of */Unknown*. The simplest way to map such an event is to start from an existing event in the Event Console. The following scenario explains this, creating a new event class mapping called *su* which maps an incoming event to the event class */Security/Su*.

1. Generate a syslog “authentication failure” event at the Zenoss system.
2. Open an Event Console that shows the event and inspect its details.
3. Select the event and use the Reclassify Event icon at the top of the console. Select your new */Security/Su* class from the dropdown list. You should be shown the event class mapping panel. Click the lefthand *Edit* menu.
4. You should find that the name of the new event class mapping is set to s*u* and the Event Class Key is set to s*u* (note lower case s in both cases). The *eventClassKey* field is actually derived from the *component* field of the incoming event in SyslogProcessing.py (around line 289). The summary field of the event should have been copied into the mapping Example box.
5. Add a text string to the Explanation box such as “Auto added by event mapping”.
6. Add a text string to the Resolution box such as “This is a dummy resolution”.
7. Open a Zenoss GUI window that shows all Su events (you may find it useful to have several browser tabs open to focus on different aspects of the Zenoss GUI). Select all the Su events and *Close* them.
8. Generate a new Su event.
9. Check the details of the new event in the Event Console. The event should have mapped to eventClass */Security/Su* . The severity should be *Info* (blue). The

details of the event should show the eventClassMapping field set to */Security/Su/su* .

Any existing event mapping can be modified in a similar fashion.



State at time: 2012/12/18 12:38:08

Name: su

Event Class Key: su

Sequence: 1

Rule

Regex

Example

pam_unix(su:auth): authentication failure; logname=jane uid=1337 euid=0 tty=pts/3 ruser=zenoss rhost= user=root

Transform

Explanation

Auto added by event mapping

Figure 27: Edit dialogue for event class mapping

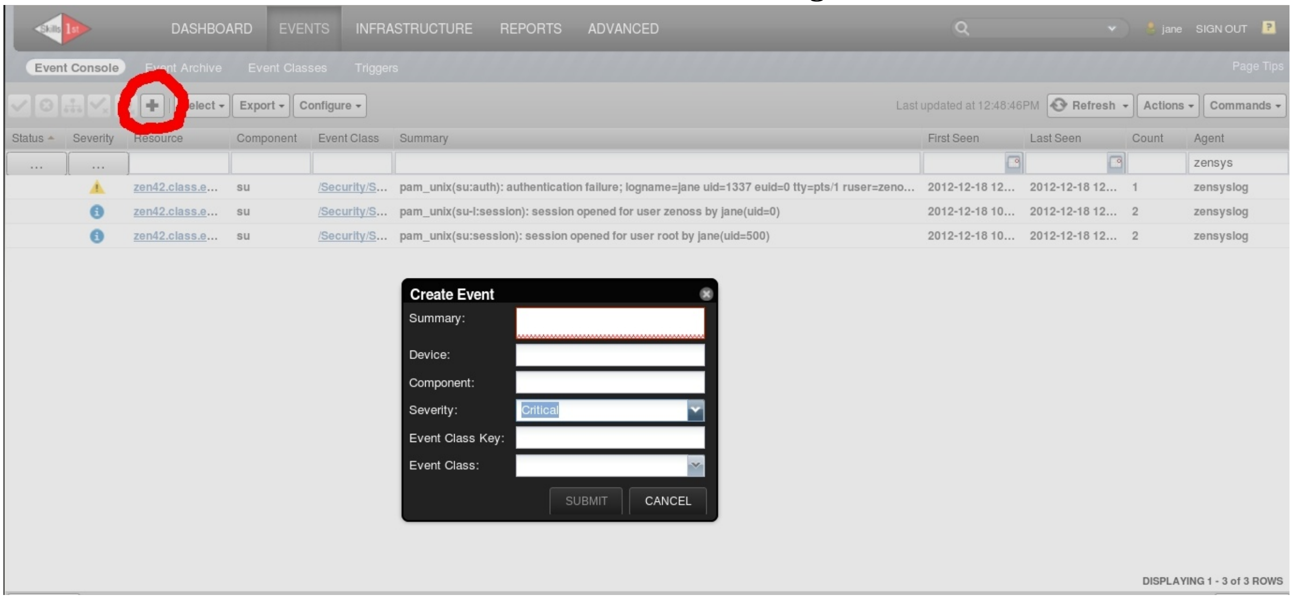
Whenever you change an event mapping, it is advisable to clear any existing events of that category before testing the new configuration.

When you are working with event mappings, don't forget the *Event* menu which filters an Event Console by Event Class.

It is useful to refer to event classes using the **breadcrumb** path seen at the top of a page, such as */Events/Security/Su* .

6.1.1 Generating test events

Test events can be created from the Event Console using the "+" icon.



Status	Severity	Resource	Component	Event Class	Summary	First Seen	Last Seen	Count	Agent
...	...								zensys
	⚠	zen42.class.e...	su	/Security/S...	pam_unix(su:auth): authentication failure; logname=jane uid=1337 euid=0 tty=pts/1 ruser=zeno...	2012-12-18 12...	2012-12-18 12...	1	zensyslog
	ℹ	zen42.class.e...	su	/Security/S...	pam_unix(su-l:session): session opened for user zenoss by jane(uid=0)	2012-12-18 10...	2012-12-18 12...	2	zensyslog
	ℹ	zen42.class.e...	su	/Security/S...	pam_unix(su:session): session opened for user root by jane(uid=500)	2012-12-18 10...	2012-12-18 12...	2	zensyslog

Create Event

Summary:	
Device:	
Component:	
Severity:	Critical
Event Class Key:	
Event Class:	

SUBMIT CANCEL

DISPLAYING 1 - 3 of 3 ROWS

Figure 28: Dialogue to create a test event

Alternatively, the command line *zensendevent* can be used (you should ensure you are the zenoss user). This takes parameters:

-d	device
-p	component
-k	eventClassKey
-s	severity
-c	eventClass
-y	eventKey
-i	IP address
-h	help
-o	<field> = <value> (for any other attribute; can have multiple -o)
--monitor	collector this event came from
--port=PORT	default is 8081
--server=SERVER	default is localhost
--auth=AUTH	default is admin:zenoss

- The remainder of the line after these options is used for the summary field (strictly the Message field in the GUI dialogue populates the event *summary* field)

The core-autodeploy script delivered with Zenoss 4.2.3 has new functionality to increase security on a Zenoss installation. For many years the Zenoss user of **admin** with a password of **zenoss** has been configured as standard. The new installation script changes this, generating a robust password which is stored in several configuration files in *$ZENHOME/etc*, including *global.conf* and *hubpasswd*.

zensendevent is a standalone Python utility in *$ZENHOME/bin* that communicates with the **zenhub** daemon. Note in the usage description above, that the default *--auth* parameter value is *admin:zenoss*; typically this means that zensendevent commands will fail with an Unauthorized message unless the *--auth* parameter is added with the correct user and password, found in *$ZENHOME/etc/hubpasswd.*

A discussion on modifying zensendevent to automatically look-up the correct authentication parameters, can be found on the Zenoss wiki at http://wiki.zenoss.org/Zensendevent_in_Zenoss_4.2.3
The code is supplied in Appendix A.

6.2 Regex in event mappings

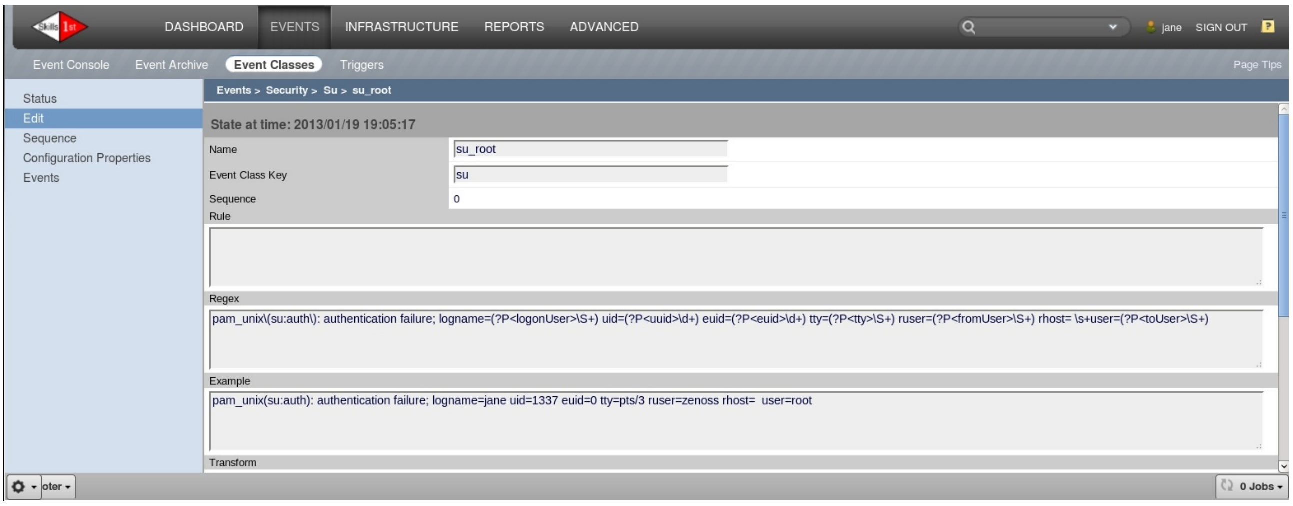
The **Regex** element of an event class mapping can be used to parse the summary field of the incoming event, which is presented by the parsing daemon (zensyslog, zeneventlog, zentrap). The Regex element uses the Python format for regular expressions and can use the Python **named group** syntax to not only check for literal strings but also to define regular expressions for variable parts of a string, and associate that variable part with a name. Variable parts of the string are captured into Python **named groups** – this means that:

- You can have one expression match lots of similar but different incoming events
- The variable part (typically between the *(?P and \S+)*) can be passed to the rest of the event processing mechanism as a named field of the event.
- Thus, in the product-shipped *dropbear* event mapping for */Security/Login/Fail*, the Regex is as follows:
  - exit before auth \(user '(?P<eventKey>\S+)', (?P<failures>\S+) fails \): Max auth tries reached
  - (?P<eventKey>\S+) will parse the characters after *user '* upto the next single quote and place that string into the eventKey field of the event. Similarly (?P<failures>\S+) will parse the string that follows a comma and space and is ended by space and *fails*, into a new event attribute called *failures*.
  - Matching the literal string representing a bracket requires the backslash escape or the bracket will be interpreted as a metacharacter.
  - The rest of the event summary must match the literal text in the Regex; however, other text can appear beyond the end after *tries reached* .
  - The Example box should shows a sample event summary that is matched by the regular expression in the Regex box. If you attempt to *Save* a regex that does not match the example, the regex field will be shown in red.

For more information on Python regular expressions, see http://docs.python.org/2/library/re.html .

See Figure 29 for an example of a more specific mapping, *su_root*, for the event class */Security/Su*. The regex is used to ensure that the summary has the string *pam_unix(su:auth): authentication failure;* followed by some fixed and some variable elements.

```
pam_unix\(su:auth\): authentication failure; logname=(?P<logonUser>\S+)
uid=(?P<uuid>\d+) euid=(?P<euid>\d+) tty=(?P<tty>\S+) ruser=(?
P<fromUser>\S+) rhost=\s+user=(?P<toUser>\S+)
```



Events > Security > Su > su_root

State at time: 2013/01/19 19:05:17

Name	su_root
Event Class Key	su
Sequence	0
Rule	

Regex

```
pam_unix\(su:auth\): authentication failure; logname=(?P<logonUser>\S+) uid=(?P<uuid>\d+) euid=(?P<euid>\d+) tty=(?P<tty>\S+) ruser=(?P<fromUser>\S+) rhost= \s+user=(?P<toUser>\S+)
```

Example

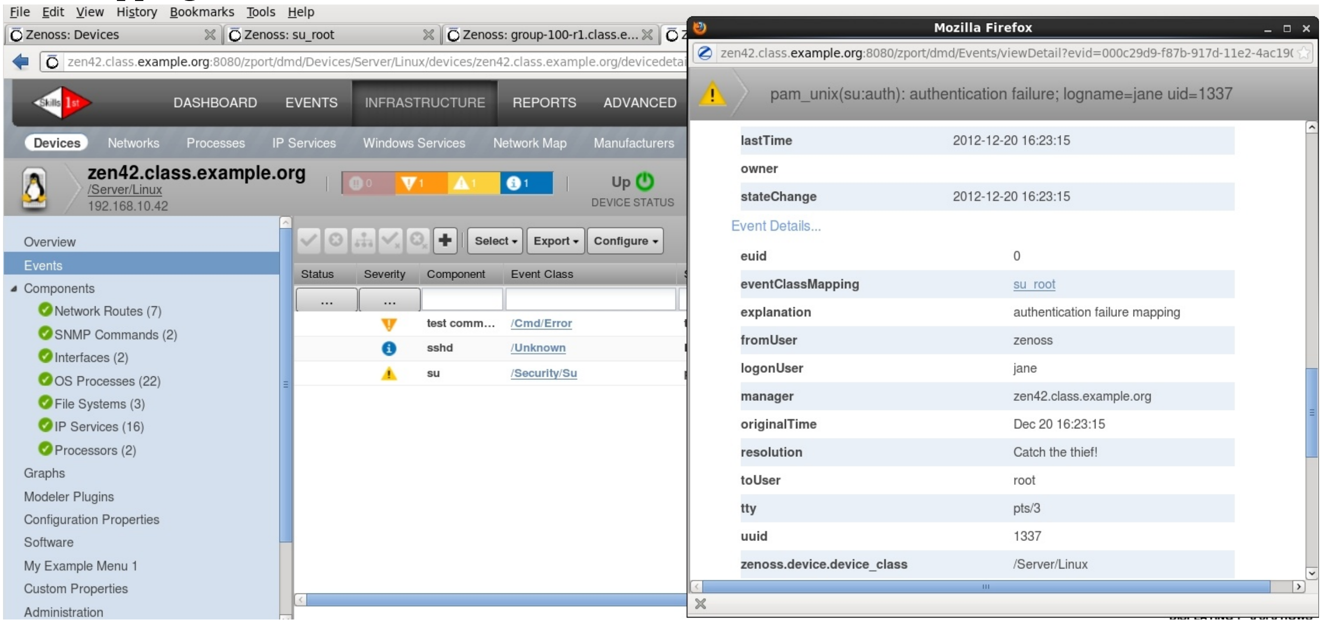
```
pam_unix(su:auth): authentication failure; logname=jane uid=1337 euid=0 tty=pts/3 ruser=zenoss rhost=  user=root
```

Transform

Figure 29: Event mapping dialogue with Regex for authentication failure

The event summary field can be parsed to generate new, user-defined fields for the event which will be shown in the details of the event and can be used in any subsequent event transforms.

Additionally, the Configuration Property of *zEventSeverity* has been set to *Warning* for this mapping.



zen42.class.example.org:8080/zport/dmd/Events/viewDetail?evid=000c29d9-f87b-917d-11e2-4ac19(

pam_unix(su:auth): authentication failure; logname=jane uid=1337

lastTime	2012-12-20 16:23:15
owner	
stateChange	2012-12-20 16:23:15

Event Details...

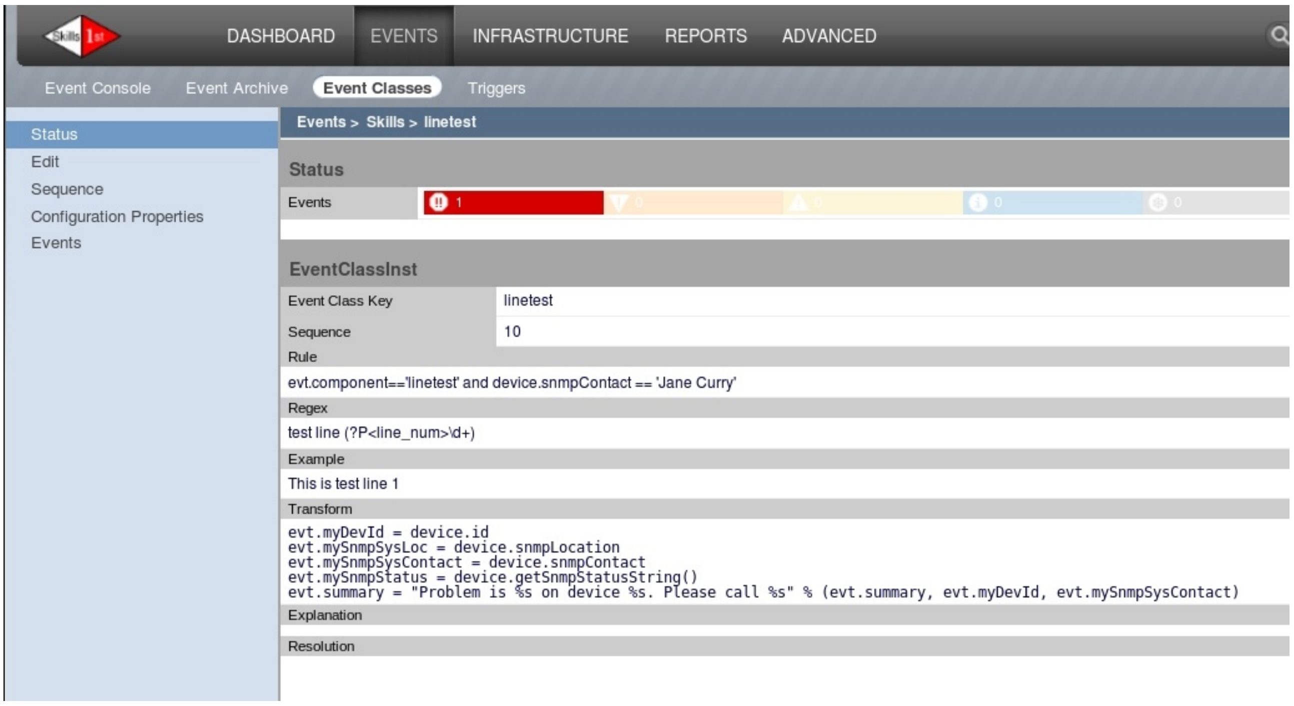
euid	0
eventClassMapping	su_root
explanation	authentication failure mapping
fromUser	zenoss
logonUser	jane
manager	zen42.class.example.org
originalTime	Dec 20 16:23:15
resolution	Catch the thief!
toUser	root
tty	pts/3
uuid	1337
zenoss.device.device_class	/Server/Linux

Figure 30Event details for authentication failure event showing new event fields created by the regex

The Regex element is **only** used if both the eventClassKey and the Rule (if any) are satisfied. If the Rule fails, the Regex will not be tested, nor will any named group, user-defined fields be generated. If a Rule does **not** exist and the Regex does not match, the user-defined fields will not be generated and the event class mapping to this event class will fail. No event transforms will take place. If a Rule **does** exist and is satisfied but the Regex fails then any user-defined fields will **not** be generated but the event class mapping **will** be successful and any mapping transform **will** take place.

6.3 Rules in event mappings

The **Rule** element of an event class mapping uses Python expressions to test any instantiated field of the incoming event against a value. Expressions can be complex including Python method calls and logical ANDs and ORs. The default event fields that are defined, are given in Appendix D3 of the Zenoss Core 4 Administration Guide. **Note** that some of these fields are **not** actually available at event mapping time – notably **evid, stateChange, count, dedupid, firstTime, lastTime** and **eventClassMapping** .



EventClassInst

Event Class Key	linetest
Sequence	10

Rule

evt.component=='linetest' and device.snmpContact == 'Jane Curry'

Regex

test line (?P<line_num>\d+)

Example

This is test line 1

Transform

```
evt.myDevId = device.id
evt.mySnmpSysLoc = device.snmpLocation
evt.mySnmpSysContact = device.snmpContact
evt.mySnmpStatus = device.getSnmpStatusString()
evt.summary = "Problem is %s on device %s. Please call %s" % (evt.summary, evt.myDevId, evt.mySnmpSysContact)
```

Explanation

Resolution

Figure 31: Event mapping linetest, showing complex Rule testing event and device attributes

The Rule element can also use Python expressions to test for values of attributes of the **device** that generated the event. Some of the methods and attributes that are available for devices are documented in Appendix D2 of the Zenoss Core 4 Administration Guide, under the section on TALES expressions (**T**emplate **A**ttribute

Language **E**xpression **S**yntax is part of Zope. Zope is the application server that Zenoss is built on).

The Rule element will only be used if the *eventClassKey* field in the mapping has achieved a match with the incoming event. After that, if a Rule exists, it must be satisfied before this mapping (and hence class) is applied.

6.4 Other elements of event mappings

The **Example** element of an event class mapping is a sample string that is useful when constructing a Regex. The Regex will turn red if the Regex does not match the Example string when the *Save* button is used.

The **Explanation** and **Resolution** elements of an event class mapping are strings that can be configured to provide further information to Zenoss users. They appear in the event detail. **Note** that these elements can only be literal strings; they cannot use either standard or user-defined fields from the event.

The combination of eventClassKey, Rule and Regex determine the event class that will be associated with the incoming event and what transforms (if any) will take place. There may still be multiple combinations of these that satisfy any given incoming event. If so, the *Sequence* menu is used to decide the precedence of evaluation of matching event mappings. The mappings will be tested from the lowest to the highest sequence number. Once a match is found, any subsequent mappings (with higher sequence numbers) will be ignored. Generally, a mapping with more specific matching criteria will have a lower sequence number.

In the examples above for the */Security/Su* class, the generic *su* mapping has sequence number 1 and the more specific *su_root* mapping has sequence 0.

A particular example of event mappings that use sequence numbers, is the event class mapping called **defaultmapping** which must have an eventClassKey of *defaultmapping* . There are at least 6 mappings, all called *defaultmapping* , out-of-the-box. Each maps to a different class. A default mapping is a special case that is used by the event mapping process if no match can be found for the eventClassKey field (note that if the eventClassKey field does not exist then no mapping at all will be applied). In the case where an eventClassKey match is not found, the mapping process re-evaluates looking for a match with the special eventClassKey of *defaultmapping* . It is possible to create new mappings, either with the name of *defaultmapping* or, indeed, with a different name, provided the eventClassKey is *defaultmapping* . The sequence numbers of all such default mappings should be adjusted to prioritise these default mappings.

7 Event transforms

Transforms can be used to modify fields of an event, create new, user-defined fields or fields can be retrieved from events already in the MySQL database.

7.1 Different ways to apply transforms

You can have simple assignments of field values or set them based on complex Python programs. The transform mechanism can be applied in two ways:

- event class transforms
- event class mapping transforms

Prior to Zenoss 2.4, an event **class** transform was only used for events inserted directly to that exact event class by the parsing mechanism (zenping, zenperfsnmp, zencommand, AddEvent with Event Class specified, etc). If a transform existed in an event class **mapping** that was used, the event class transform was **not** used.

Zenoss 2.4 introduced **cascading event transforms**. This changed things in two ways. Given an event class */Toptest* with a subclass of */T1*, if an event arrives that already has class */Toptest/T1*, then the Toptest transform will be applied, followed by the T1 transform. If an event arrives that does not have a pre-allocated class but whose event class is determined to be */Toptest/T1*, by the Rule / Regex of the event class mapping, t1, then transforms will be applied in the order:

- Toptest class -> T1 class -> t1 event class mapping

It is perfectly possible for a transform to use user-defined event fields instantiated by earlier transforms; however, be very aware that if any statement in a transform fails (perhaps because a field doesn't exist), then the processing of that transform will stop at that point and no further statements will be executed. Any further transforms **will** be executed (at least until an error is reached).

All transforms are executed once the Rule and Regex elements of a mapping have been successfully tested and after device and event context have been applied. Thus, at transform time, most of the standard event fields are available, **except** those populated at database insertions time (evid, stateChange, eventState, dedupid, count, eventClassMapping, firstTime and lastTime). Any user-defined fields created by the Regex are also available.

Event class transforms can be useful on the */Unknown* class to selectively change the class for events that would otherwise be */Unknown* .

Note that if a transform tries to reference a field of an event that does not yet exist (like *count*) then that line of the transform **and any subsequent lines** will be ignored. Such an error will not trigger any error messages in the transform dialogue. Transforms are implemented by the zeneventd daemon so inspect the end of *$ZENHOME/log/zeneventd.log* to see the error message reporting the absence of the attribute.

A class transform is configured from the Action icon at the bottom of the lefthand menu for an event class.

A mapping transform is specified as part of the same event mapping dialogue that defines the Rule and Regex fields. In each case, if the Python syntax is incorrect, when you use the *Save* button, then the transform is all displayed in red text, indicating an error.

Figure 31 on page 57 showed an event mapping called linetest which includes a transform to create several user-defined event fields, some based on values from the event and some with values from the device that generated the event. The event summary field is set to a string constructed from literal text, standard event fields and user-defined fields.

```
evt.myDevId = device.id
evt.mySnmpSysLoc = device.snmpLocation
evt.mySnmpSysContact = device.snmpContact
evt.mySnmpStatus = device.getSnmpStatusString()
evt.summary = "Problem is %s on device %s.  Please call %s" % (evt.summary,
                   evt.myDevId, evt.mySnmpSysContact)
```

Most of the user-defined fields are assigned to simple **attributes** of either the event or the device; for example, *device.snmpContact*. The line before the end demonstrates using a Python **method** to get values; for example *device.getSnmpStatusString()* (note the *()* at the end – this is the clue that it is a method rather than an attribute).

7.2 Understanding fields available for event processing

So – how does one work out what attributes and methods are available? The Zenoss Core 4 Administration Guide documents the **TALES Event Attributes** in Appendix D3 but this is only a starting point.

Similarly, Appendix D2 documents **TALES Device Attributes** and methods but this information is very incomplete.

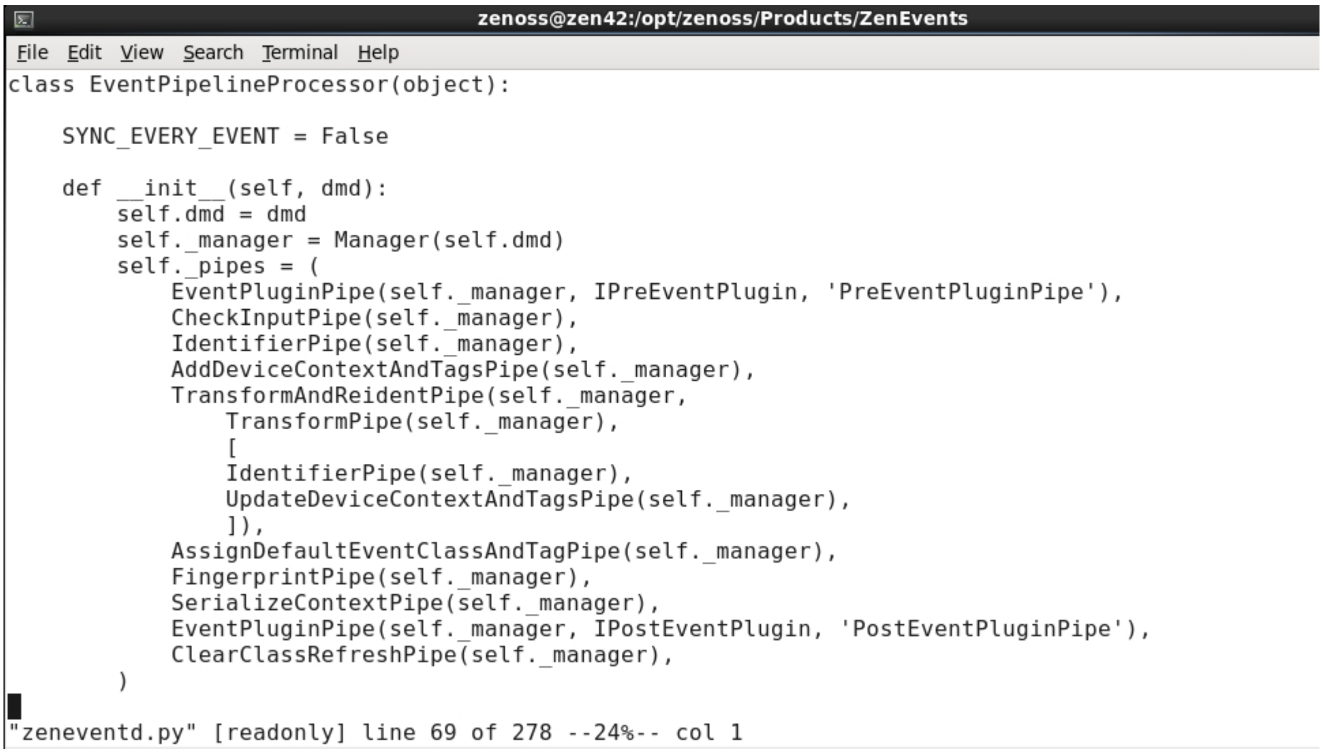
When zeneventd is processing an event, strictly it is working on a number of Python **dictionaries** that make up a **ZepRawEventProxy** object class. Remember from the architecture section that zeneventd takes elements from the rawevents queue, processes them and outputs the result to the zenevents queue to be further processed by the zeneventserver daemon (Figure 12, Zenoss 4 event architecture). The messages on the rawevent queue (like all other queue messages) are blobs of binary data.

There are a number of modules in *$ZENHOME/lib/python/zenoss/protocols* that manipulate this message data using Google **protobufs** as a data interchange format for the structured queue message data.

$ZENHOME/Products/ZenEvents/events2 contains three Python files that are crucial for understanding the details of how zeneventd processes the raw event:

- processing.py
- fields.py
- proxy.py

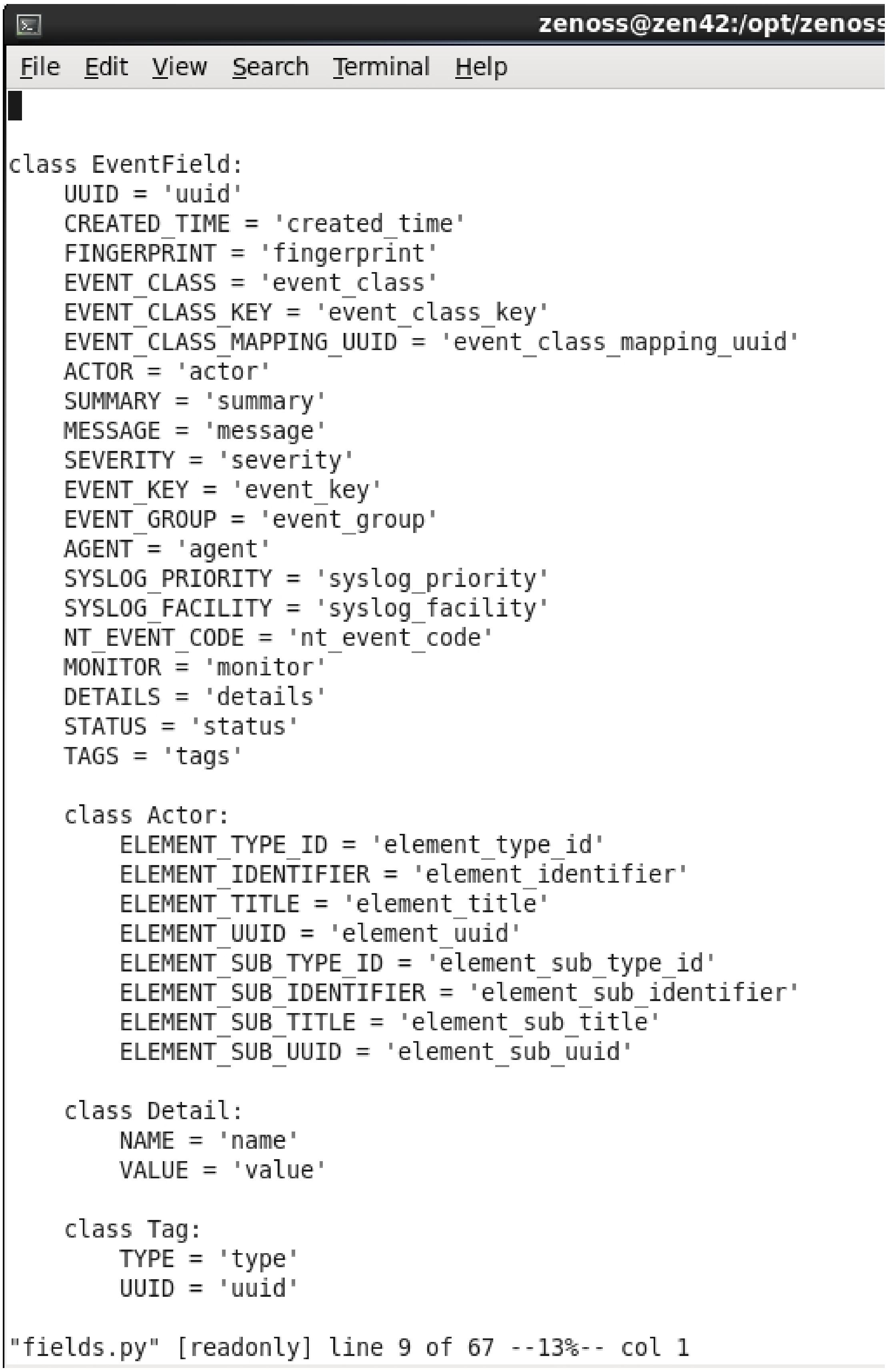
$ZENHOME/Products/ZenEvents/zeneventd.py has a number of pipelines that an event passes through. Their effect can be seen be analysing zeneventd.log if the Debug logging level is turned on.


zenoss@zen42:/opt/zenoss/Products/ZenEvents
File Edit View Search Terminal Help

```
class EventPipelineProcessor(object):

    SYNC_EVERY_EVENT = False

    def __init__(self, dmd):
        self.dmd = dmd
        self._manager = Manager(self.dmd)
        self._pipes = (
            EventPluginPipe(self._manager, IPreEventPlugin, 'PreEventPluginPipe'),
            CheckInputPipe(self._manager),
            IdentifierPipe(self._manager),
            AddDeviceContextAndTagsPipe(self._manager),
            TransformAndReidentPipe(self._manager,
                TransformPipe(self._manager),
                [
                IdentifierPipe(self._manager),
                UpdateDeviceContextAndTagsPipe(self._manager),
                ]),
            AssignDefaultEventClassAndTagPipe(self._manager),
            FingerprintPipe(self._manager),
            SerializeContextPipe(self._manager),
            EventPluginPipe(self._manager, IPostEventPlugin, 'PostEventPluginPipe'),
            ClearClassRefreshPipe(self._manager),
        )

"zeneventd.py" [readonly] line 69 of 278 --24%-- col 1
```

Figure 32Event Pipeline Processor object class in zeneventd.py

processing.py contains the code to implement each of the pipeline stages executed by zeneventd. There are methods to processes a raw event, add device and event context, process rule and regex to establish an event class, and to perform transforms. There is also a method to generate the fingerprint field.

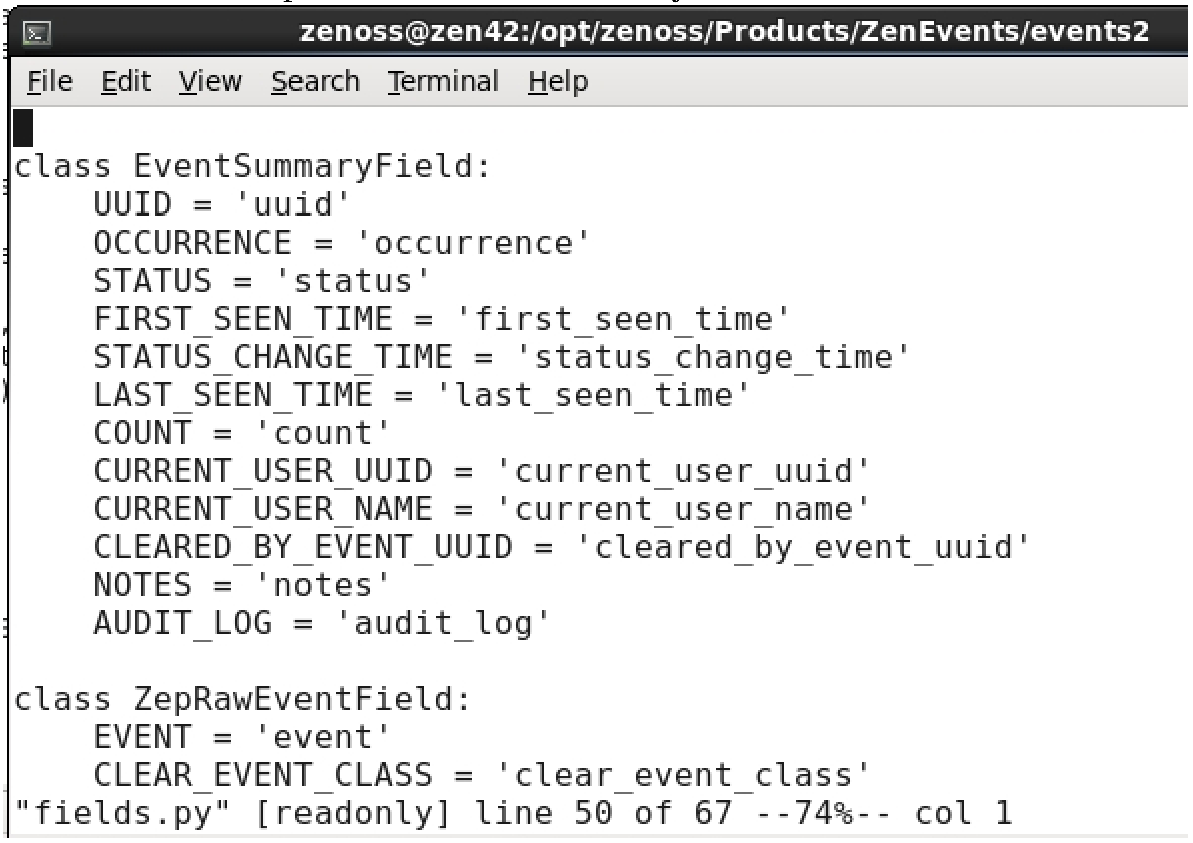


```
class EventField:
    UUID = 'uuid'
    CREATED_TIME = 'created_time'
    FINGERPRINT = 'fingerprint'
    EVENT_CLASS = 'event_class'
    EVENT_CLASS_KEY = 'event_class_key'
    EVENT_CLASS_MAPPING_UUID = 'event_class_mapping_uuid'
    ACTOR = 'actor'
    SUMMARY = 'summary'
    MESSAGE = 'message'
    SEVERITY = 'severity'
    EVENT_KEY = 'event_key'
    EVENT_GROUP = 'event_group'
    AGENT = 'agent'
    SYSLOG_PRIORITY = 'syslog_priority'
    SYSLOG_FACILITY = 'syslog_facility'
    NT_EVENT_CODE = 'nt_event_code'
    MONITOR = 'monitor'
    DETAILS = 'details'
    STATUS = 'status'
    TAGS = 'tags'

    class Actor:
        ELEMENT_TYPE_ID = 'element_type_id'
        ELEMENT_IDENTIFIER = 'element_identifier'
        ELEMENT_TITLE = 'element_title'
        ELEMENT_UUID = 'element_uuid'
        ELEMENT_SUB_TYPE_ID = 'element_sub_type_id'
        ELEMENT_SUB_IDENTIFIER = 'element_sub_identifier'
        ELEMENT_SUB_TITLE = 'element_sub_title'
        ELEMENT_SUB_UUID = 'element_sub_uuid'

    class Detail:
        NAME = 'name'
        VALUE = 'value'

    class Tag:
        TYPE = 'type'
        UUID = 'uuid'

"fields.py" [readonly] line 9 of 67 --13%-- col 1
```

Figure 33EventField object class in $ZENHOME/Products/ZenEvents/events2/fields.py

$ZENHOME/Products/ZenEvents/events2/fields.py contains object class definitions for:

- EventField
    - The EventField attributes match up with the base MySQL database fields in zenoss_zep.
    - The Actor, Detail and Tag fields are defined as sub classes of the object
- EventSummaryField
    - Has the additional fields that are populated when the event is inserted into the zenoss_zep database event_summary table.



zenoss@zen42:/opt/zenoss/Products/ZenEvents/events2

```
class EventSummaryField:
    UUID = 'uuid'
    OCCURRENCE = 'occurrence'
    STATUS = 'status'
    FIRST_SEEN_TIME = 'first_seen_time'
    STATUS_CHANGE_TIME = 'status_change_time'
    LAST_SEEN_TIME = 'last_seen_time'
    COUNT = 'count'
    CURRENT_USER_UUID = 'current_user_uuid'
    CURRENT_USER_NAME = 'current_user_name'
    CLEARED_BY_EVENT_UUID = 'cleared_by_event_uuid'
    NOTES = 'notes'
    AUDIT_LOG = 'audit_log'

class ZepRawEventField:
    EVENT = 'event'
    CLEAR_EVENT_CLASS = 'clear_event_class'
"fields.py" [readonly] line 50 of 67 --74%-- col 1
```

Figure 34EventSummaryField and ZepRawEventField definitions

- ZepRawEventField
    - Has the same fields as EventField but also has clear_event_class as that is needed by the zeneventd processing pipelines as it is part of the event context.

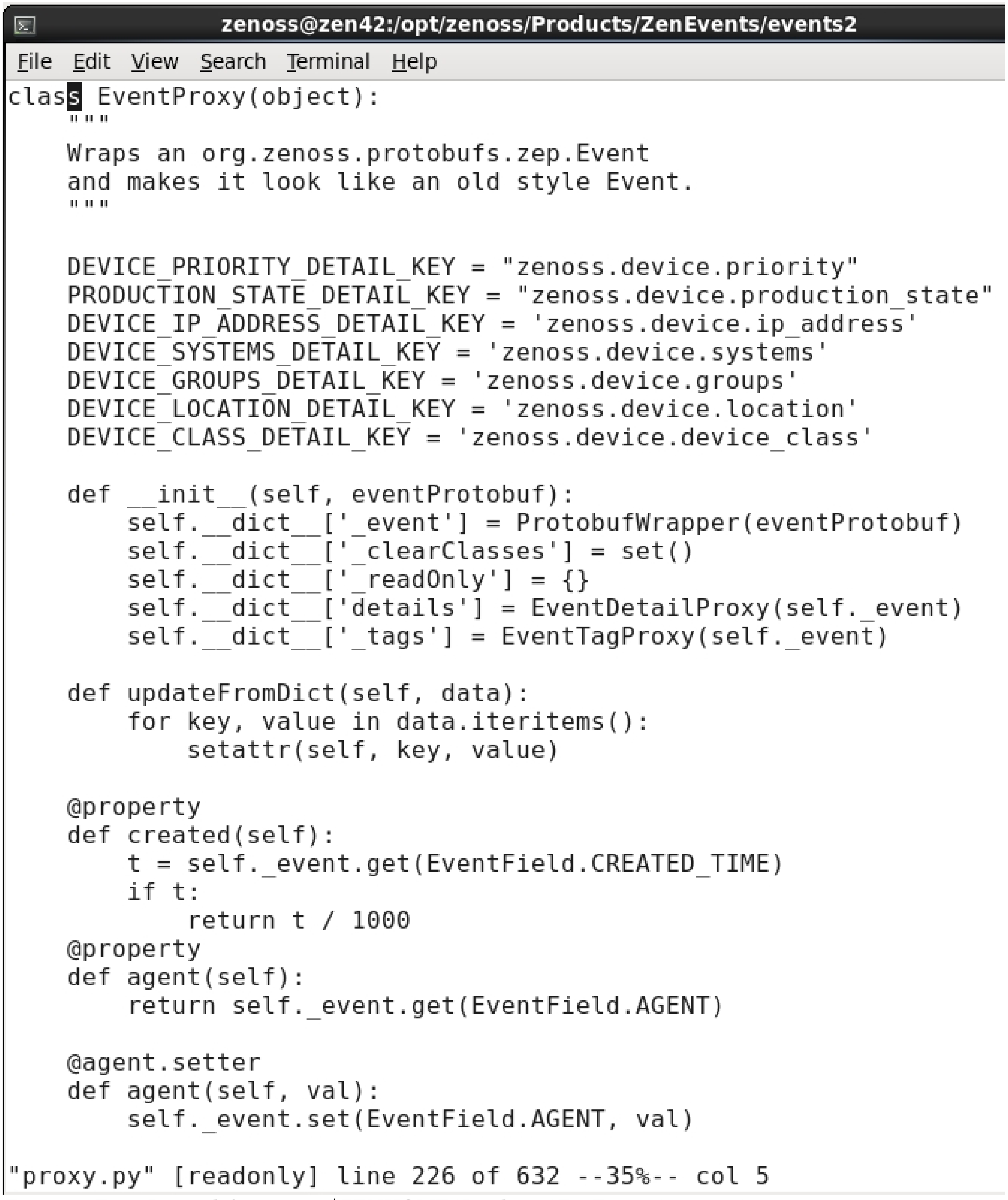
Note that the definitions in fields.py are **not** helpful when deciding what attributes are available to transforms; these are the fields one finds in the zenoss_zep database.

7.2.1 Event Proxies

$ZENHOME/Products/ZenEvents/events2/proxy.py is the key to understanding what attributes are available when writing rules and transforms. proxy.py provides

translations between encoded formats of events and a human-readable JSON (JavaScript Object Notation) format.

As far as possible, the attributes presented by a proxy are the same in Zenoss 4 as they were in previous versions.


zenoss@zen42:/opt/zenoss/Products/ZenEvents/events2
File Edit View Search Terminal Help

```
class EventProxy(object):
    """
    Wraps an org.zenoss.protobufs.zep.Event
    and makes it look like an old style Event.
    """

    DEVICE_PRIORITY_DETAIL_KEY = "zenoss.device.priority"
    PRODUCTION_STATE_DETAIL_KEY = "zenoss.device.production_state"
    DEVICE_IP_ADDRESS_DETAIL_KEY = 'zenoss.device.ip_address'
    DEVICE_SYSTEMS_DETAIL_KEY = 'zenoss.device.systems'
    DEVICE_GROUPS_DETAIL_KEY = 'zenoss.device.groups'
    DEVICE_LOCATION_DETAIL_KEY = 'zenoss.device.location'
    DEVICE_CLASS_DETAIL_KEY = 'zenoss.device.device_class'

    def __init__(self, eventProtobuf):
        self.__dict__['_event'] = ProtobufWrapper(eventProtobuf)
        self.__dict__['_clearClasses'] = set()
        self.__dict__['_readOnly'] = {}
        self.__dict__['details'] = EventDetailProxy(self._event)
        self.__dict__['_tags'] = EventTagProxy(self._event)

    def updateFromDict(self, data):
        for key, value in data.iteritems():
            setattr(self, key, value)

    @property
    def created(self):
        t = self._event.get(EventField.CREATED_TIME)
        if t:
            return t / 1000
    @property
    def agent(self):
        return self._event.get(EventField.AGENT)

    @agent.setter
    def agent(self, val):
        self._event.set(EventField.AGENT, val)

"proxy.py" [readonly] line 226 of 632 --35%-- col 5
```

Figure 35EventProxy definition in $ZENHOME/Products/ZenEvents/events2/proxy.py

An **EventProxy** is several Python dictionaries:

- The main body of the event is a dictionary called _event
- A details dictionary
- An _tags dictionary
- A dictionary for _clearClasses
- A dictionary for _readOnly attributes

There are a large number of Python *@property* decorator constructs whose purpose is to present an attribute using a method, for example:

```
@property
def device(self):
    return self._event.actor.element_identifier
```

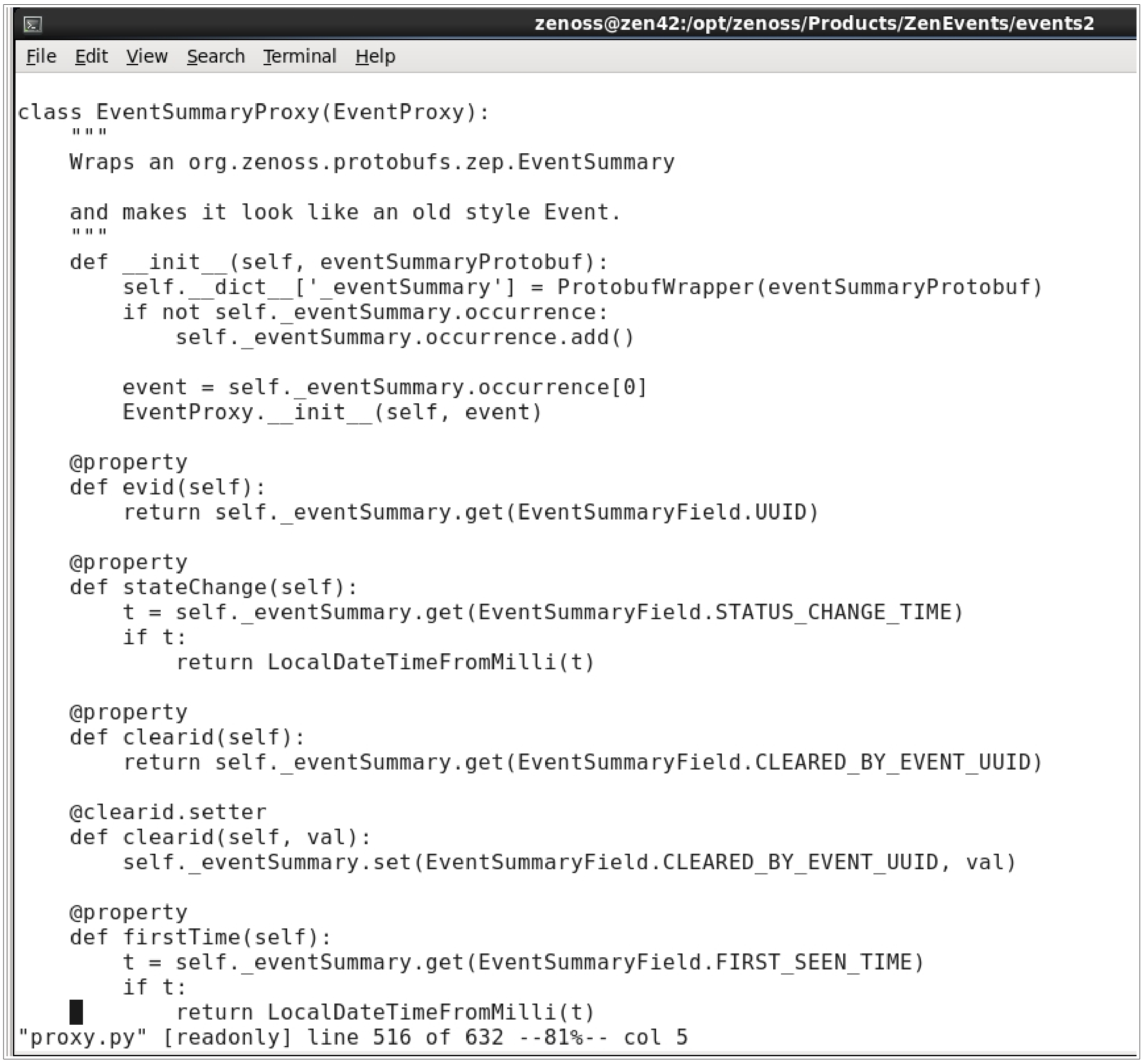
defines an attribute called **device** which is delivered by a method that returns the value of the **event's actor's element_identifier**. *device* is the field that we have (have always had) to manipulate in transforms.

The @property definitions at the end of Figure 35 show simpler definitions that return the value of a basic field of an event (using the EventField definitions defined in fields.py).

When a user views event details using the Zenoss GUI or accesses data from from the event_summary table of the zenoss_zep database using the JSON API, the event data presented is an **EventSummaryProxy**, which is a JSON format. The EventSummaryProxy inherits from the EventProxy but also has attributes that are added on database insertion:

- evid
- stateChange
- clearid
- firstTime
- lastTime
- count
- ownerid
- eventState

The EventSummaryProxy was originally designed with an idea of keeping **all** event data, treating duplicates as multiple **occurrences** within the EventSummaryProxy; however the scalability was not feasible so, in practise the fields of an event are in the zero'th element of an EventSummary occurrence list.


File Edit View Search Terminal Help

```
class EventSummaryProxy(EventProxy):
    """
    Wraps an org.zenoss.protobufs.zep.EventSummary

    and makes it look like an old style Event.
    """
    def __init__(self, eventSummaryProtobuf):
        self.__dict__['_eventSummary'] = ProtobufWrapper(eventSummaryProtobuf)
        if not self._eventSummary.occurrence:
            self._eventSummary.occurrence.add()

        event = self._eventSummary.occurrence[0]
        EventProxy.__init__(self, event)

    @property
    def evid(self):
        return self._eventSummary.get(EventSummaryField.UUID)

    @property
    def stateChange(self):
        t = self._eventSummary.get(EventSummaryField.STATUS_CHANGE_TIME)
        if t:
            return LocalDateTimeFromMilli(t)

    @property
    def clearid(self):
        return self._eventSummary.get(EventSummaryField.CLEARED_BY_EVENT_UUID)

    @clearid.setter
    def clearid(self, val):
        self._eventSummary.set(EventSummaryField.CLEARED_BY_EVENT_UUID, val)

    @property
    def firstTime(self):
        t = self._eventSummary.get(EventSummaryField.FIRST_SEEN_TIME)
        if t:
            return LocalDateTimeFromMilli(t)
"proxy.py" [readonly] line 516 of 632 --81%-- col 5
```

Figure 36EventSummaryProxy object class

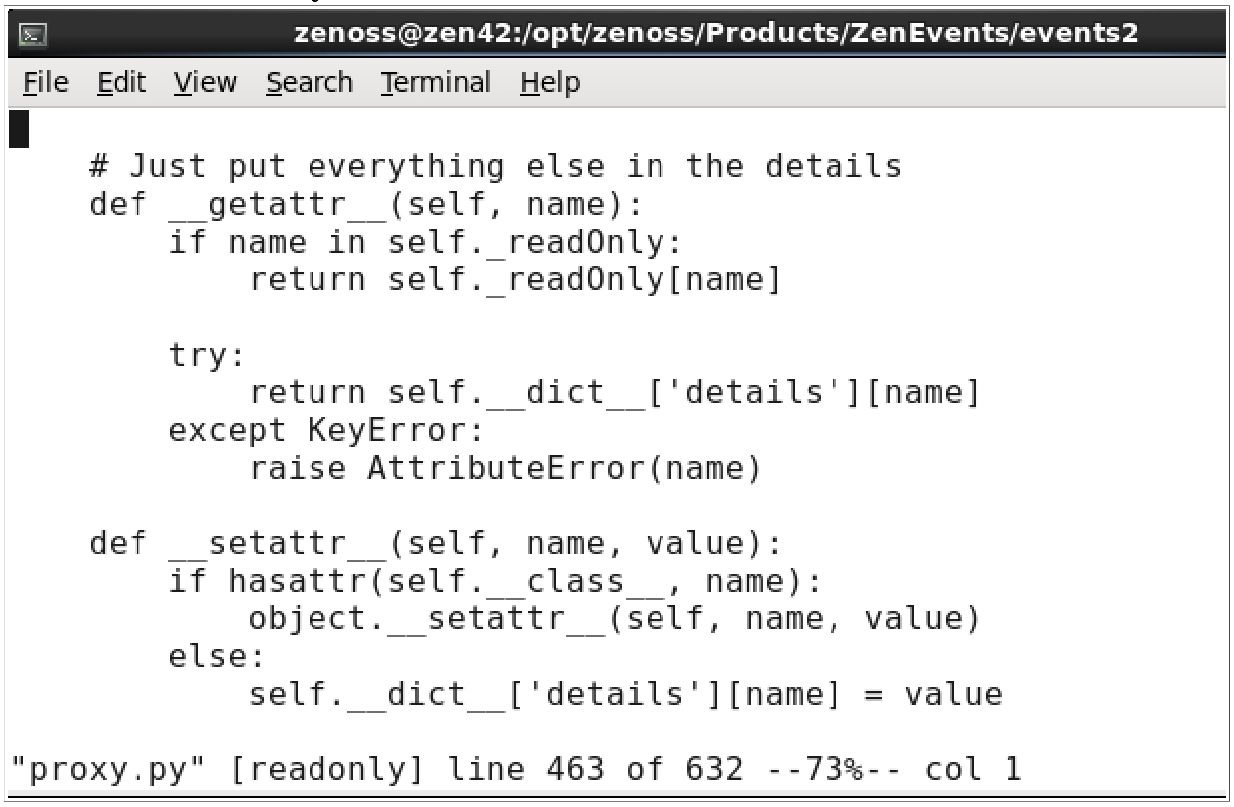
proxy.py also defines a class for **ZepRawEventProxy** which inherits from EventProxy. The additional properties for ZepRawEventProxy are for **_ClearClasses, _action** and **eventClassMapping.**

It is the attributes defined in proxy.py for the ZepRawEventProxy object class that are available for use in rules and transforms.

7.2.2 Event Details

So what happens to a user-defined event attribute generated, say, by the varbinds that come in on an SNMP TRAP?

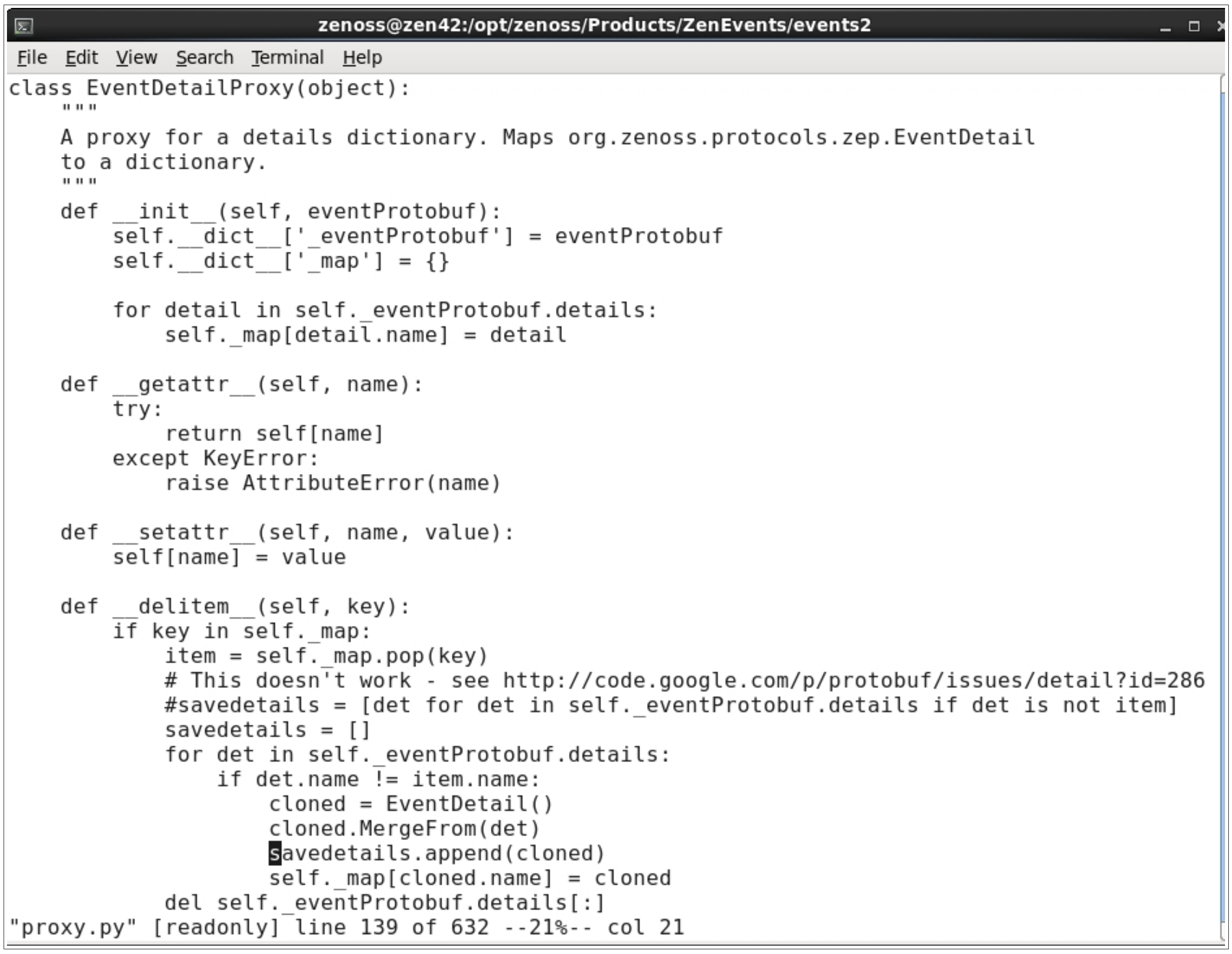
Remember that the EventProxy has a number of dictionaries, including a **details** dictionary. Examination of the EventProxy object class in proxy.py shows that any fields that don't match the standard fields are put in <name> , <value> pairs in the event's details dictionary.


zenoss@zen42:/opt/zenoss/Products/ZenEvents/events2
File Edit View Search Terminal Help

```

    # Just put everything else in the details
    def __getattr__(self, name):
        if name in self._readOnly:
            return self._readOnly[name]

        try:
            return self.__dict__['details'][name]
        except KeyError:
            raise AttributeError(name)

    def __setattr__(self, name, value):
        if hasattr(self.__class__, name):
            object.__setattr__(self, name, value)
        else:
            self.__dict__['details'][name] = value

"proxy.py" [readonly] line 463 of 632 --73%-- col 1
```

Figure 37Processing event details in proxy.py

The evt.details dictionary is available as an **EventDetailProxy** object (also defined in proxy.py).


zenoss@zen42:/opt/zenoss/Products/ZenEvents/events2
File Edit View Search Terminal Help

```
class EventDetailProxy(object):
    """
    A proxy for a details dictionary. Maps org.zenoss.protocols.zep.EventDetail
    to a dictionary.
    """
    def __init__(self, eventProtobuf):
        self.__dict__['_eventProtobuf'] = eventProtobuf
        self.__dict__['_map'] = {}

        for detail in self._eventProtobuf.details:
            self._map[detail.name] = detail

    def __getattr__(self, name):
        try:
            return self[name]
        except KeyError:
            raise AttributeError(name)

    def __setattr__(self, name, value):
        self[name] = value

    def __delitem__(self, key):
        if key in self._map:
            item = self._map.pop(key)
            # This doesn't work - see http://code.google.com/p/protobuf/issues/detail?id=286
            #savedetails = [det for det in self._eventProtobuf.details if det is not item]
            savedetails = []
            for det in self._eventProtobuf.details:
                if det.name != item.name:
                    cloned = EventDetail()
                    cloned.MergeFrom(det)
                    savedetails.append(cloned)
                    self._map[cloned.name] = cloned
            del self._eventProtobuf.details[:]
"proxy.py" [readonly] line 139 of 632 --21%-- col 21
```

Figure 38EventDetailProxy object class in proxy.py

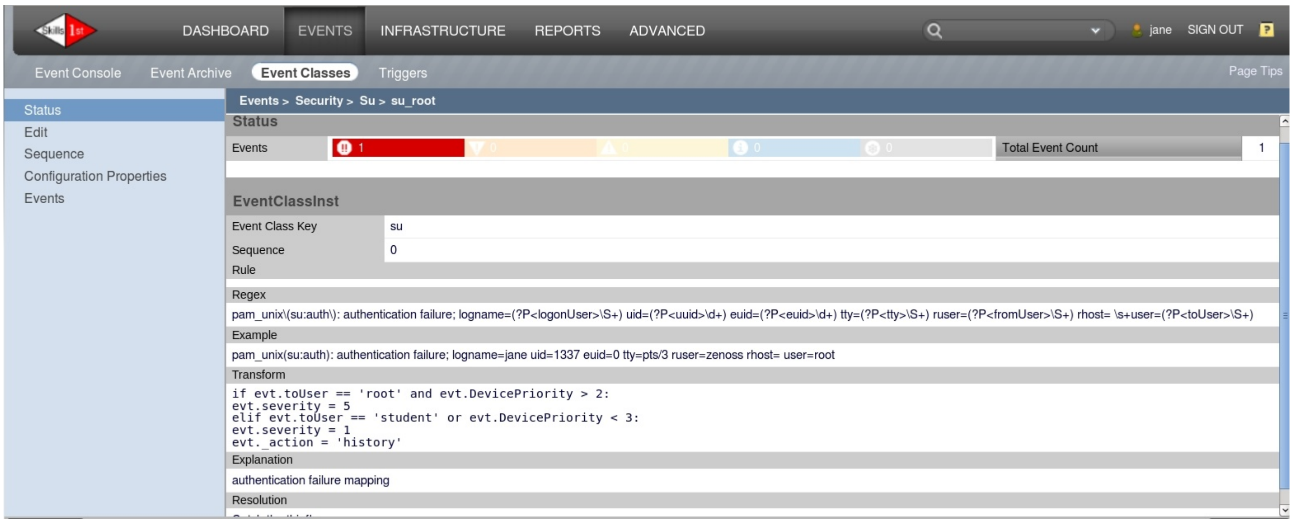
To access these details in a rule or transform they can be referred to as *evt.<name of detail field>* if the name does **not** include a . (dot); otherwise to use these details in a rule or transform, they need to be accessed through the **_map** dictionary.

7.3 Transform examples

7.3.1 Combining user defined fields from Regex with transform

In this example, we will return to the */Security/Su* subclass of events and combine regular expressions and transforms. The objective is, for “important devices”, to escalate the event severity if a user tries to su to root but to decrease the severity if the su event comes **either** from an “unimportant” device or if the su is to a particular userid (*student* in this case). “Important” devices are determined by the event field *DevicePriority* (note two capital letters in this field name). The device priority for a device can be changed from the *Overview* menu on a device's details page.

This example is the same as shown in Figure 29 but a transform has been added.



Status

Events	1 0 0 0 0
Total Event Count	1

EventClassInst

Event Class Key	su
Sequence	0

Rule

Regex

pam_unix\(su:auth\): authentication failure; logname=(?P<logonUser>\S+) uid=(?P<uuid>\d+) euid=(?P<euid>\d+) tty=(?P<tty>\S+) ruser=(?P<fromUser>\S+) rhost= \s+user=(?P<toUser>\S+)

Example

pam_unix(su:auth): authentication failure; logname=jane uid=1337 euid=0 tty=pts/3 ruser=zenoss rhost= user=root

Transform

```
if evt.toUser == 'root' and evt.DevicePriority > 2:
evt.severity = 5
elif evt.toUser == 'student' or evt.DevicePriority < 3:
evt.severity = 1
evt._action = 'history'
```

Explanation

authentication failure mapping

Resolution

Figure 39: su_root event mapping with transform

Note that the Status menu of a mapping loses any Python indentations you have carefully created! The transform should be entered as:

```
if evt.toUser == 'root' and evt.DevicePriority > 2:
    evt.severity = 5
elif evt.toUser == 'student' or evt.DevicePriority < 3:
    evt.severity = 1
    evt._action = 'history'
```

The user-defined field *toUser*, created by the Regex, is tested against the literal string *'root'*. The result is logically AND'ed with a test of the standard event field *DevicePriority* for *> 2* . If the result is True then the standard event field *severity* is set to *5* (Critical). Remember that the default severity for the *su_root* mapping was set to *Warning* by the *zEventSeverity* event context zProperty.

In the *elif* statement, if this condition is True then the event's severity is set to *1* (Debug) and the zProperty *zEventAction* is overridden by setting *evt._action = 'history'* in the transform. In this case, the event's *eventState* is set to *Closed*.

Note with any Python test that includes multiple clauses, the test fails as soon as a condition fails so in the *if* statement if *evt.toUser* is not 'root' then *evt.DevicePriority* will not be evaluated. Performance can be improved by careful consideration and ordering of such tests.

7.3.2 Applying event and device context in relation to transforms

Event context (zEventSeverity, zEventAction, zEventClearClasses) is applied through the *Configuration Properties* menu of an event class or event class mapping.

Device context comprises the event fields **prodState, Location, DeviceClass, DeviceGroups, Systems** and **DevicePriority**. **ipAddress** and the **monitor** (collector) responsible for the event also tend to be bracketed with the device context but these latter fields are information received on the incoming event, rather than the device context data that is looked-up in the Zope database.

The following *device_context* event mapping example demonstrates the order in which device context, event context and the mapping transform are applied.

Create a new event subclass, *Device_context*, under the */Skills* class.

Create a mapping, *device_context*, for this new event class. Ensure that the eventClassKey is *device_context*. Set the Regex to the literal string:

```
        This is a device context event
```

Set a Rule as follows (all on one line):

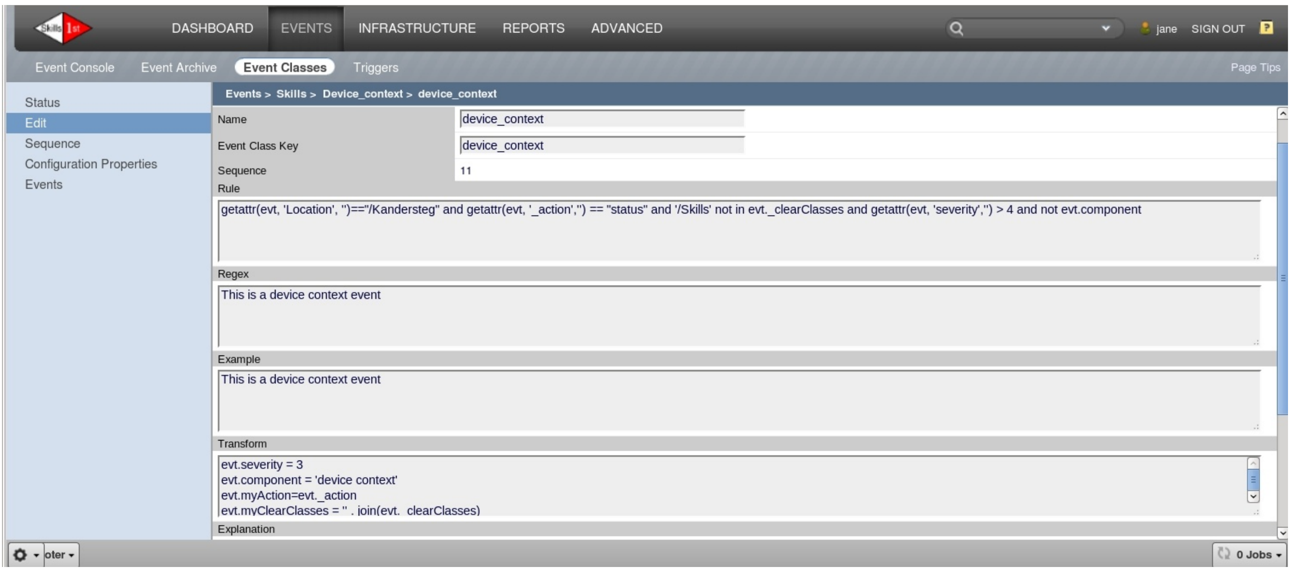
```
        getattr(evt, 'Location', '')=="/Kandersteg" and getattr(evt, '_action','')
        == "status" and '/Skills' not in evt._clearClasses and getattr(evt,
        'severity','') > 4 and not evt.component
```

Using the *Configuration Properties* menu for the mapping, set the *zEventSeverity* event context value to *Error* (4), *zEventAction* to *history* and *zEventClearClasses* to */Skills*.

Test the mapping with a zensendevent (all on one line):

```
        zensendevent -d group-100-r1.class.example.org -s Critical -k
        device_context This is a device context event 1
```

The test event set the device field to *group-100-r1.class.example.org* which is included in the Location called */Kandersteg* . The eventClassKey should be set to *device_context,* the component field should be blank and the eventClass should be blank.



Events > Skills > Device_context > device_context

Name: device_context
Event Class Key: device_context
Sequence: 11
Rule:

```
getattr(evt, 'Location', '')=="/Kandersteg" and getattr(evt, '_action','') == "status" and '/Skills' not in evt._clearClasses and getattr(evt, 'severity','') > 4 and not evt.component
```

Regex: This is a device context event
Example: This is a device context event
Transform:

```
evt.severity = 3
evt.component = 'device context'
evt.myAction=evt._action
evt.myClearClasses = '' . join(evt._clearClasses)
```

Explanation

Figure 40: Combining a Rule, context and a transform for the device_context event mapping

The Rule demonstrates the Python *getattr* function to test:

- The *evt.Location* field set by device context, which should evaluate TRUE at Rule time ie. device context **has** been applied
- The *evt._action* field that is set by event context to *history*. The test shown above actually evaluates TRUE showing that event context has **not** been applied at Rule time.

- Similarly, the *evt._clearClasses* field test evaluates TRUE showing that event context has **not** been applied. The Python syntax for checking *evt._clearClasses* is a little different as this attribute is defined as a Python **list** rather than a string.
- The *evt.severity* starts at 5 in the generated event and event context sets it to 4. This test evaluates TRUE confirming that event context has **not** been applied.
- The *evt.component* must be blank (the null string evaluates to the boolean False)
- **Note** that the syntax for the last field of the getattr is two single quotes to supply a null default

In summary, the Rule and Regex should evaluate successfully and the transform will be applied.

The transform demonstrates:

- Changing the *evt.severity* field again – it would have been modified from the original value (5) down to (4) when the event context was applied after Rule and Regex processing. The transform changes it to 3.
- Changing the *evt.component* field is interesting. Remember that the fingerprint dedupid field includes the component. Although the raw event did **not** include a component field, the fingerprint is generated after the transform as the dedupid in the event **does** contain the component.
- Several user-defined variables are created. The evt.myClearClasses line demonstrates that all user-defined fields appear to be of type string but *evt._clearClasses* is defined as a Python list . You cannot assign evt.myClearClasses to something of type list unless you use the **join** function to stick together the list elements back into a string type.
- The user-defined fields demonstrate that both device context and event context have been applied by transform time

8 Testing and debugging aids

8.1 Log files

8.1.1 zeneventd.log

Device context, event context, rule, regex and transforms are all applied by the zeneventd daemon. It also constructs the dedupid fingerprint field. See the event processing pipeline code for zeneventd in Figure 32 on page 61.

Turning up the debug flag in *$ZENHOME/etc/zeneventd.conf* to *10* (*Debug*) provides an opportunity to track the progress of each of the stages in this pipeline in *$ZENHOME/log/zeneventd.log*, noting that the event gains more fields as processing continues.

zeneventd.log is also the place to look for problems with event processing. Even with the usual debug level of *20* (*Info*), rule, regex and transform problems are highlighted. Search for *WARNING* in the log.

The following extract shows a transform attempting to change evt.Location (which appears not to be allowed). Note that although the message is definitely helpful, its ideas about line numbers are way out!

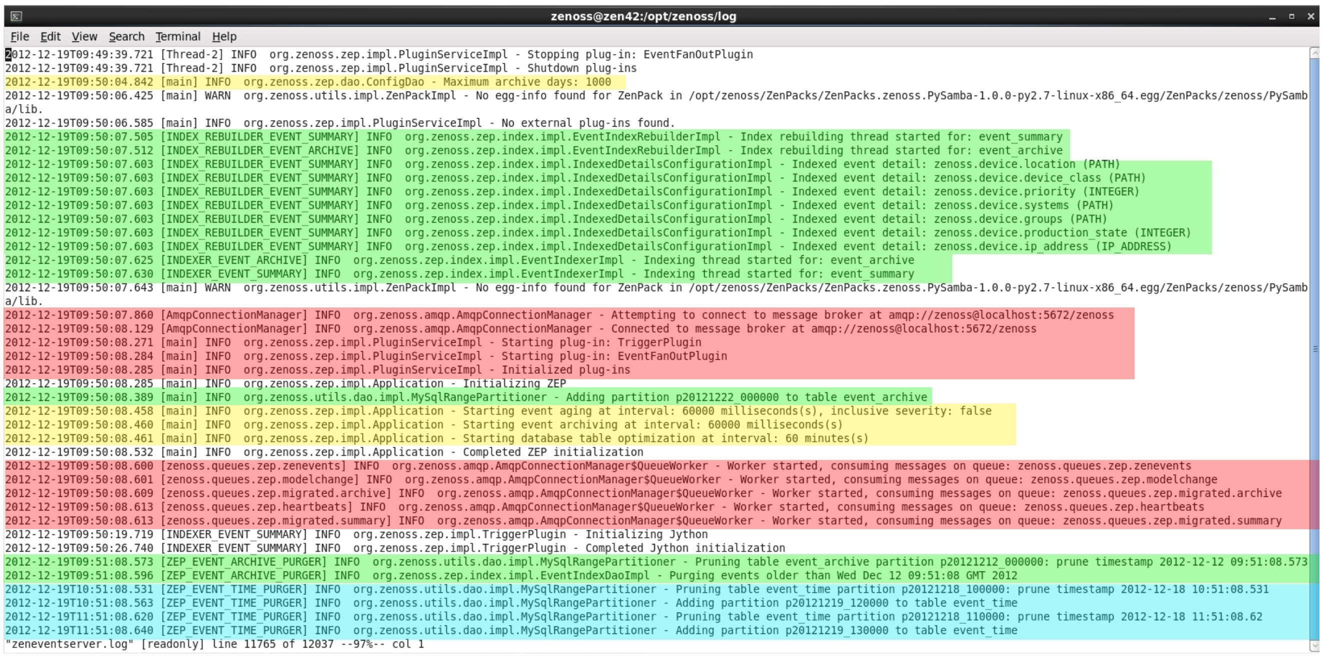
```
2012-12-20 10:02:01,923 WARNING zen.Events: Error processing
transform/mapping on Event Class
/Skills/Device_context/instances/device_context
Problem on line 475: AttributeError: can't set attribute

Transform:
  0 evt.Location ='/Taplow'
  1 evt.severity = 3
  2 evt.myProdState = evt.prodState
  3 evt.myDeviceClass = evt.DeviceClass
  4 evt.myDeviceGroups = evt.DeviceGroups
  5 evt.mySystems=evt.Systems
  6 evt.myAction=evt._action
  7 evt.myClearClasses = '' . join(evt._clearClasses)
```

With Zenoss 4, you will also receive an event from the Zenoss server with similar information (and equally creative line numbers!). With versions of Zenoss prior to 4 there was no warning event and all the event processing was performed by zenhub so *zenhub.log* was the place to search for errors.

8.1.2 zeneventserver.log

The zeneventserver daemon is written in Java. This means that error messages are difficult to comprehend in *$ZENHOME/log/zeneventserver.log* without an intimate knowledge of the Java code.

What is useful to help understanding of the architecture is to inspect this log around daemon start-up.



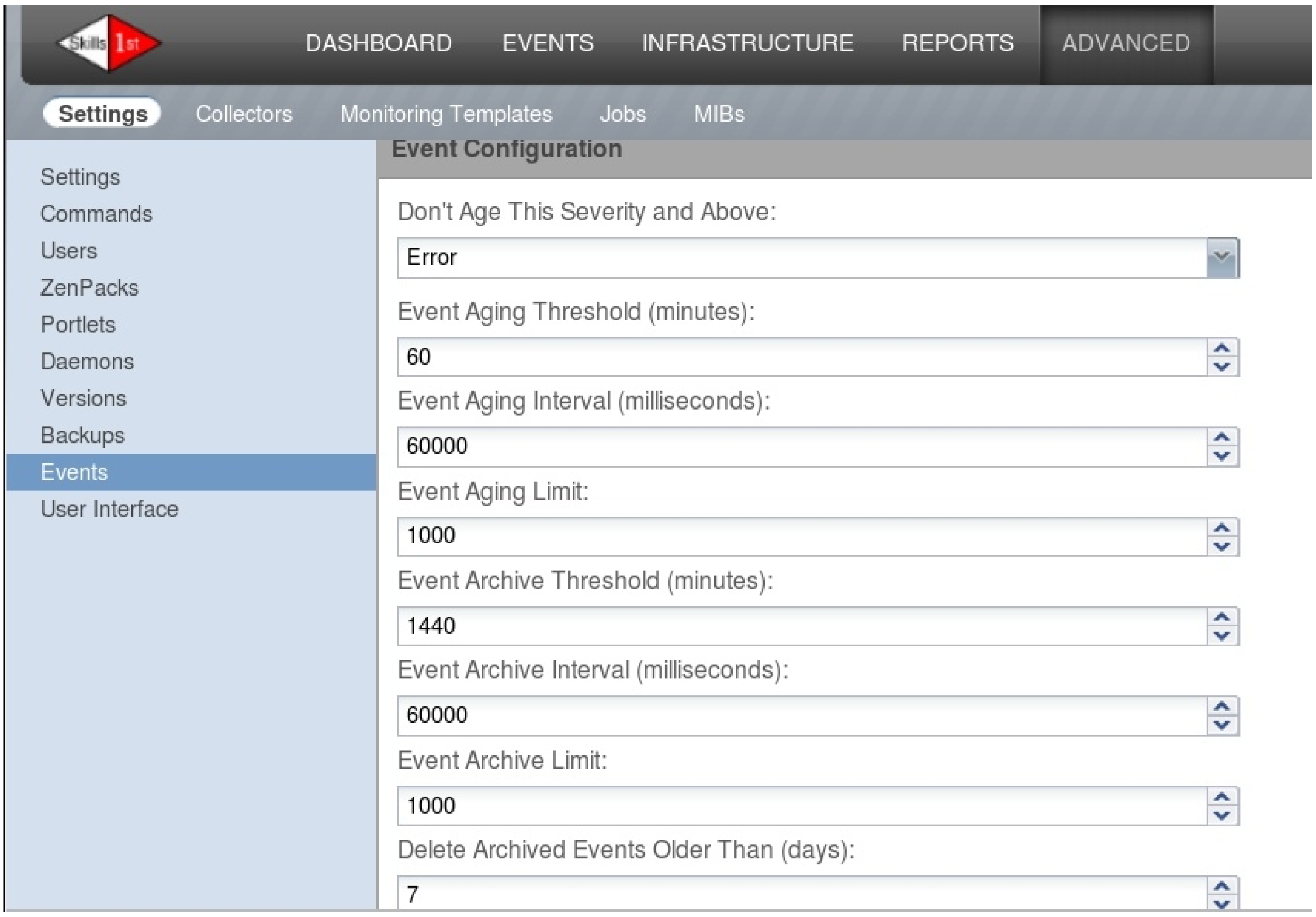
```
2012-12-19T09:49:39.721 [Thread-2] INFO  org.zenoss.zep.impl.PluginServiceImpl - Stopping plug-in: EventFanOutPlugin
2012-12-19T09:49:39.721 [Thread-2] INFO  org.zenoss.zep.impl.PluginServiceImpl - Shutdown plug-ins
2012-12-19T09:50:04.842 [main] INFO  org.zenoss.zep.dao.ConfigDao - Maximum archive days: 1000
2012-12-19T09:50:06.425 [main] WARN  org.zenoss.utils.impl.ZenPackImpl - No egg-info found for ZenPack in /opt/zenoss/ZenPacks/ZenPacks.zenoss.PySamba-1.0.0-py2.7-linux-x86_64.egg/ZenPacks/zenoss/PySamb
a/lib.
2012-12-19T09:50:06.585 [main] INFO  org.zenoss.zep.impl.PluginServiceImpl - No external plug-ins found.
2012-12-19T09:50:07.505 [INDEX_REBUILDER_EVENT_SUMMARY] INFO  org.zenoss.zep.index.impl.EventIndexRebuilderImpl - Index rebuilding thread started for: event_summary
2012-12-19T09:50:07.512 [INDEX_REBUILDER_EVENT_ARCHIVE] INFO  org.zenoss.zep.index.impl.EventIndexRebuilderImpl - Index rebuilding thread started for: event_archive
2012-12-19T09:50:07.603 [INDEX_REBUILDER_EVENT_SUMMARY] INFO  org.zenoss.zep.index.impl.IndexedDetailsConfigurationImpl - Indexed event detail: zenoss.device.location (PATH)
2012-12-19T09:50:07.603 [INDEX_REBUILDER_EVENT_SUMMARY] INFO  org.zenoss.zep.index.impl.IndexedDetailsConfigurationImpl - Indexed event detail: zenoss.device.device_class (PATH)
2012-12-19T09:50:07.603 [INDEX_REBUILDER_EVENT_SUMMARY] INFO  org.zenoss.zep.index.impl.IndexedDetailsConfigurationImpl - Indexed event detail: zenoss.device.priority (INTEGER)
2012-12-19T09:50:07.603 [INDEX_REBUILDER_EVENT_SUMMARY] INFO  org.zenoss.zep.index.impl.IndexedDetailsConfigurationImpl - Indexed event detail: zenoss.device.systems (PATH)
2012-12-19T09:50:07.603 [INDEX_REBUILDER_EVENT_SUMMARY] INFO  org.zenoss.zep.index.impl.IndexedDetailsConfigurationImpl - Indexed event detail: zenoss.device.groups (PATH)
2012-12-19T09:50:07.603 [INDEX_REBUILDER_EVENT_SUMMARY] INFO  org.zenoss.zep.index.impl.IndexedDetailsConfigurationImpl - Indexed event detail: zenoss.device.production_state (INTEGER)
2012-12-19T09:50:07.603 [INDEX_REBUILDER_EVENT_SUMMARY] INFO  org.zenoss.zep.index.impl.IndexedDetailsConfigurationImpl - Indexed event detail: zenoss.device.ip_address (IP_ADDRESS)
2012-12-19T09:50:07.625 [INDEXER_EVENT_ARCHIVE] INFO  org.zenoss.zep.index.impl.EventIndexerImpl - Indexing thread started for: event_archive
2012-12-19T09:50:07.630 [INDEXER_EVENT_SUMMARY] INFO  org.zenoss.zep.index.impl.EventIndexerImpl - Indexing thread started for: event_summary
2012-12-19T09:50:07.643 [main] WARN  org.zenoss.utils.impl.ZenPackImpl - No egg-info found for ZenPack in /opt/zenoss/ZenPacks/ZenPacks.zenoss.PySamba-1.0.0-py2.7-linux-x86_64.egg/ZenPacks/zenoss/PySamb
a/lib.
2012-12-19T09:50:07.860 [AmqpConnectionManager] INFO  org.zenoss.amqp.AmqpConnectionManager - Attempting to connect to message broker at amqp://zenoss@localhost:5672/zenoss
2012-12-19T09:50:08.129 [AmqpConnectionManager] INFO  org.zenoss.amqp.AmqpConnectionManager - Connected to message broker at amqp://zenoss@localhost:5672/zenoss
2012-12-19T09:50:08.271 [main] INFO  org.zenoss.zep.impl.PluginServiceImpl - Starting plug-in: TriggerPlugin
2012-12-19T09:50:08.284 [main] INFO  org.zenoss.zep.impl.PluginServiceImpl - Starting plug-in: EventFanOutPlugin
2012-12-19T09:50:08.285 [main] INFO  org.zenoss.zep.impl.PluginServiceImpl - Initialized plug-ins
2012-12-19T09:50:08.285 [main] INFO  org.zenoss.zep.impl.Application - Initializing ZEP
2012-12-19T09:50:08.389 [main] INFO  org.zenoss.utils.dao.impl.MySqlRangePartitioner - Adding partition p20121222_000000 to table event_archive
2012-12-19T09:50:08.458 [main] INFO  org.zenoss.zep.impl.Application - Starting event aging at interval: 60000 milliseconds(s), inclusive severity: false
2012-12-19T09:50:08.460 [main] INFO  org.zenoss.zep.impl.Application - Starting event archiving at interval: 60000 milliseconds(s)
2012-12-19T09:50:08.461 [main] INFO  org.zenoss.zep.impl.Application - Starting database table optimization at interval: 60 minutes(s)
2012-12-19T09:50:08.532 [main] INFO  org.zenoss.zep.impl.Application - Completed ZEP initialization
2012-12-19T09:50:08.600 [zenoss.queues.zep.zenevents] INFO  org.zenoss.amqp.AmqpConnectionManager$QueueWorker - Worker started, consuming messages on queue: zenoss.queues.zep.zenevents
2012-12-19T09:50:08.601 [zenoss.queues.zep.modelchange] INFO  org.zenoss.amqp.AmqpConnectionManager$QueueWorker - Worker started, consuming messages on queue: zenoss.queues.zep.modelchange
2012-12-19T09:50:08.609 [zenoss.queues.zep.migrated.archive] INFO  org.zenoss.amqp.AmqpConnectionManager$QueueWorker - Worker started, consuming messages on queue: zenoss.queues.zep.migrated.archive
2012-12-19T09:50:08.613 [zenoss.queues.zep.heartbeats] INFO  org.zenoss.amqp.AmqpConnectionManager$QueueWorker - Worker started, consuming messages on queue: zenoss.queues.zep.heartbeats
2012-12-19T09:50:08.613 [zenoss.queues.zep.migrated.summary] INFO  org.zenoss.amqp.AmqpConnectionManager$QueueWorker - Worker started, consuming messages on queue: zenoss.queues.zep.migrated.summary
2012-12-19T09:50:19.719 [INDEXER_EVENT_SUMMARY] INFO  org.zenoss.zep.impl.TriggerPlugin - Initializing Jython
2012-12-19T09:50:26.740 [INDEXER_EVENT_SUMMARY] INFO  org.zenoss.zep.impl.TriggerPlugin - Completed Jython initialization
2012-12-19T09:51:08.573 [ZEP_EVENT_ARCHIVE_PURGER] INFO  org.zenoss.utils.dao.impl.MySqlRangePartitioner - Pruning table event_archive partition p20121212_000000: prune timestamp 2012-12-12 09:51:08.573
2012-12-19T09:51:08.596 [ZEP_EVENT_ARCHIVE_PURGER] INFO  org.zenoss.zep.index.impl.EventIndexDaoImpl - Purging events older than Wed Dec 12 09:51:08 GMT 2012
2012-12-19T10:51:08.531 [ZEP_EVENT_TIME_PURGER] INFO  org.zenoss.utils.dao.impl.MySqlRangePartitioner - Pruning table event_time partition p20121218_100000: prune timestamp 2012-12-18 10:51:08.531
2012-12-19T10:51:08.563 [ZEP_EVENT_TIME_PURGER] INFO  org.zenoss.utils.dao.impl.MySqlRangePartitioner - Adding partition p20121219_120000 to table event_time
2012-12-19T11:51:08.620 [ZEP_EVENT_TIME_PURGER] INFO  org.zenoss.utils.dao.impl.MySqlRangePartitioner - Pruning table event_time partition p20121218_110000: prune timestamp 2012-12-18 11:51:08.62
2012-12-19T11:51:08.640 [ZEP_EVENT_TIME_PURGER] INFO  org.zenoss.utils.dao.impl.MySqlRangePartitioner - Adding partition p20121219_130000 to table event_time
"zeneventserver.log" [readonly] line 11765 of 12037 --97%-- col 1
```

Figure 41: zeneventserver.log showing daemon startup

In Figure 41 lines highlighted in yellow show Event Manager configuration parameters that can be check against the *ADVANCED* -> *Settings* -> *Events* menu.

- Maximum archive days: 1000
- Starting event ageing at interval: 60000 milliseconds(s)
- Starting event archiving at interval: 60000 milliseconds(s)
- Starting database table optimization at interval: 60 minutes(s)

Lines highlighted in green show operations associated with the MySQL database and the associated Lucene indexes.



Event Configuration

Don't Age This Severity and Above: Error

Event Aging Threshold (minutes): 60

Event Aging Interval (milliseconds): 60000

Event Aging Limit: 1000

Event Archive Threshold (minutes): 1440

Event Archive Interval (milliseconds): 60000

Event Archive Limit: 1000

Delete Archived Events Older Than (days): 7

Figure 42: Event Manager parameters that match with zeneventserver.log startup log

Lines highlighted in red are interacting with Rabbit MQ AMQP system. The first section shows zeneventserver connecting to the MQ subsystem; if this is unsuccessful then many of the Zenoss daemons will fail.

The second section shows the threads starting up to consume the various queues that zeneventserver processes.

- zenoss.queues.zep.zenevents
- zenoss.queues.zep.modelchange
- zenoss.queues.zep.heartbeats
- zenoss.queues.zep.migrated.summary
- zenoss.queues.zep.migrated.archive

Note that you would **not** expect to see zeneventserver working on zenoss.queues.zep.rawevents - the consumer of that queue is the zeneventd daemon.

Lines highlighted in light blue are subsequent, periodic operations by zeneventserver performing maintenance on the MySQL database. The log shows an event table partition being pruned every hour and a new one being created, as a section of events are aged into the event_archive table.

8.1.3 Other log files

Other log files that may have a bearing on events are:

● zenhub.log	interactions between daemons)	more useful prior
● event.log	problems seen by event.log)	to V4 for event issues
● zenperfsnmp.log	issues with performance data and threshold events	
● zenwinperf.log	issues with performance data and threshold events	
● zencommand.log	issues with performance data and threshold events	
● zensyslog.log	daemon that receives syslog events	
● zeneventlog.log	daemon that receives Windows events	
● zentrap.log	daemon that receives SNMP TRAPs	

8.2 Using zendmd to run Python commands

Zenoss provides a Python command line interface, **zendmd**, where code for transforms can be tested out and the attributes and methods available can be explored.

Note carefully the indentation of statements. Python is very particular about indentation to interpret structure such as for loops. It doesn't matter how many spaces you indent the body of the for loop but it **must** be indented from the *for* line and each line in the main body of that for loop must have the same indentation. The body of a for loop, inside a for loop, would indent further – and so on.

You should run zendmd as the *zenoss* user. This section is not supposed to be a Python tutorial; that said, here are a couple of tricks with zendmd.

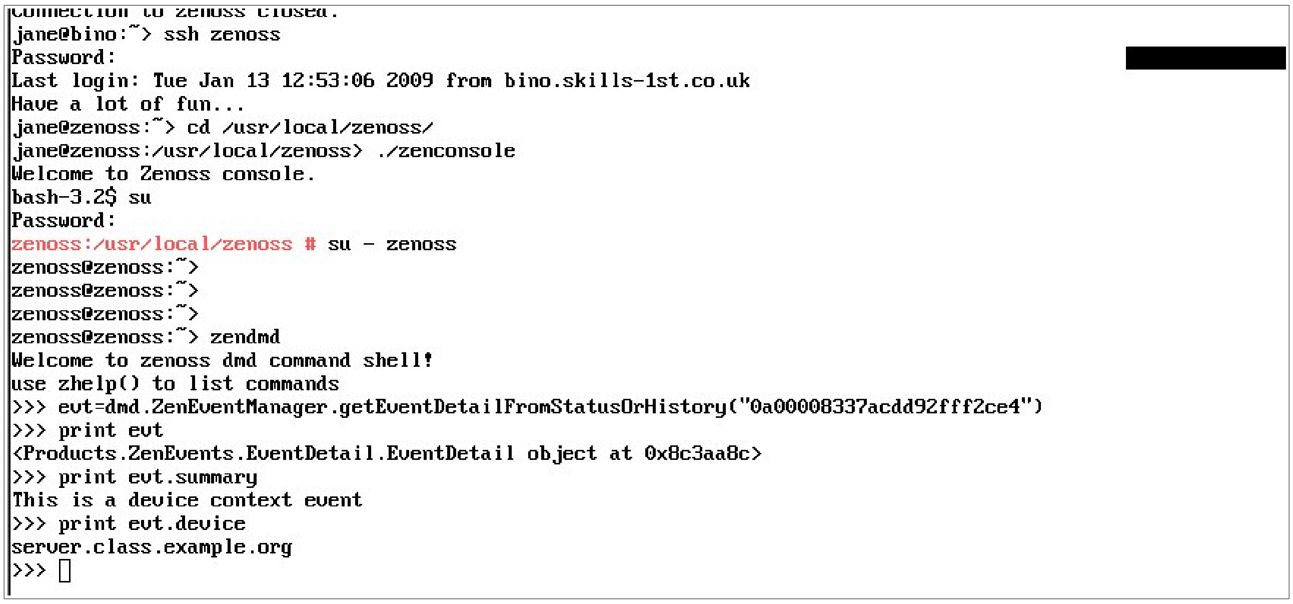
Note that these techniques for accessing events have changed substantially between previous versions of Zenoss and Zenoss 4.

8.2.1 Referencing an existing Zenoss event for use in zendmd

If you want to explore the attributes and methods available for an event or the device that generated the event, using zendmd, you need a way to reference an event. When executing a transform, these objects are made available to you automatically as the *evt* variable and the *device* variable – but in a zendmd test environment you need to supply these.

With earlier versions of Zenoss there was a method on the ZenEventManager, *getEventDetailFromStatusOrHistory*, which took as a parameter the string value of a unique evid and delivered an EventDetail object (see Figure 43).

To find the *evid,* simply display an appropriate event in the Event Console, bring up the detailed data, and cut and paste the *evid* value into the statement in zendmd.

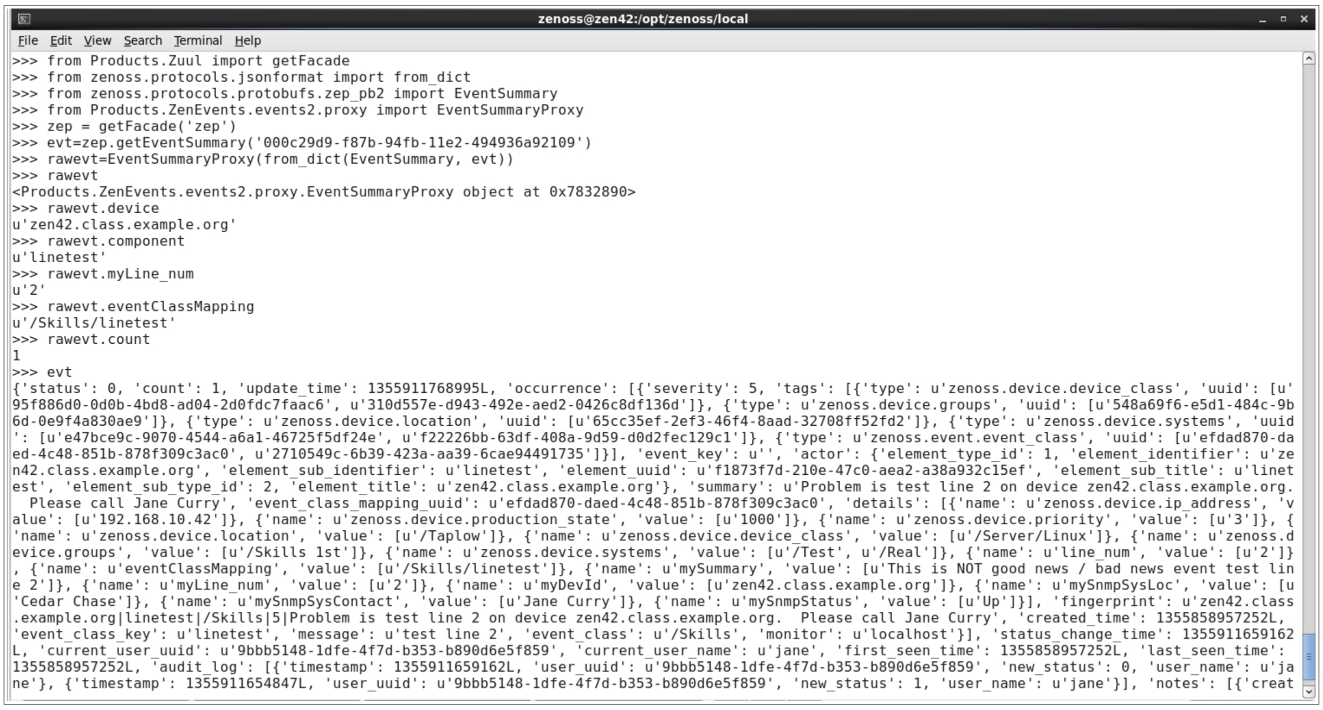


```
Connection to zenoss closed.
jane@bino:~> ssh zenoss
Password:                                                              [redacted]
Last login: Tue Jan 13 12:53:06 2009 from bino.skills-1st.co.uk
Have a lot of fun...
jane@zenoss:~> cd /usr/local/zenoss/
jane@zenoss:/usr/local/zenoss> ./zenconsole
Welcome to Zenoss console.
bash-3.2$ su
Password:
zenoss:/usr/local/zenoss # su - zenoss
zenoss@zenoss:~>
zenoss@zenoss:~>
zenoss@zenoss:~>
zenoss@zenoss:~> zendmd
Welcome to zenoss dmd command shell!
use zhelp() to list commands
>>> evt=dmd.ZenEventManager.getEventDetailFromStatusOrHistory("0a00008337acdd92fff2ce4")
>>> print evt
<Products.ZenEvents.EventDetail.EventDetail object at 0x8c3aa8c>
>>> print evt.summary
This is a device context event
>>> print evt.device
server.class.example.org
>>> 
```

Figure 43: Using zendmd to set the evt variable to an existing Zenoss event - Zenoss prior to V4

With Zenoss 4, it is a little more complex. We really need to get back to the ZepRawEventProxy format to test transform code, but that is no longer available - the data from the rawevent queue is gone.

What we do have access to is the event in the MySQL database; however we don't want it with database-style attributes, we want EventProxy attributes.


zenoss@zen42:/opt/zenoss/local
File Edit View Search Terminal Help

```
>>> from Products.Zuul import getFacade
>>> from zenoss.protocols.jsonformat import from_dict
>>> from zenoss.protocols.protobufs.zep_pb2 import EventSummary
>>> from Products.ZenEvents.events2.proxy import EventSummaryProxy
>>> zep = getFacade('zep')
>>> evt=zep.getEventSummary('000c29d9-f87b-94fb-11e2-494936a92109')
>>> rawevt=EventSummaryProxy(from_dict(EventSummary, evt))
>>> rawevt
<Products.ZenEvents.events2.proxy.EventSummaryProxy object at 0x7832890>
>>> rawevt.device
u'zen42.class.example.org'
>>> rawevt.component
u'linetest'
>>> rawevt.myLine_num
u'2'
>>> rawevt.eventClassMapping
u'/Skills/linetest'
>>> rawevt.count
1
>>> evt
{'status': 0, 'count': 1, 'update_time': 1355911768995L, 'occurrence': [{'severity': 5, 'tags': [{'type': u'zenoss.device.device_class', 'uuid': [u'
95f886d0-0d0b-4bd8-ad04-2d0fdc7faac6', u'310d557e-d943-492e-aed2-0426c8df136d']}, {'type': u'zenoss.device.groups', 'uuid': [u'548a69f6-e5d1-484c-9b
6d-0e9f4a830ae9']}, {'type': u'zenoss.device.location', 'uuid': [u'65cc35ef-2ef3-46f4-8aad-32708ff52fd2']}, {'type': u'zenoss.device.systems', 'uuid
': [u'e47bce9c-9070-4544-a6a1-46725f5df24e', u'f22226bb-63df-408a-9d59-d0d2fec129c1']}, {'type': u'zenoss.event.event_class', 'uuid': [u'efdad870-da
ed-4c48-851b-878f309c3ac0', u'2710549c-6b39-423a-aa39-6cae94491735']}], 'event_key': u'', 'actor': {'element_type_id': 1, 'element_identifier': u'ze
n42.class.example.org', 'element_sub_identifier': u'linetest', 'element_uuid': u'f1873f7d-210e-47c0-aea2-a38a932c15ef', 'element_sub_title': u'linet
est', 'element_sub_type_id': 2, 'element_title': u'zen42.class.example.org'}, 'summary': u'Problem is test line 2 on device zen42.class.example.org.
  Please call Jane Curry', 'event_class_mapping_uuid': u'efdad870-daed-4c48-851b-878f309c3ac0', 'details': [{'name': u'zenoss.device.ip_address', 'v
alue': [u'192.168.10.42']}, {'name': u'zenoss.device.production_state', 'value': [u'1000']}, {'name': u'zenoss.device.priority', 'value': [u'3']}, {
'name': u'zenoss.device.location', 'value': [u'/Taplow']}, {'name': u'zenoss.device.device_class', 'value': [u'/Server/Linux']}, {'name': u'zenoss.d
evice.groups', 'value': [u'/Skills 1st']}, {'name': u'zenoss.device.systems', 'value': [u'/Test', u'/Real']}, {'name': u'line_num', 'value': [u'2']}
, {'name': u'eventClassMapping', 'value': [u'/Skills/linetest']}, {'name': u'mySummary', 'value': [u'This is NOT good news / bad news event test lin
e 2']}, {'name': u'myLine_num', 'value': [u'2']}, {'name': u'myDevId', 'value': [u'zen42.class.example.org']}, {'name': u'mySnmpSysLoc', 'value': [u
'Cedar Chase']}, {'name': u'mySnmpSysContact', 'value': [u'Jane Curry']}, {'name': u'mySnmpStatus', 'value': [u'Up']}], 'fingerprint': u'zen42.class
.example.org|linetest|/Skills|5|Problem is test line 2 on device zen42.class.example.org.  Please call Jane Curry', 'created_time': 1355858957252L,
'event_class_key': u'linetest', 'message': u'test line 2', 'event_class': u'/Skills', 'monitor': u'localhost'}], 'status_change_time': 1355911659162
L, 'current_user_uuid': u'9bbb5148-1dfe-4f7d-b353-b890d6e5f859', 'current_user_name': u'jane', 'first_seen_time': 1355858957252L, 'last_seen_time':
1355858957252L, 'audit_log': [{'timestamp': 1355911659162L, 'user_uuid': u'9bbb5148-1dfe-4f7d-b353-b890d6e5f859', 'new_status': 0, 'user_name': u'ja
ne'}, {'timestamp': 1355911654847L, 'user_uuid': u'9bbb5148-1dfe-4f7d-b353-b890d6e5f859', 'new_status': 1, 'user_name': u'jane'}], 'notes': [{'creat
```

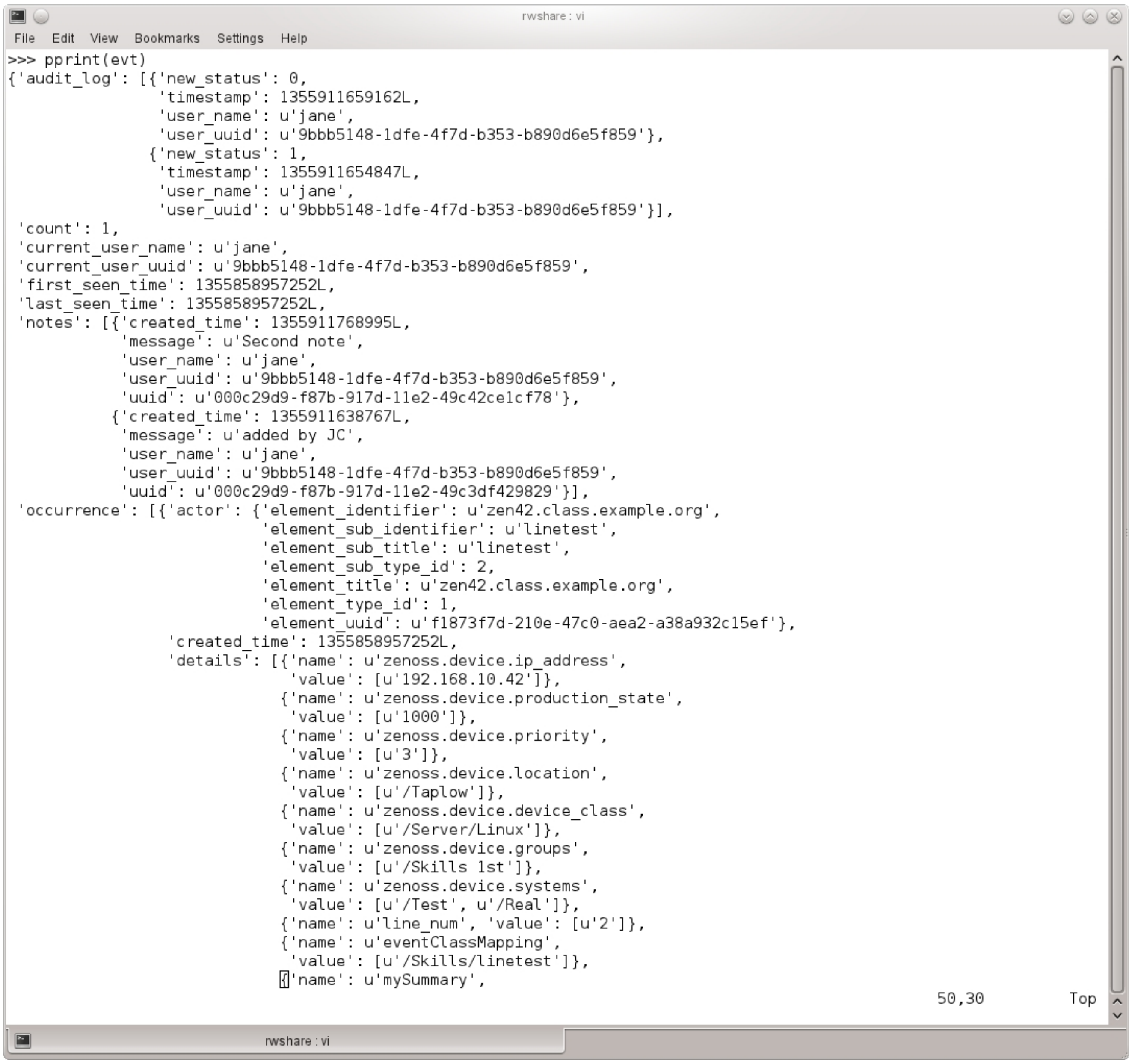
Figure 44: Using zendmd to retrieve an event from the MySQL database, convert to an EventSummaryProxy and extract various fields

$ZENHOME/Products/Zuul/facades/zepfacade.py provides a number of utilities to access data from the zenoss_zep database and manipulate it, typically providing JSON-format data.

Figure 44 demonstrates using zendmd to access events in the MySQL database, convert them to EventSummaryProxy format and then print out various fields.

- `zep = getFacade('zep')        provides access to the zenoss_zep database`
- `evt=zep.getEventSummary('000c29d9-f87b-94fb-11e2-494936a92109')`
  - Retrieves the event with the specified uuid - the result is in JSON
- `rawevt=EventSummaryProxy(from_dict(EventSummary, evt))`
  - The EventSummaryProxy class takes a protobuf-style event as parameter, not the JSON-style event we currently have. Use from_dict to convert from JSON to protobuf
- `rawevt.device                 standard attribute`
- `rawevt.myLineNum              attribute from details`
- **REMEMBER** that this is an EventSummaryProxy, not a ZepRawEventProxy so you have access to fields that are not available at transform time (like count, firstTime, ...)
- `evt                           the JSON-format event (dictionary)`

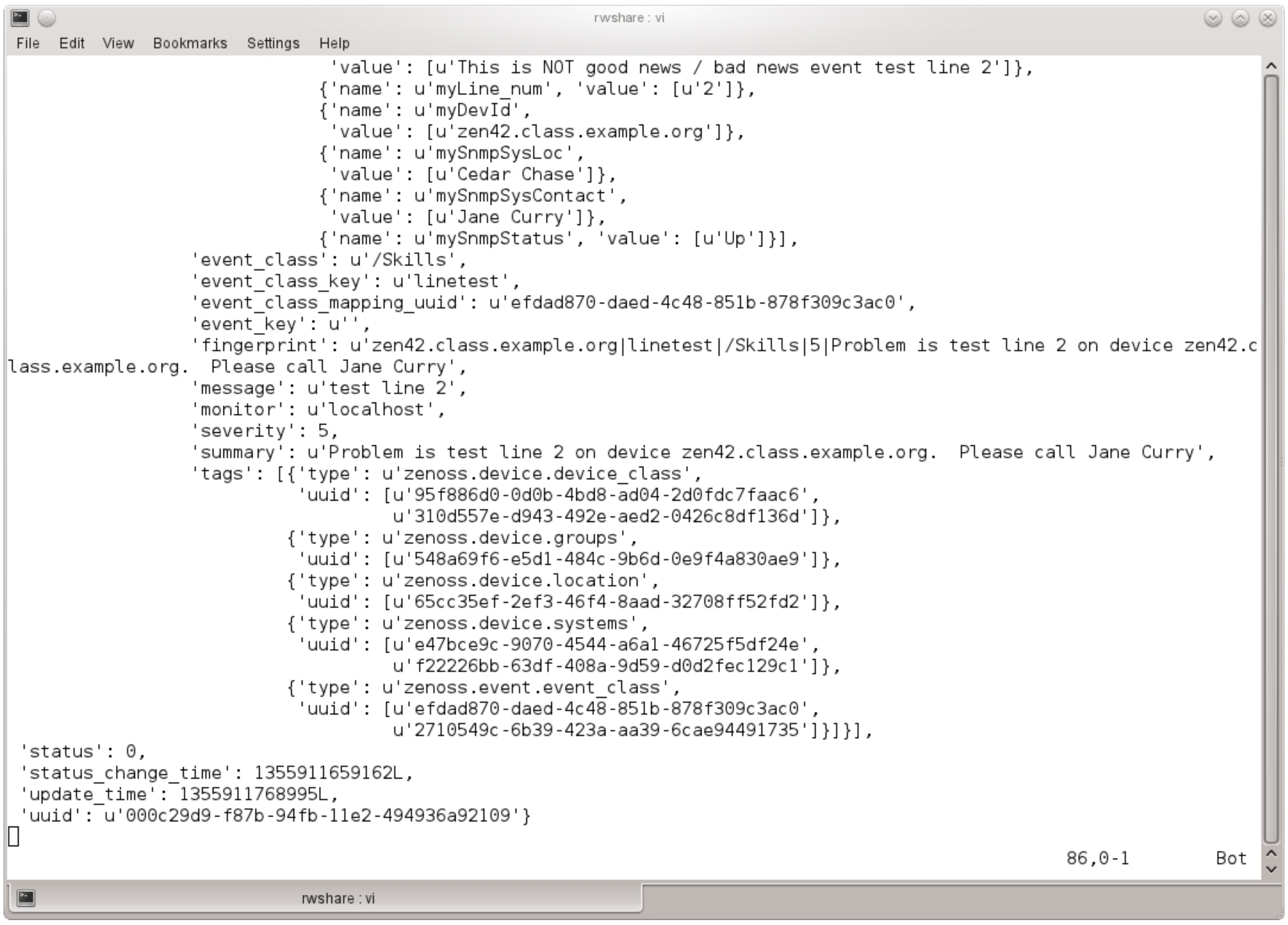
The JSON-style events are very hard to read as shown above. zendmd understands the *pprint* method to pretty-print complex structures. It can be useful to capture the output of *pprint(evt)* into a file and then use the vi editor % technique to help match opening and closing brackets.



```
>>> pprint(evt)
{'audit_log': [{'new_status': 0,
                'timestamp': 1355911659162L,
                'user_name': u'jane',
                'user_uuid': u'9bbb5148-1dfe-4f7d-b353-b890d6e5f859'},
               {'new_status': 1,
                'timestamp': 1355911654847L,
                'user_name': u'jane',
                'user_uuid': u'9bbb5148-1dfe-4f7d-b353-b890d6e5f859'}],
 'count': 1,
 'current_user_name': u'jane',
 'current_user_uuid': u'9bbb5148-1dfe-4f7d-b353-b890d6e5f859',
 'first_seen_time': 1355858957252L,
 'last_seen_time': 1355858957252L,
 'notes': [{'created_time': 1355911768995L,
            'message': u'Second note',
            'user_name': u'jane',
            'user_uuid': u'9bbb5148-1dfe-4f7d-b353-b890d6e5f859',
            'uuid': u'000c29d9-f87b-917d-11e2-49c42ce1cf78'},
           {'created_time': 1355911638767L,
            'message': u'added by JC',
            'user_name': u'jane',
            'user_uuid': u'9bbb5148-1dfe-4f7d-b353-b890d6e5f859',
            'uuid': u'000c29d9-f87b-917d-11e2-49c3df429829'}],
 'occurrence': [{'actor': {'element_identifier': u'zen42.class.example.org',
                           'element_sub_identifier': u'linetest',
                           'element_sub_title': u'linetest',
                           'element_sub_type_id': 2,
                           'element_title': u'zen42.class.example.org',
                           'element_type_id': 1,
                           'element_uuid': u'f1873f7d-210e-47c0-aea2-a38a932c15ef'},
                 'created_time': 1355858957252L,
                 'details': [{'name': u'zenoss.device.ip_address',
                              'value': [u'192.168.10.42']},
                             {'name': u'zenoss.device.production_state',
                              'value': [u'1000']},
                             {'name': u'zenoss.device.priority',
                              'value': [u'3']},
                             {'name': u'zenoss.device.location',
                              'value': [u'/Taplow']},
                             {'name': u'zenoss.device.device_class',
                              'value': [u'/Server/Linux']},
                             {'name': u'zenoss.device.groups',
                              'value': [u'/Skills 1st']},
                             {'name': u'zenoss.device.systems',
                              'value': [u'/Test', u'/Real']},
                             {'name': u'line_num', 'value': [u'2']},
                             {'name': u'eventClassMapping',
                              'value': [u'/Skills/linetest']},
                             {'name': u'mySummary',
                                                                       50,30         Top
```

Figure 45First part of zendmd pprint(evt) command displaying summary event in JSON format

Remember that Figure 45 and Figure 46 are showing the JSON-style event, not the EventSummaryProxy that delivers suitable attributes for transform manipulation.

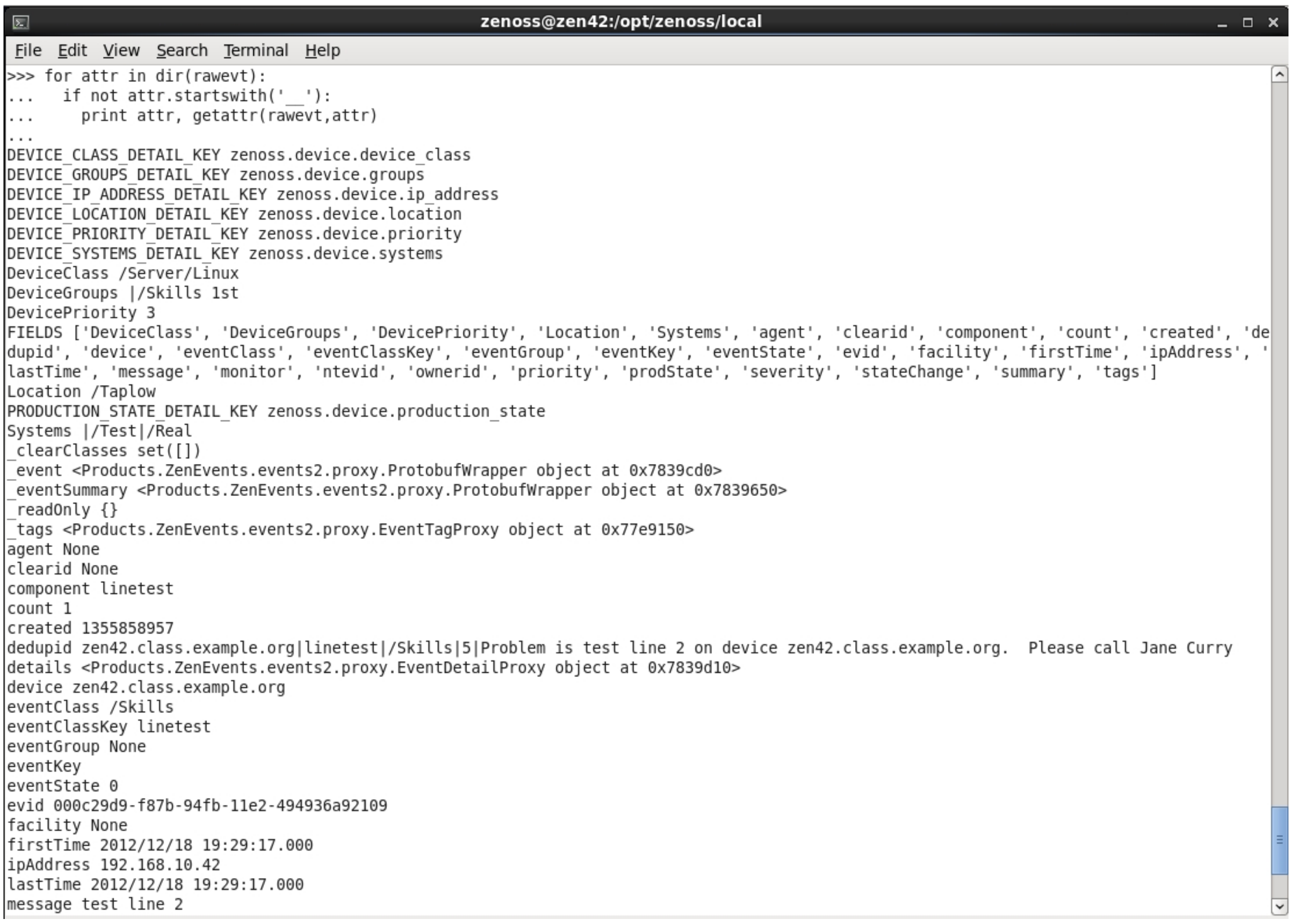


```
                            'value': [u'This is NOT good news / bad news event test line 2']},
                           {'name': u'myLine_num', 'value': [u'2']},
                           {'name': u'myDevId',
                            'value': [u'zen42.class.example.org']},
                           {'name': u'mySnmpSysLoc',
                            'value': [u'Cedar Chase']},
                           {'name': u'mySnmpSysContact',
                            'value': [u'Jane Curry']},
                           {'name': u'mySnmpStatus', 'value': [u'Up']}],
                'event_class': u'/Skills',
                'event_class_key': u'linetest',
                'event_class_mapping_uuid': u'efdad870-daed-4c48-851b-878f309c3ac0',
                'event_key': u'',
                'fingerprint': u'zen42.class.example.org|linetest|/Skills|5|Problem is test line 2 on device zen42.c
lass.example.org.  Please call Jane Curry',
                'message': u'test line 2',
                'monitor': u'localhost',
                'severity': 5,
                'summary': u'Problem is test line 2 on device zen42.class.example.org.  Please call Jane Curry',
                'tags': [{'type': u'zenoss.device.device_class',
                          'uuid': [u'95f886d0-0d0b-4bd8-ad04-2d0fdc7faac6',
                                   u'310d557e-d943-492e-aed2-0426c8df136d']},
                         {'type': u'zenoss.device.groups',
                          'uuid': [u'548a69f6-e5d1-484c-9b6d-0e9f4a830ae9']},
                         {'type': u'zenoss.device.location',
                          'uuid': [u'65cc35ef-2ef3-46f4-8aad-32708ff52fd2']},
                         {'type': u'zenoss.device.systems',
                          'uuid': [u'e47bce9c-9070-4544-a6a1-46725f5df24e',
                                   u'f22226bb-63df-408a-9d59-d0d2fec129c1']},
                         {'type': u'zenoss.event.event_class',
                          'uuid': [u'efdad870-daed-4c48-851b-878f309c3ac0',
                                   u'2710549c-6b39-423a-aa39-6cae94491735']}]}],
 'status': 0,
 'status_change_time': 1355911659162L,
 'update_time': 1355911768995L,
 'uuid': u'000c29d9-f87b-94fb-11e2-494936a92109'}

                                                                                86,0-1        Bot
```

Figure 46Second part of zendmd pprint(evt) command displaying summary event in JSON format

8.2.2 Using zendmd to understand attributes for an EventSummaryProxy

An EventSummaryProxy is an object class representing a Zenoss event – it is a Python **dictionary** data type – a data structure of <key> , <value> pairs. To see what keys (attributes) are available, use the method shown in the following figure. Built-in methods starting with a double underscore are deliberately excluded.

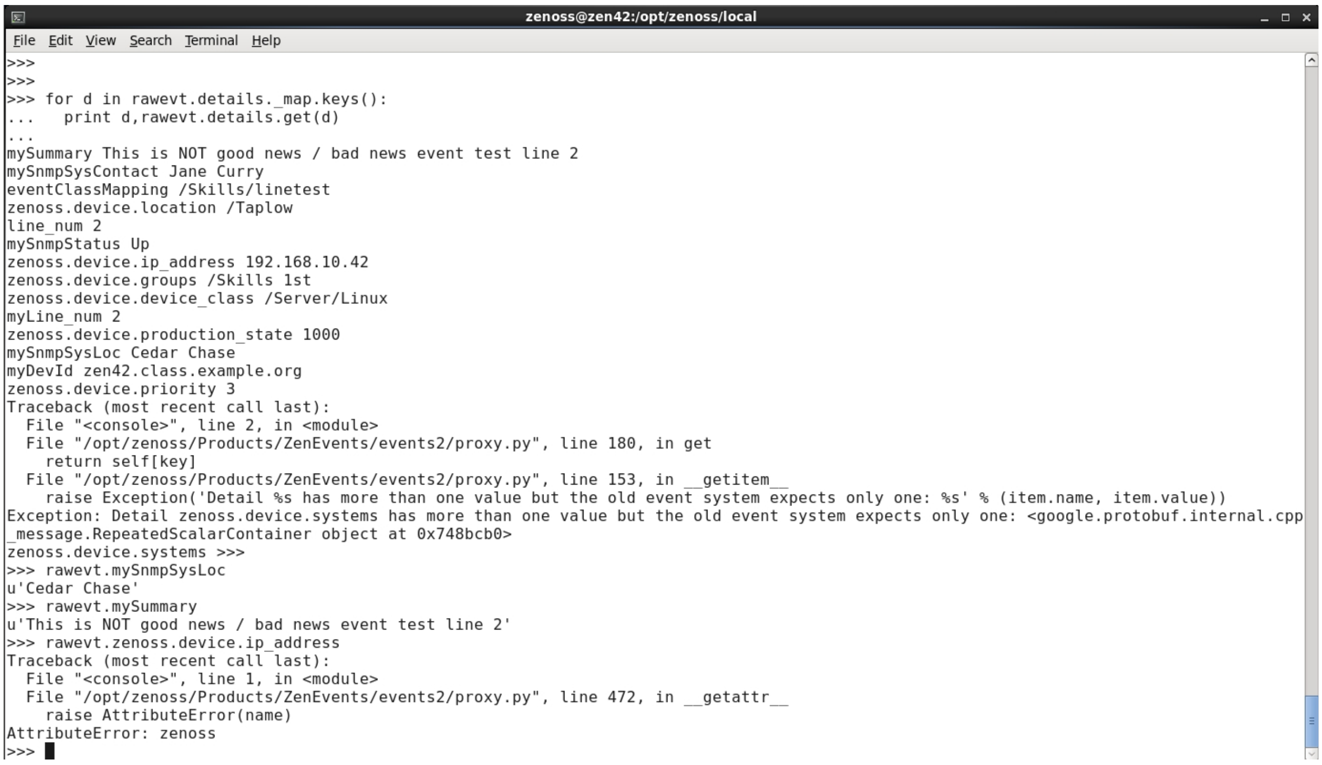

zenoss@zen42:/opt/zenoss/local
File Edit View Search Terminal Help

```
>>> for attr in dir(rawevt):
...   if not attr.startswith('__'):
...     print attr, getattr(rawevt,attr)
...
DEVICE_CLASS_DETAIL_KEY zenoss.device.device_class
DEVICE_GROUPS_DETAIL_KEY zenoss.device.groups
DEVICE_IP_ADDRESS_DETAIL_KEY zenoss.device.ip_address
DEVICE_LOCATION_DETAIL_KEY zenoss.device.location
DEVICE_PRIORITY_DETAIL_KEY zenoss.device.priority
DEVICE_SYSTEMS_DETAIL_KEY zenoss.device.systems
DeviceClass /Server/Linux
DeviceGroups |/Skills 1st
DevicePriority 3
FIELDS ['DeviceClass', 'DeviceGroups', 'DevicePriority', 'Location', 'Systems', 'agent', 'clearid', 'component', 'count', 'created', 'de
dupid', 'device', 'eventClass', 'eventClassKey', 'eventGroup', 'eventKey', 'eventState', 'evid', 'facility', 'firstTime', 'ipAddress', '
lastTime', 'message', 'monitor', 'ntevid', 'ownerid', 'priority', 'prodState', 'severity', 'stateChange', 'summary', 'tags']
Location /Taplow
PRODUCTION_STATE_DETAIL_KEY zenoss.device.production_state
Systems |/Test|/Real
_clearClasses set([])
_event <Products.ZenEvents.events2.proxy.ProtobufWrapper object at 0x7839cd0>
_eventSummary <Products.ZenEvents.events2.proxy.ProtobufWrapper object at 0x7839650>
_readOnly {}
_tags <Products.ZenEvents.events2.proxy.EventTagProxy object at 0x77e9150>
agent None
clearid None
component linetest
count 1
created 1355858957
dedupid zen42.class.example.org|linetest|/Skills|5|Problem is test line 2 on device zen42.class.example.org.  Please call Jane Curry
details <Products.ZenEvents.events2.proxy.EventDetailProxy object at 0x7839d10>
device zen42.class.example.org
eventClass /Skills
eventClassKey linetest
eventGroup None
eventKey
eventState 0
evid 000c29d9-f87b-94fb-11e2-494936a92109
facility None
firstTime 2012/12/18 19:29:17.000
ipAddress 192.168.10.42
lastTime 2012/12/18 19:29:17.000
message test line 2
```

Figure 47: Using zendmd to print event attribute <key> <value> pairs (partial listing)

These are the primary event fields that are available to use in a transform (remembering to also exclude those that don't exist at rawevent time eg. count, firstTime, eventState, ...).

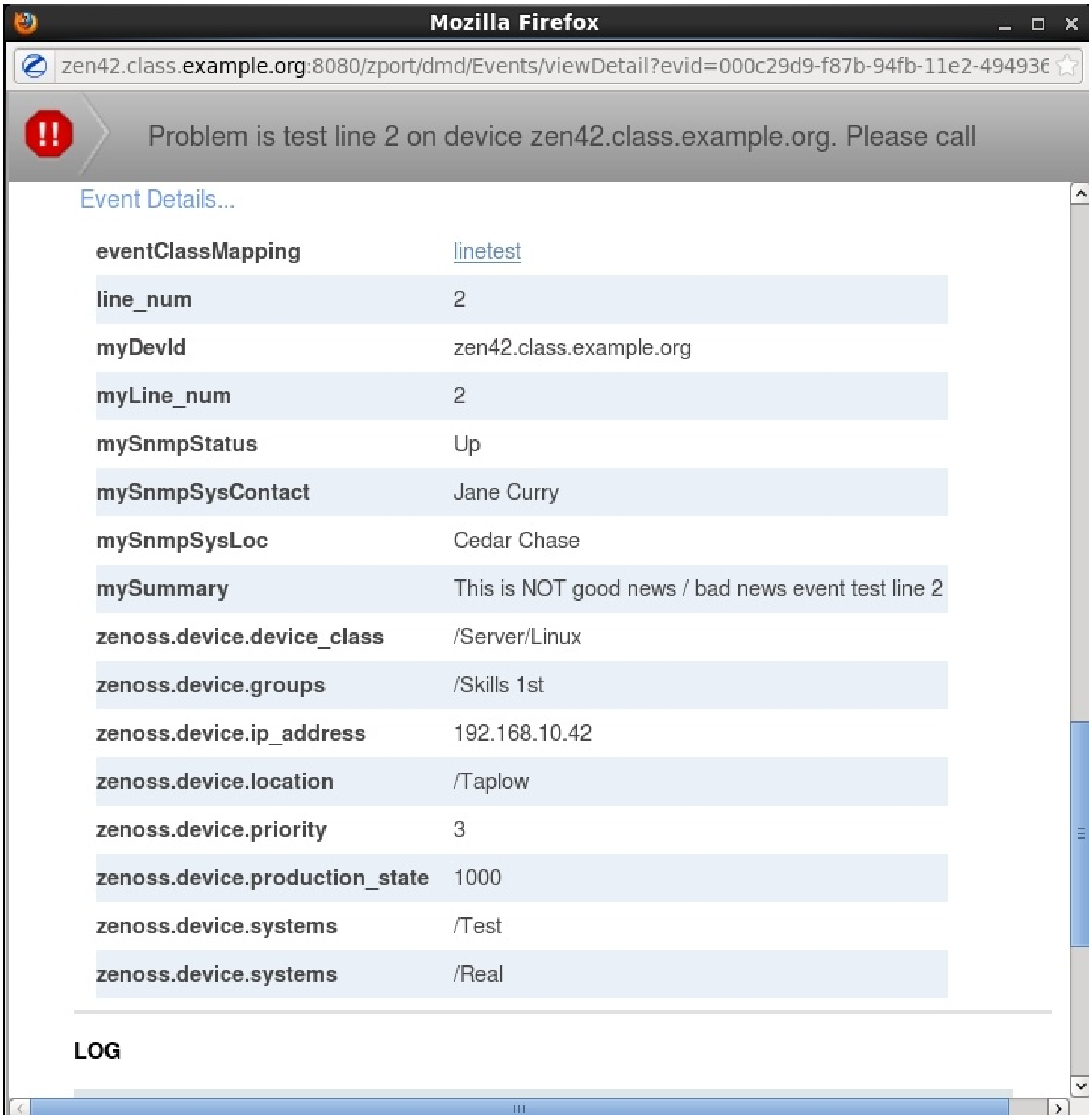
Note that some of the dictionary elements are themselves dictionaries eg. *details*. To find out what the details attributes are, see Figure 48. Remember from Figure 38, that the EventDetailProxy class has an _map dictionary with *name,value* pairs in it.


zenoss@zen42:/opt/zenoss/local
File Edit View Search Terminal Help

```
>>>
>>>
>>> for d in rawevt.details._map.keys():
...   print d,rawevt.details.get(d)
...
mySummary This is NOT good news / bad news event test line 2
mySnmpSysContact Jane Curry
eventClassMapping /Skills/linetest
zenoss.device.location /Taplow
line_num 2
mySnmpStatus Up
zenoss.device.ip_address 192.168.10.42
zenoss.device.groups /Skills 1st
zenoss.device.device_class /Server/Linux
myLine_num 2
zenoss.device.production_state 1000
mySnmpSysLoc Cedar Chase
myDevId zen42.class.example.org
zenoss.device.priority 3
Traceback (most recent call last):
  File "<console>", line 2, in <module>
  File "/opt/zenoss/Products/ZenEvents/events2/proxy.py", line 180, in get
    return self[key]
  File "/opt/zenoss/Products/ZenEvents/events2/proxy.py", line 153, in __getitem__
    raise Exception('Detail %s has more than one value but the old event system expects only one: %s' % (item.name, item.value))
Exception: Detail zenoss.device.systems has more than one value but the old event system expects only one: <google.protobuf.internal.cpp
_message.RepeatedScalarContainer object at 0x748bcb0>
zenoss.device.systems >>>
>>> rawevt.mySnmpSysLoc
u'Cedar Chase'
>>> rawevt.mySummary
u'This is NOT good news / bad news event test line 2'
>>> rawevt.zenoss.device.ip_address
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/opt/zenoss/Products/ZenEvents/events2/proxy.py", line 472, in __getattr__
    raise AttributeError(name)
AttributeError: zenoss
>>> 
```

Figure 48zendmd to display event details dictionary name,value pairs

The **get** method of EventDetailProxy delivers values when that item is a **single, scalar** value. If the item has multiple values, a list for example, then the get method breaks as shown above on the *zenoss.device.systems* attribute. Note that it “gets away with” the *zenoss.device.groups* attribute because, although a device may be in multiple groups, in this case the device is only in a single group, whereas it is a member of two Systems. This is also echoed in the Event Details of the Zenoss GUI.


Mozilla Firefox

zen42.class.example.org:8080/zport/dmd/Events/viewDetail?evid=000c29d9-f87b-94fb-11e2-49493

Problem is test line 2 on device zen42.class.example.org. Please call

Event Details...

eventClassMapping	linetest
line_num	2
myDevId	zen42.class.example.org
myLine_num	2
mySnmpStatus	Up
mySnmpSysContact	Jane Curry
mySnmpSysLoc	Cedar Chase
mySummary	This is NOT good news / bad news event test line 2
zenoss.device.device_class	/Server/Linux
zenoss.device.groups	/Skills 1st
zenoss.device.ip_address	192.168.10.42
zenoss.device.location	/Taplow
zenoss.device.priority	3
zenoss.device.production_state	1000
zenoss.device.systems	/Test
zenoss.device.systems	/Real

LOG

Figure 49Zenoss GUI Event Details showing one instance of zenoss.device.groups and 2 instances of zenoss.device.systems

If an event details attribute is not a scalar, use the **getAll** method rather than the get method. For example:

```
>>> print list(rawevt.details.getAll('zenoss.device.systems'))
[u'/Test', u'/Real']
>>>
```

Also note in Figure 48 that user-defined detail attributes can simply be referred to as *rawevt.mySummary* or *rawevt.mySnmpSysLoc* but you cannot refer to detail fields that contain a . (dot) in this way thus excluding the default details attributes (those starting with *zenoss.*) and excluding SNMP TRAP varbind fields that typically contain a dot; use the get and getAll methods to access such detail fields.

8.3 Using the Python debugger in transforms

A very powerful aid when debugging any Python is to use the Python Debugger, **pdb**. See http://docs.python.org/2/library/pdb.html for detailed documentation.

pdb allows you to break execution, display the state of objects and their values and step through the code. When used in transforms, this means running zeneventd in the foreground in debug mode (so **definitely not** a technique for use in production).

When using pdb to examine transforms, it is not easy to step through the transform code (using *s* to step or *n* for next) as you end up nested many layers deep in the methods of the zeneventd code; however it is very useful to examine the state of the event (*evt*) and also explore the device (*device*).

If you are doing this, you may wish to reduce the Zenoss system to a minimum set of daemons to avoid events from lots of other sources.

If *$ZENHOME/etc/DAEMONS_TXT_ONLY* exists then the only Zenoss daemons that will be manipulated by a *zenoss start* / *zenoss stop* / *zenoss status* will be those listed in *$ZENHOME/etc/daemons.txt*. A minimum set of daemons would be:

- zeneventserver
- zopectl
- zeneventd
- zenhub
- zenjobs
- zenactiond

When you have restarted Zenoss, go to *ADVANCED* -> *Settings* -> *Events*, scroll to the bottom of the page and click *Clear* . This prevents the heartbeat from periodically checking all those daemons that are now down and generating heartbeat events..

To put a break point at the start of a transform, add the following line:

```
import pdb; pdb.set_trace()
```

Stop the zeneventd daemon and start it in the foreground in debug mode:

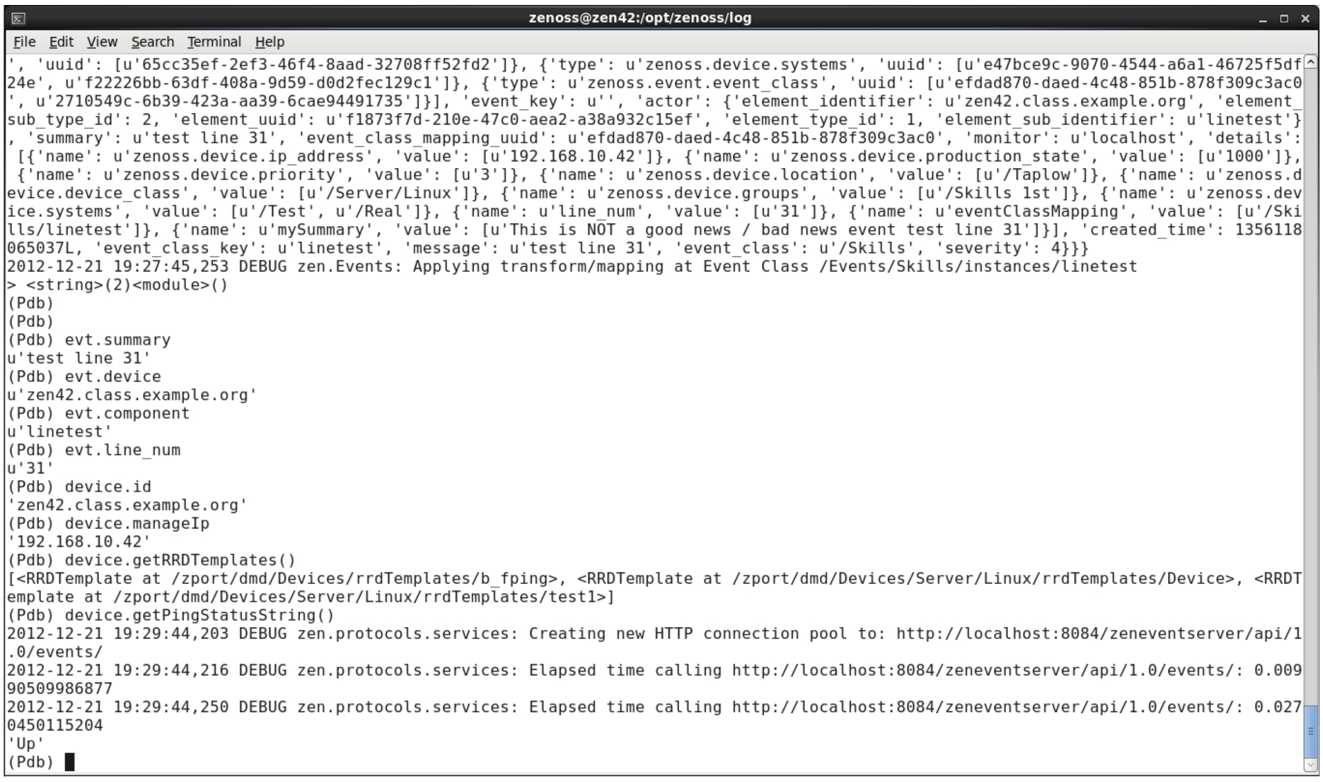
```
zeneventd stop
zeneventd run -v 10
```

Generate an event that will trigger the transform; for example:

```
zensendevent -d zen42.class.example.org -s Error -k linetest -p linetest test line 24
```

In the zeneventd foreground window you should see a pdb prompt. You should now have access to:

evt	a ZepRawEventProxy object
device	a Device object


zenoss@zen42:/opt/zenoss/log
File Edit View Search Terminal Help

```
', 'uuid': [u'65cc35ef-2ef3-46f4-8aad-32708ff52fd2']}, {'type': u'zenoss.device.systems', 'uuid': [u'e47bce9c-9070-4544-a6a1-46725f5df
24e', u'f22226bb-63df-408a-9d59-d0d2fec129c1']}, {'type': u'zenoss.event.event_class', 'uuid': [u'efdad870-daed-4c48-851b-878f309c3ac0
', u'2710549c-6b39-423a-aa39-6cae94491735']}], 'event_key': u'', 'actor': {'element_identifier': u'zen42.class.example.org', 'element_
sub_type_id': 2, 'element_uuid': u'f1873f7d-210e-47c0-aea2-a38a932c15ef', 'element_type_id': 1, 'element_sub_identifier': u'linetest'}
, 'summary': u'test line 31', 'event_class_mapping_uuid': u'efdad870-daed-4c48-851b-878f309c3ac0', 'monitor': u'localhost', 'details':
 [{'name': u'zenoss.device.ip_address', 'value': [u'192.168.10.42']}, {'name': u'zenoss.device.production_state', 'value': [u'1000']},
 {'name': u'zenoss.device.priority', 'value': [u'3']}, {'name': u'zenoss.device.location', 'value': [u'/Taplow']}, {'name': u'zenoss.d
evice.device_class', 'value': [u'/Server/Linux']}, {'name': u'zenoss.device.groups', 'value': [u'/Skills 1st']}, {'name': u'zenoss.dev
ice.systems', 'value': [u'/Test', u'/Real']}, {'name': u'line_num', 'value': [u'31']}, {'name': u'eventClassMapping', 'value': [u'/Ski
lls/linetest']}, {'name': u'mySummary', 'value': [u'This is NOT a good news / bad news event test line 31']}], 'created_time': 1356118
065037L, 'event_class_key': u'linetest', 'message': u'test line 31', 'event_class': u'/Skills', 'severity': 4}}}
2012-12-21 19:27:45,253 DEBUG zen.Events: Applying transform/mapping at Event Class /Events/Skills/instances/linetest
> <string>(2)<module>()
(Pdb)
(Pdb)
(Pdb) evt.summary
u'test line 31'
(Pdb) evt.device
u'zen42.class.example.org'
(Pdb) evt.component
u'linetest'
(Pdb) evt.line_num
u'31'
(Pdb) device.id
'zen42.class.example.org'
(Pdb) device.manageIp
'192.168.10.42'
(Pdb) device.getRRDTemplates()
[<RRDTemplate at /zport/dmd/Devices/rrdTemplates/b_fping>, <RRDTemplate at /zport/dmd/Devices/Server/Linux/rrdTemplates/Device>, <RRDT
emplate at /zport/dmd/Devices/Server/Linux/rrdTemplates/test1>]
(Pdb) device.getPingStatusString()
2012-12-21 19:29:44,203 DEBUG zen.protocols.services: Creating new HTTP connection pool to: http://localhost:8084/zeneventserver/api/1
.0/events/
2012-12-21 19:29:44,216 DEBUG zen.protocols.services: Elapsed time calling http://localhost:8084/zeneventserver/api/1.0/events/: 0.009
90509986877
2012-12-21 19:29:44,250 DEBUG zen.protocols.services: Elapsed time calling http://localhost:8084/zeneventserver/api/1.0/events/: 0.027
0450115204
'Up'
(Pdb) █
```

Figure 50: pdb dialogue in zeneventd foreground - generated by pdb.set_trace() in transform

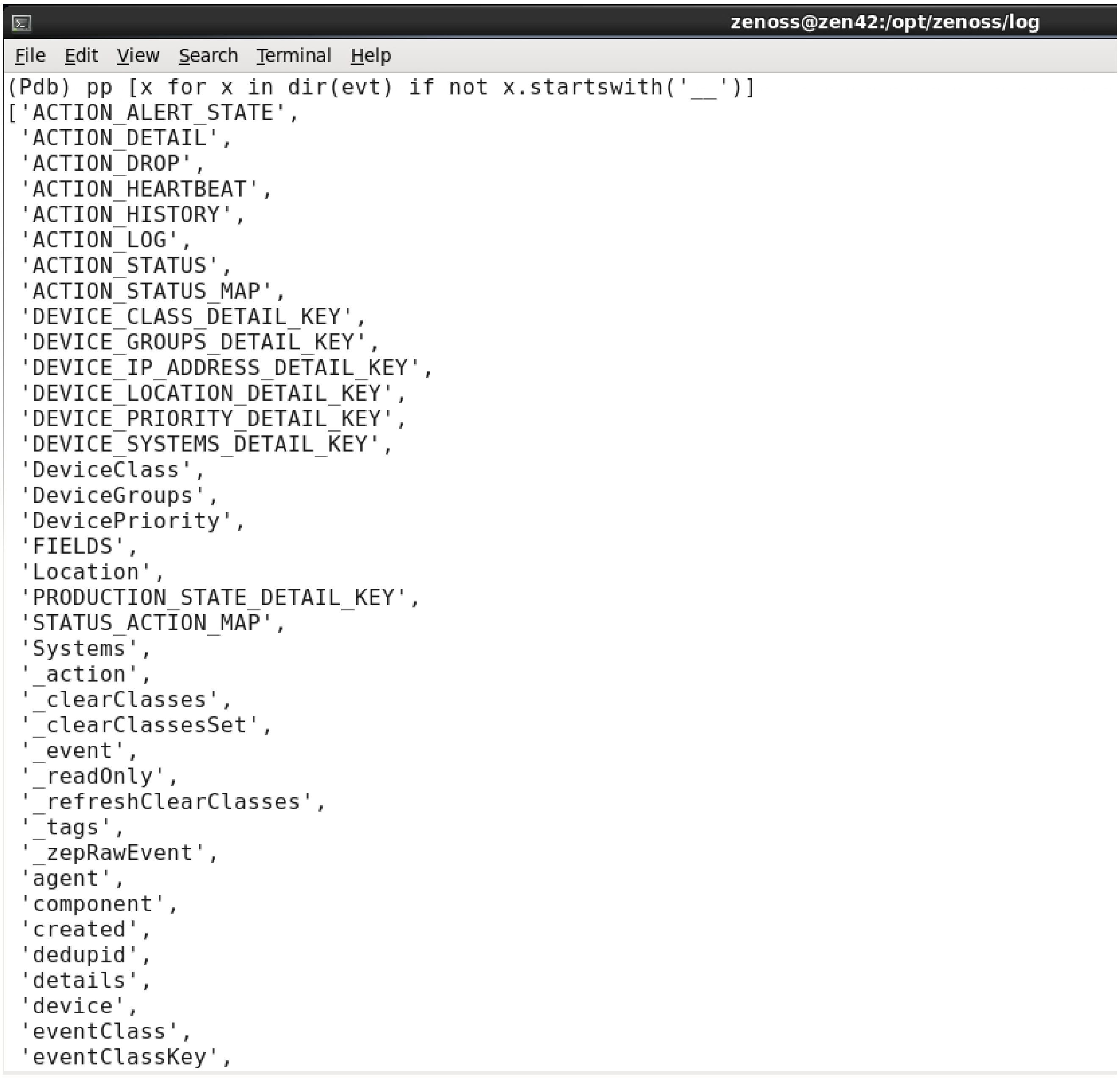
Figure 50 demonstrates exploring some of the attributes of both *evt* and *device*. Note that entering a simple carriage-return repeats the previous pdb command.

c in pdb continues execution.

To see legal attributes and methods for the Device object, examine the *Device* class definition in *$ZENHOME/Products/ZenModel/Device.py*.

pdb does not have the pprint method seen in zendmd but it does have an equivalent *pp* utility. For example, to print all primary event fields, excluding built-in methods, use:

```
        pp [x for x in dir(evt) if not x.startswith('__')]
```



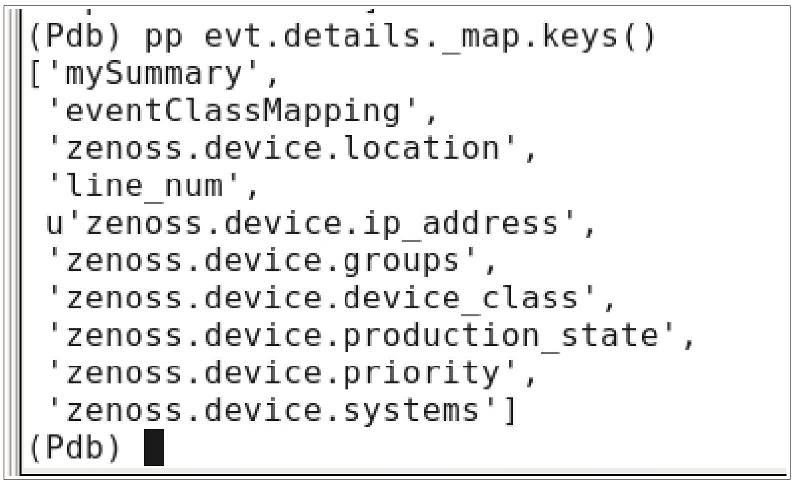
zenoss@zen42:/opt/zenoss/log
File Edit View Search Terminal Help

```
(Pdb) pp [x for x in dir(evt) if not x.startswith('__')]
['ACTION_ALERT_STATE',
 'ACTION_DETAIL',
 'ACTION_DROP',
 'ACTION_HEARTBEAT',
 'ACTION_HISTORY',
 'ACTION_LOG',
 'ACTION_STATUS',
 'ACTION_STATUS_MAP',
 'DEVICE_CLASS_DETAIL_KEY',
 'DEVICE_GROUPS_DETAIL_KEY',
 'DEVICE_IP_ADDRESS_DETAIL_KEY',
 'DEVICE_LOCATION_DETAIL_KEY',
 'DEVICE_PRIORITY_DETAIL_KEY',
 'DEVICE_SYSTEMS_DETAIL_KEY',
 'DeviceClass',
 'DeviceGroups',
 'DevicePriority',
 'FIELDS',
 'Location',
 'PRODUCTION_STATE_DETAIL_KEY',
 'STATUS_ACTION_MAP',
 'Systems',
 '_action',
 '_clearClasses',
 '_clearClassesSet',
 '_event',
 '_readOnly',
 '_refreshClearClasses',
 '_tags',
 '_zepRawEvent',
 'agent',
 'component',
 'created',
 'dedupid',
 'details',
 'device',
 'eventClass',
 'eventClassKey',
```

Figure 51: Using pdb to pretty print all primary event fields

To show detail fields:

```
pp evt.details._map.keys()
```



```
(Pdb) pp evt.details._map.keys()
['mySummary',
 'eventClassMapping',
 'zenoss.device.location',
 'line_num',
 u'zenoss.device.ip_address',
 'zenoss.device.groups',
 'zenoss.device.device_class',
 'zenoss.device.production_state',
 'zenoss.device.priority',
 'zenoss.device.systems']
(Pdb)
```

Figure 52: Using pdb to display detail event fields

To print a scalar value for a detail event field, try:

```
(Pdb) print evt.details.get('zenoss.device.device_class')
/Server/Linux
(Pdb) print evt.details.get('mySummary')
This is NOT a good news / bad news event test line 31
(Pdb)
```

To print a non-scalar (a list for example):

```
(Pdb) print list(evt.details.getAll('zenoss.device.systems'))
[u'/Test', u'/Real']
```

An attempt to print all detail field names and values might be:

```
pp [(v,evt.details.get(v)) for v in evt.details._map.keys()]
*** Exception: Exception(u'Detail zenoss.device.systems has more than one
value but the old event system expects only one:
<google.protobuf.internal.cpp_message.RepeatedScalarContainer object at
0x5e01ef0>',)
(Pdb)
```

This comes up against the problem described in the zendmd section where the *get* method fails with non-scalar values. A partial circumvention, given the knowledge that none of the user-defined variables are non-scalar, would be:

```
(Pdb) pp [(v,evt.details.get(v)) for v in evt.details._map.keys() if not
                v.startswith('zenoss.device')]
[('mySummary', u'This is NOT a good news / bad news event test line 31'),
 ('eventClassMapping', u'/Skills/linetest'),
 ('line_num', u'31')]
(Pdb)
```

Perhaps a better solution is to accept all values as lists and use the getAll method, which then works for all event details name,value pairs.

```
(Pdb) pp [(v, list(evt.details.getAll(v))) for v in evt.details._map.keys()]
[('mySummary', [u'This is NOT a good news / bad news event test line 31']),
 ('eventClassMapping', [u'/Skills/linetest']),
 ('zenoss.device.location', [u'/Taplow']),
 ('line_num', [u'31']),
 (u'zenoss.device.ip_address', [u'192.168.10.42']),
 ('zenoss.device.groups', [u'/Skills 1st']),
 ('zenoss.device.device_class', [u'/Server/Linux']),
 ('zenoss.device.production_state', [u'1000']),
 ('zenoss.device.priority', [u'3']),
 ('zenoss.device.systems', [u'/Test', u'/Real'])]
(Pdb)
```

9 Zenoss and SNMP

9.1 SNMP introduction

The Simple Network Management Protocol (SNMP) defines Management Information Base (MIB) variables that can be polled to provide performance and configuration information. The SNMP standard also provides for agents to send “events” to a manager. Version 1 of SNMP defines these as TRAPs; versions 2 and 3 of the standard calls them NOTIFICATIONs (Zenoss supports all three versions of SNMP). Both MIB variables and TRAPs / NOTIFICATIONs use Object Identifiers (OIDs) to denote different variables and events.

SNMP TRAPs are distinguished by their Enterprise Object Id (OID), the generic TRAP number and the specific TRAP number.

Natively, OIDs are defined as strings of dotted decimals that represent a path through a tree-based hierarchy, where the root of the tree is 1 and represents the ISO organisation; it has a sub-branch, 3, which represents organisations (org); it has a sub-branch, 6, which represents the US Department of Defense (dod); it has a sub-branch, 1, which represents internet, and so on. Thus, all OIDs start with 1.3.6.1 .

There is a standard, **MIB-2**, which defines a number of variables that every SNMP-capable device must support; these are largely simple, network-related variables, such as interfaceInOctets. In addition to MIB-2, there are a large number of standardised MIBs defined in Request For Comment (RFC) documents; an example would be RFC 1493 defining the bridge MIB.

The third category of MIBs are known as **Enterprise Specific**, which are specific to a particular vendor's particular agent – for example , the Cisco Firewall MIB. Enterprise specific MIBs often include definitions of Enterprise Specific TRAPs , in addition to MIB variables.

MIB source files translate dotted-decimal OIDs into more meaningful text. MIB files are available for many standards (like the HOST-RESOURCES MIB) and, typically, any supplier who generates their own enterprise specific MIB variables and TRAPs, should make available a source MIB file to aid this translation.

SNMP agents typically come as part of the base Operating System (Windows, Unix, Linux, Cisco IOS); however they may not be activated automatically and will require some configuration. Some agents support little more than MIB-2; others support a wide range of standard MIBs and enterprise specific MIBs.

The SNMP communication protocol varies depending on the version of SNMP. Versions 1 and 2 (strictly 2c) use a **community** name string as an authentication mechanism between SNMP manager and agent. Managers must be configured with the correct community names to use for an agent; SNMP agents must be configured for which manager(s) are allowed access to them, and which SNMP manager(s) to send TRAPs to.

SNMP V3 is more complex to configure but provides facilities for strong authentication on SNMP packets and for encryption of data if so desired.

In addition to requesting MIB-2 variables, Zenoss will try to access the standard Host Resources MIB to get process information for server machines. It will also attempt to access the Windows Informant MIB for all Windows server systems, in order to get CPU and file system information. The Informant MIB is a free extension subagent and MIB available from Informant at http://www.wtcs.org/informant/index.htm . Note that the base Windows SNMP agent should be installed and configured before installing the Informant extension.

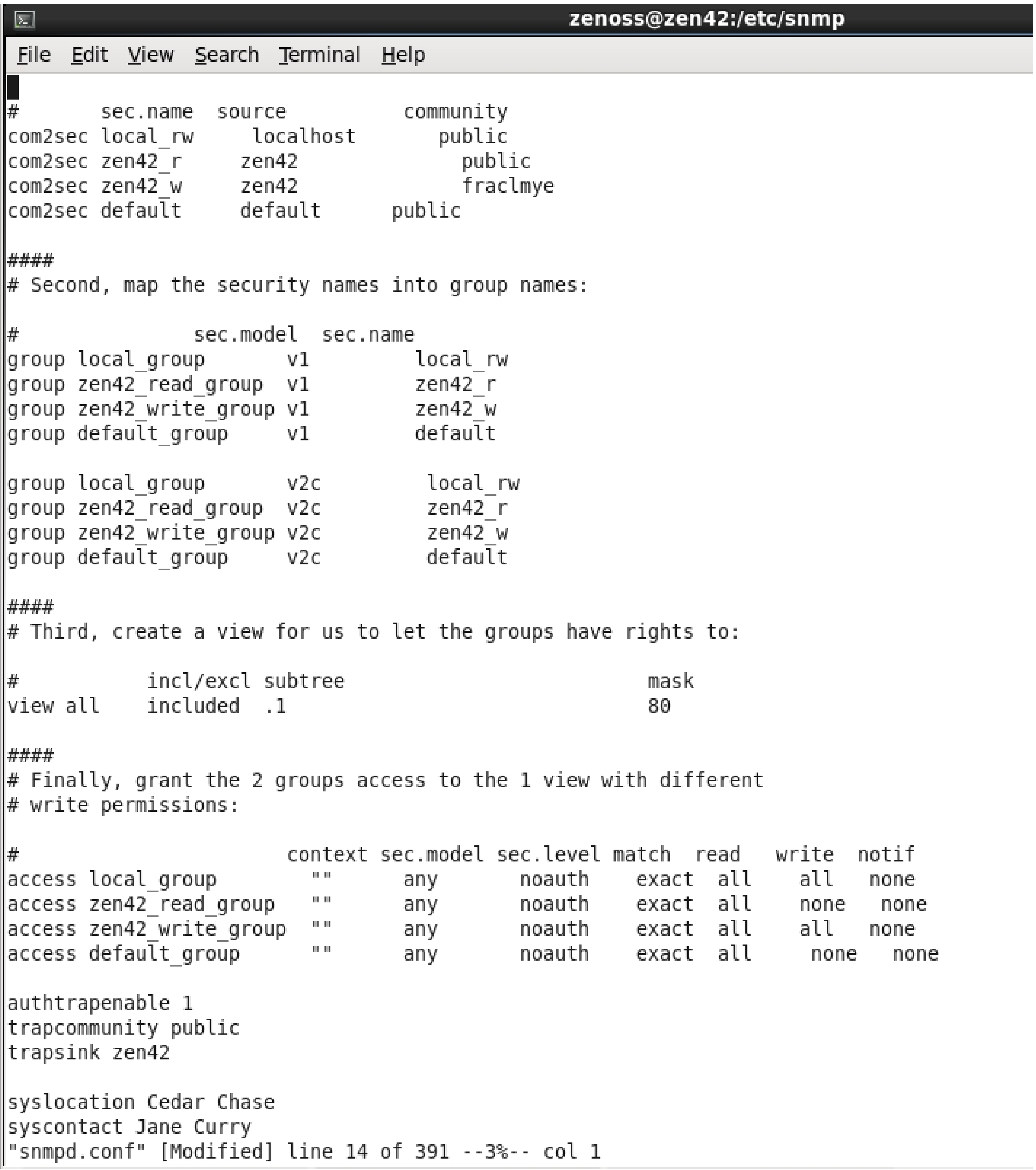
Once SNMP agents are configured with community name and TRAP destination, a simple way to test them is simply to recycle the SNMP agent (indeed they will need recycling after any configuration changes). On a Windows system, use the Services utility to stop and start SNMP; on a Linux system, */etc/init.d/snmpd restart* will usually suffice. In either case you should either see a **cold start** TRAP (generic TRAP 0) or a **warm start** TRAP (generic TRAP 1) in the Zenoss Event Console. The event details should show the community name from the TRAP packet.

Another good way of generating TRAPs is to force an authentication TRAP (generic TRAP 4). An easy way to do this is to use the *snmpwalk* command with a bad community name. If the community is *public*, for a host system called *zenoss*, try:

```
snmpwalk -v 1 -c public zenoss system          test with good community
snmpwalk -v 1 -c fred zenoss system            to generate several TRAP 4's
```

9.2 SNMP on Linux systems

Most Linux systems come with some flavour of the net-snmp agent (formerly the UCD agent). Many Linux default configurations for this agent provide very limited SNMP access. The snmp agent configuration is typically called *snmpd.conf*; the location of this file varies between different Linux implementations but */etc/snmp* is a common choice. You will need *root* authority to manipulate the SNMP configuration and daemon.



```
#       sec.name  source          community
com2sec local_rw     localhost       public
com2sec zen42_r     zen42             public
com2sec zen42_w     zen42             fraclmye
com2sec default     default      public

####
# Second, map the security names into group names:

#               sec.model  sec.name
group local_group       v1         local_rw
group zen42_read_group  v1         zen42_r
group zen42_write_group v1         zen42_w
group default_group     v1         default

group local_group       v2c         local_rw
group zen42_read_group  v2c         zen42_r
group zen42_write_group v2c         zen42_w
group default_group     v2c         default

####
# Third, create a view for us to let the groups have rights to:

#           incl/excl subtree                          mask
view all    included  .1                               80

####
# Finally, grant the 2 groups access to the 1 view with different
# write permissions:

#                       context sec.model sec.level match  read   write  notif
access local_group        ""      any       noauth    exact  all    all   none
access zen42_read_group   ""      any       noauth    exact  all    none   none
access zen42_write_group  ""      any       noauth    exact  all    all   none
access default_group      ""      any       noauth    exact  all     none   none

authtrapenable 1
trapcommunity public
trapsink zen42

syslocation Cedar Chase
syscontact Jane Curry
"snmpd.conf" [Modified] line 14 of 391 --3%-- col 1
```

Figure 53: snmpd.conf for net-snmp agent

Figure 53 shows an snmpd.conf that configures for SNMP V1 and SNMP V2c, providing access to the entire MIB (the *all* view). TRAPs , including Authentication TRAPs, are sent to the *zen42* host. The *sysContact* and *sysLocation* variables are set (these are retrieved as standard by a Zenoss modeler poll).

The snmpd agent should be stopped and restarted after any changes to snmpd.conf.

```
/etc/init.d/snmpd stop
/etc/init.d/snmpd start
```

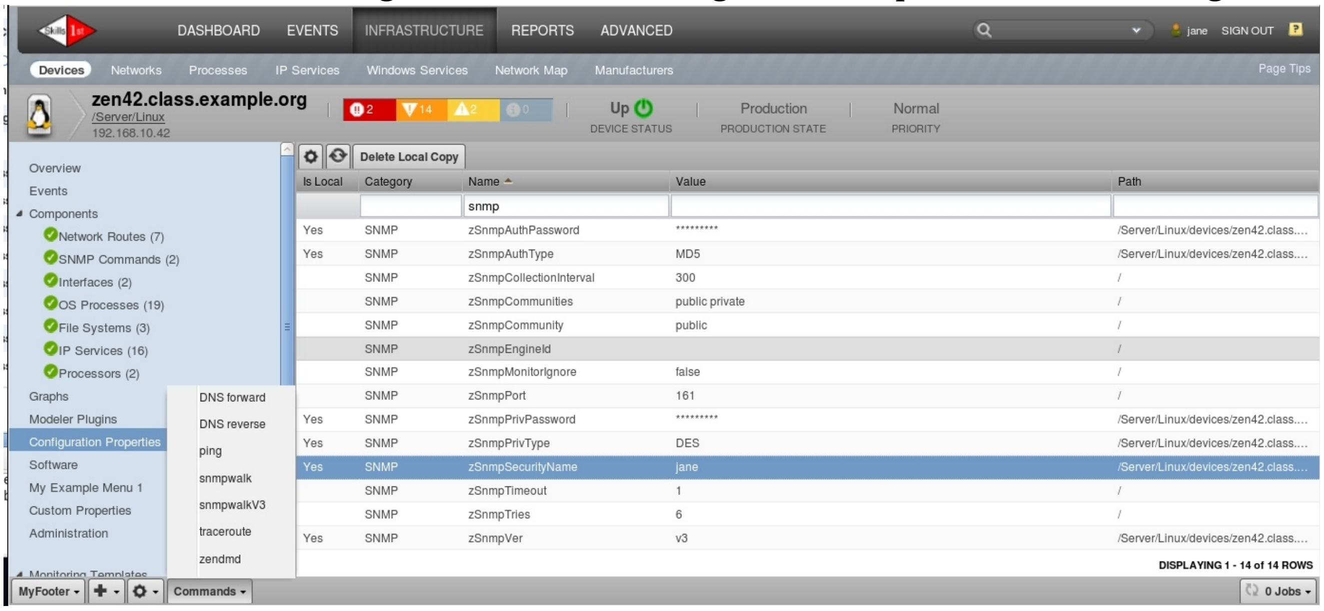
A simple way to test that TRAPs are configured is to generate an Authentication TRAP.

```
snmpwalk -v 1 -c public zen42 system        test with good community
snmpwalk -v 1 -c fred zen42 system          to generate several TRAP 4's
```

Where available, the V3 of the SNMP standard should really be used as it provides strong authentication (not just a community name that passes over the network in clear) and it also provides data encryption if desired. Although slightly harder to set up, it is not too onerous. On the agent, a user id must be generated with parameters for authentication and encryption (privacy), specifying the encryption algorithm and the encryption password to be used.

```
# For SNMP V3
# Uncomment next 5 lines
com2sec snmpv3test localhost    dummycontext
com2sec snmpv3test zen42        dummycontext
group snmpv3group       usm     snmpv3test
#access snmpv3group       ""      usm       auth    exact  all     all   all
access snmpv3group       ""      usm       priv    exact  all     all   all
rwuser jane

#
# rwuser jane created by STOPPING SNMPD and running
# net-snmp-config --create-snmpv3-user -a fraclmyea -x fraclmyex -X DES -A MD5 jane
# /var/lib/net-snmp/snmpd.conf is modified with (hidden) encryption key and
#  rwuser jane is added to this file (/etc/snmp/snmpd.conf)
# test with following if no privacy (data encryption)
#  snmpwalk -v 3 -a MD5 -A fraclmyea -l authNoPriv -u jane zen42 system
# or, with encryption
# snmpwalk -v 3 -a MD5 -A fraclmyea -X fraclmyex -l authPriv -u jane zen42 system
#
# Restart the snmpd daemon
# Note that on CentOS net-snmp-devel must be installed to provide
#    net-snmp-config
```

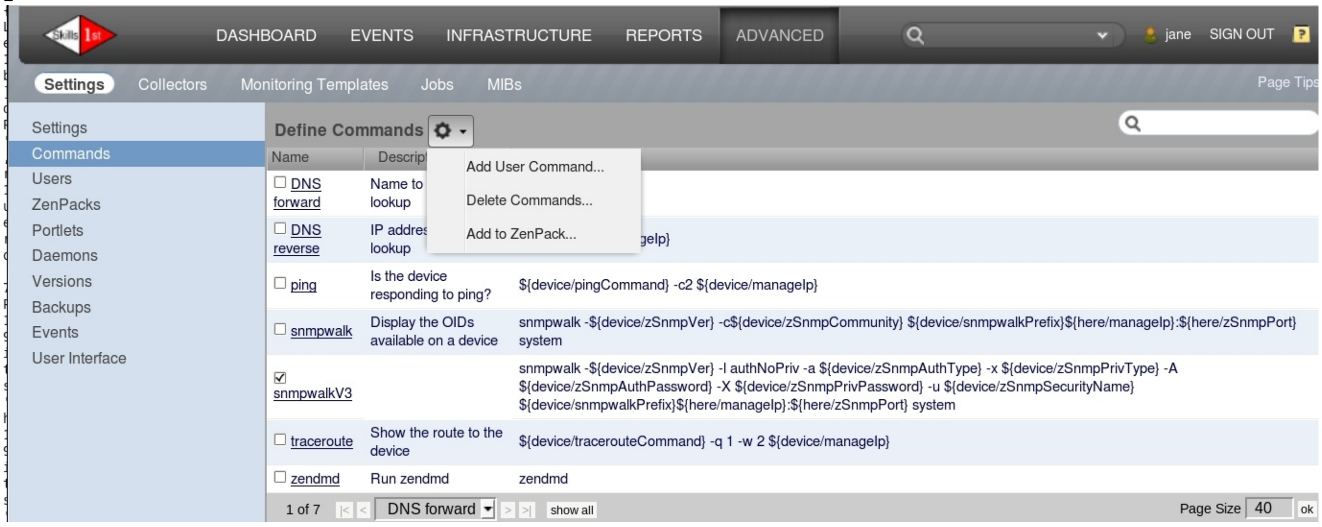
Zenoss must also be configured to have matching SNMP V3 parameters for this agent.



Is Local	Category	Name	Value	Path
		snmp		
Yes	SNMP	zSnmpAuthPassword	*********	/Server/Linux/devices/zen42.class....
Yes	SNMP	zSnmpAuthType	MD5	/Server/Linux/devices/zen42.class....
	SNMP	zSnmpCollectionInterval	300	/
	SNMP	zSnmpCommunities	public private	/
	SNMP	zSnmpCommunity	public	/
	SNMP	zSnmpEngineId		/
	SNMP	zSnmpMonitorIgnore	false	/
	SNMP	zSnmpPort	161	/
Yes	SNMP	zSnmpPrivPassword	*********	/Server/Linux/devices/zen42.class....
Yes	SNMP	zSnmpPrivType	DES	/Server/Linux/devices/zen42.class....
Yes	SNMP	zSnmpSecurityName	jane	/Server/Linux/devices/zen42.class....
	SNMP	zSnmpTimeout	1	/
	SNMP	zSnmpTries	6	/
Yes	SNMP	zSnmpVer	v3	/Server/Linux/devices/zen42.class....

Figure 54: Configuration Properties for agent with SNMP V3

Note that the standard snmpwalk command from the Command icon does not work for SNMP V3 but it is relatively easy to create a new command from *ADVANCED -> Settings -> Commands* which runs an appropriate snmpwalk with the SNMP V3 parameters substituted.



Name	Descript	
☐ DNS forward	Name to lookup	
☐ DNS reverse	IP addres lookup	gelp}
☐ ping	Is the device responding to ping?	${device/pingCommand} -c2 ${device/manageIp}
☐ snmpwalk	Display the OIDs available on a device	snmpwalk -${device/zSnmpVer} -c${device/zSnmpCommunity} ${device/snmpwalkPrefix}${here/manageIp}:${here/zSnmpPort} system
☑ snmpwalkV3		snmpwalk -${device/zSnmpVer} -l authNoPriv -a ${device/zSnmpAuthType} -x ${device/zSnmpPrivType} -A ${device/zSnmpAuthPassword} -X ${device/zSnmpPrivPassword} -u ${device/zSnmpSecurityName} ${device/snmpwalkPrefix}${here/manageIp}:${here/zSnmpPort} system
☐ traceroute	Show the route to the device	${device/tracerouteCommand} -q 1 -w 2 ${device/manageIp}
☐ zendmd	Run zendmd	zendmd

Figure 55: Creating a new Command option to run snmpwalkV3

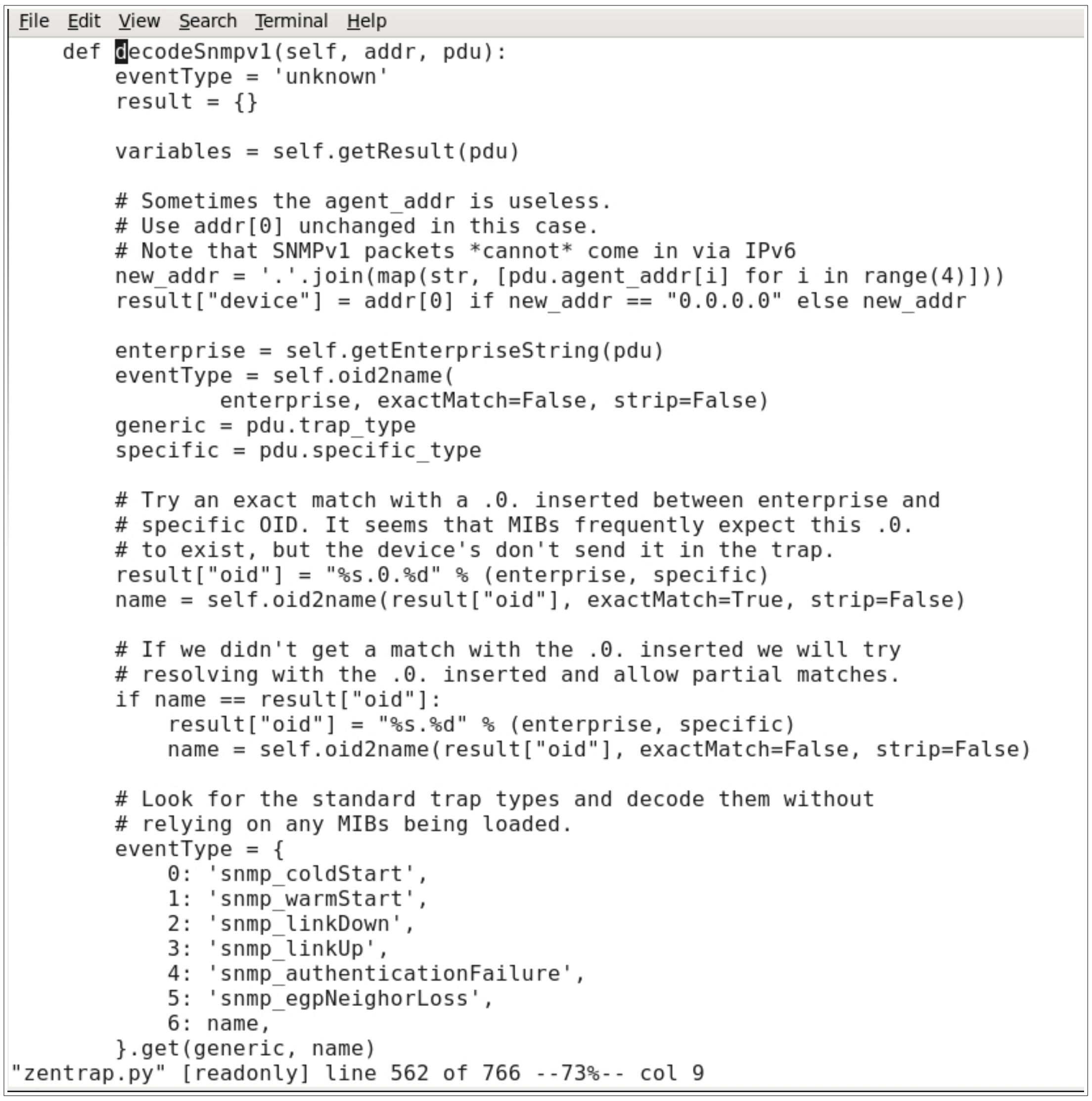
Note that different implementations of net-snmp on different Operating Systems may work slightly differently. For example, Open SuSE does not need the *net-snmp-devel* package and the *rwuser* is created in a separate snmpd.conf under */usr/share/snmp* (which is created automatically if it doesn't exist).

9.3 Zenoss SNMP architecture

9.3.1 The zentrap daemon

zentrap is the Zenoss daemon that processes incoming SNMP TRAPs. By default, zentrap will sit on the well-know SNMP TRAP port of UDP/162 – this can be reconfigured, if required. Both SNMP version 1 TRAPs and SNMP version 2 NOTIFICATIONs are supported.

zentrap processing is implemented by the Python program *$ZENHOME/Products/ZenEvents/zentrap.py*.



```
    def decodeSnmpv1(self, addr, pdu):
        eventType = 'unknown'
        result = {}

        variables = self.getResult(pdu)

        # Sometimes the agent_addr is useless.
        # Use addr[0] unchanged in this case.
        # Note that SNMPv1 packets *cannot* come in via IPv6
        new_addr = '.'.join(map(str, [pdu.agent_addr[i] for i in range(4)]))
        result["device"] = addr[0] if new_addr == "0.0.0.0" else new_addr

        enterprise = self.getEnterpriseString(pdu)
        eventType = self.oid2name(
                enterprise, exactMatch=False, strip=False)
        generic = pdu.trap_type
        specific = pdu.specific_type

        # Try an exact match with a .0. inserted between enterprise and
        # specific OID. It seems that MIBs frequently expect this .0.
        # to exist, but the device's don't send it in the trap.
        result["oid"] = "%s.0.%d" % (enterprise, specific)
        name = self.oid2name(result["oid"], exactMatch=True, strip=False)

        # If we didn't get a match with the .0. inserted we will try
        # resolving with the .0. inserted and allow partial matches.
        if name == result["oid"]:
            result["oid"] = "%s.%d" % (enterprise, specific)
            name = self.oid2name(result["oid"], exactMatch=False, strip=False)

        # Look for the standard trap types and decode them without
        # relying on any MIBs being loaded.
        eventType = {
            0: 'snmp_coldStart',
            1: 'snmp_warmStart',
            2: 'snmp_linkDown',
            3: 'snmp_linkUp',
            4: 'snmp_authenticationFailure',
            5: 'snmp_egpNeighorLoss',
            6: name,
        }.get(generic, name)
"zentrap.py" [readonly] line 562 of 766 --73%-- col 9
```

Figure 56: zentrap.py part 1 - checking for extra 0 and processing of generic TRAPs

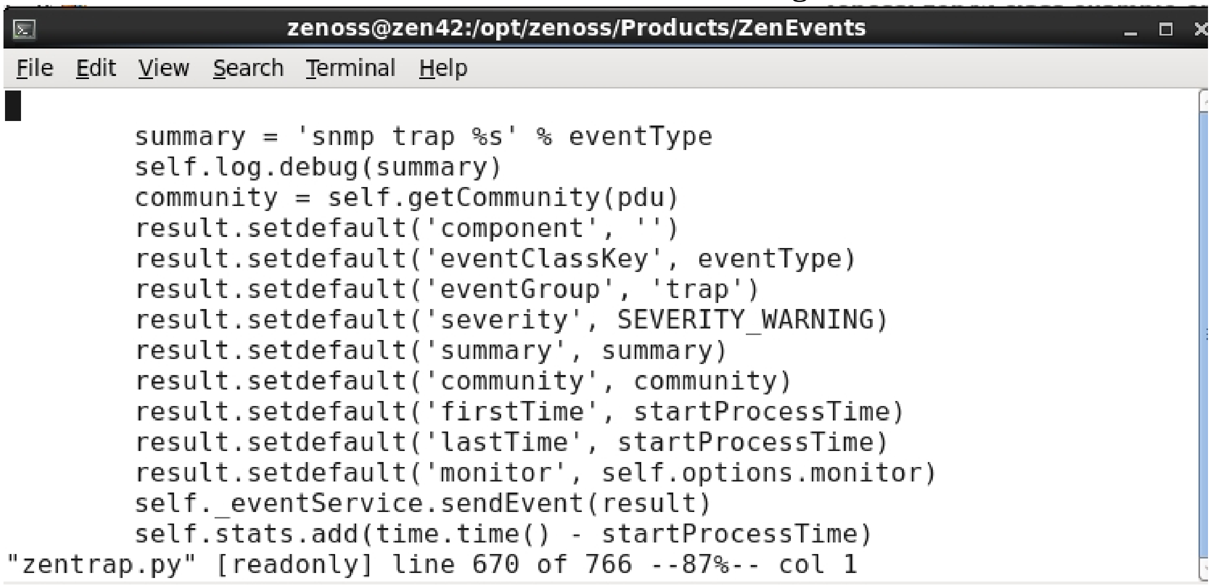
zentrap.py parses the incoming SNMP Protocol Data Unit (PDU) to retrieve the enterprise OID, the generic TRAP number and the specific TRAP number.

The algorithm for interpreting incoming TRAP Enterprise fields has changed several times over the years because some agents have an extra 0 defined in their MIB which they do not send on an actual TRAP (see the comments in the code in Figure 56). In Zenoss 4.2, the algorithm first tries to find a MIB in the ZODB database that corresponds with the incoming TRAP, **with** the extra 0; if this fails, then a partial match is attempted **without** the extra 0 (note that the comment in the code is inaccurate). Either way, the *oid* field of the event is set to the concatenation of the enterprise and the specific trap number, with or without the 0 in the middle, depending on the outcome of the *oid2name* lookup function.

The generic TRAPs (0 through 5) are translated to strings such as *snmp_coldStart*. using the *eventType* dictionary. For specific TRAPs (generic TRAP 6), *eventType* delivers the concatenation of the enterprise OID and the specific TRAP number; for example, 1.3.6.1.4.1.123 is the enterprise, the specific trap number is 1234, so *eventType* delivers

1.3.6.1.4.1.123.1234. Any variables of the TRAP (varbinds) are also parsed out into OID / value pairs if the MIB provides this translation.

The *oid2name* function looks up in the ZODB database to see if translations are available for the enterprise OID, the specific TRAP number and the varbind identifiers, to translate from dotted-decimal notation to textual strings.


zenoss@zen42:/opt/zenoss/Products/ZenEvents
File Edit View Search Terminal Help

```
        summary = 'snmp trap %s' % eventType
        self.log.debug(summary)
        community = self.getCommunity(pdu)
        result.setdefault('component', '')
        result.setdefault('eventClassKey', eventType)
        result.setdefault('eventGroup', 'trap')
        result.setdefault('severity', SEVERITY_WARNING)
        result.setdefault('summary', summary)
        result.setdefault('community', community)
        result.setdefault('firstTime', startProcessTime)
        result.setdefault('lastTime', startProcessTime)
        result.setdefault('monitor', self.options.monitor)
        self._eventService.sendEvent(result)
        self.stats.add(time.time() - startProcessTime)
"zentrap.py" [readonly] line 670 of 766 --87%-- col 1
```

Figure 57: zentrap.py part 2 - event field settings

The following event fields are then set:

● component	left blank
● eventClassKey	set to *eventType*
● eventGroup	*trap*
● severity	3
● summary	*snmp trap* followed by *eventType*
● community	set to community name string (this is a user-defined field)
● firstTime	set to timestamp
● lastTime	set to timestamp
● monitor	set to Collector that received the TRAP

9.4 Interpreting MIBs

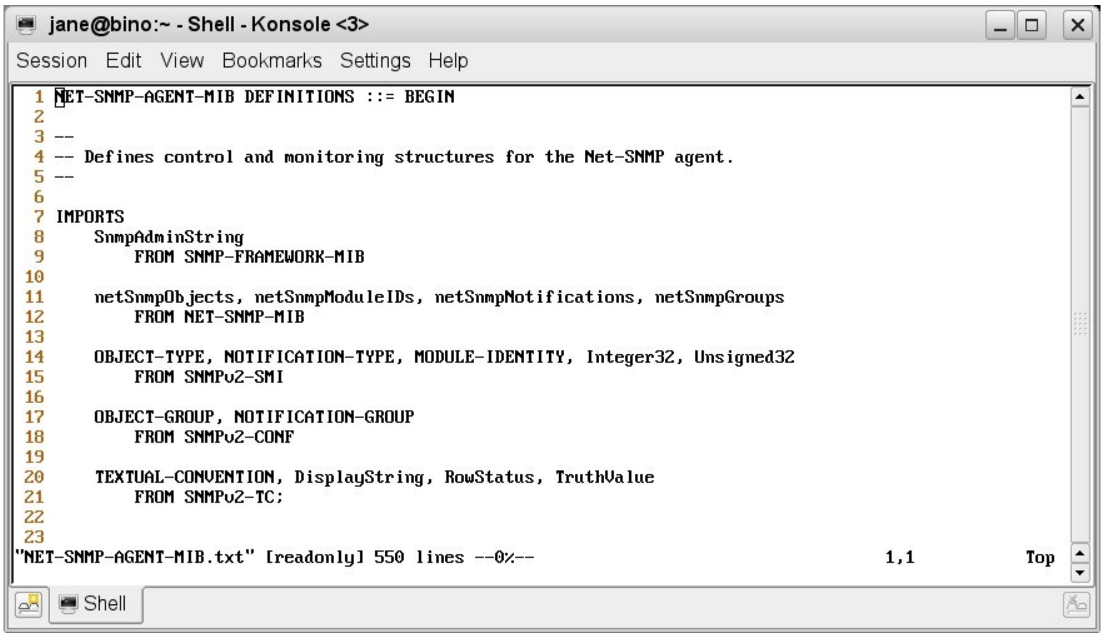
To help decode SNMP TRAP enterprise OIDs from dotted decimal (such as .*1.3.6.1.4.1.8072.4.0.2*) into slightly more meaningful text (like *nsNotifyShutdown)* the **zenmib** command can be used to import both standard MIB source files (such as SNMPv2-SMI which defines standard OIDs) and vendor-specific MIBs. The base directory for MIBs in later versions of Zenoss is *$ZENHOME/share/mibs.*

The zenmib command without parameters will try to import all MIB files that are in *$ZENHOME/share/mibs/site* . A specific MIB file can be provided as a parameter; the command should either be run from the *$ZENHOME/share/mibs/site* directory (in which case a full pathname is not required and the file is expected to be in that directory) or a fully qualified pathname can be specified.

9.4.1 zenmib example

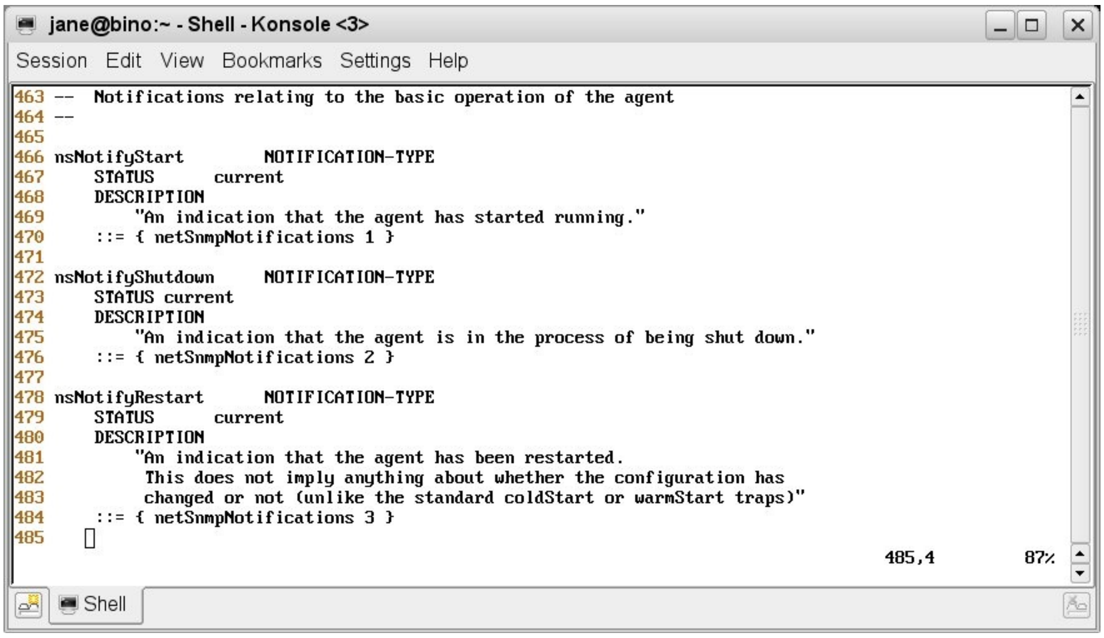
To help understand the zenmib command, here is a worked example. It uses the agent for net-snmp which is the agent typically shipped with a Linux system. The enterprise OID for net-snmp is .1.3.6.1.4.1.8072.

1. Recycle a net-snmp agent with */etc/init.d/snmpd restart* . In addition to the generic cold start TRAP, you should also see TRAP .1.3.6.1.4.1.8072.4.2 . This comes from the net-snmp enterprise (.1.3.6.1.4.1.8072).
2. The actual TRAP is defined in the file *NET-SNMP-AGENT-MIB.txt* which should be shipped as part of the Operating System net-snmp package. Typically this MIB file can be found under */usr/share/snmp/mibs* . Find and examine *NET-SNMP-AGENT-MIB.txt*. Strictly, the MIB file is defining SNMP V2 NOTIFICATIONs , rather than SNMP V1 TRAPs – search in the file for the string *NOTIFI* to find the relevant lines. Also note the *IMPORTS* section at the top of the MIB file, especially the import from NET-SNMP-MIB. This indicates that NET-SNMP-AGENT-MIB is dependent on also loading NET-SNMP-MIB in addition to some standard SNMPv2 MIBs.


jane@bino:~ - Shell - Konsole <3>
Session Edit View Bookmarks Settings Help
Shell

```
NET-SNMP-AGENT-MIB DEFINITIONS ::= BEGIN

--
-- Defines control and monitoring structures for the Net-SNMP agent.
--

IMPORTS
    SnmpAdminString
        FROM SNMP-FRAMEWORK-MIB

    netSnmpObjects, netSnmpModuleIDs, netSnmpNotifications, netSnmpGroups
        FROM NET-SNMP-MIB

    OBJECT-TYPE, NOTIFICATION-TYPE, MODULE-IDENTITY, Integer32, Unsigned32
        FROM SNMPv2-SMI

    OBJECT-GROUP, NOTIFICATION-GROUP
        FROM SNMPv2-CONF

    TEXTUAL-CONVENTION, DisplayString, RowStatus, TruthValue
        FROM SNMPv2-TC;


"NET-SNMP-AGENT-MIB.txt" [readonly] 550 lines --0%--                1,1          Top
```

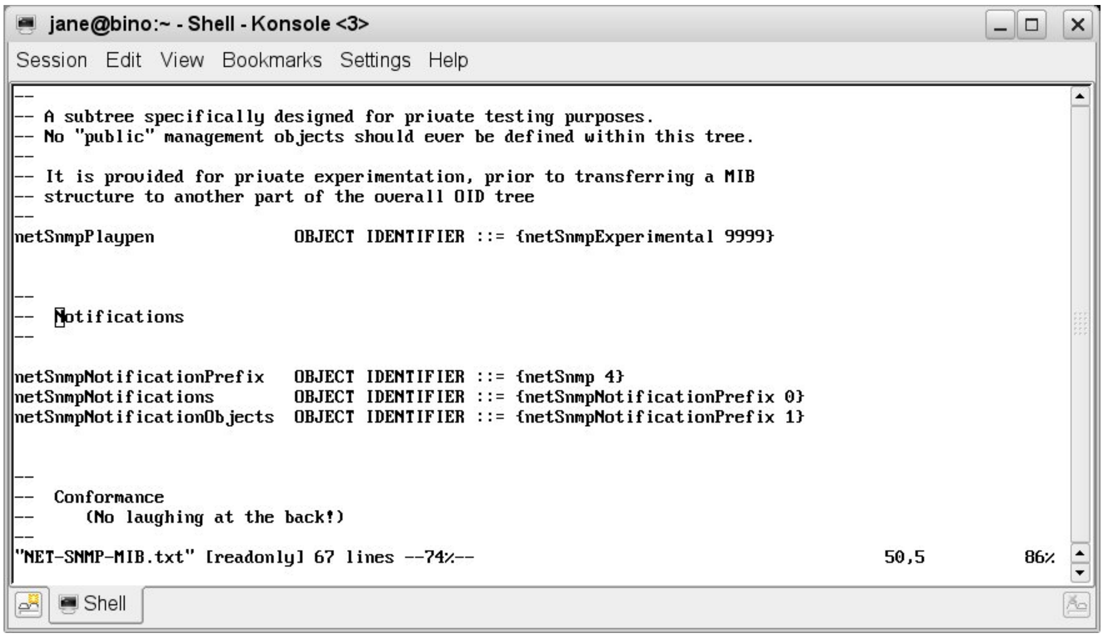
Figure 58: MIB file for NET_SNMP_AGENT-MIB showing IMPORTS section


jane@bino:~ - Shell - Konsole <3>
Session Edit View Bookmarks Settings Help
Shell

```
--  Notifications relating to the basic operation of the agent
--

nsNotifyStart        NOTIFICATION-TYPE
    STATUS      current
    DESCRIPTION
        "An indication that the agent has started running."
    ::= { netSnmpNotifications 1 }

nsNotifyShutdown     NOTIFICATION-TYPE
    STATUS current
    DESCRIPTION
        "An indication that the agent is in the process of being shut down."
    ::= { netSnmpNotifications 2 }

nsNotifyRestart      NOTIFICATION-TYPE
    STATUS      current
    DESCRIPTION
        "An indication that the agent has been restarted.
         This does not imply anything about whether the configuration has
         changed or not (unlike the standard coldStart or warmStart traps)"
    ::= { netSnmpNotifications 3 }

                                                                485,4         87%
```

Figure 59: MIB file for NET-SNMP-AGENT-MIB showing notifications

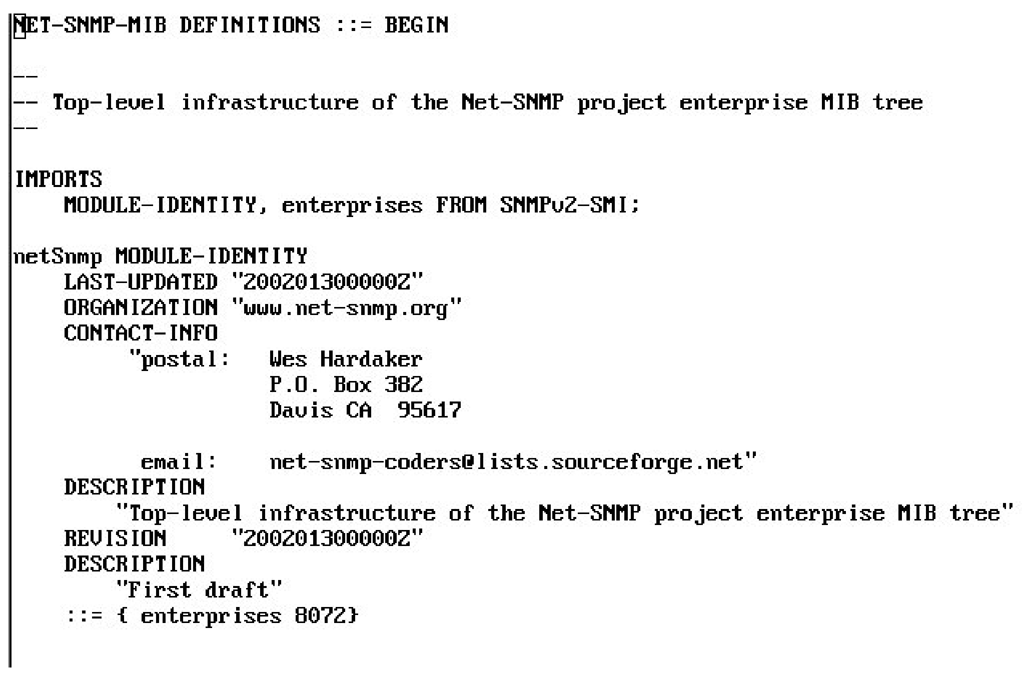
3. Inspect the NET-SNMP-MIB.txt file and search for the string *Notifications*. You should see that the netSnmpNotificationPrefix is defined as branch 4 beneath netSnmp and that netSnmpNotifications is branch 0 under netSnmpNotificationPrefix .


jane@bino:~ - Shell - Konsole <3>
Session Edit View Bookmarks Settings Help
Shell

```
--
-- A subtree specifically designed for private testing purposes.
-- No "public" management objects should ever be defined within this tree.
--
-- It is provided for private experimentation, prior to transferring a MIB
-- structure to another part of the overall OID tree
--
netSnmpPlaypen              OBJECT IDENTIFIER ::= {netSnmpExperimental 9999}


--
--  Notifications
--

netSnmpNotificationPrefix   OBJECT IDENTIFIER ::= {netSnmp 4}
netSnmpNotifications        OBJECT IDENTIFIER ::= {netSnmpNotificationPrefix 0}
netSnmpNotificationObjects  OBJECT IDENTIFIER ::= {netSnmpNotificationPrefix 1}


--
--  Conformance
--     (No laughing at the back!)
--
"NET-SNMP-MIB.txt" [readonly] 67 lines --74%--                 50,5          86%
```

Figure 60: MIB file for NET-SNMP-MIB showing OIDs for notification hierarchy

4. At the top of the file you should find the lines that define the enterprise OID for netSnmp .



```
NET-SNMP-MIB DEFINITIONS ::= BEGIN

--
-- Top-level infrastructure of the Net-SNMP project enterprise MIB tree
--

IMPORTS
    MODULE-IDENTITY, enterprises FROM SNMPv2-SMI;

netSnmp MODULE-IDENTITY
    LAST-UPDATED "200201300000Z"
    ORGANIZATION "www.net-snmp.org"
    CONTACT-INFO
         "postal:   Wes Hardaker
                    P.O. Box 382
                    Davis CA  95617

          email:    net-snmp-coders@lists.sourceforge.net"
    DESCRIPTION
        "Top-level infrastructure of the Net-SNMP project enterprise MIB tree"
    REVISION     "200201300000Z"
    DESCRIPTION
        "First draft"
    ::= { enterprises 8072}
```

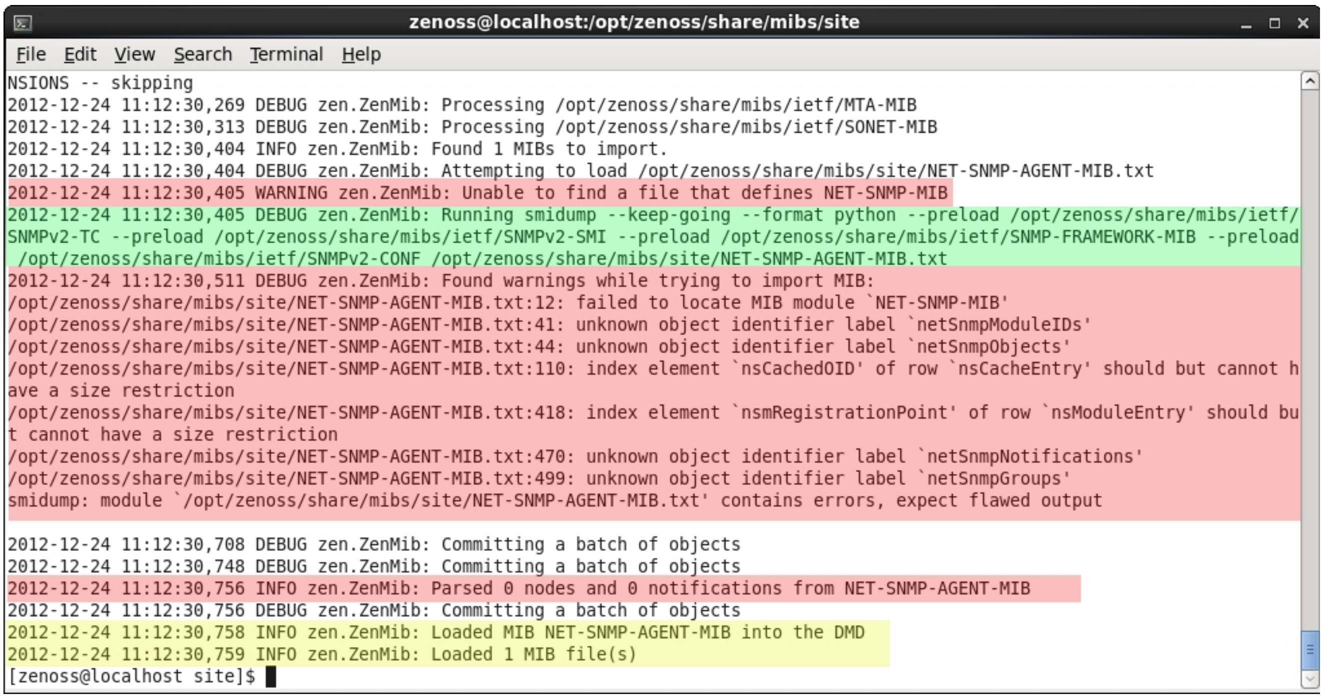
Figure 61: MIB file for NET-SNMP-MIB showing OID for netSnmp

5. Between them, these files give us (almost) the OID for the unknown TRAP we received - 1.3.6.1.4.1.8072.4.0.2 .
    - 1.3.6.1.4.1 is the standard iso.org.dod.internet.private.enterprises OID which is defined in the IMPORT from SNMPv2-SMI
    - netSnmp is {enterprises 8072 }
    - netSnmpNotificationPrefix is branch 4 under netSnmp
    - netSnmpNotifications is branch 0 under netSnmpNotificationPrefix
    - nsNotifyShutdown is NOTIFICATION 2 under netSnmpNotifications
6. Note that some SNMP agents (including the net-snmp agent) are known to omit the 0 from the TRAP that they actually generate, which is why the *oid* field in the details of the event does not quite match the OID specified in the MIB file.
7. *$ZENHOME/share/mibs* contains five subdirectories four of which contain source MIB files provided with Zenoss (*iana, ietf, irtf, tubs*). The fifth directory, *site*, is where other MIBs to be imported, should be placed.
8. The *site* directory should contain ZENOSS-MIB.txt which is provided as standard to define TRAPs that are sent by the Notification function (this will be discussed later).
9. Copy *NET-SNMP-AGENT-MIB.txt* to the *site* directory. At this point do **not** copy *NET-SNMP-MIB.txt*; we will demonstrate the error message when corequisite MIBs are not available.

10. To import into Zenoss use:

```
zenmib run -v10  NET-SNMP-AGENT-MIB.txt
```

11. You should see that the *NET-SNMP-AGENT-MIB.txt* file **is** imported but with errors; there should be a *WARNING* message saying the NET-SNMP-MIB could not be found.

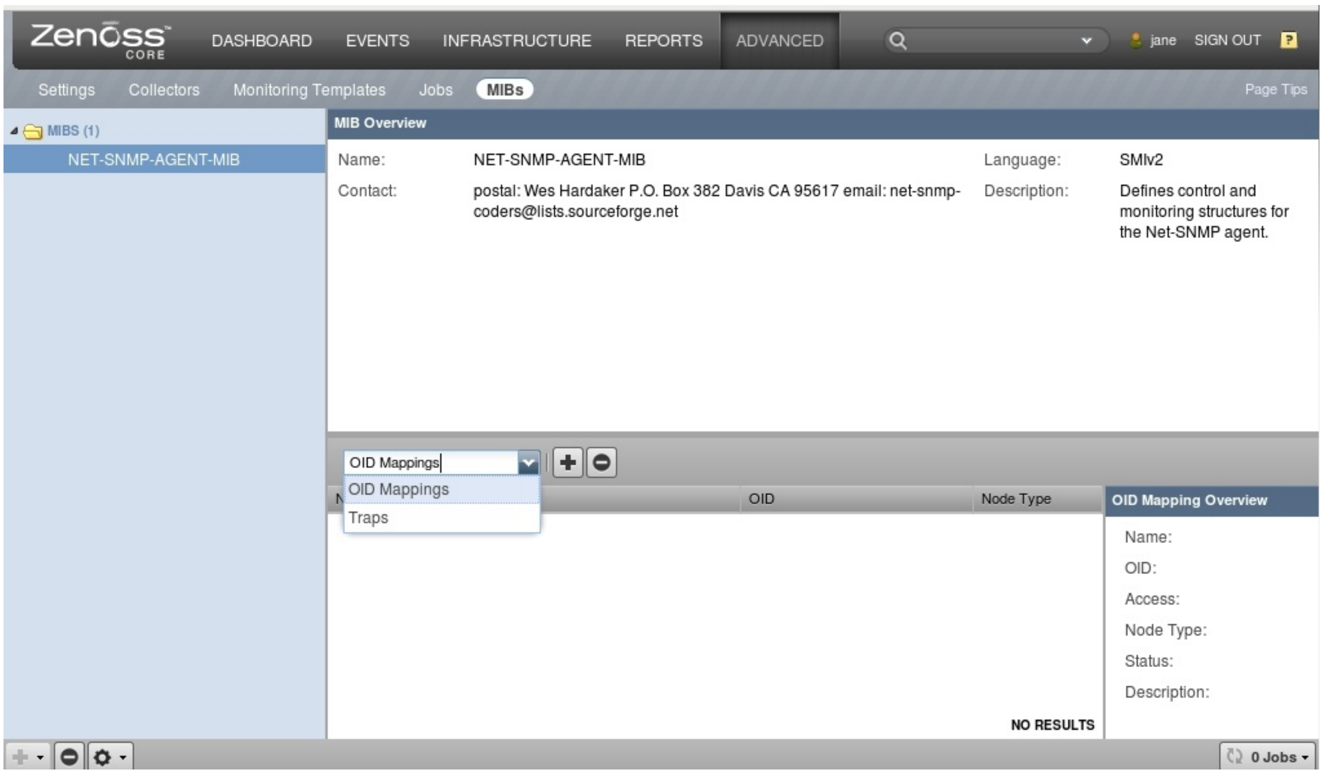


```
NSIONS -- skipping
2012-12-24 11:12:30,269 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/MTA-MIB
2012-12-24 11:12:30,313 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/SONET-MIB
2012-12-24 11:12:30,404 INFO zen.ZenMib: Found 1 MIBs to import.
2012-12-24 11:12:30,404 DEBUG zen.ZenMib: Attempting to load /opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt
2012-12-24 11:12:30,405 WARNING zen.ZenMib: Unable to find a file that defines NET-SNMP-MIB
2012-12-24 11:12:30,405 DEBUG zen.ZenMib: Running smidump --keep-going --format python --preload /opt/zenoss/share/mibs/ietf/
SNMPv2-TC --preload /opt/zenoss/share/mibs/ietf/SNMPv2-SMI --preload /opt/zenoss/share/mibs/ietf/SNMP-FRAMEWORK-MIB --preload
 /opt/zenoss/share/mibs/ietf/SNMPv2-CONF /opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt
2012-12-24 11:12:30,511 DEBUG zen.ZenMib: Found warnings while trying to import MIB:
/opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt:12: failed to locate MIB module `NET-SNMP-MIB'
/opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt:41: unknown object identifier label `netSnmpModuleIDs'
/opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt:44: unknown object identifier label `netSnmpObjects'
/opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt:110: index element `nsCachedOID' of row `nsCacheEntry' should but cannot h
ave a size restriction
/opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt:418: index element `nsmRegistrationPoint' of row `nsModuleEntry' should bu
t cannot have a size restriction
/opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt:470: unknown object identifier label `netSnmpNotifications'
/opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt:499: unknown object identifier label `netSnmpGroups'
smidump: module `/opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt' contains errors, expect flawed output

2012-12-24 11:12:30,708 DEBUG zen.ZenMib: Committing a batch of objects
2012-12-24 11:12:30,748 DEBUG zen.ZenMib: Committing a batch of objects
2012-12-24 11:12:30,756 INFO zen.ZenMib: Parsed 0 nodes and 0 notifications from NET-SNMP-AGENT-MIB
2012-12-24 11:12:30,756 DEBUG zen.ZenMib: Committing a batch of objects
2012-12-24 11:12:30,758 INFO zen.ZenMib: Loaded MIB NET-SNMP-AGENT-MIB into the DMD
2012-12-24 11:12:30,759 INFO zen.ZenMib: Loaded 1 MIB file(s)
[zenoss@localhost site]$
```

Figure 62: Importing NET-SNMP-AGENT before pre-requisites in place

12. Note in the *Running smidump* line that the standard SNMPv2 prerequisite files that were listed as IMPORTs in Figure 58 **have** automatically been located in *$ZENHOME/share/mibs/ietf*; however ultimately *0 nodes and 0 notifications* were loaded.

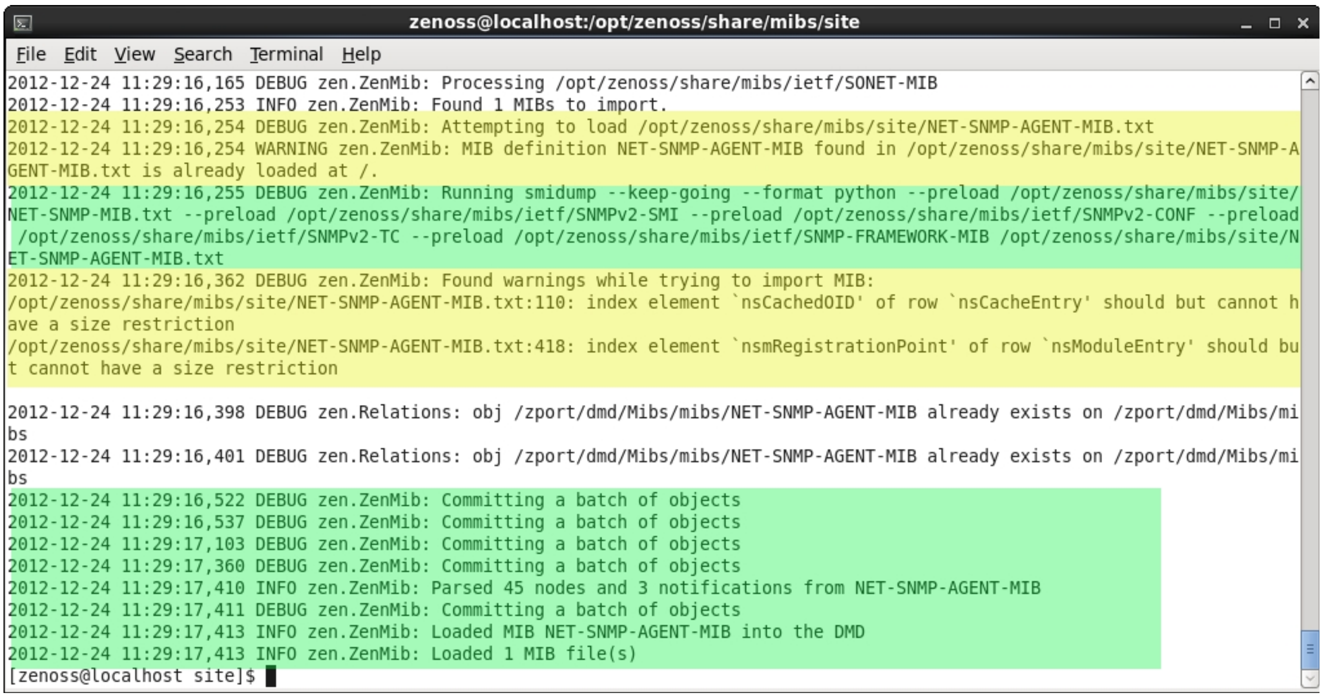
13. From the Zenoss GUI, use the *ADVANCED -> MIBs* menu. The NET-SNMP-AGENT-MIB **is** listed but, as suggested, it has no OID Mappings and no TRAPs.



Name: NET-SNMP-AGENT-MIB
Language: SMIv2
Contact: postal: Wes Hardaker P.O. Box 382 Davis CA 95617 email: net-snmp-coders@lists.sourceforge.net
Description: Defines control and monitoring structures for the Net-SNMP agent.

Figure 63: MIB GUI with imported NET-SNMP-AGENT-MIB but no OIDs or TRAPs

14. Copy *NET-SNMP-MIB.txt* to *$ZENHOME/share/mibs/site* and rerun the *zenmib* command.



zenoss@localhost:/opt/zenoss/share/mibs/site
File Edit View Search Terminal Help

```
2012-12-24 11:29:16,165 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/SONET-MIB
2012-12-24 11:29:16,253 INFO zen.ZenMib: Found 1 MIBs to import.
2012-12-24 11:29:16,254 DEBUG zen.ZenMib: Attempting to load /opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt
2012-12-24 11:29:16,254 WARNING zen.ZenMib: MIB definition NET-SNMP-AGENT-MIB found in /opt/zenoss/share/mibs/site/NET-SNMP-A
GENT-MIB.txt is already loaded at /.
2012-12-24 11:29:16,255 DEBUG zen.ZenMib: Running smidump --keep-going --format python --preload /opt/zenoss/share/mibs/site/
NET-SNMP-MIB.txt --preload /opt/zenoss/share/mibs/ietf/SNMPv2-SMI --preload /opt/zenoss/share/mibs/ietf/SNMPv2-CONF --preload
 /opt/zenoss/share/mibs/ietf/SNMPv2-TC --preload /opt/zenoss/share/mibs/ietf/SNMP-FRAMEWORK-MIB /opt/zenoss/share/mibs/site/N
ET-SNMP-AGENT-MIB.txt
2012-12-24 11:29:16,362 DEBUG zen.ZenMib: Found warnings while trying to import MIB:
/opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt:110: index element `nsCachedOID' of row `nsCacheEntry' should but cannot h
ave a size restriction
/opt/zenoss/share/mibs/site/NET-SNMP-AGENT-MIB.txt:418: index element `nsmRegistrationPoint' of row `nsModuleEntry' should bu
t cannot have a size restriction

2012-12-24 11:29:16,398 DEBUG zen.Relations: obj /zport/dmd/Mibs/mibs/NET-SNMP-AGENT-MIB already exists on /zport/dmd/Mibs/mi
bs
2012-12-24 11:29:16,401 DEBUG zen.Relations: obj /zport/dmd/Mibs/mibs/NET-SNMP-AGENT-MIB already exists on /zport/dmd/Mibs/mi
bs
2012-12-24 11:29:16,522 DEBUG zen.ZenMib: Committing a batch of objects
2012-12-24 11:29:16,537 DEBUG zen.ZenMib: Committing a batch of objects
2012-12-24 11:29:17,103 DEBUG zen.ZenMib: Committing a batch of objects
2012-12-24 11:29:17,360 DEBUG zen.ZenMib: Committing a batch of objects
2012-12-24 11:29:17,410 INFO zen.ZenMib: Parsed 45 nodes and 3 notifications from NET-SNMP-AGENT-MIB
2012-12-24 11:29:17,411 DEBUG zen.ZenMib: Committing a batch of objects
2012-12-24 11:29:17,413 INFO zen.ZenMib: Loaded MIB NET-SNMP-AGENT-MIB into the DMD
2012-12-24 11:29:17,413 INFO zen.ZenMib: Loaded 1 MIB file(s)
[zenoss@localhost site]$
```

Figure 64: Successful import of NET-SNMP-AGENT given correct pre-requisites

15. There is a DEBUG line noting that the NET-SNMP-AGENT-MIB is already imported; this is not an issue. This import will overwrite any existing MIB of that name.

16. Note that the *Running smidump* line also looks in the *site* directory and finds the pre-requisite NET-SNMP-MIB.txt in addition to finding the standard SNMPv2 MIBs in the *ietf* directory. 45 nodes and 3 notifications have been loaded.

17. Return to the Zenoss GUI and refresh the *MIBs* menu. Clicking on the NET-SNMP-AGENT-MIB should now display 45 OID Mappings and three TRAPs, including *nsNotifyShutdown*.

18. Restart the snmp agent on the Zenoss system with */etc/init.d/snmpd restart*. You **should** see an event in the Event Console that now contains *snmp trap nsNotifyShutdown* in the summary field, rather than *snmp trap 1.3.6.1.4.1.8072.4.2* . If this does **not** work, you may need to recycle the zentrap daemon. You can do this with the GUI from the *ADVANCED -> Settings -> Daemons* menu or, as the *zenoss* user from a command line, use *zentrap restart*.

19. Zenoss has implemented a number of changes in the way MIBs are interpreted. Remember from Figure 60 that netSnmpNotifications is branch 0 under netSnmpNotificationPrefix; however, some agents omit this 0 when they actually generate TRAPs. Zenoss 4.2 has processing in *$ZENHOME/Products/ZenEvent/zentrap.py* to try and interpret actual TRAPs both with and without the extra 0. The event console showed an event with OID 1.3.6.1.4.1.8072.4.2 for the original event; compare the event details of the original event with the new one that contains *nsNotifyShutdown* in the summary field. You should find that the new event has an *oid* field of 1.3.6.1.4.1.8072.4.**0**.2.

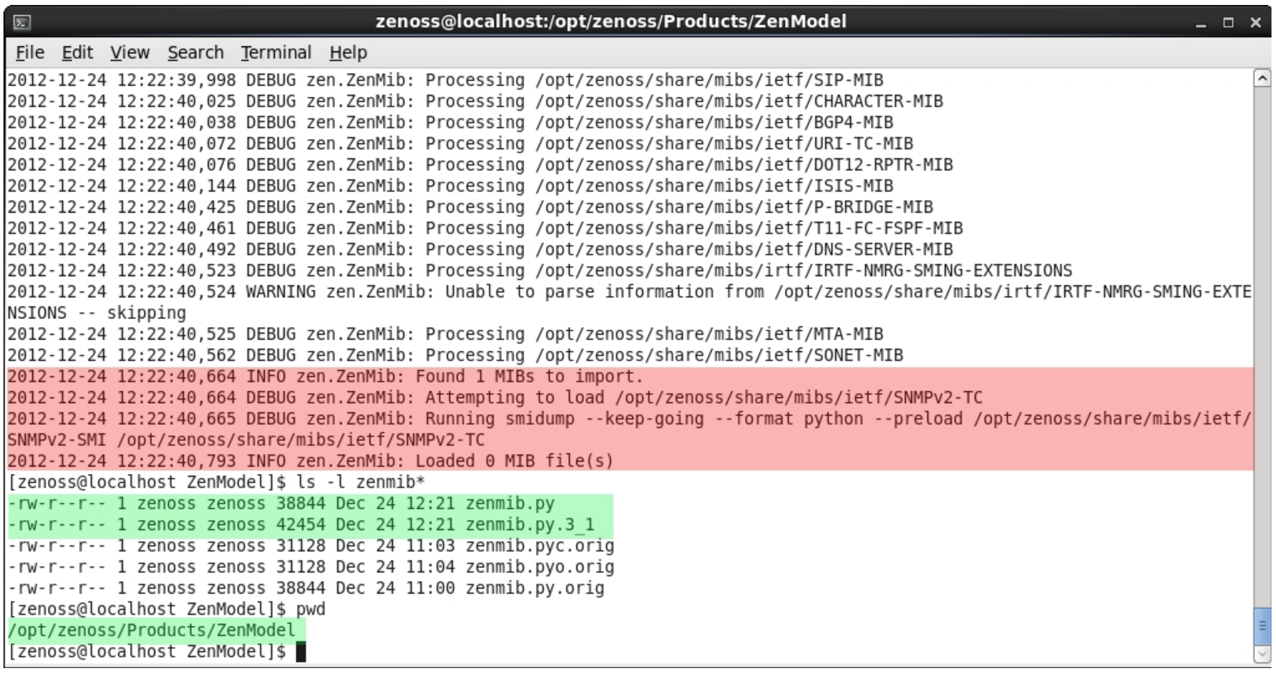
20. Examine *$ZENHOME/Products/ZenEvent/zentrap.py* (around line 580 in Zenoss Core 4.2) to see the code that handles this extra 0 digit processing.

9.4.2 A few comments on importing MIBs with Zenoss

There are a few quirks to do with importing MIBs into Zenoss and the quirks have changed subtly over several versions of Zenoss.

- **Note** that MIBs imported into Zenoss are **only** used for interpreting SNMP V1 TRAPs and SNMP V2 NOTIFICATIONs for use in the Event subsystem. Although the OIDs are imported from MIBs, they cannot be used for MIB browsing or when working with OIDs for performance sampling, thresholding and graphing.
- **Always** ensure you do MIB work as the *zenoss* user .
- By default. *zenmib run -v10* will try and import **everything** under *$ZENHOME/share/mibs/site*. The *-v10* simply adds more verbose output. *zenmib* should check in the other directories for prerequisites.
- Whenever you have imported a MIB, check at the GUI on the *MIBs* page. You should see the name of the MIB **and** you should usually see non-zero counts under the *OID Mappings* and *TRAP* dropdown menus.

- There are some MIBs that **will** result in zero counts, for example if the MIB source file only defines SNMP structure and does not include the definition for any OIDs or TRAPs .
- Check the output of the zenmib command carefully for error messages.
- If OID translations do not appear to be working in events after importing a MIB, recycle the *zentrap* daemon from the *ADVANCED -> Settings -> Daemons* menu or, as the *zenoss* user, run *zentrap restart*.
- If event mappings and transforms are built assuming that a MIB **has** been imported, for example, testing the eventClassKey field for *enterprises.8072.4.2*, and that MIB is then removed from the Zope database, then the mapping and/or transform will **fail**. Especial care should be taken with any ZenPack that imports MIBs as the removal of the ZenPack is likely to remove those MIBs.
- Zenoss 4.2 (and 3.2.1) appear to have a timing bug that affects some installations. The symptom is that zenmib apparently satisfies its checks but then reports *Loaded 0 MIB file(s)*. The only solution I have found (which appears to work perfectly) is to use a *zenmib.py* from a Zenoss 3.1 installation. This file belongs in *$ZENHOME/Products/ZenModel*.


zenoss@localhost:/opt/zenoss/Products/ZenModel
File Edit View Search Terminal Help

```
2012-12-24 12:22:39,998 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/SIP-MIB
2012-12-24 12:22:40,025 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/CHARACTER-MIB
2012-12-24 12:22:40,038 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/BGP4-MIB
2012-12-24 12:22:40,072 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/URI-TC-MIB
2012-12-24 12:22:40,076 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/DOT12-RPTR-MIB
2012-12-24 12:22:40,144 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/ISIS-MIB
2012-12-24 12:22:40,425 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/P-BRIDGE-MIB
2012-12-24 12:22:40,461 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/T11-FC-FSPF-MIB
2012-12-24 12:22:40,492 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/DNS-SERVER-MIB
2012-12-24 12:22:40,523 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/irtf/IRTF-NMRG-SMING-EXTENSIONS
2012-12-24 12:22:40,524 WARNING zen.ZenMib: Unable to parse information from /opt/zenoss/share/mibs/irtf/IRTF-NMRG-SMING-EXTE
NSIONS -- skipping
2012-12-24 12:22:40,525 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/MTA-MIB
2012-12-24 12:22:40,562 DEBUG zen.ZenMib: Processing /opt/zenoss/share/mibs/ietf/SONET-MIB
2012-12-24 12:22:40,664 INFO zen.ZenMib: Found 1 MIBs to import.
2012-12-24 12:22:40,664 DEBUG zen.ZenMib: Attempting to load /opt/zenoss/share/mibs/ietf/SNMPv2-TC
2012-12-24 12:22:40,665 DEBUG zen.ZenMib: Running smidump --keep-going --format python --preload /opt/zenoss/share/mibs/ietf/
SNMPv2-SMI /opt/zenoss/share/mibs/ietf/SNMPv2-TC
2012-12-24 12:22:40,793 INFO zen.ZenMib: Loaded 0 MIB file(s)
[zenoss@localhost ZenModel]$ ls -l zenmib*
-rw-r--r-- 1 zenoss zenoss 38844 Dec 24 12:21 zenmib.py
-rw-r--r-- 1 zenoss zenoss 42454 Dec 24 12:21 zenmib.py.3_1
-rw-r--r-- 1 zenoss zenoss 31128 Dec 24 11:03 zenmib.pyc.orig
-rw-r--r-- 1 zenoss zenoss 31128 Dec 24 11:04 zenmib.pyo.orig
-rw-r--r-- 1 zenoss zenoss 38844 Dec 24 11:00 zenmib.py.orig
[zenoss@localhost ZenModel]$ pwd
/opt/zenoss/Products/ZenModel
[zenoss@localhost ZenModel]$ 
```

Figure 65: Occasional timing bug with Zenoss 4.2. Replace zenmib.py with a Zenoss 3.1 version.

9.5 The MIB Browser ZenPack

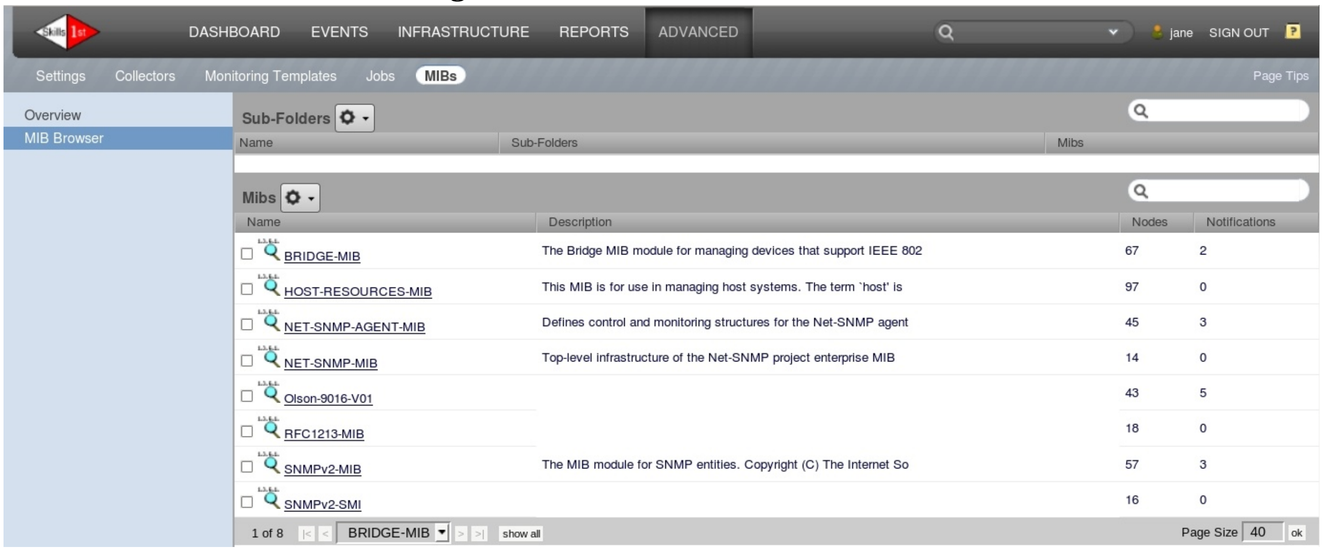
There is an excellent community ZenPack available to perform MIB Browsing. This is not directly relevant to TRAP / NOTIFICATION processing, but it is useful for investigating MIBs with a view to building SNMP performance templates.

It can be downloaded from http://wiki.zenoss.org/ZenPack:MIB_Browser . Unfortunately this ZenPack keeps getting broken by new versions of Zenoss. If you follow the link to

Download for Zenoss Core 3.1, this does indeed work for Core 3.1; this version should be downloaded and modified for Core 3.2; for Zenoss 4.2, follow the *Download for Zenoss Core 4.2* link and perform the same modifications which are documented in the comments if you follow the documentation link http://community.zenoss.org/docs/DOC-10321 . Basically you revert the later Core files back to the 3.1 level of code.

It provides a MIB browser to explore any OID that has been loaded into Zenoss, along with a test facility to *snmpwalk* a configurable device to retrieve values for any selected part of the MIB tree. Note that it only supports SNMP V1.

The MIB Browser ZenPack changes the *ADVANCED -> MIBs* menu and creates a *MIB Browser* lefthand menu. Selecting the *MIB Browser* menu offers a similar layout to the *Overview* menu but it introduces new icons alongside the name of a MIB. Clicking the icon starts the MIB Browser against the selected MIB.



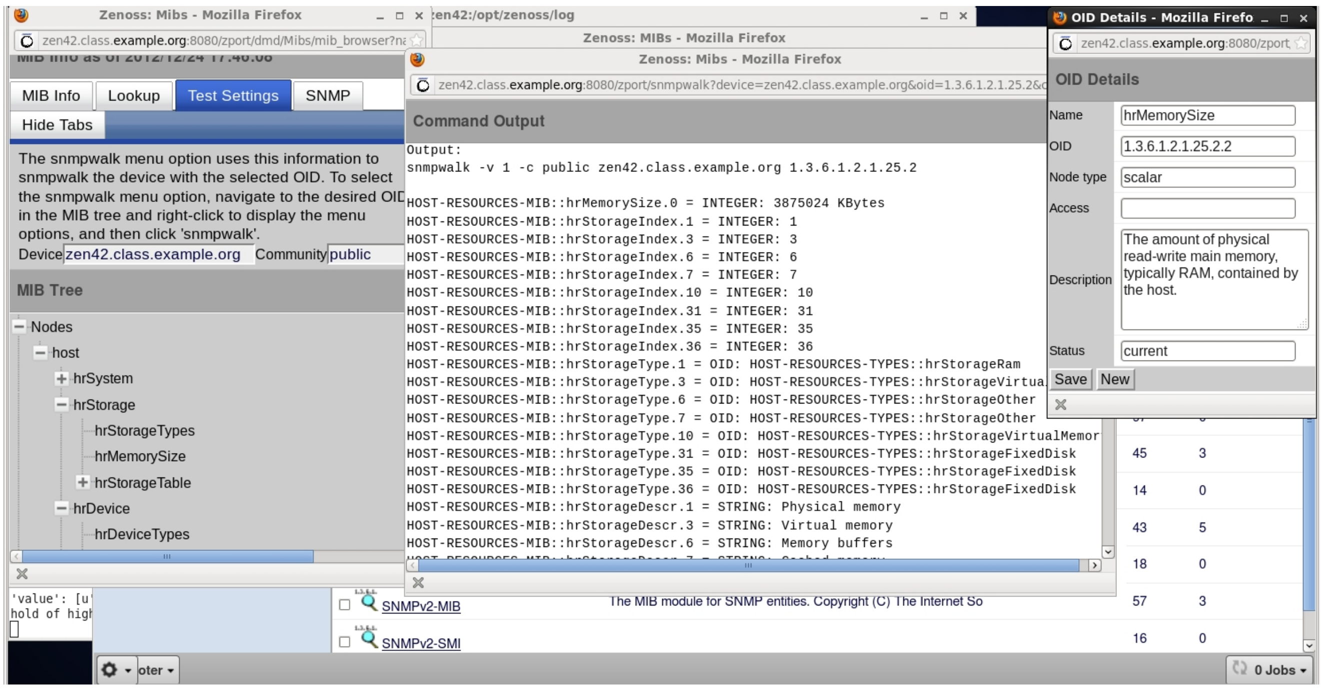
Name	Sub-Folders	Mibs

Name	Description	Nodes	Notifications
BRIDGE-MIB	The Bridge MIB module for managing devices that support IEEE 802	67	2
HOST-RESOURCES-MIB	This MIB is for use in managing host systems. The term `host' is	97	0
NET-SNMP-AGENT-MIB	Defines control and monitoring structures for the Net-SNMP agent	45	3
NET-SNMP-MIB	Top-level infrastructure of the Net-SNMP project enterprise MIB	14	0
Olson-9016-V01		43	5
RFC1213-MIB		18	0
SNMPv2-MIB	The MIB module for SNMP entities. Copyright (C) The Internet So	57	3
SNMPv2-SMI		16	0

Figure 66: Starting the MIB Browser - click against the magnifier icon for a given MIB

In order to perform an snmpwalk, you need to provide a target device and an SNMP v1 community name under the *Test Settings* tab. A **right-hand** mouse click then provides the snmpwalk menu against the level of the MIB tree that you are positioned on.

The *OID Details* window gives the same information you would see if you inspected the MIB source file. Use this window to cut-and-paste into OID fields in performance templates.


Zenoss: Mibs - Mozilla Firefox
zen42.class.example.org:8080/zport/dmd/Mibs/mib_browser?na
MIB Info
Lookup
Test Settings
SNMP
Hide Tabs
MIB Tree
Nodes
host
hrSystem
hrStorage
hrStorageTypes
hrMemorySize
hrStorageTable
hrDevice
hrDeviceTypes
Zenoss: MIBs - Mozilla Firefox
Zenoss: Mibs - Mozilla Firefox
zen42.class.example.org:8080/zport/snmpwalk?device=zen42.class.example.org&oid=1.3.6.1.2.1.25.2&c
Command Output
OID Details - Mozilla Firefox
OID Details
Save
New
SNMPv2-MIB
SNMPv2-SMI
The MIB module for SNMP entities. Copyright (C) The Internet So
0 Jobs

The snmpwalk menu option uses this information to snmpwalk the device with the selected OID. To select the snmpwalk menu option, navigate to the desired OID in the MIB tree and right-click to display the menu options, and then click 'snmpwalk'.

Device zen42.class.example.org Community public

```
Output:
snmpwalk -v 1 -c public zen42.class.example.org 1.3.6.1.2.1.25.2

HOST-RESOURCES-MIB::hrMemorySize.0 = INTEGER: 3875024 KBytes
HOST-RESOURCES-MIB::hrStorageIndex.1 = INTEGER: 1
HOST-RESOURCES-MIB::hrStorageIndex.3 = INTEGER: 3
HOST-RESOURCES-MIB::hrStorageIndex.6 = INTEGER: 6
HOST-RESOURCES-MIB::hrStorageIndex.7 = INTEGER: 7
HOST-RESOURCES-MIB::hrStorageIndex.10 = INTEGER: 10
HOST-RESOURCES-MIB::hrStorageIndex.31 = INTEGER: 31
HOST-RESOURCES-MIB::hrStorageIndex.35 = INTEGER: 35
HOST-RESOURCES-MIB::hrStorageIndex.36 = INTEGER: 36
HOST-RESOURCES-MIB::hrStorageType.1 = OID: HOST-RESOURCES-TYPES::hrStorageRam
HOST-RESOURCES-MIB::hrStorageType.3 = OID: HOST-RESOURCES-TYPES::hrStorageVirtua
HOST-RESOURCES-MIB::hrStorageType.6 = OID: HOST-RESOURCES-TYPES::hrStorageOther
HOST-RESOURCES-MIB::hrStorageType.7 = OID: HOST-RESOURCES-TYPES::hrStorageOther
HOST-RESOURCES-MIB::hrStorageType.10 = OID: HOST-RESOURCES-TYPES::hrStorageVirtualMemor
HOST-RESOURCES-MIB::hrStorageType.31 = OID: HOST-RESOURCES-TYPES::hrStorageFixedDisk
HOST-RESOURCES-MIB::hrStorageType.35 = OID: HOST-RESOURCES-TYPES::hrStorageFixedDisk
HOST-RESOURCES-MIB::hrStorageType.36 = OID: HOST-RESOURCES-TYPES::hrStorageFixedDisk
HOST-RESOURCES-MIB::hrStorageDescr.1 = STRING: Physical memory
HOST-RESOURCES-MIB::hrStorageDescr.3 = STRING: Virtual memory
HOST-RESOURCES-MIB::hrStorageDescr.6 = STRING: Memory buffers
```

Name	hrMemorySize
OID	1.3.6.1.2.1.25.2.2
Node type	scalar
Access	
Description	The amount of physical read-write main memory, typically RAM, contained by the host.
Status	current

Figure 67: Using the MIB Browser ZenPack

9.5.1 Modifying Zenoss Core 4.2 to make the MIB Browser ZenPack work

1. Download the egg file and install in the normal way. It should install with no errors.

```
zenpack --install ZenPacks.community.mib_browser-1.2-py2.7.egg
zenhub restart
zopectl restart
```

2. Change to *$ZENHOME/Products/ZenUI3/browser*. Backup *backcompat.py, navigation.zcml* and *backcompat.zcml.*

3. In *backcompat.py,* comment out the lines at the end defining *MibClass*. If there are also similar lines for *MibNode* and *MibNotification*, comment them out too.

```
#def MibClass(ob):
#    id = '/'.join(ob.getPhysicalPath())
#    return '/zport/dmd/mibs#mibtree:' + id
```

4. In *navigation.zcml*, around line 233, change the url line to be url="/zport/dmd/Mibs/mibOrganizerOverview". Note carefully the case sensitivity on mibs / Mibs.

```
-  url="/zport/dmd/mibs"
+  url="/zport/dmd/Mibs/mibOrganizerOverview"
```

5. In *backcompat.zcml*, around line 260 comment out lines for the adapter for *Products.ZenModel.MibOrganizer.MibOrganizer*. If adapter stanzas also exist for *MibNode, MibNotification* and *MibModule*, comment them out too.

6. Change directory to *$ZENHOME/Products/ZenModel/skins/zenmodel* and backup *viewMibModule.pt*.

7. Modify *viewMibModule.py*. Change the template in the first line .

```
-   <tal:block metal:use-macro="here/templates/macros/page2">
+   <tal:block metal:use-macro="here/page_macros/old-new">
```

8. You will need to completely restart Zenoss and **make sure your browser cache is cleared**.

9.6 Mapping SNMP events

Zenoss provides some event mappings for SNMP TRAPs out-of-the-box. As discussed in an earlier section, the file *$ZENHOME/Products/ZenModel/data/events.xml* configures all the standard mappings so searching this file for SNMP provides insight for default customisation.

Most SNMP TRAPs map to the Zenoss **/Unknown** event class. There are one-or-two exceptions for some generic TRAPs such as Link Up (3), Link Down (2) and the Authentication TRAP (4). Event fields that are automatically populated by the **zentrap** processing include *summary, eventClassKey* and *agent*. The event details shows the **community** and **oid** Name / Value pairs. Note that the value of the oid field is always in numeric format, not translated through an imported MIB.

This means that, typically, the event only maps on the Event Class Key, which is interpreted by zentrap.py as *enterprises.<enterprise number>.<specific trap>* if the SNMPv2-SMI has been imported or *1.3.6.1.4.1.<enterprise number>.<specific trap>* otherwise. The *summary* field will be *snmp trap <enterprise OID><specific trap>* and the *agent* field will be set to *zentrap* . These translations assume that the enterprise-specific MIB has **not** been imported.

TRAPs and NOTIFICATIONs may have one or more TRAP variables (varbinds). These varbinds appear in the event details where the field **name** is the varbind OID (possibly translated through a MIB lookup) and the corresponding field **value** is the value of that varbind.

Event class mappings can be devised with various Rule, Regex and Transform elements, to parse out the intelligence from SNMP TRAPs and either create new user-defined event fields or modify existing fields (such as *evt.summary*).

Note that event mappings that parse out SNMP OIDs and varbinds must be aware of whether the relevant MIBs have been imported, or not. If a MIB is imported, OID mapping based on matching dotted-decimal notation will fail as the MIB OID translations happen **before** event mapping.

9.6.1 SNMP event mapping example

In order to interpret enterprise specific TRAPs, mappings are usually required. Often an action or modification is required, effectively based on what enterprise the TRAP came from (Cisco, net-snmp, ...), so a subclass of events are required that inherit some common characteristics but some event details vary depending on the exact enterprise specific TRAP number.

Many enterprise TRAPs also include several varbinds that need to be interpreted and processed.

In the mapping example shown here, three small scripts are used to generate TRAPs from the 1.3.6.1.4.1.123 enterprise – one for each of specific TRAPs 1234, 1235 and 1236. The first two have a single varbind whose string-type value is "Hello world 4", where the end number is 4 or 5; the third script generates a TRAP with 2 varbinds. Note that each of the varbinds exhibit the "extra 0" behaviour, ie. the varbind field will be 1.3.6.1.4.1.123.0.1234.

```
#!/bin/bash
#
# Generate a sample trap
# Send trap using the snmptrap supplied with net-snmp
# Trap here is Enterprise 1.3.6.1.4.1.123, trap 1236
# Ensure you change the line for MANAGER to be your Zenoss Server
#
# Uncomment next line for extra debugging
#set -x
MANAGER=zen42.class.example.org
HOST=zen42.class.example.org
ENTERPRISE=.1.3.6.1.4.1.123
GENTRAP=6
SPECTRAP=1236
TRAPVAR1=.1.3.6.1.4.1.123.0.12361
TRAPVAR2=.1.3.6.1.4.1.123.0.12362
VARBIND1="Hello world varbind1 61"
VARBIND2="Hello world varbind2 62"
TIMESTAMP=1
#
/usr/bin/snmptrap -v 1 -c public $MANAGER $ENTERPRISE $HOST $GENTRAP
$SPECTRAP $TIMESTAMP \
  $TRAPVAR1 s "$VARBIND1" \
  $TRAPVAR2 s "$VARBIND2"
#
```

1. Without any mapping, when *gen_mytrap_1234.sh* is run, it will map to the */Unknown* event class.
2. Create a new event subclass *Snmp* under the class */Skills* .
3. Map the "1234" event by selecting it and using the *Reclassify an Event* icon. Choose */Skills/Snmp* from the dropdown selection box. Leave the rest of the Event Class Mapping parameters as defaults for now. This means that the event only maps on the eventClassKey, which translates to *<enterprise OID>.<specific trap>* . The mapping name is automatically assigned the name of the eventClassKey (*1.3.6.1.4.1.123.1234* if SNMPv2-SMI is **not** imported; *enterprises.123.1234* if it is). Refer back to the snippet of the zentrap code in Figure 57 for more information on the parsing of the TRAP into event fields. Check that your event class mapping works.

From here, ensure that the SNMPv2-SMI MIB is imported; thus any TRAP enterprise field (and hence eventClassKey) will start with **enterprises**, not 1.3.6.1.4.1. In most cases, the same will apply to the name field of a TRAP varbind.

The next step is to interpret the varbind. Each of the TRAPs generated by the test scripts come from the Enterprise 1.3.6.1.4.1.123 and the name of each of the varbinds also starts with 1.3.6.1.4.1.123 thus, in the detail of the interpreted event, the varbind name fields will start with *enterprises*. A transform will extract that part of the OID **after** *enterprises* . It will also substitute the value of the varbind into the event summary.

At transform time, strictly the event is a ZepRawEventProxy object, which has a details dictionary (an EventDetailProxy object) as part of it (refer back to Figure 35, Figure 37 and Figure 38). Also remember that although one can refer to detail event fields by name (eg. evt.line_num) if they are simple names, you **cannot** use this method if the detail name has a dot in it.

If one is interested in the values of such fields, the *get* or *getAll* methods are needed. Since the *get* method fails with an attribute error if the value is non-scalar, it is safer to assume that **all** values may be non-scalar and use the *getAll* method.

In versions of Zenoss prior to 4, a transform to interpret TRAP varbinds would look like this:

```
for attr in dir(evt):
    if attr.startswith('enterprises.123.'):
         evt.myRestOfOID=attr.replace('enterprises.123.','')
         evt.myFieldValue=getattr(evt,attr)
         evt.summary=(evt.summary + “  “ + evt.myFieldValue)
```

This will fail with Zenoss 4 as the new event structure does not deliver detail event fields as a result of *dir(evt)*. A Zenoss 4 version would be:

```
for attr in evt.details._map.keys():
    if attr.startswith('enterprises.123'):
      evt.myRestOfOID=attr.replace('enterprises.123.','')
      evt.myFieldValue=' '.join(list(evt.details.getAll(attr)))
      evt.summary = (evt.summary + “  ” + evt.myFieldValue)
```

1. The first line cycles through the event details attribute names.
2. The “startswith” line ensures that transforms only take place for attributes that start with enterprises.123 – ie. varbind attribute fields.
3. Note that the “replace” line is replacing the OID specified, with the null string – the syntax after the comma is single-quote single-quote . The rest of the attribute (ie. the 0.1234 bit) is kept and becomes the value of the user-field myRestOfOID .
4. The *evt.myFieldValue* line uses the *getAll* method in case the varbind value is non-scalar. To concatenate the resulting list with the *evt.summary* string, the list is converted into a string with the *join* function.

5. Running the script to generate a “1234” TRAP should now generate an event with:
   - The event mapped to the */Skills/Snmp* class
   - The summary field should say *“snmp trap enterprises.123.1234 Hello world 4”*.
   - The *Event Details* should show values for *community*, *oid*, *myFieldValue* and *myRestOfOID*, in addition to the default varbind name/value pair of enterprises.*123.0.1234 / Hello world 4*
6. Running the script to generate a “1235” TRAP will still generate an event with the */Unknown* class as the event class mapping is based on the eventClassKey of enterprises.*123.1234* .

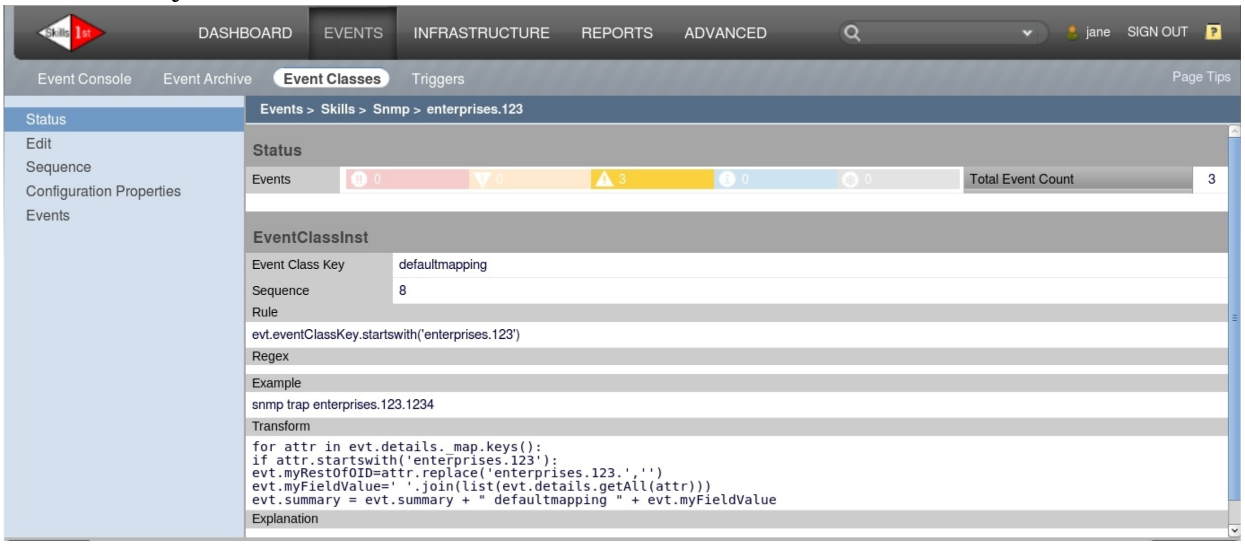
So far, we are only matching a single SNMP TRAP with the eventClassKey field. The objective is to map all events from the enterprise 1.3.6.1.4.1.123 . With SNMP, you often want to apply a transform to several similar events which are only distinguished by the later parts of the OID field. The test scripts all generate events whose eventClassKey start with *1.3.6.1.4.1.123.* but they differ in the last number.

A **Rule** will be used to match all appropriate events. However, a Rule is only inspected if the eventClassKey has already matched successfully and we have no control over the eventClassKey – that is set by *zentrap.py* . Thus, the **defaultmapping** concept will be used.

1. Clear all SNMP events for your Zenoss system.
2. Edit the *enterprises.123.1234* mapping.
   - In the *Rule* box put *evt.eventClassKey.startswith('enterprises.123.')*
   - Change the *Name* of the mapping to *enterprises.123*
   - In the *Transform* box put:

```
for attr in evt.details._map.keys():
    if attr.startswith('enterprises.123'):
      evt.myRestOfOID=attr.replace('enterprises.123.','')
      evt.myFieldValue=' '.join(list(evt.details.getAll(attr)))
      evt.summary = evt.summary + " defaultmapping " + evt.myFieldValue
```
   - Save the mapping away
3. Run the *gen_mytrap_1234.sh* script and the *gen_mytrap_1235.sh* script.
4. Check the events in the Event Console
5. You should find that the 1234 TRAP maps successfully but the 1235 TRAP doesn't. This is because the initial test for event class mapping checks the eventClassKey – that is still set to *enterprises.123.1234* so the processing never even gets as far as checking our Rule! **Note** that we have no control over how the eventClassKey field is populated by the event processing mechanism – it is parsed out for us by *zentrap.py* (see Figure 57 again).

6. This is where the "magic string" of *defaultmapping* can be used in the Event Class Key field. Set the Event Class Key to *defaultmapping* (**Note** it **must** be all lower case). If the process of mapping an event cannot find a match for the Event Class Key then it will re-run the mapping process with an Event Class Key of *defaultmapping*.
7. Save the mapping.
8. Check the *Sequence* menu. There are several mappings that all map on an Event Class Key of *defaultmapping*. Choose a suitable sequence number for the new *defaultmapping*. Save the mapping.
9. Clear existing events. Rerun both scripts. Check that both events now map correctly.



Events > Skills > Snmp > enterprises.123

Status

Events	0	0	3	0	0	Total Event Count	3

EventClassInst

Event Class Key	defaultmapping
Sequence	8

Rule

evt.eventClassKey.startswith('enterprises.123')

Regex

Example

snmp trap enterprises.123.1234

Transform

```
for attr in evt.details._map.keys():
if attr.startswith('enterprises.123'):
evt.myRestOfOID=attr.replace('enterprises.123.','')
evt.myFieldValue=' '.join(list(evt.details.getAll(attr)))
evt.summary = evt.summary + " defaultmapping " + evt.myFieldValue
```

Explanation

Figure 68: Mapping for SNMP TRAP with rule, transform and eventClassKey of defaultmapping

The test events used so far, only have one varbind. What if your TRAP has several varbinds and you want to use information from each of them? The script *gen_mytrap_1236.sh* generates a specific TRAP 1236, with two varbinds:

● varbind 1	1.3.6.1.4.1.123.0.12361	Hello world varbind1 61"
● varbind 2	1.3.6.1.4.1.123.0.12362	Hello world varbind1 62"

Running the script *gen_mytrap_1236.sh* should result in an event that maps to the */Skills/Snmp* class, with the *myFieldValue* and *myRestOfOID* fields matching the data in the **last** varbind that was processed, and the summary reflecting the data from **all** varbinds.

To provide a more elegant transform solution where you do not know if a detail value is scalar or not, the Python *try / except* construct could be used:

```
for attr in evt.details._map.keys():
    if attr.startswith('enterprises.123'):
      evt.myRestOfOID=attr.replace('enterprises.123.','')
      try:
        evt.myFieldValue=evt.details.get(attr)
      except:
        evt.myFieldValue=' '.join(list(evt.details.getAll(attr)))
      evt.summary = evt.summary + " defaultmapping " + evt.myFieldValue
```

Check the end of *$ZENHOME/log/zeneventd.log* for debugging help.

10 Event Triggers and Notifications

10.1 Zenoss prior to V4

Prior to Zenoss 4, there were two ways of automating responses to events.

- User **Alerting Rules**
  - Email to users
  - Paging to users
- **Event Commands**
  - Scripts run in the background

The user actions were configured on a per-user or per-user-group basis. This meant that similar emails / pages for many users or groups had to be created individually; there was no easy way to copy an Alerting Rule from one user to another.

Event Commands used a very similar method to define when a command should be automatically run in the background.

Alerting Rules and Event Commands were executed by the zenactiond daemon which processed any requests every 60 seconds. Duplicate events did **not** create multiple actions and this was handled by the **alert_state** table of the MySQL events database.

This is probably the area that has changed most for users of Zenoss 4.

10.2 Zenoss 4 architecture

Zenoss 4 has completely changed the architecture of the MySQL events database. There is no alert_state table in the zenoss_zep database. zenactiond is still responsible for executing actions but it has been completely rewritten and takes input from a RabbitMQ queue called **signal** which is fed by the zeneventserver daemon. This makes alerting much more responsive.

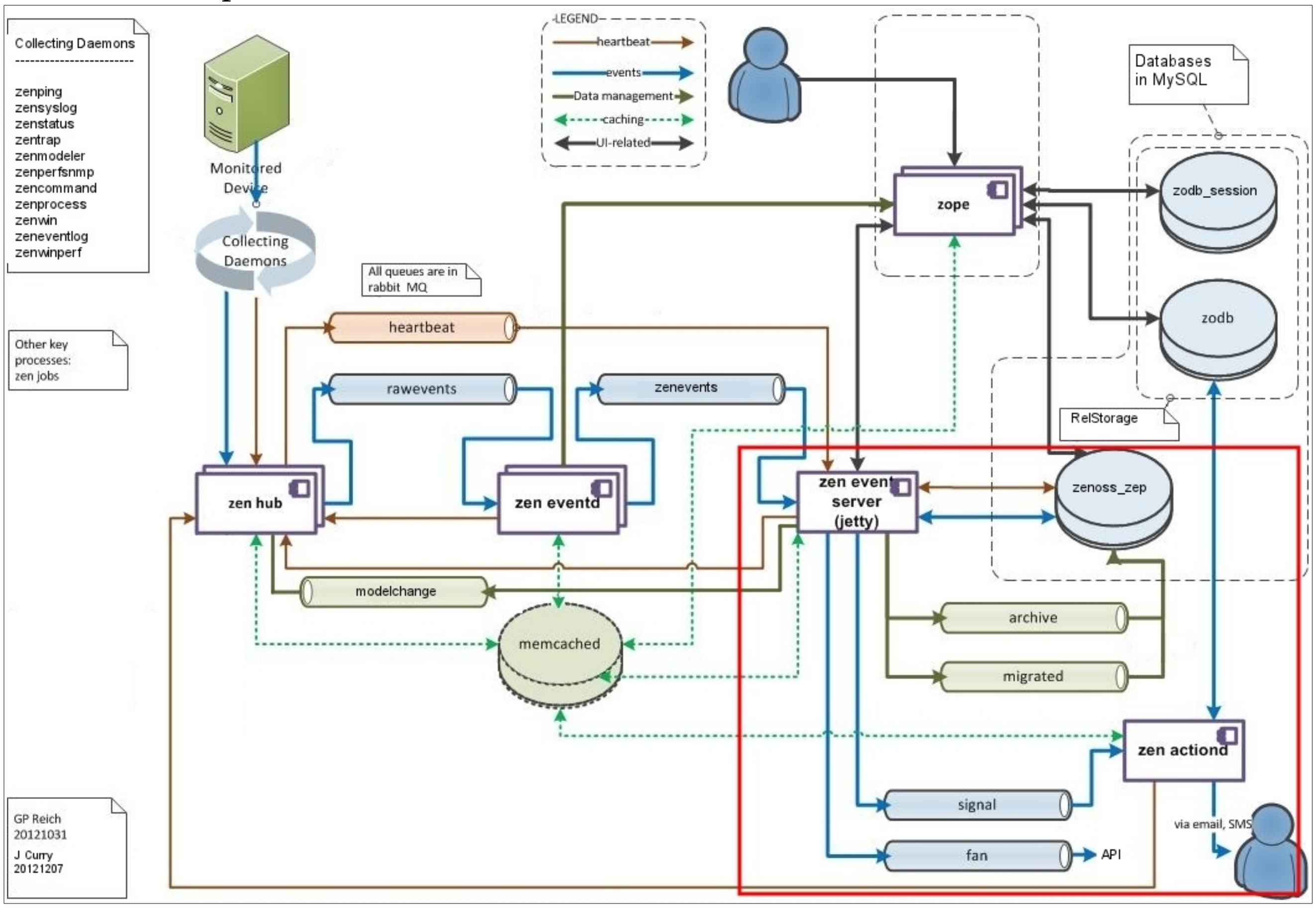


Figure 69: Zenoss event architecture - action processing in bottom-right

Alerting Rules have gone in Zenoss 4 and are replaced by the concepts of:

- Triggers
- Notifications

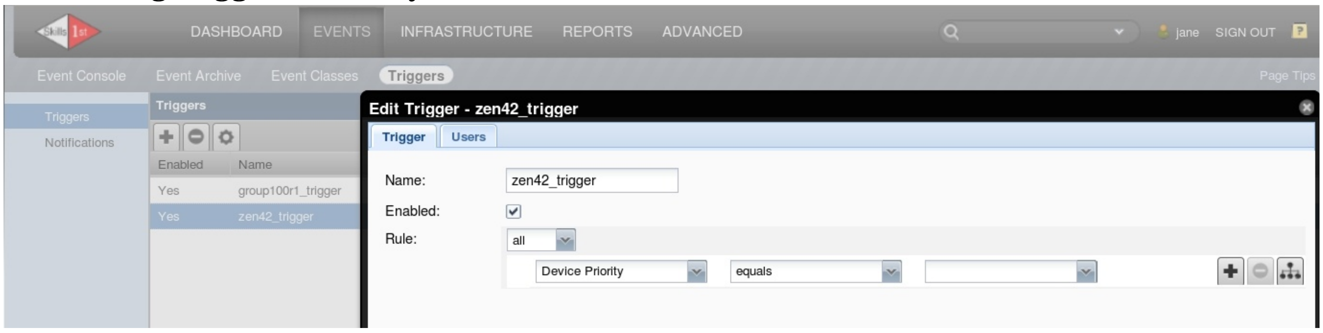
Triggers define what causes a response. A **Notification** is the response. This is better in several ways. Both mechanisms are decoupled from users and from each other. Notifications now include event commands as well as the traditional email and paging, and SNMP TRAPs have also been added as a notification action.

Trigger and NotificationSubscriptions objects are defined in the Zope database (though the Trigger is a stub object that is used for managing permissions and does not contain the actual trigger rules).

There is a new *EVENTS -> Triggers* menu for defining both Triggers and Notifications.

10.3 Triggers

Triggers define under what conditions some action should take place. They are defined from the *EVENTS -> Triggers* menu. Use the + icon to add a new trigger; double-click an existing trigger to modify it.



Enabled	Name
Yes	group100r1_trigger
Yes	zen42_trigger

Name: zen42_trigger

Enabled: ✓

Rule: all

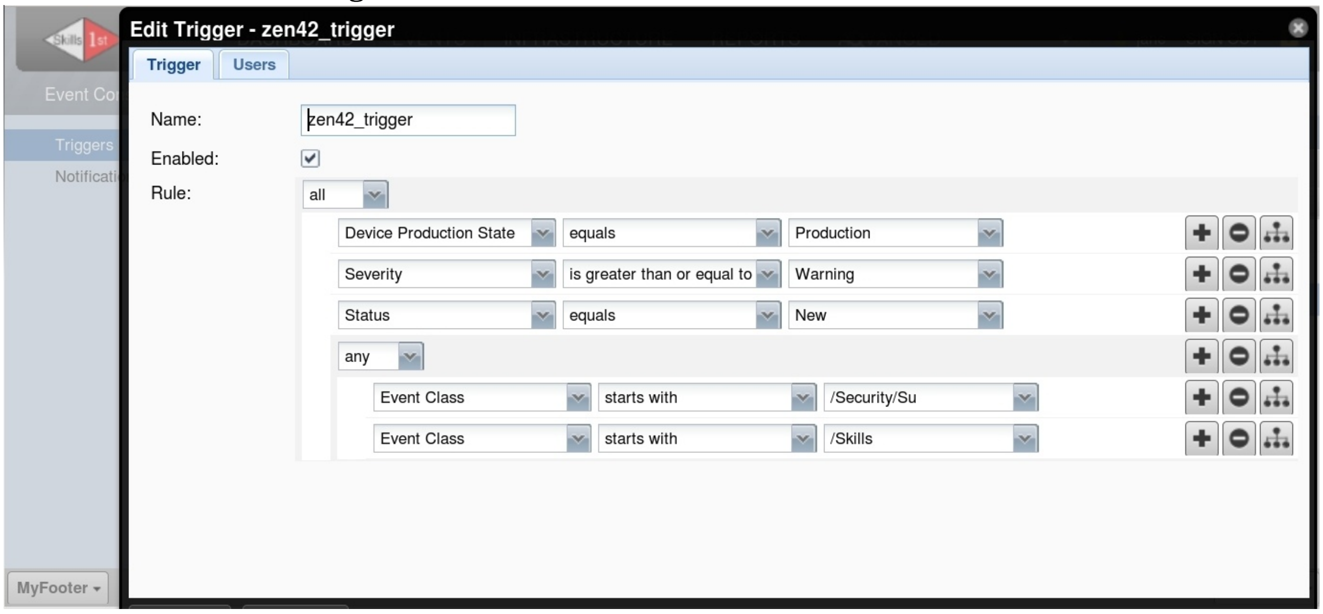
Device Priority | equals |

Figure 70: Creating a new Trigger

Note that by default, a new trigger is created as **Enabled** but with an illegal rule! *DevicePriority equals* without a value will cause lots of errors in zeneventserver.log.

When creating the Trigger rule, combinations or logical ANDs and ORs can be used (the *all* and *any* options). Use the + icon to add further conditions. All the standard event attributes are available to select from the dropdown boxes. User-defined event fields are **not** available here although it is possible in ZenPacks to provide for user-defined event fields.

Unlike earlier versions of Zenoss, it is also possible to nest criteria to build up the overall rule. Use the right-most icon to add a nested clause.



Name: zen42_trigger

Enabled: ✓

Rule: all

Device Production State	equals	Production
Severity	is greater than or equal to	Warning
Status	equals	New

any

Event Class	starts with	/Security/Su
Event Class	starts with	/Skills

Figure 71: A Trigger rule with nested clause

The *Users* tab of the Trigger definition is to control who can manipulate this Trigger. Both global and specific roles can be allocated. Users who have either the global Manager or ZenManager role will automatically have manage access to triggers, as will the trigger "owner" (creator).

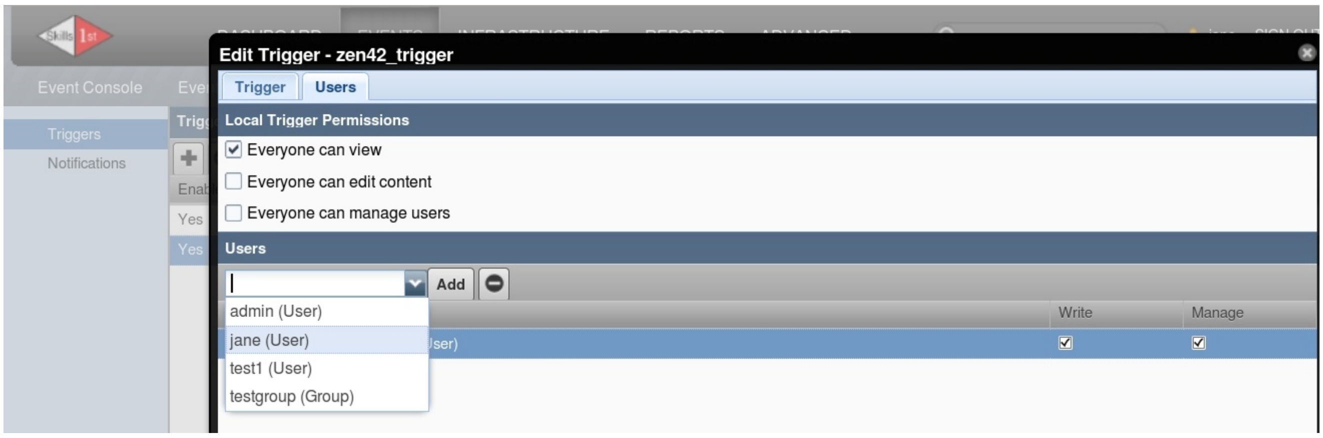

Edit Trigger - zen42_trigger
Trigger
Users
Local Trigger Permissions
Everyone can view
Everyone can edit content
Everyone can manage users
Users
Add
admin (User)
jane (User)
test1 (User)
testgroup (Group)
Write
Manage

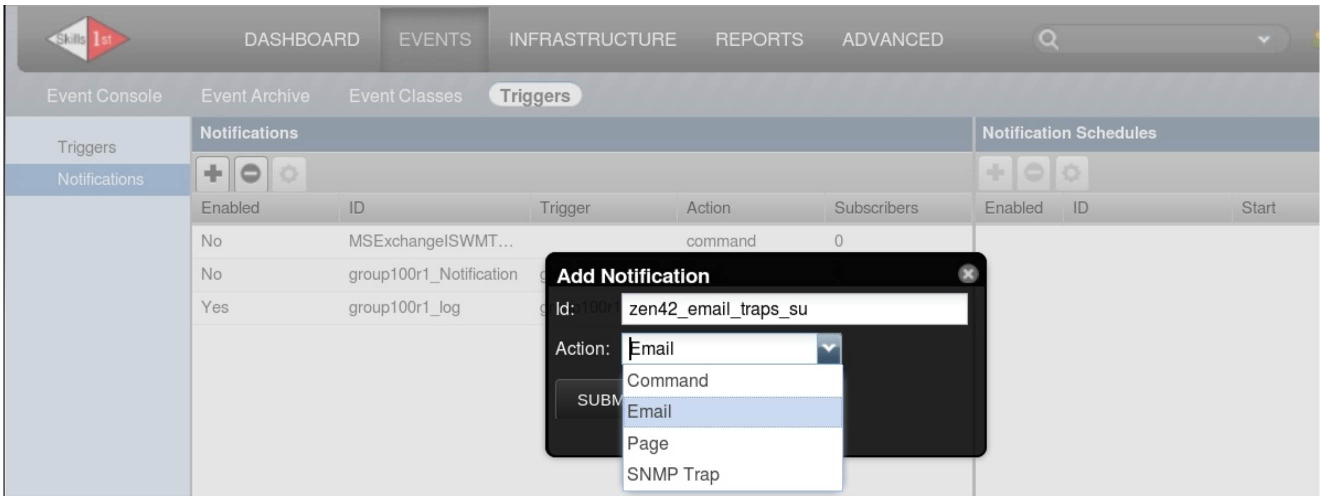
Figure 72: Trigger Users tab for global and user-specific roles

Note that this *Users* tab has no effect on who receives any related Notifications.

10.4 Notifications

Notifications are created from the same menu path as Triggers. A name and a notification type are the initial requirements.

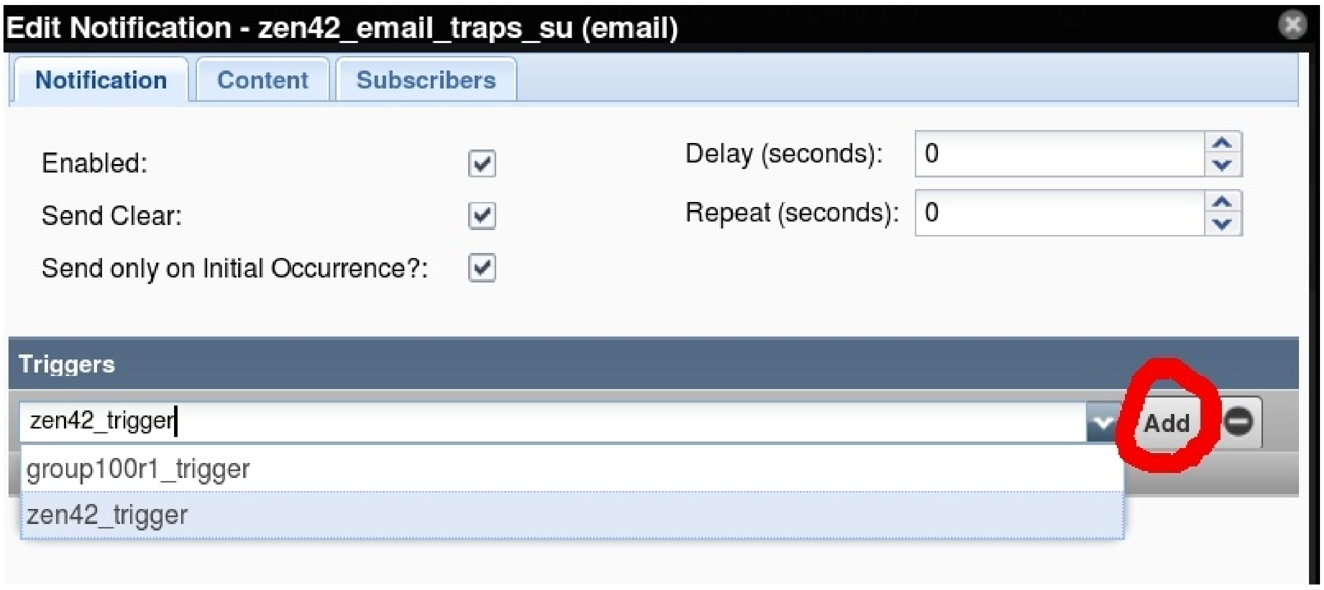
Note that a careful naming convention for Triggers and Notifications makes the environment much easier to work with.



Enabled	ID	Trigger	Action	Subscribers
No	MSExchangeISWMT...		command	0
No	group100r1_Notification			
Yes	group100r1_log			

Figure 73: Creating an email Notification

The Notification is created **not** Enabled by default. You can choose whether to send “good news” Clear notifications and whether to delay a Notification (useful for less critical events that may self-clear). Events can be sent repeatedly or only on the initial occurrence.


Edit Notification - zen42_email_traps_su (email)

Notification | Content | Subscribers

Enabled: ✓

Send Clear: ✓

Send only on Initial Occurrence?: ✓

Delay (seconds): 0

Repeat (seconds): 0

Triggers

zen42_trigger

group100r1_trigger

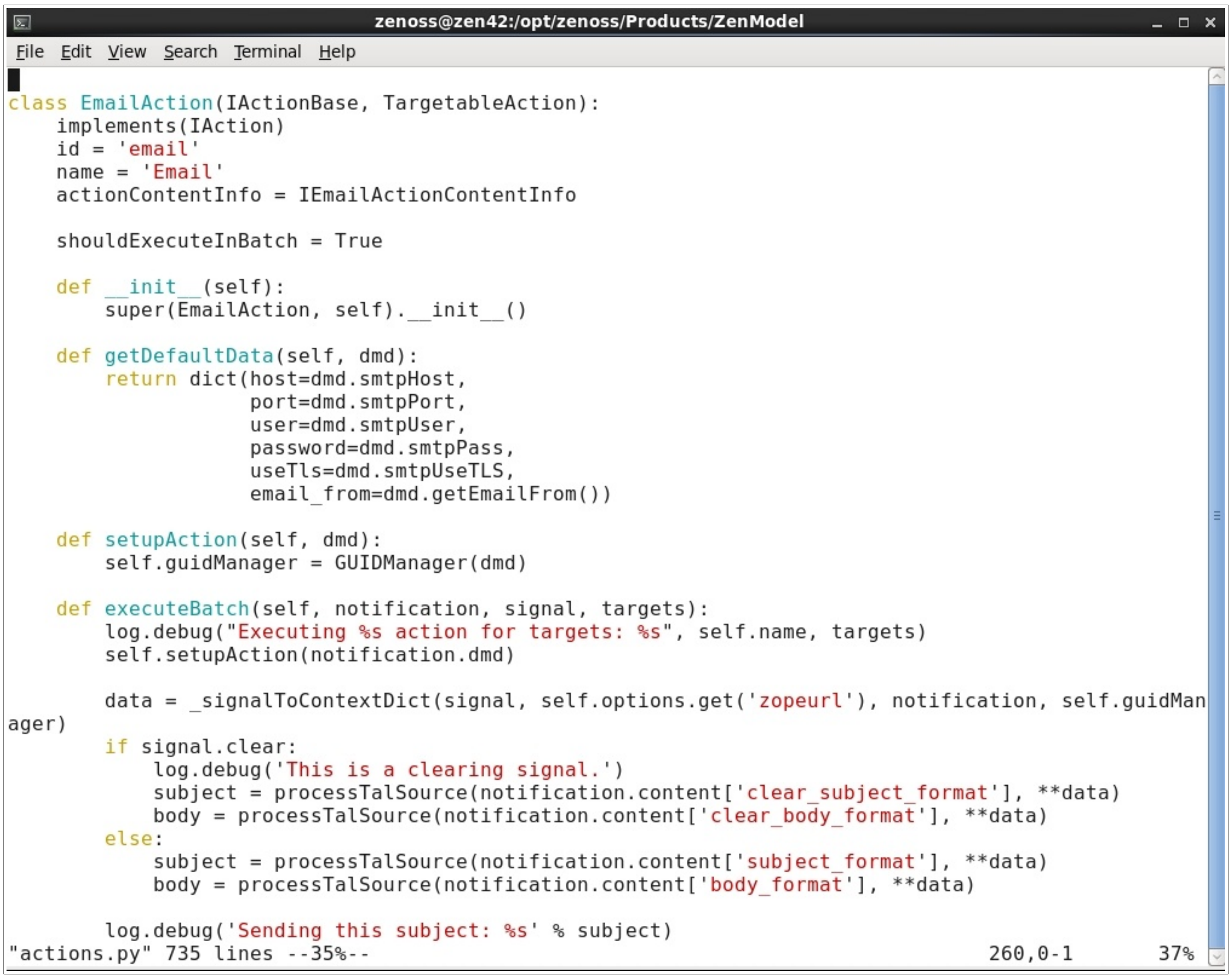
zen42_trigger

Figure 74: Notification details

A key field for a Notification is the Trigger that causes the Notification. Configured Triggers will be offered in the dropdown box. **Make sure** you select a Trigger and click *Add* - if you simply select the Trigger and then *SUBMIT* the entire Notification, the Trigger will **not** be saved.

Depending on the Notification type selected when the Notification is created, the *Content* tab will vary; the others remain the same, though for Command and Trap notifications the *Subscriber* tab is not relevant to whether the action takes place as these are background actions not user-related actions.

The different Notification actions are encoded in *$ZENHOME/Products/ZenModel/actions.py*.


zenoss@zen42:/opt/zenoss/Products/ZenModel
File Edit View Search Terminal Help

```
class EmailAction(IActionBase, TargetableAction):
    implements(IAction)
    id = 'email'
    name = 'Email'
    actionContentInfo = IEmailActionContentInfo

    shouldExecuteInBatch = True

    def __init__(self):
        super(EmailAction, self).__init__()

    def getDefaultData(self, dmd):
        return dict(host=dmd.smtpHost,
                    port=dmd.smtpPort,
                    user=dmd.smtpUser,
                    password=dmd.smtpPass,
                    useTls=dmd.smtpUseTLS,
                    email_from=dmd.getEmailFrom())

    def setupAction(self, dmd):
        self.guidManager = GUIDManager(dmd)

    def executeBatch(self, notification, signal, targets):
        log.debug("Executing %s action for targets: %s", self.name, targets)
        self.setupAction(notification.dmd)

        data = _signalToContextDict(signal, self.options.get('zopeurl'), notification, self.guidMan
ager)
        if signal.clear:
            log.debug('This is a clearing signal.')
            subject = processTalSource(notification.content['clear_subject_format'], **data)
            body = processTalSource(notification.content['clear_body_format'], **data)
        else:
            subject = processTalSource(notification.content['subject_format'], **data)
            body = processTalSource(notification.content['body_format'], **data)

        log.debug('Sending this subject: %s' % subject)
"actions.py" 735 lines --35%--                                              260,0-1       37%
```

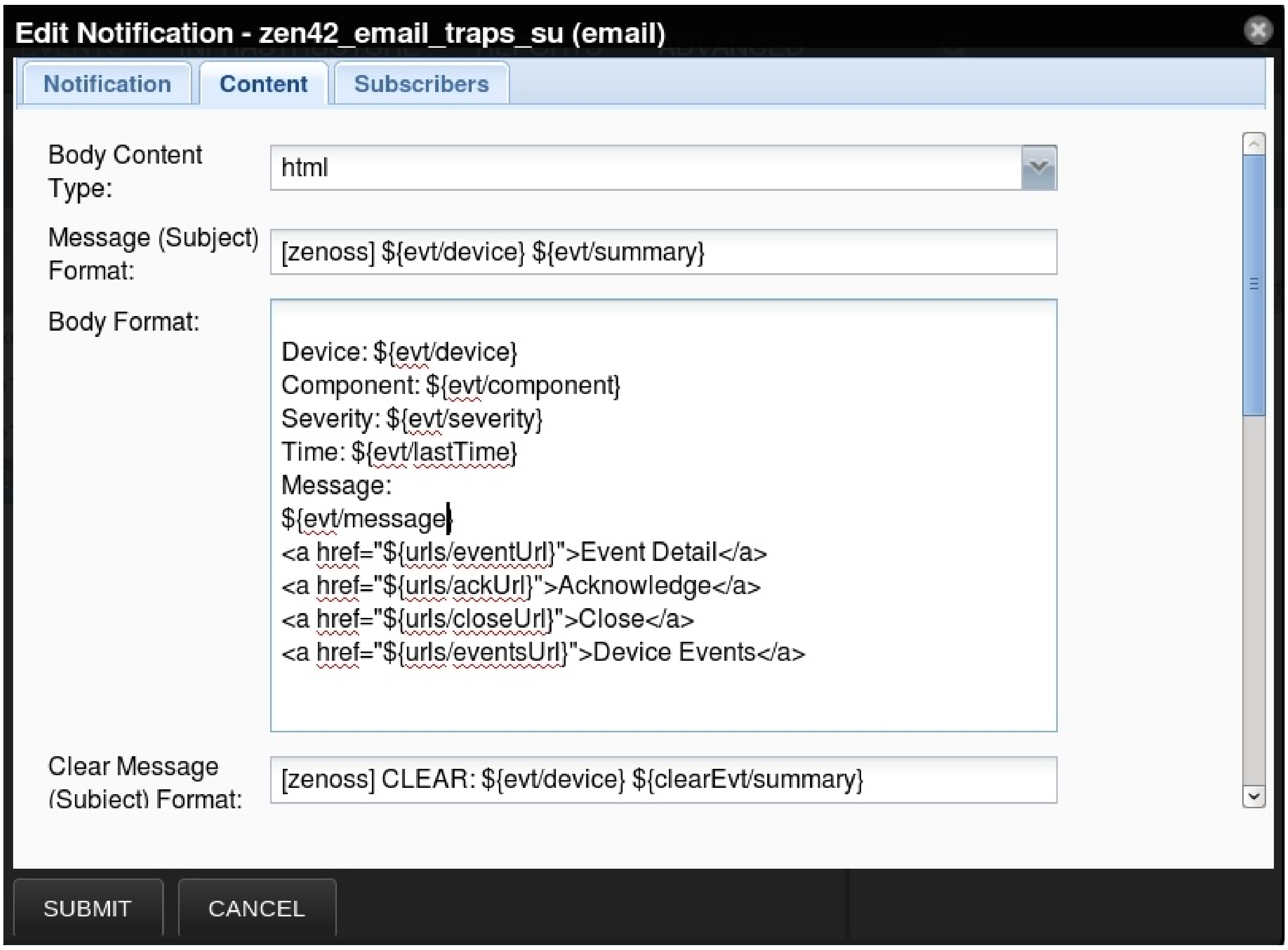
Figure 75: $ZENHOME/Products/ZenModel/actions.py implements Notification actions

10.4.1 email Notifications

The *Content* tab for email allows you to customise the email subject and body, using standard fields from the event, using TALES expressions (**T**emplate **A**ttribute **L**anguage **E**xpression **S**yntax, from Zope) to reference fields of the event, *evt*. See Appendix D of the Zenoss Administration Guide for more details. **Note** that you **must** use TALES – the *evt.<event field>* syntax used in mapping rules and transforms does not work in event commands. TALES syntax takes the form:

```
${evt/<event field>}
```

Also see section 2.6 of the Zenoss Core 4 Administrators Guide.


Edit Notification - zen42_email_traps_su (email)

Body Content Type: html

Message (Subject) Format: [zenoss] ${evt/device} ${evt/summary}

Body Format:

```
Device: ${evt/device}
Component: ${evt/component}
Severity: ${evt/severity}
Time: ${evt/lastTime}
Message:
${evt/message}
<a href="${urls/eventUrl}">Event Detail</a>
<a href="${urls/ackUrl}">Acknowledge</a>
<a href="${urls/closeUrl}">Close</a>
<a href="${urls/eventsUrl}">Device Events</a>
```

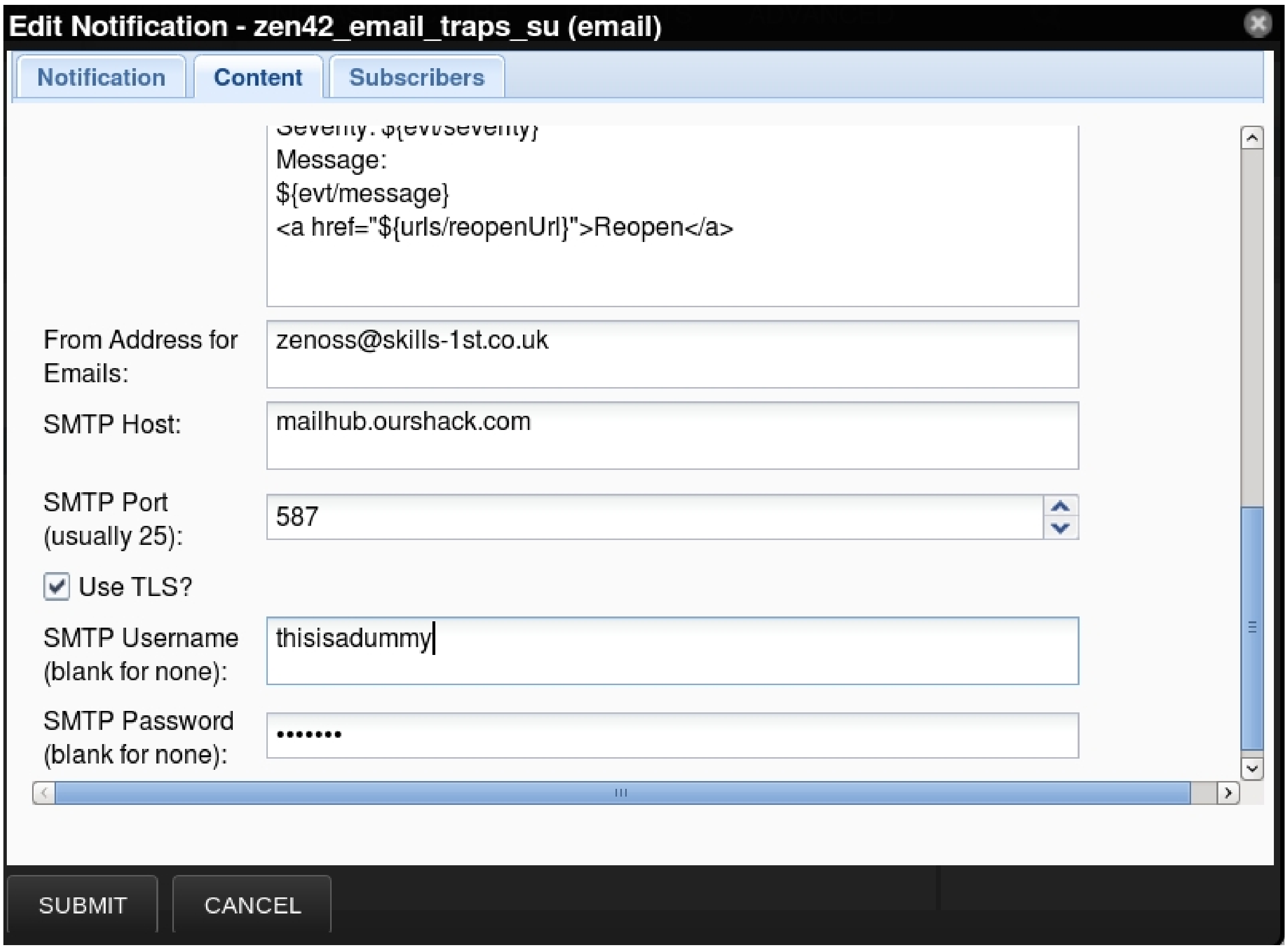
Clear Message (Subject) Format: [zenoss] CLEAR: ${evt/device} ${clearEvt/summary}

Figure 76: The Content tab of a Notification - part 1

Also note that previous versions of Zenoss provided access to the *dev* variable to access attributes of the device that caused the event. The *dev* variable is no longer legal for use in Notification content.

Separate definitions can be provided for the problem and clearing Notifications.

The bottom of the Notification configuration panel allows you to override default configurations for mail host parameters.

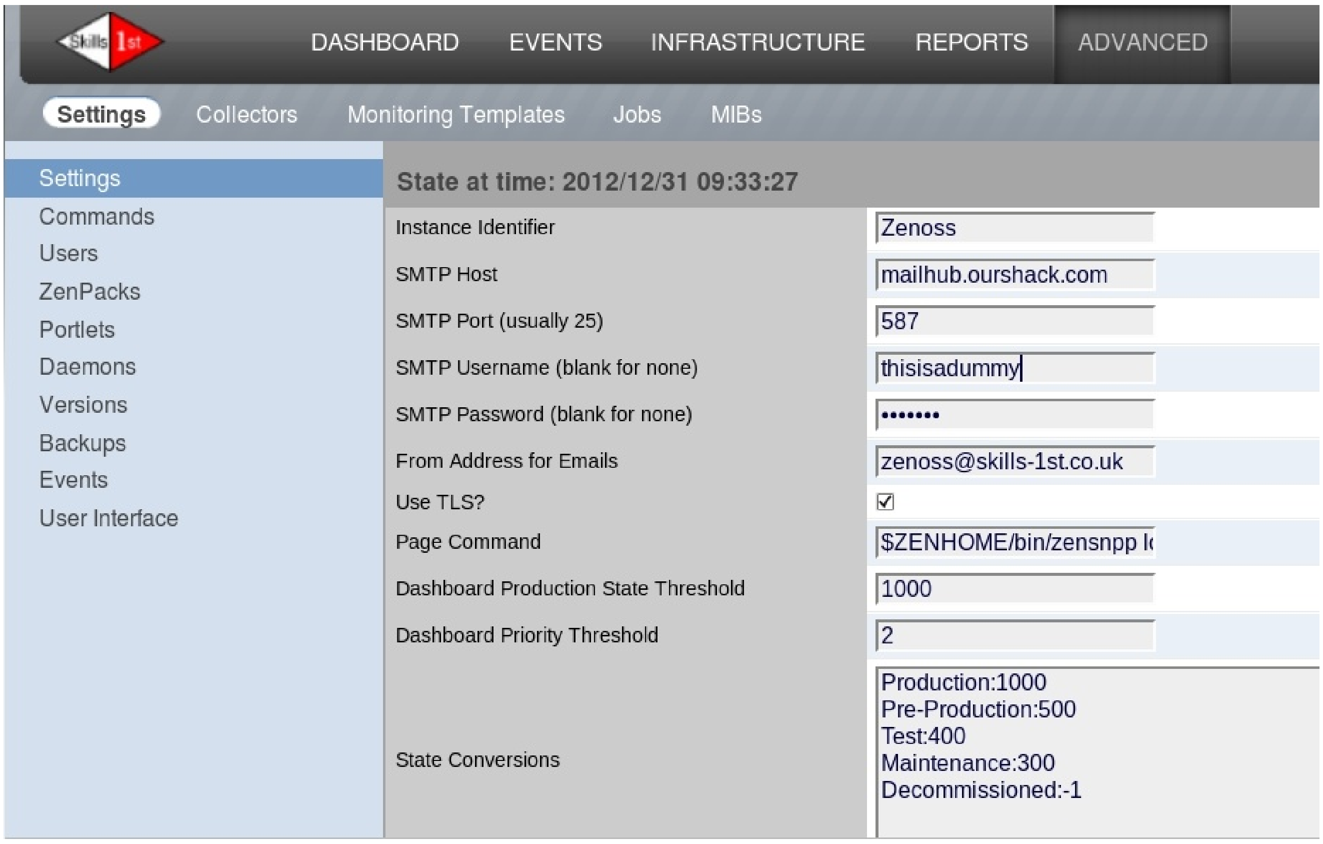


```
Message:
${evt/message}
<a href="${urls/reopenUrl}">Reopen</a>
```

From Address for Emails:	zenoss@skills-1st.co.uk
SMTP Host:	mailhub.ourshack.com
SMTP Port (usually 25):	587
Use TLS?	✓
SMTP Username (blank for none):	thisisadummy
SMTP Password (blank for none):	•••••••

Figure 77: Notification Content with mail server parameters

These parameters are specified globally from the *ADVANCED* -> *Settings* -> *Settings* menu.



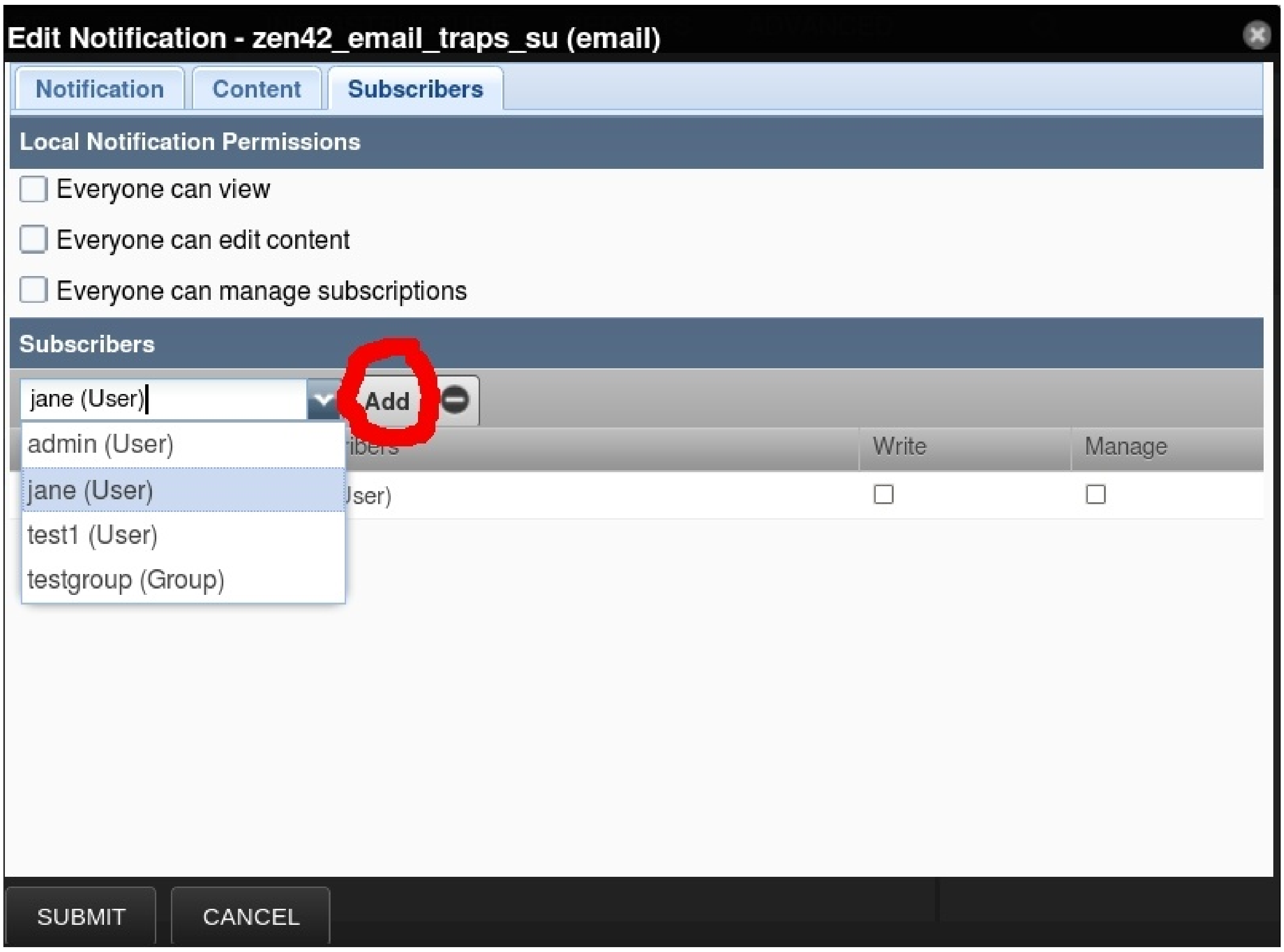
State at time: 2012/12/31 09:33:27

Instance Identifier	Zenoss
SMTP Host	mailhub.ourshack.com
SMTP Port (usually 25)	587
SMTP Username (blank for none)	thisisadummy
SMTP Password (blank for none)	•••••••
From Address for Emails	zenoss@skills-1st.co.uk
Use TLS?	✓
Page Command	$ZENHOME/bin/zensnpp l
Dashboard Production State Threshold	1000
Dashboard Priority Threshold	2
State Conversions	Production:1000 Pre-Production:500 Test:400 Maintenance:300 Decommissioned:-1

Figure 78: Default settings for mail server and paging

Do ensure that the *From Address for Emails* settings are legal for mailservers. A difficult scenario to debug is where email notifications never arrive because they are discarded by a mail server because of the *From* address.

The third tab, *Subscribers*, on the Notification definition panel defines who receives the notification. In addition, this panel also servers a similar purpose to the *Users* tab for Triggers in that it defines who is allowed to **manage** the Notification definition. Unlike Triggers, if no subscriber (user or user-group) is specified (and explicitly **Add**ed) then no email will be received. It is not necessary to specify any management roles though.


Edit Notification - zen42_email_traps_su (email)

Local Notification Permissions

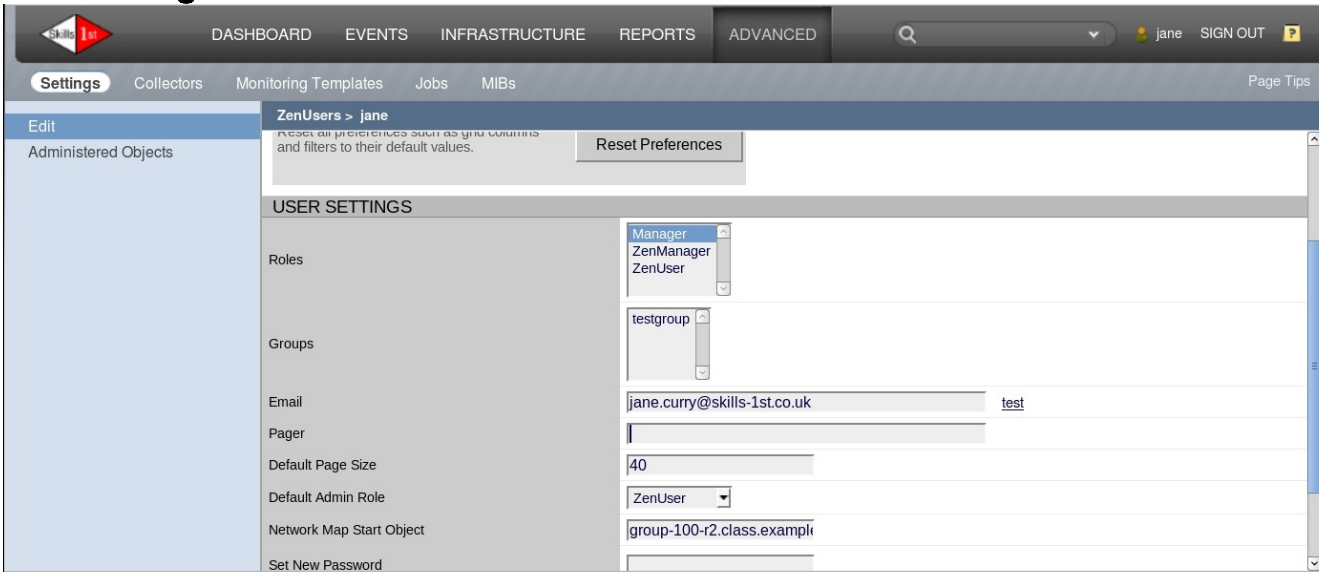
- [ ] Everyone can view
- [ ] Everyone can edit content
- [ ] Everyone can manage subscriptions

Subscribers

jane (User)

Figure 79: Subscribers to Notifications

10.4.2 Page Notifications



Reset all preferences such as grid columns and filters to their default values.

USER SETTINGS

Roles	Manager, ZenManager, ZenUser
Groups	testgroup
Email	jane.curry@skills-1st.co.uk (test)
Pager	
Default Page Size	40
Default Admin Role	ZenUser
Network Map Start Object	group-100-r2.class.exampl
Set New Password	

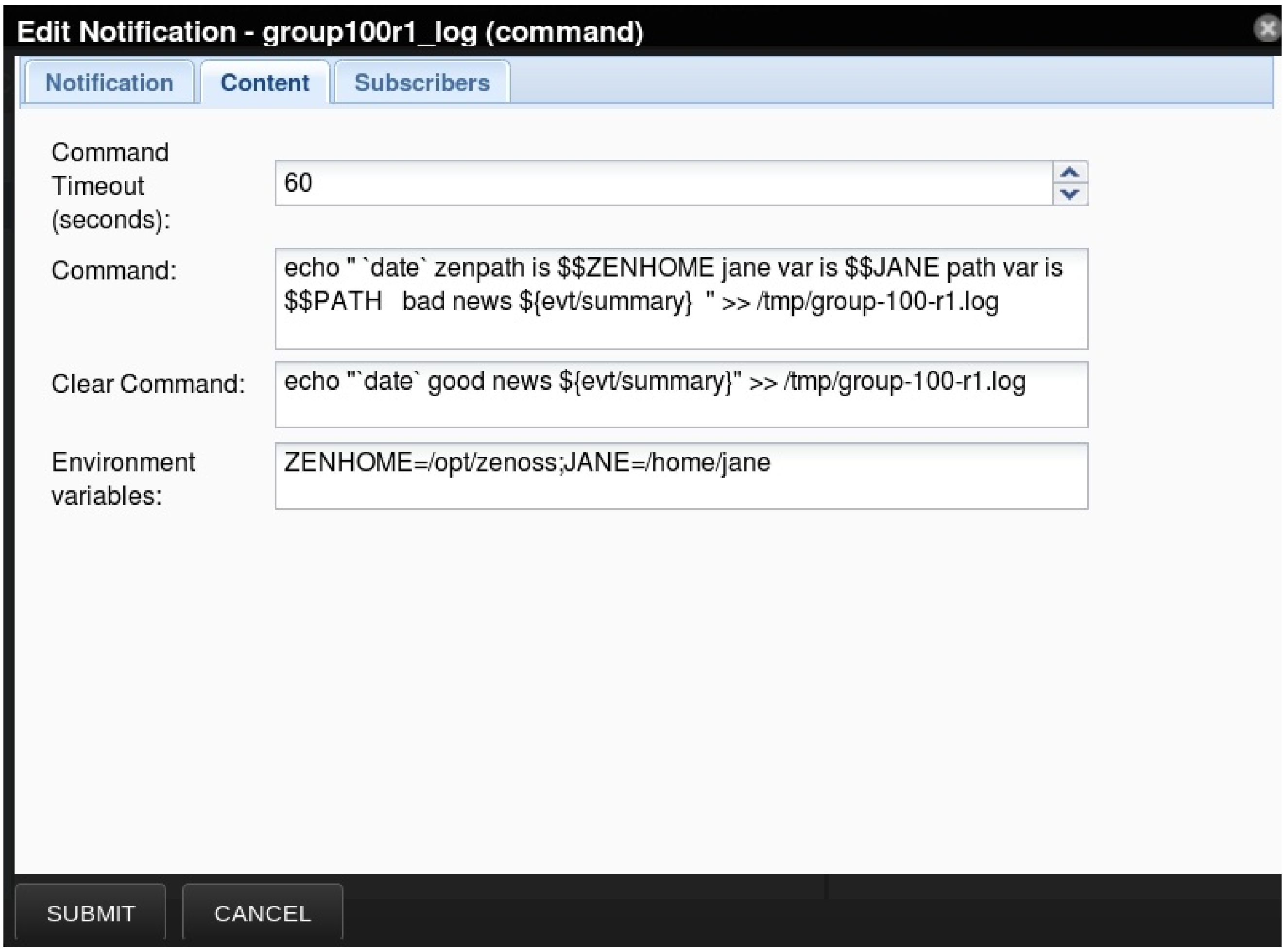
Figure 80: User settings showing email and page parameters

A *Page* notification is very similar to email, simply providing a Content tab to specify a Message format and a Clear Message format. As with email, the *evt* variable is available for parameter substitution. The command used to send page messages is that specified globally from *ADVANCED* -> *Settings* -> *Settings* (see Figure 78). The individual recipient comes from those users / groups specified in the *Subscribers* tab who must have their pager details configured on that users home page (this is also where a user's email address is specified).

10.4.3 Command Notifications

The Content tab for a Command Notification specifies a “bad news” and a “good news” command, a time parameter for how long the command may run until it is deemed to have failed, and environment variables can also be specified as <*variable*>=<*value*>.

The latter is useful as in past versions of Zenoss a common issue was to create an Event Command but forget to source any necessary environment variables in the script. Since the script is run by zenactiond, it has very little default context in which to run so things like $ZENHOME, $PATH were not automatically set.



Command Timeout (seconds):	60
Command:	echo " `date` zenpath is $$ZENHOME jane var is $$JANE path var is $$PATH bad news ${evt/summary} " >> /tmp/group-100-r1.log
Clear Command:	echo "`date` good news ${evt/summary}" >> /tmp/group-100-r1.log
Environment variables:	ZENHOME=/opt/zenoss;JANE=/home/jane

Figure 81: A Command Notification

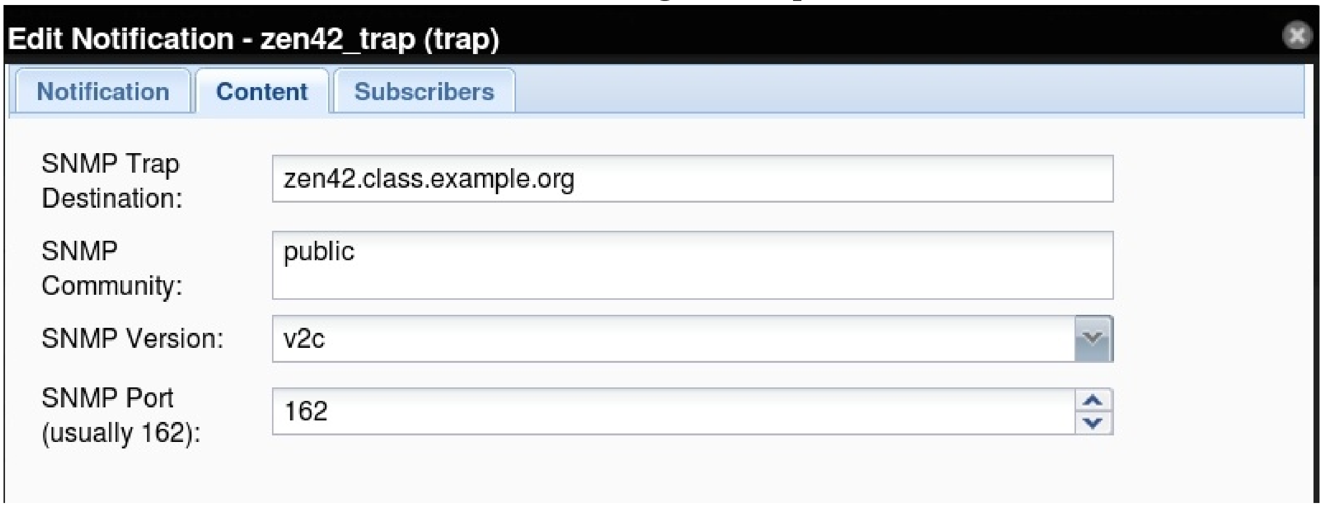
Note that to use these environment variables in a script you need to escape the dollar with a dollar eg. $$ZENHOME. Multiple environment variable are semicolon separated and you do not include the dollar when you specify the name of the environment variable.

Also note that, although a subscriber is not typically required as the Command notification is a background script, due to a bug In Core 4.2, environment variables will be ignored unless there **is** a subscriber. It is not onerous to setup a dummy user subscriber as a circumvention to this issue.

Command Notifications may be simple built-in shell commands as shown above or they can be complex scripts in other languages, provided they can be executed from a shell environment. Again, standard fields from the event can be substituted using TALES expressions. Note in the figure above the use of back-tics around the *date* command to run the date command before adding the output of the environment variables and the good news / bad news message.

10.4.4 TRAP Notifications

SNMP TRAP notifications are new with Zenoss 4. It was possible to create TRAP forwarding scenarios using Event Commands in the past but this ability is now standard. The *Content* tab in this case configures trap destination.



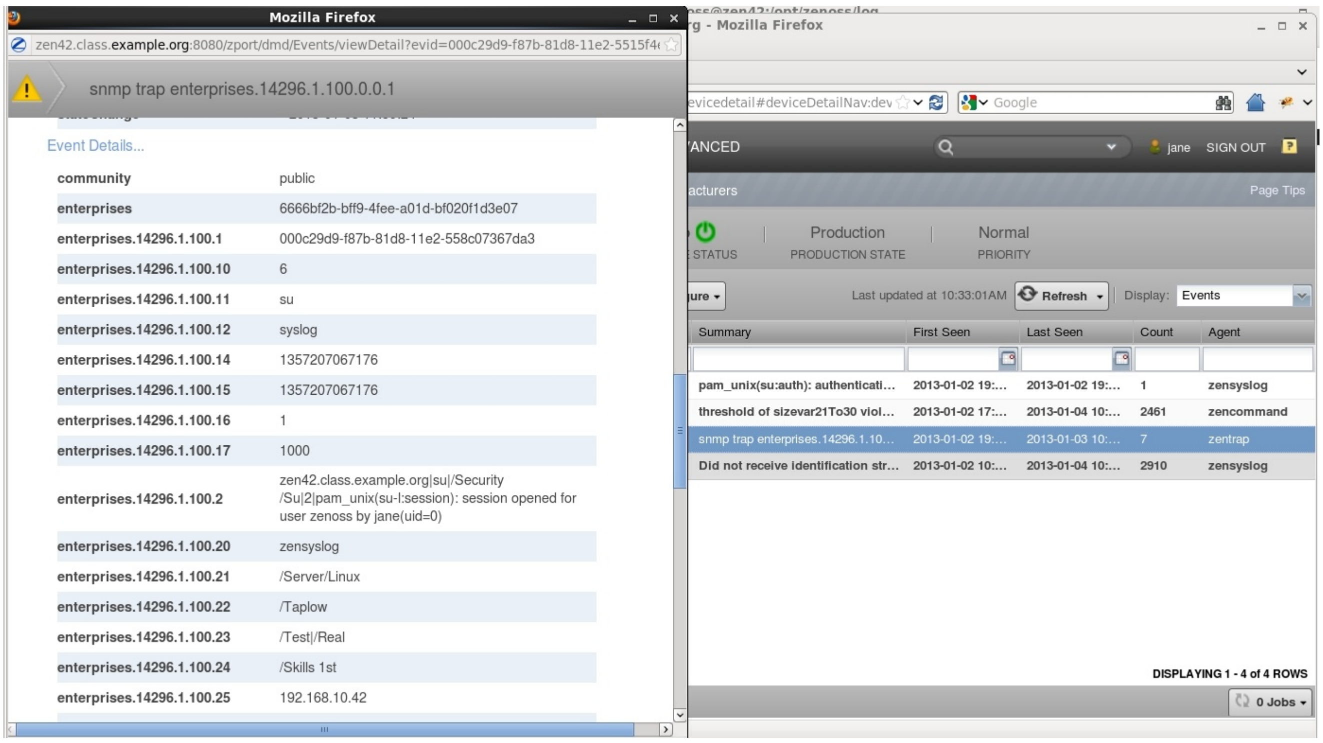
SNMP Trap Destination: zen42.class.example.org
SNMP Community: public
SNMP Version: v2c
SNMP Port (usually 162): 162

Figure 82: Trap notification

The trap destination may either be a resolvable name or an IP address.

Note that with Zenoss Core 4.2 there is a bug that means selecting SNMP v1 results in no TRAP being issued, even though zenactiond.log reports that a TRAP has been successfully sent.

The TRAP is defined in $ZENHOME/share/mibs/site/ZENOSS-MIB.txt. It is a single TRAP with many varbinds that are populated with the fields of the original event. It would be good practise to import this MIB into a Zenoss server that is receiving such notification TRAPs.



Mozilla Firefox

zen42.class.example.org:8080/zport/dmd/Events/viewDetail?evid=000c29d9-f87b-81d8-11e2-5515f4

snmp trap enterprises.14296.1.100.0.0.1

Event Details...

community	public
enterprises	6666bf2b-bff9-4fee-a01d-bf020f1d3e07
enterprises.14296.1.100.1	000c29d9-f87b-81d8-11e2-558c07367da3
enterprises.14296.1.100.10	6
enterprises.14296.1.100.11	su
enterprises.14296.1.100.12	syslog
enterprises.14296.1.100.14	1357207067176
enterprises.14296.1.100.15	1357207067176
enterprises.14296.1.100.16	1
enterprises.14296.1.100.17	1000
enterprises.14296.1.100.2	zen42.class.example.org|su|/Security/Su|2|pam_unix(su-l:session): session opened for user zenoss by jane(uid=0)
enterprises.14296.1.100.20	zensyslog
enterprises.14296.1.100.21	/Server/Linux
enterprises.14296.1.100.22	/Taplow
enterprises.14296.1.100.23	/Test|/Real
enterprises.14296.1.100.24	/Skills 1st
enterprises.14296.1.100.25	192.168.10.42

Production STATUS | Production PRODUCTION STATE | Normal PRIORITY

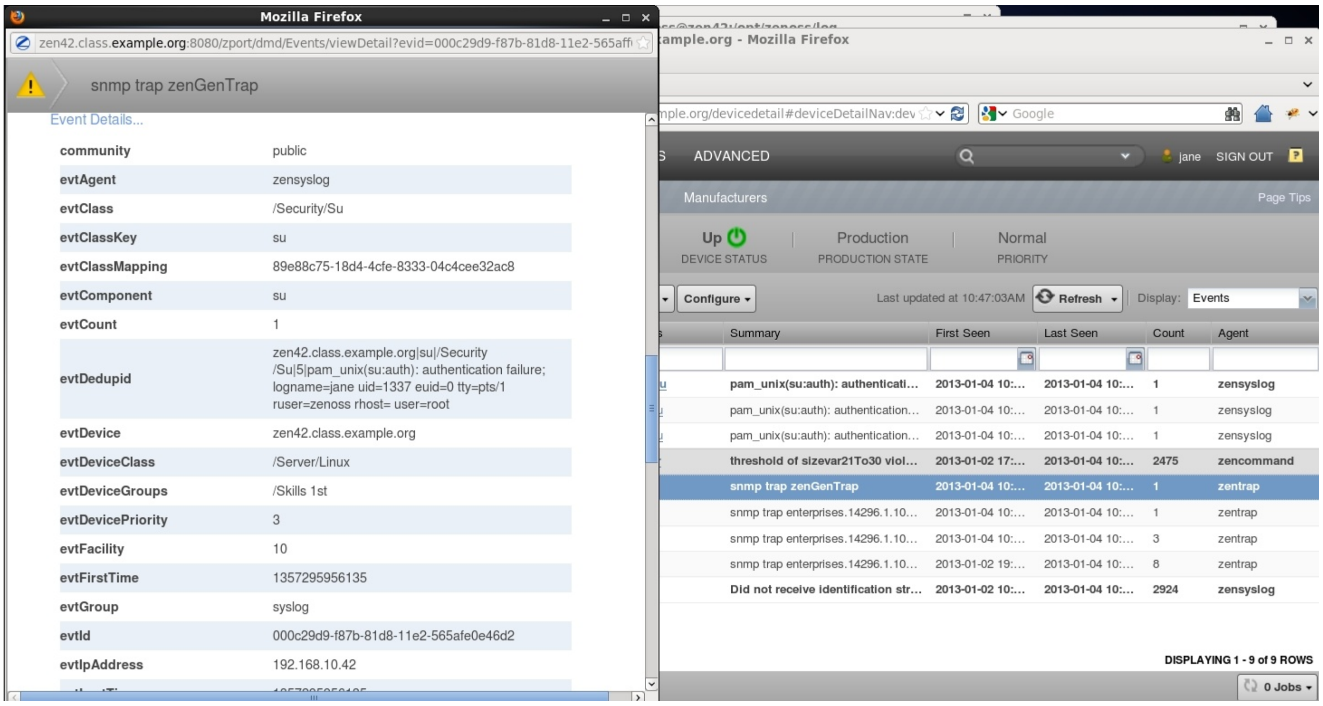
Last updated at 10:33:01AM Refresh Display: Events

Summary	First Seen	Last Seen	Count	Agent
pam_unix(su:auth): authenticati...	2013-01-02 19:...	2013-01-02 19:...	1	zensyslog
threshold of sizevar21To30 viol...	2013-01-02 17:...	2013-01-04 10:...	2461	zencommand
snmp trap enterprises.14296.1.10...	2013-01-02 19:...	2013-01-03 10:...	7	zentrap
Did not receive identification str...	2013-01-02 10:...	2013-01-04 10:...	2910	zensyslog

DISPLAYING 1 - 4 of 4 ROWS

*Figure 83: Trap resulting from a Notification TRAP **without** the ZENOSS-MIB.txt imported*

Thus the varbind names will be translated to something more helpful.



Mozilla Firefox

zen42.class.example.org:8080/zport/dmd/Events/viewDetail?evid=000c29d9-f87b-81d8-11e2-565aff

snmp trap zenGenTrap

Event Details...

community	public
evtAgent	zensyslog
evtClass	/Security/Su
evtClassKey	su
evtClassMapping	89e88c75-18d4-4cfe-8333-04c4cee32ac8
evtComponent	su
evtCount	1
evtDedupid	zen42.class.example.org|su|/Security/Su|5|pam_unix(su:auth): authentication failure; logname=jane uid=1337 euid=0 tty=pts/1 ruser=zenoss rhost= user=root
evtDevice	zen42.class.example.org
evtDeviceClass	/Server/Linux
evtDeviceGroups	/Skills 1st
evtDevicePriority	3
evtFacility	10
evtFirstTime	1357295956135
evtGroup	syslog
evtId	000c29d9-f87b-81d8-11e2-565afe0e46d2
evtIpAddress	192.168.10.42

Up DEVICE STATUS | Production PRODUCTION STATE | Normal PRIORITY

Configure Last updated at 10:47:03AM Refresh Display: Events

Summary	First Seen	Last Seen	Count	Agent
pam_unix(su:auth): authenticati...	2013-01-04 10:...	2013-01-04 10:...	1	zensyslog
pam_unix(su:auth): authentication...	2013-01-04 10:...	2013-01-04 10:...	1	zensyslog
pam_unix(su:auth): authentication...	2013-01-04 10:...	2013-01-04 10:...	1	zensyslog
threshold of sizevar21To30 viol...	2013-01-02 17:...	2013-01-04 10:...	2475	zencommand
snmp trap zenGenTrap	2013-01-04 10:...	2013-01-04 10:...	1	zentrap
snmp trap enterprises.14296.1.10...	2013-01-04 10:...	2013-01-04 10:...	1	zentrap
snmp trap enterprises.14296.1.10...	2013-01-04 10:...	2013-01-04 10:...	3	zentrap
snmp trap enterprises.14296.1.10...	2013-01-02 19:...	2013-01-04 10:...	8	zentrap
Did not receive identification str...	2013-01-02 10:...	2013-01-04 10:...	2924	zensyslog

DISPLAYING 1 - 9 of 9 ROWS

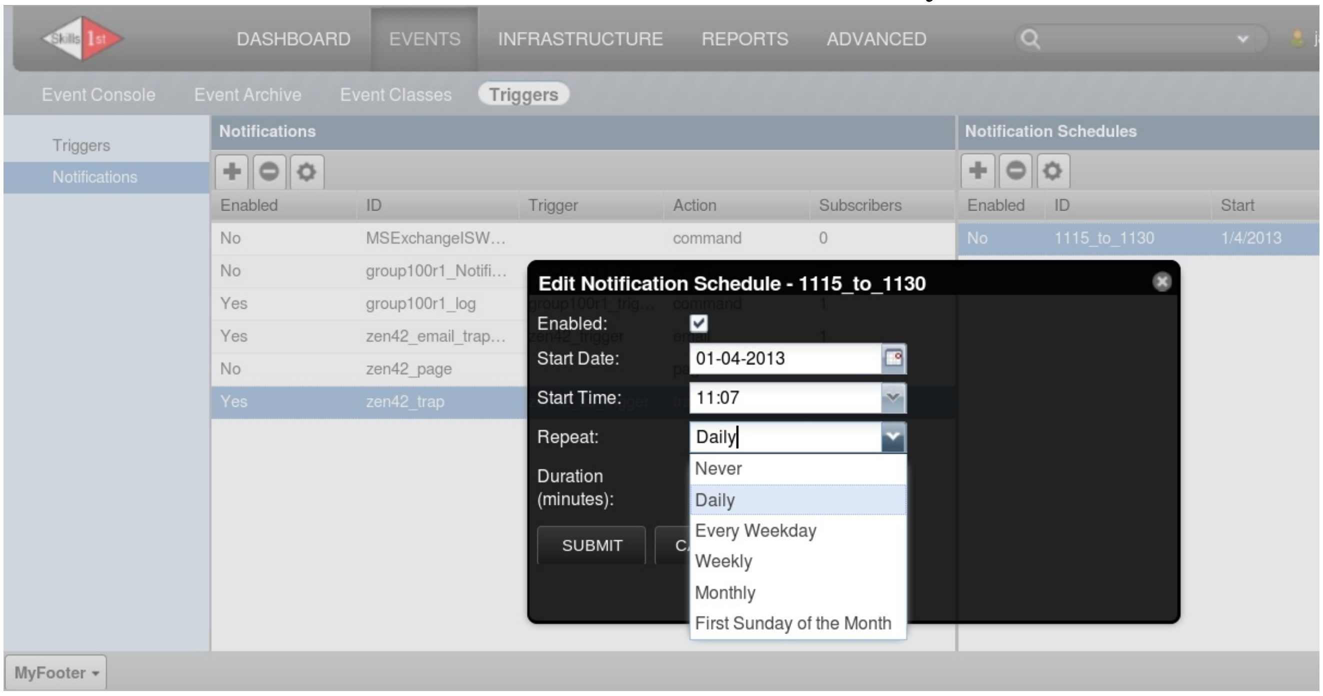
Figure 84: Trap resulting from a Notification TRAP with the Zenoss MIB imported

Careful inspection of the TRAP with the Zenoss MIB imported reveals an omission in the MIB; varbind 8 for the message field is not defined so it shows in the event details with the name **zenTrapDef.8**.

Note that the version of ZENOSS-MIB.txt shipped with Core 4.2.3 has been modified from the 4.2 version in such a way that it does not import cleanly (there are non-printing characters in the file). For a description of the problem and a working file, see http://jira.zenoss.com/jira/browse/ZEN-5060 .

10.5 Notification Schedules

Any Notification type may have one or more schedules associated with it. These are effectively Maintenance Windows (and are indeed implemented by the same code as Maintenance Windows). They allow different responses to take place at different times. If no Notification Schedule exists then the Notification is always active.



Notifications

Enabled	ID	Trigger	Action	Subscribers
No	MSExchangeISW…		command	0
No	group100r1_Notifi…			
Yes	group100r1_log			
Yes	zen42_email_trap…			
No	zen42_page			
Yes	zen42_trap			

Notification Schedules

Enabled	ID	Start
No	1115_to_1130	1/4/2013

Edit Notification Schedule - 1115_to_1130

Enabled: ✓
Start Date: 01-04-2013
Start Time: 11:07
Repeat: Daily
Duration (minutes):

Never
Daily
Every Weekday
Weekly
Monthly
First Sunday of the Month

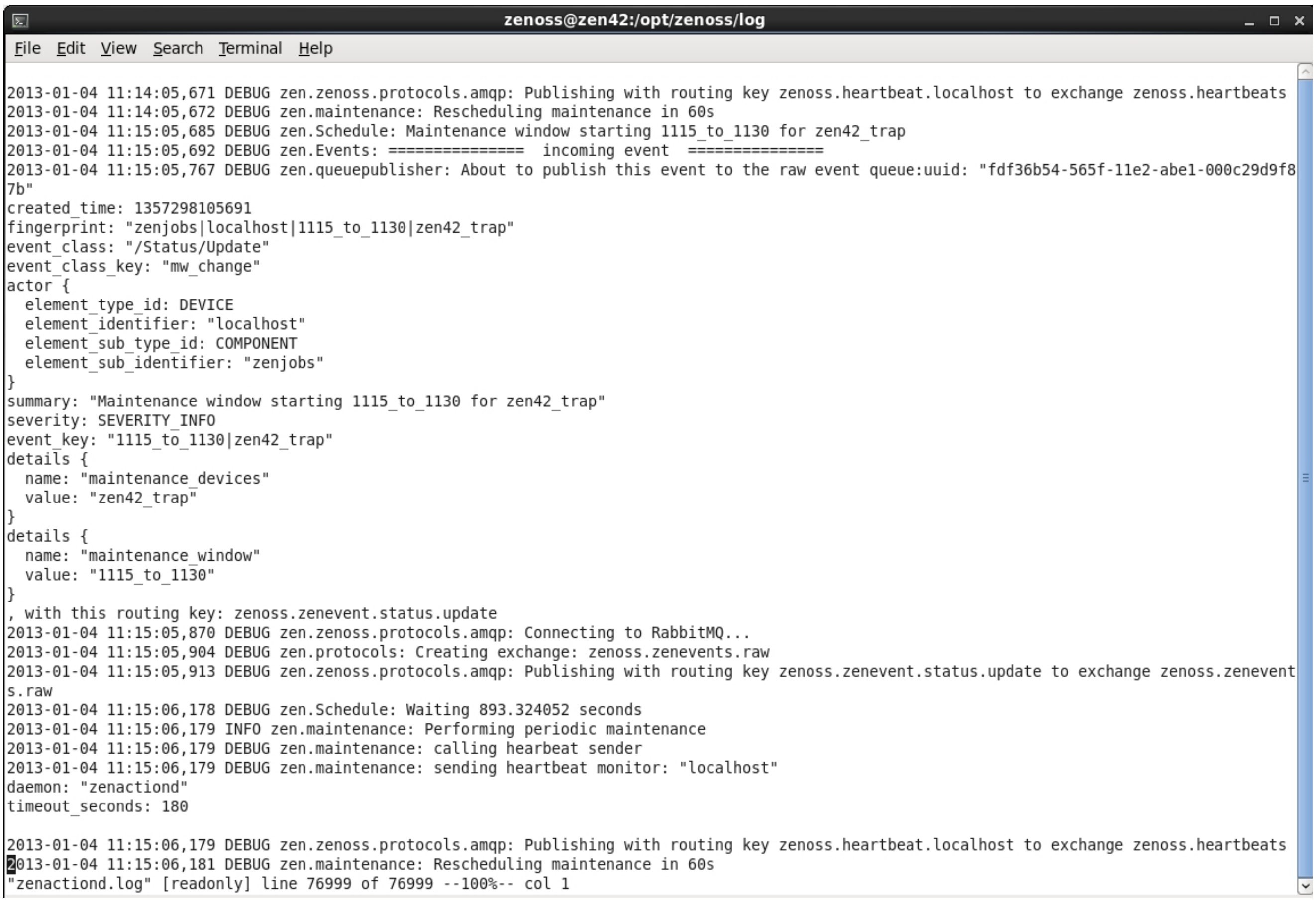
SUBMIT

Figure 85: Notification schedule

The schedule is created as **not** Enabled by default. Typically the schedule will repeat over certain periods - see Figure 85.

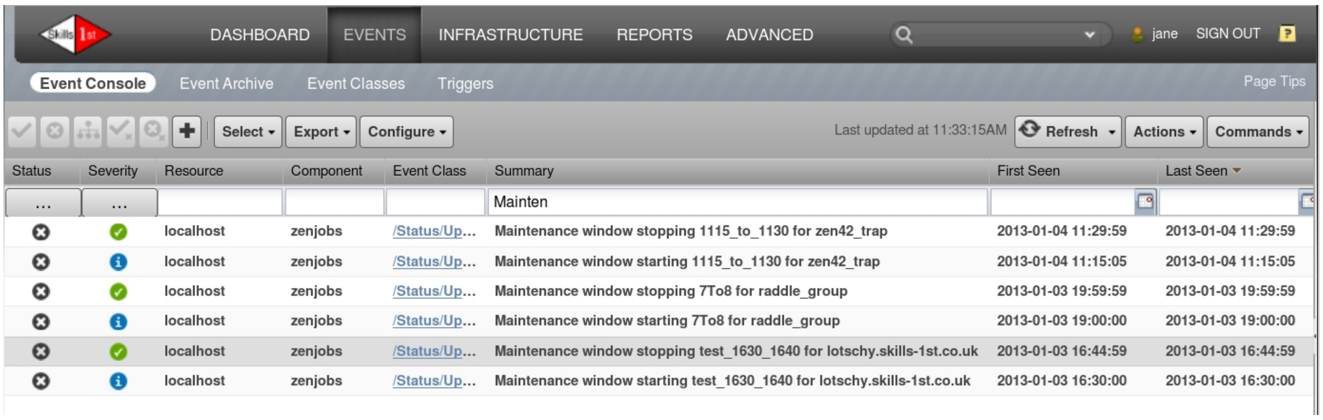
With debug logging turned on for the zenactiond daemon, the start of a Notification schedule can be clearly seen.

An **Info** severity event is created when any Maintenance Window starts and it is cleared by the **Clear** severity event generated when the Maintenance Window ends.


zenoss@zen42:/opt/zenoss/log
File Edit View Search Terminal Help

```
2013-01-04 11:14:05,671 DEBUG zen.zenoss.protocols.amqp: Publishing with routing key zenoss.heartbeat.localhost to exchange zenoss.heartbeats
2013-01-04 11:14:05,672 DEBUG zen.maintenance: Rescheduling maintenance in 60s
2013-01-04 11:15:05,685 DEBUG zen.Schedule: Maintenance window starting 1115_to_1130 for zen42_trap
2013-01-04 11:15:05,692 DEBUG zen.Events: ==============  incoming event  ==============
2013-01-04 11:15:05,767 DEBUG zen.queuepublisher: About to publish this event to the raw event queue:uuid: "fdf36b54-565f-11e2-abe1-000c29d9f8
7b"
created_time: 1357298105691
fingerprint: "zenjobs|localhost|1115_to_1130|zen42_trap"
event_class: "/Status/Update"
event_class_key: "mw_change"
actor {
  element_type_id: DEVICE
  element_identifier: "localhost"
  element_sub_type_id: COMPONENT
  element_sub_identifier: "zenjobs"
}
summary: "Maintenance window starting 1115_to_1130 for zen42_trap"
severity: SEVERITY_INFO
event_key: "1115_to_1130|zen42_trap"
details {
  name: "maintenance_devices"
  value: "zen42_trap"
}
details {
  name: "maintenance_window"
  value: "1115_to_1130"
}
, with this routing key: zenoss.zenevent.status.update
2013-01-04 11:15:05,870 DEBUG zen.zenoss.protocols.amqp: Connecting to RabbitMQ...
2013-01-04 11:15:05,904 DEBUG zen.protocols: Creating exchange: zenoss.zenevents.raw
2013-01-04 11:15:05,913 DEBUG zen.zenoss.protocols.amqp: Publishing with routing key zenoss.zenevent.status.update to exchange zenoss.zenevent
s.raw
2013-01-04 11:15:06,178 DEBUG zen.Schedule: Waiting 893.324052 seconds
2013-01-04 11:15:06,179 INFO zen.maintenance: Performing periodic maintenance
2013-01-04 11:15:06,179 DEBUG zen.maintenance: calling hearbeat sender
2013-01-04 11:15:06,179 DEBUG zen.maintenance: sending heartbeat monitor: "localhost"
daemon: "zenactiond"
timeout_seconds: 180

2013-01-04 11:15:06,179 DEBUG zen.zenoss.protocols.amqp: Publishing with routing key zenoss.heartbeat.localhost to exchange zenoss.heartbeats
2013-01-04 11:15:06,181 DEBUG zen.maintenance: Rescheduling maintenance in 60s
"zenactiond.log" [readonly] line 76999 of 76999 --100%-- col 1
```

Figure 86: zenactiond.log showing the start of a Notification Schedule



Status	Severity	Resource	Component	Event Class	Summary	First Seen	Last Seen
...	...				Mainten		
⊗	✓	localhost	zenjobs	/Status/Up...	Maintenance window stopping 1115_to_1130 for zen42_trap	2013-01-04 11:29:59	2013-01-04 11:29:59
⊗	ℹ	localhost	zenjobs	/Status/Up...	Maintenance window starting 1115_to_1130 for zen42_trap	2013-01-04 11:15:05	2013-01-04 11:15:05
⊗	✓	localhost	zenjobs	/Status/Up...	Maintenance window stopping 7To8 for raddle_group	2013-01-03 19:59:59	2013-01-03 19:59:59
⊗	ℹ	localhost	zenjobs	/Status/Up...	Maintenance window starting 7To8 for raddle_group	2013-01-03 19:00:00	2013-01-03 19:00:00
⊗	✓	localhost	zenjobs	/Status/Up...	Maintenance window stopping test_1630_1640 for lotschy.skills-1st.co.uk	2013-01-03 16:44:59	2013-01-03 16:44:59
⊗	ℹ	localhost	zenjobs	/Status/Up...	Maintenance window starting test_1630_1640 for lotschy.skills-1st.co.uk	2013-01-03 16:30:00	2013-01-03 16:30:00

Figure 87: Events for Maintenance Windows starting / stopping

10.6 Using zenactiond.log

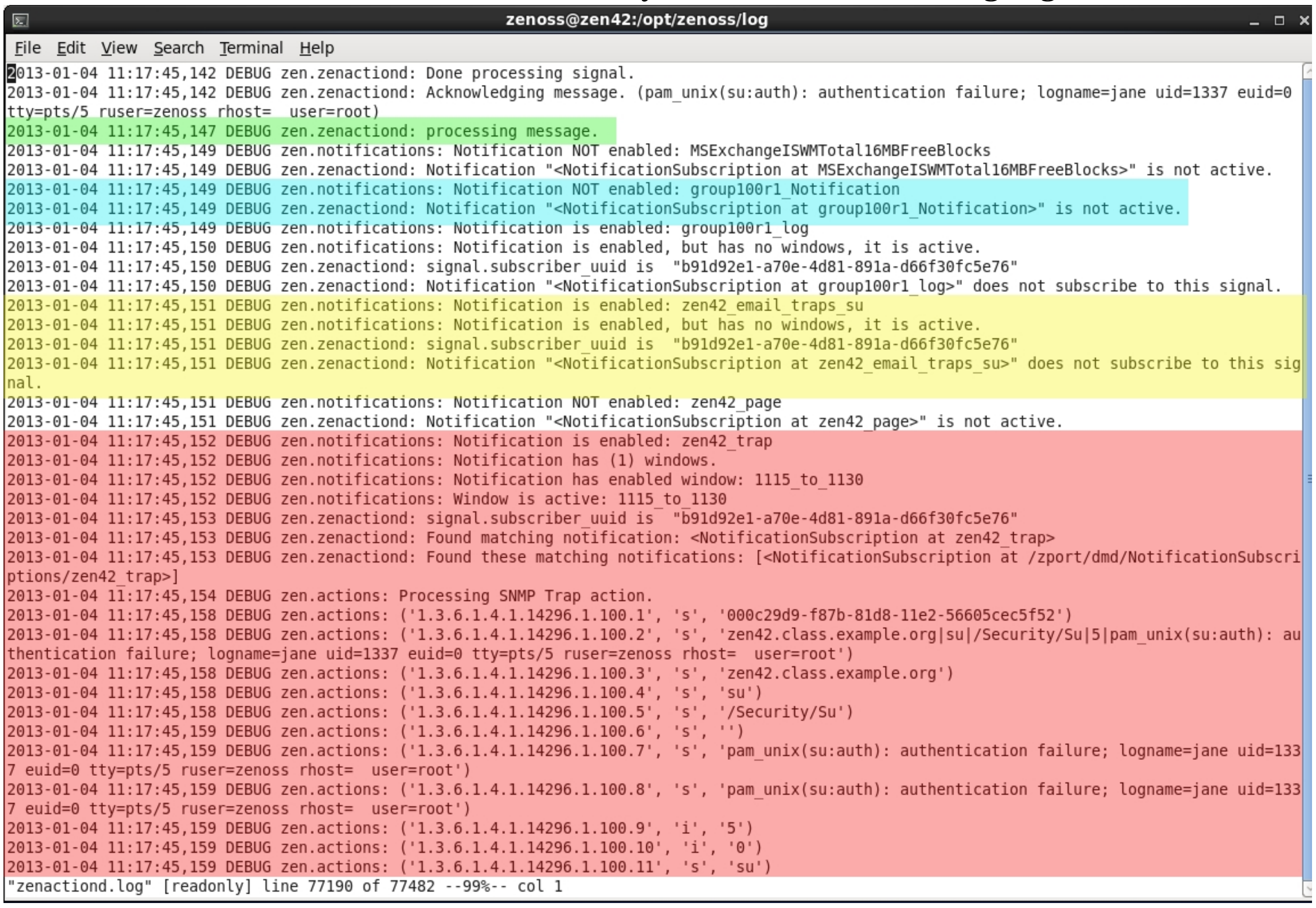
All Notifications are processed by the zenactiond daemon. To debug issues and also as a learning aid, it is helpful to set the debugging level to *Debug* (*logseverity 10*), remembering to recycle zenactiond.

Inspecting zenactiond.log provides a good insight into how zenactiond processes events from the RabbitMQ **signal** queue and then tests them against the configured Notifications.

The Triggers are processed by the zeneventserver daemon to decide what to place on the signal queue. There are obviously different signals for each Notification type.

A processing cycle starts with a *processing message* entry (highlighted in green) in Figure 88.

Notifications are checked as to whether they are enabled or not (highlighted in blue).



```
zenoss@zen42:/opt/zenoss/log
File  Edit  View  Search  Terminal  Help
2013-01-04 11:17:45,142 DEBUG zen.zenactiond: Done processing signal.
2013-01-04 11:17:45,142 DEBUG zen.zenactiond: Acknowledging message. (pam_unix(su:auth): authentication failure; logname=jane uid=1337 euid=0
tty=pts/5 ruser=zenoss rhost=  user=root)
2013-01-04 11:17:45,147 DEBUG zen.zenactiond: processing message.
2013-01-04 11:17:45,149 DEBUG zen.notifications: Notification NOT enabled: MSExchangeISWMTotal16MBFreeBlocks
2013-01-04 11:17:45,149 DEBUG zen.zenactiond: Notification "<NotificationSubscription at MSExchangeISWMTotal16MBFreeBlocks>" is not active.
2013-01-04 11:17:45,149 DEBUG zen.notifications: Notification NOT enabled: group100r1_Notification
2013-01-04 11:17:45,149 DEBUG zen.zenactiond: Notification "<NotificationSubscription at group100r1_Notification>" is not active.
2013-01-04 11:17:45,149 DEBUG zen.notifications: Notification is enabled: group100r1_log
2013-01-04 11:17:45,150 DEBUG zen.notifications: Notification is enabled, but has no windows, it is active.
2013-01-04 11:17:45,150 DEBUG zen.zenactiond: signal.subscriber_uuid is  "b91d92e1-a70e-4d81-891a-d66f30fc5e76"
2013-01-04 11:17:45,150 DEBUG zen.zenactiond: Notification "<NotificationSubscription at group100r1_log>" does not subscribe to this signal.
2013-01-04 11:17:45,151 DEBUG zen.notifications: Notification is enabled: zen42_email_traps_su
2013-01-04 11:17:45,151 DEBUG zen.notifications: Notification is enabled, but has no windows, it is active.
2013-01-04 11:17:45,151 DEBUG zen.zenactiond: signal.subscriber_uuid is  "b91d92e1-a70e-4d81-891a-d66f30fc5e76"
2013-01-04 11:17:45,151 DEBUG zen.zenactiond: Notification "<NotificationSubscription at zen42_email_traps_su>" does not subscribe to this sig
nal.
2013-01-04 11:17:45,151 DEBUG zen.notifications: Notification NOT enabled: zen42_page
2013-01-04 11:17:45,151 DEBUG zen.zenactiond: Notification "<NotificationSubscription at zen42_page>" is not active.
2013-01-04 11:17:45,152 DEBUG zen.notifications: Notification is enabled: zen42_trap
2013-01-04 11:17:45,152 DEBUG zen.notifications: Notification has (1) windows.
2013-01-04 11:17:45,152 DEBUG zen.notifications: Notification has enabled window: 1115_to_1130
2013-01-04 11:17:45,152 DEBUG zen.notifications: Window is active: 1115_to_1130
2013-01-04 11:17:45,153 DEBUG zen.zenactiond: signal.subscriber_uuid is  "b91d92e1-a70e-4d81-891a-d66f30fc5e76"
2013-01-04 11:17:45,153 DEBUG zen.zenactiond: Found matching notification: <NotificationSubscription at zen42_trap>
2013-01-04 11:17:45,153 DEBUG zen.zenactiond: Found these matching notifications: [<NotificationSubscription at /zport/dmd/NotificationSubscri
ptions/zen42_trap>]
2013-01-04 11:17:45,154 DEBUG zen.actions: Processing SNMP Trap action.
2013-01-04 11:17:45,158 DEBUG zen.actions: ('1.3.6.1.4.1.14296.1.100.1', 's', '000c29d9-f87b-81d8-11e2-56605cec5f52')
2013-01-04 11:17:45,158 DEBUG zen.actions: ('1.3.6.1.4.1.14296.1.100.2', 's', 'zen42.class.example.org|su|/Security/Su|5|pam_unix(su:auth): au
thentication failure; logname=jane uid=1337 euid=0 tty=pts/5 ruser=zenoss rhost=  user=root')
2013-01-04 11:17:45,158 DEBUG zen.actions: ('1.3.6.1.4.1.14296.1.100.3', 's', 'zen42.class.example.org')
2013-01-04 11:17:45,158 DEBUG zen.actions: ('1.3.6.1.4.1.14296.1.100.4', 's', 'su')
2013-01-04 11:17:45,158 DEBUG zen.actions: ('1.3.6.1.4.1.14296.1.100.5', 's', '/Security/Su')
2013-01-04 11:17:45,159 DEBUG zen.actions: ('1.3.6.1.4.1.14296.1.100.6', 's', '')
2013-01-04 11:17:45,159 DEBUG zen.actions: ('1.3.6.1.4.1.14296.1.100.7', 's', 'pam_unix(su:auth): authentication failure; logname=jane uid=133
7 euid=0 tty=pts/5 ruser=zenoss rhost=  user=root')
2013-01-04 11:17:45,159 DEBUG zen.actions: ('1.3.6.1.4.1.14296.1.100.8', 's', 'pam_unix(su:auth): authentication failure; logname=jane uid=133
7 euid=0 tty=pts/5 ruser=zenoss rhost=  user=root')
2013-01-04 11:17:45,159 DEBUG zen.actions: ('1.3.6.1.4.1.14296.1.100.9', 'i', '5')
2013-01-04 11:17:45,159 DEBUG zen.actions: ('1.3.6.1.4.1.14296.1.100.10', 'i', '0')
2013-01-04 11:17:45,159 DEBUG zen.actions: ('1.3.6.1.4.1.14296.1.100.11', 's', 'su')
"zenactiond.log" [readonly] line 77190 of 77482 --99%-- col 1
```

Figure 88: zenactiond.log processing a signal against various Notifications

The event that generated this signal was a /Security/Su event and should trigger both the *zen42_email_traps_su* Notification and the *zen42_trap* Notification. In Figure 88 the log shows zen42_email_traps_su being discarded (highlighted in yellow); this is because the signal message is keyed to a TRAP Notification type, not an email one (unfortunately zenactiond.log does not show this detail).

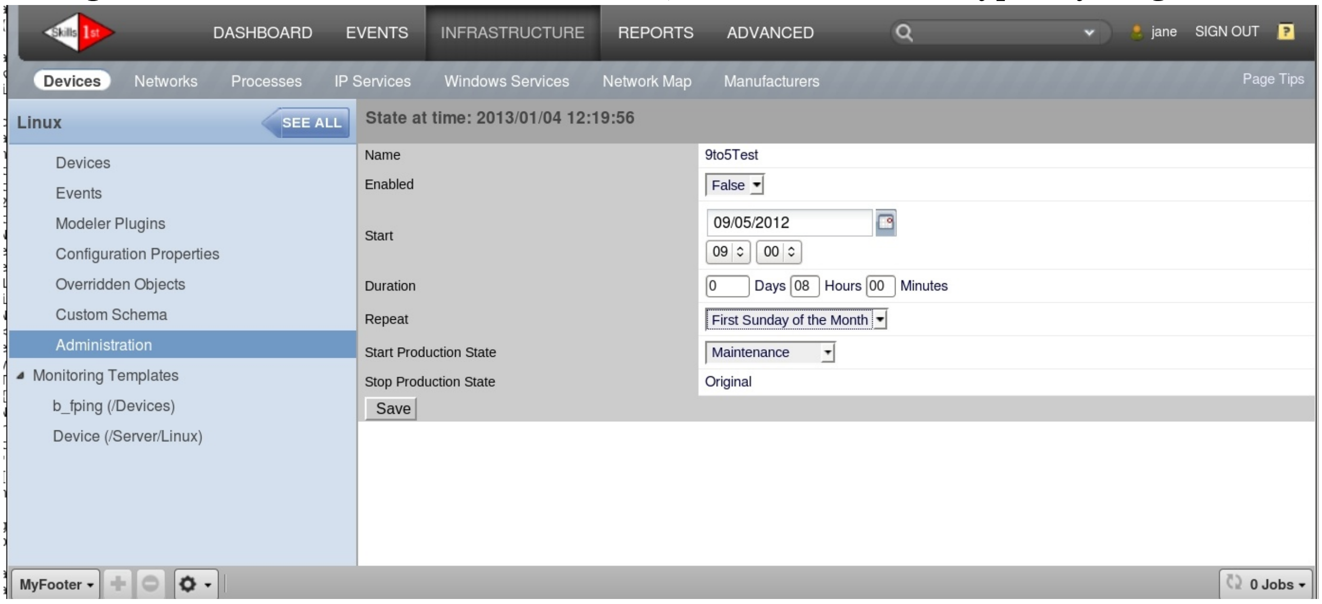
The match with *zen42_trap* is highlighted in red where the checking for a notification schedule window can also be seen. The start of the notification action to generate the TRAP is also highlighted.

Once the action is completed, zenactiond.log shows similar iterations through the Notifications list with a separate signal message, where the *zen42_email_traps_su* Notification is selected and actioned and the *zen42_trap* Notification is discarded.

10.7 The effect of device Production State

The **Production State** of a device can be used to control different management aspects of a system. Production State for a device is configured on the device's home page *Overview* and may be modified by Maintenance Windows configured for a device, device class, Group, System or Location.

When configuring a Maintenance Window, the production state is defined both for during the window and the state to return to, where the latter is typically *Original*.



State at time: 2013/01/04 12:19:56

Name	9to5Test
Enabled	False
Start	09/05/2012 09 00
Duration	0 Days 08 Hours 00 Minutes
Repeat	First Sunday of the Month
Start Production State	Maintenance
Stop Production State	Original

Save

Figure 89: Maintenance Window for device class /Server/Linux for first Sunday in the month

Chapter 8 of the Zenoss Core 4 Administration Guide describes the different Production States and the effect that these have. Three different types of "management" are defined:

● Monitoring	ping polling and event generation
● Alerting	generating alerts (emails, pagers, commands, traps)
● Dashboard	whether to include in the *Device Issues* portlet

In practise, anything to do with Notifications is controlled by the filters in the Trigger. If no *Production State* filter is configured then the Notification **will** run, by default.

A device Production State of *Production* will result in events contributing to the *Device Issues* portlet of the Zenoss Dashboard and all monitoring will take place.

A Production State of *Decommissioned* should result in all monitoring ceasing; hence, all events generated by Zenoss will cease and no related Notifications will be generated; however, externally generated events (from syslog, external TRAPs, Windows event logs) **will** continue to be received and related Notifications **will** be generated unless a trigger filter excluding on Production State exists. The device will not be recorded in the Dashboard *Device Issues* portlet. **Note** that the overall Status icon on a device's Status page will turn **green** !

Any Production State **other** than *Production* will result in the device **not** being included on the Dashboard *Device Issues* portlet.

The only Production State that automatically stops all monitoring is **Decommissioned**; however, the zProperty of **zProdStateThreshold** can be set as part of the *Configuration Properties* of a device or device class. This variable controls the Production State value beneath which all monitoring ceases. By default this value is *300* which means that setting a Production State of *Maintenance* does **not** prevent ping and snmp monitoring. If you want to prevent all monitoring for Maintenance state devices, change the *zProdStateThreshold* value at the top device class level to *301*.

11 Accessing events with the JSON API

During the life of Zenoss 3, the JSON API was introduced as a means of accessing data within Zenoss. In some ways, it is similar to using the zendmd Python environment and in many cases it reflects the same calls available in zendmd, but a great advantage of the JSON API is that it can be used remotely from the Zenoss server and it requires no intimate knowledge of Python.

11.1 Definitions

For those who are not from a development background (and possibly with apologies to those who are), here are some definitions.

An Application Programming Interface (**API**) is a way of accessing "stuff".

"Stuff" in the context of Zenoss means objects that represent real things. For example, Python objects that represent devices, network interfaces, filesystems, processes and users; database objects in the MySQL database that represent events.

JavaScript Object Notation (**JSON**) is a lightweight data-interchange format. It is easy for humans to read and write being a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others.

Thus the JSON API provides a documented way of accessing different sorts of data within Zenoss, using a common interface. Whatever "stuff" is being accessed, we present requests in a text format and the results are translated back into text format for us.

In order to present our requests for data, a URL is required plus a userid and password that has authority to access the Zenoss data requested. As users, we can construct requests in exactly the same way as the Zenoss GUI does; the Zenoss GUI itself uses the JSON API to present data to us.

Another benefit of using the JSON API rather than using Python directly, is that Zenoss Development may change the underlying Python in the Zenoss Core code but, provided they maintain the JSON API interface, any access functionality built on top of the API can remain unchanged. For this reason there is a recommendation that the API be used in preference to writing Python code to access data directly.

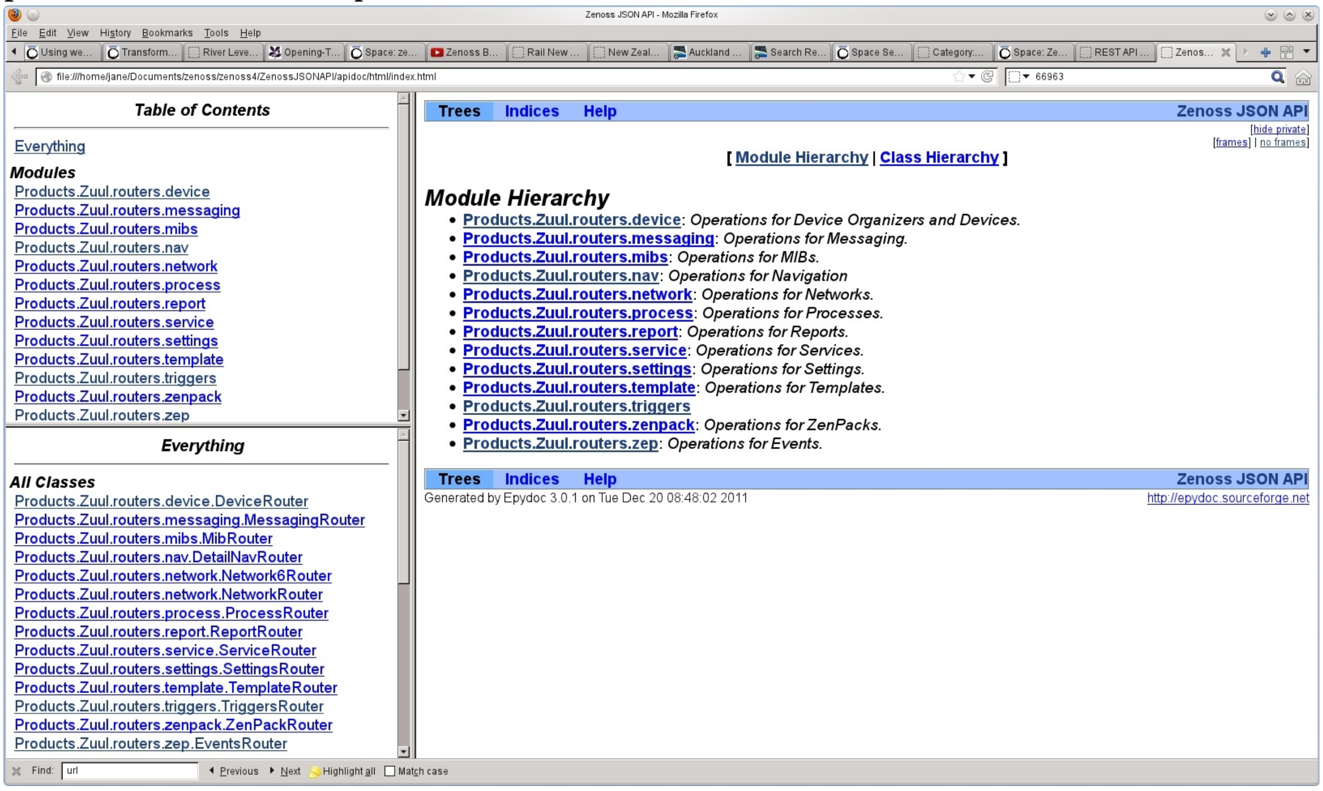
11.2 Understanding the JSON API

The JSON API is shipped as standard with Zenoss Core. The documentation can be found at http://community.zenoss.org/community/documentation/official_documentation/api ; this is actually a zipped bundle containing documentation in html format, a pdf guide and both Python and Java samples for using the API.

There are also some samples of using the JSON API with bash and curl at https://gist.github.com/1901884 .

The JSON API exposes the methods that can be found in the Zenoss code under *$ZENHOME/Products/Zuul/routers*.

The easiest way to view the documentation is to download the zip bundle, unzip it and point a browser at the *apidoc/html/index.html* file.



Zenoss JSON API - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///home/jane/Documents/zenoss/zenoss4/ZenossJSONAPI/apidoc/html/index.html
Find: url Previous Next Highlight all Match case

Table of Contents

Everything

Modules

Products.Zuul.routers.device
Products.Zuul.routers.messaging
Products.Zuul.routers.mibs
Products.Zuul.routers.nav
Products.Zuul.routers.network
Products.Zuul.routers.process
Products.Zuul.routers.report
Products.Zuul.routers.service
Products.Zuul.routers.settings
Products.Zuul.routers.template
Products.Zuul.routers.triggers
Products.Zuul.routers.zenpack
Products.Zuul.routers.zep

Everything

All Classes

Products.Zuul.routers.device.DeviceRouter
Products.Zuul.routers.messaging.MessagingRouter
Products.Zuul.routers.mibs.MibRouter
Products.Zuul.routers.nav.DetailNavRouter
Products.Zuul.routers.network.Network6Router
Products.Zuul.routers.network.NetworkRouter
Products.Zuul.routers.process.ProcessRouter
Products.Zuul.routers.report.ReportRouter
Products.Zuul.routers.service.ServiceRouter
Products.Zuul.routers.settings.SettingsRouter
Products.Zuul.routers.template.TemplateRouter
Products.Zuul.routers.triggers.TriggersRouter
Products.Zuul.routers.zenpack.ZenPackRouter
Products.Zuul.routers.zep.EventsRouter

Trees Indices Help Zenoss JSON API

[hide private]
[frames] | no frames]

[Module Hierarchy | Class Hierarchy]

Module Hierarchy

- **Products.Zuul.routers.device**: *Operations for Device Organizers and Devices.*
- **Products.Zuul.routers.messaging**: *Operations for Messaging.*
- **Products.Zuul.routers.mibs**: *Operations for MIBs.*
- **Products.Zuul.routers.nav**: *Operations for Navigation*
- **Products.Zuul.routers.network**: *Operations for Networks.*
- **Products.Zuul.routers.process**: *Operations for Processes.*
- **Products.Zuul.routers.report**: *Operations for Reports.*
- **Products.Zuul.routers.service**: *Operations for Services.*
- **Products.Zuul.routers.settings**: *Operations for Settings.*
- **Products.Zuul.routers.template**: *Operations for Templates.*
- **Products.Zuul.routers.triggers**
- **Products.Zuul.routers.zenpack**: *Operations for ZenPacks.*
- **Products.Zuul.routers.zep**: *Operations for Events.*

Trees Indices Help Zenoss JSON API

Generated by Epydoc 3.0.1 on Tue Dec 20 08:48:02 2011 http://epydoc.sourceforge.net

Figure 90: JSON API documentation in html format

The lefthand menus show the modules, effectively the files that can be found under *$ZENHOME/Zuul/Products/routers*. Typically these files each define one class though the network file has a class for each of *NetworkRouter* and *Network6Router*.

Click on a module to see an overview of what it contains. Note the *Available at* line that helps indicate the url that reaches this data.

Click on the link to the Class , *EventsRouter*, to see all the methods for this class.

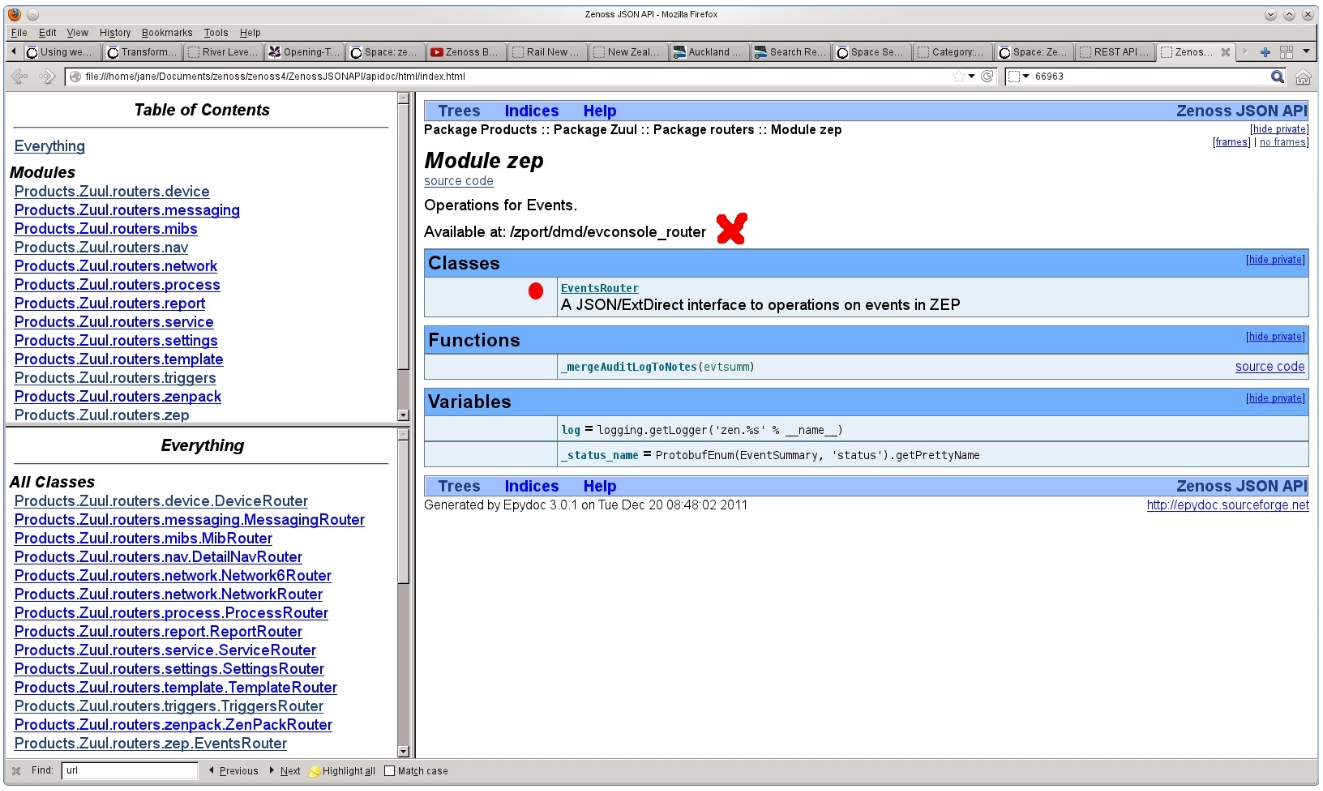


Table of Contents

Everything

Modules

- Products.Zuul.routers.device
- Products.Zuul.routers.messaging
- Products.Zuul.routers.mibs
- Products.Zuul.routers.nav
- Products.Zuul.routers.network
- Products.Zuul.routers.process
- Products.Zuul.routers.report
- Products.Zuul.routers.service
- Products.Zuul.routers.settings
- Products.Zuul.routers.template
- Products.Zuul.routers.triggers
- Products.Zuul.routers.zenpack
- Products.Zuul.routers.zep

Everything

All Classes

- Products.Zuul.routers.device.DeviceRouter
- Products.Zuul.routers.messaging.MessagingRouter
- Products.Zuul.routers.mibs.MibRouter
- Products.Zuul.routers.nav.DetailNavRouter
- Products.Zuul.routers.network.Network6Router
- Products.Zuul.routers.network.NetworkRouter
- Products.Zuul.routers.process.ProcessRouter
- Products.Zuul.routers.report.ReportRouter
- Products.Zuul.routers.service.ServiceRouter
- Products.Zuul.routers.settings.SettingsRouter
- Products.Zuul.routers.template.TemplateRouter
- Products.Zuul.routers.triggers.TriggersRouter
- Products.Zuul.routers.zenpack.ZenPackRouter
- Products.Zuul.routers.zep.EventsRouter

Trees Indices Help — Zenoss JSON API

Package Products :: Package Zuul :: Package routers :: Module zep

Module zep

source code

Operations for Events.

Available at: /zport/dmd/evconsole_router

Classes

●	EventsRouter A JSON/ExtDirect interface to operations on events in ZEP

Functions

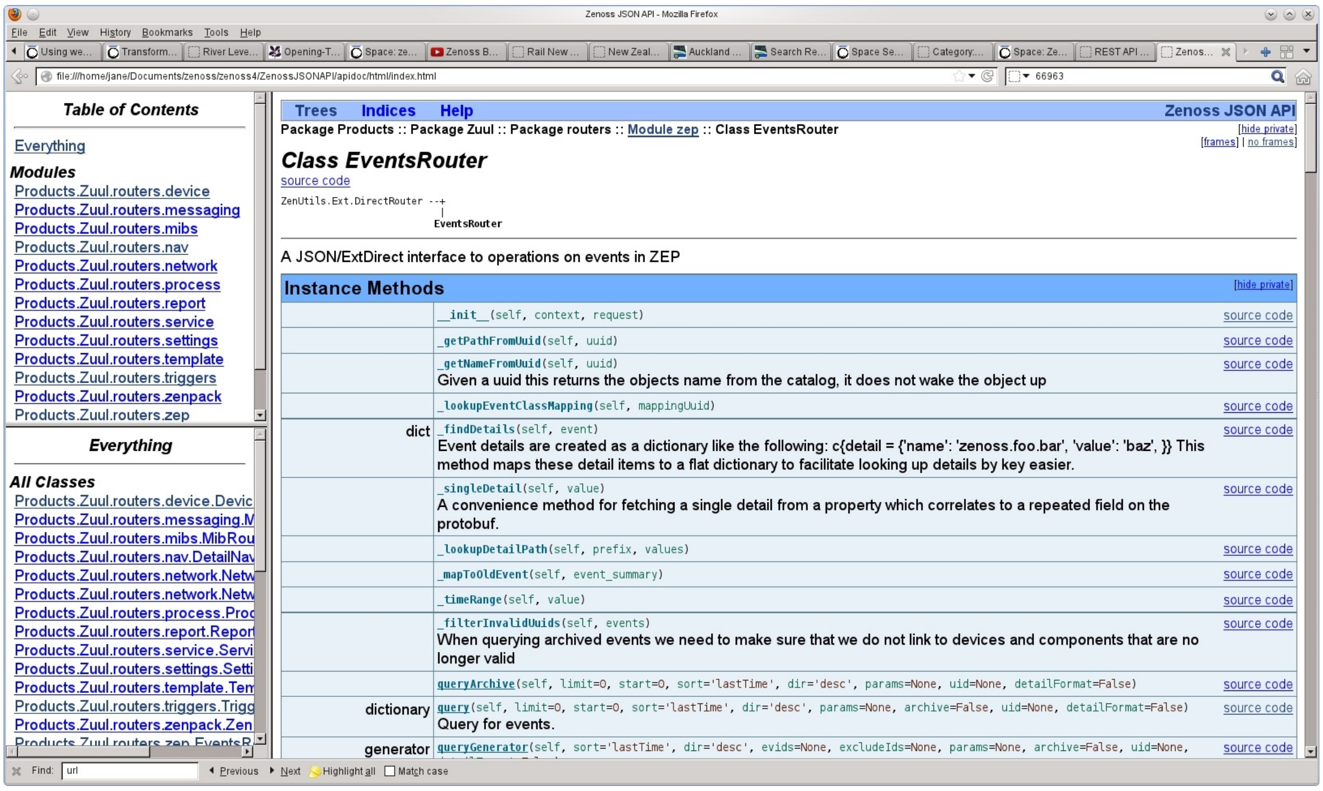
	`_mergeAuditLogToNotes(evtsumm)`	source code

Variables

	`log = logging.getLogger('zen.%s' % __name__)`
	`_status_name = ProtobufEnum(EventSummary, 'status').getPrettyName`

Generated by Epydoc 3.0.1 on Tue Dec 20 08:48:02 2011 — http://epydoc.sourceforge.net

Figure 91: JSON API - details of the zep module



Package Products :: Package Zuul :: Package routers :: Module zep :: Class EventsRouter

Class EventsRouter

source code

```
ZenUtils.Ext.DirectRouter --+
                            |
                           EventsRouter
```

A JSON/ExtDirect interface to operations on events in ZEP

Instance Methods

	`__init__(self, context, request)`	source code
	`_getPathFromUuid(self, uuid)`	source code
	`_getNameFromUuid(self, uuid)` Given a uuid this returns the objects name from the catalog, it does not wake the object up	source code
	`_lookupEventClassMapping(self, mappingUuid)`	source code
dict	`_findDetails(self, event)` Event details are created as a dictionary like the following: c{detail = {'name': 'zenoss.foo.bar', 'value': 'baz', }} This method maps these detail items to a flat dictionary to facilitate looking up details by key easier.	source code
	`_singleDetail(self, value)` A convenience method for fetching a single detail from a property which correlates to a repeated field on the protobuf.	source code
	`_lookupDetailPath(self, prefix, values)`	source code
	`_mapToOldEvent(self, event_summary)`	source code
	`_timeRange(self, value)`	source code
	`_filterInvalidUuids(self, events)` When querying archived events we need to make sure that we do not link to devices and components that are no longer valid	source code
	`queryArchive(self, limit=0, start=0, sort='lastTime', dir='desc', params=None, uid=None, detailFormat=False)`	source code
dictionary	`query(self, limit=0, start=0, sort='lastTime', dir='desc', params=None, archive=False, uid=None, detailFormat=False)` Query for events.	source code
generator	`queryGenerator(self, sort='lastTime', dir='desc', evids=None, excludeIds=None, params=None, archive=False, uid=None,`	source code

Figure 92: JSON API - methods for the EventsRouter class

Click on a method to get a more detailed overview with descriptions of the input parameters and the values returned.

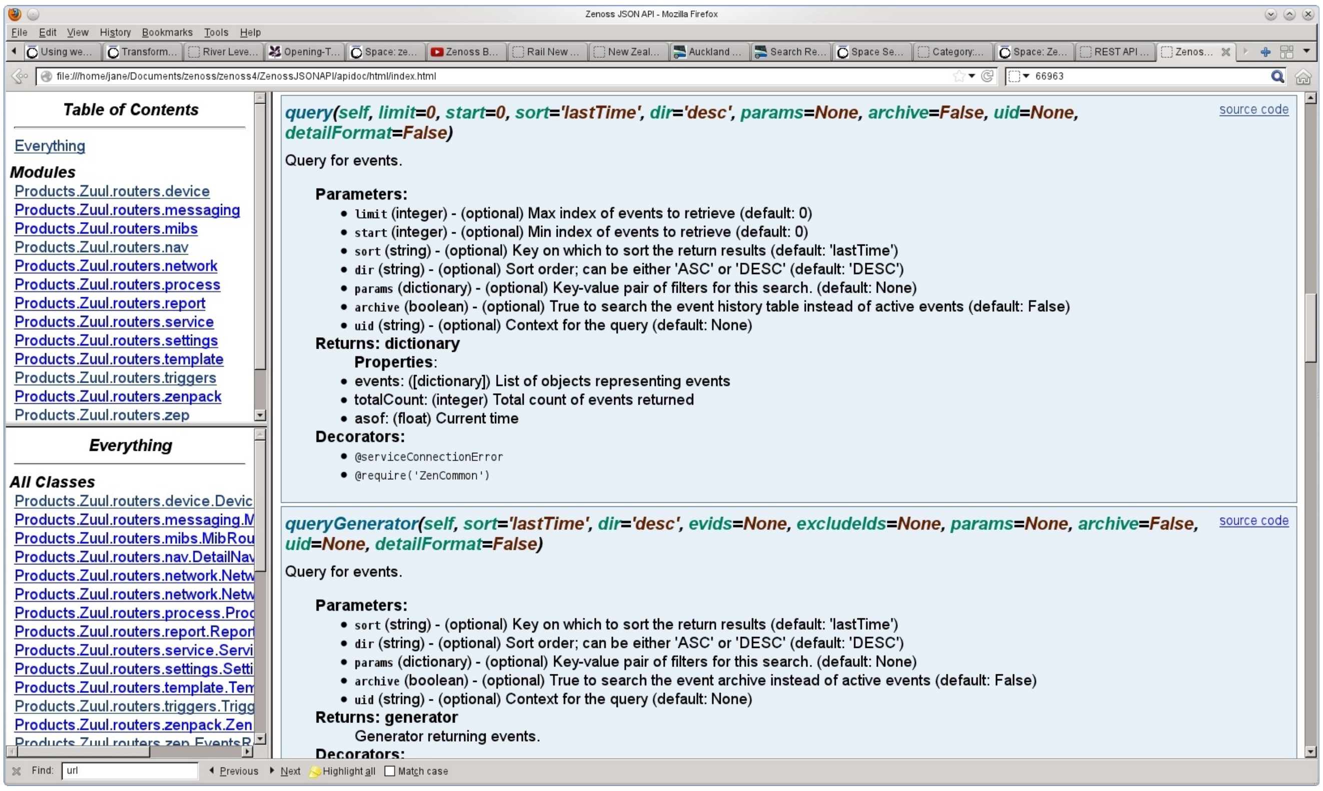


Table of Contents

Everything

Modules

- Products.Zuul.routers.device
- Products.Zuul.routers.messaging
- Products.Zuul.routers.mibs
- Products.Zuul.routers.nav
- Products.Zuul.routers.network
- Products.Zuul.routers.process
- Products.Zuul.routers.report
- Products.Zuul.routers.service
- Products.Zuul.routers.settings
- Products.Zuul.routers.template
- Products.Zuul.routers.triggers
- Products.Zuul.routers.zenpack
- Products.Zuul.routers.zep

***query**(self, limit=0, start=0, sort='lastTime', dir='desc', params=None, archive=False, uid=None, detailFormat=False)*

Query for events.

Parameters:
- `limit` (integer) - (optional) Max index of events to retrieve (default: 0)
- `start` (integer) - (optional) Min index of events to retrieve (default: 0)
- `sort` (string) - (optional) Key on which to sort the return results (default: 'lastTime')
- `dir` (string) - (optional) Sort order; can be either 'ASC' or 'DESC' (default: 'DESC')
- `params` (dictionary) - (optional) Key-value pair of filters for this search. (default: None)
- `archive` (boolean) - (optional) True to search the event history table instead of active events (default: False)
- `uid` (string) - (optional) Context for the query (default: None)

Returns: dictionary

Properties:
- events: ([dictionary]) List of objects representing events
- totalCount: (integer) Total count of events returned
- asof: (float) Current time

Decorators:
- `@serviceConnectionError`
- `@require('ZenCommon')`

***queryGenerator**(self, sort='lastTime', dir='desc', evids=None, excludeIds=None, params=None, archive=False, uid=None, detailFormat=False)*

Query for events.

Parameters:
- `sort` (string) - (optional) Key on which to sort the return results (default: 'lastTime')
- `dir` (string) - (optional) Sort order; can be either 'ASC' or 'DESC' (default: 'DESC')
- `params` (dictionary) - (optional) Key-value pair of filters for this search. (default: None)
- `archive` (boolean) - (optional) True to search the event archive instead of active events (default: False)
- `uid` (string) - (optional) Context for the query (default: None)

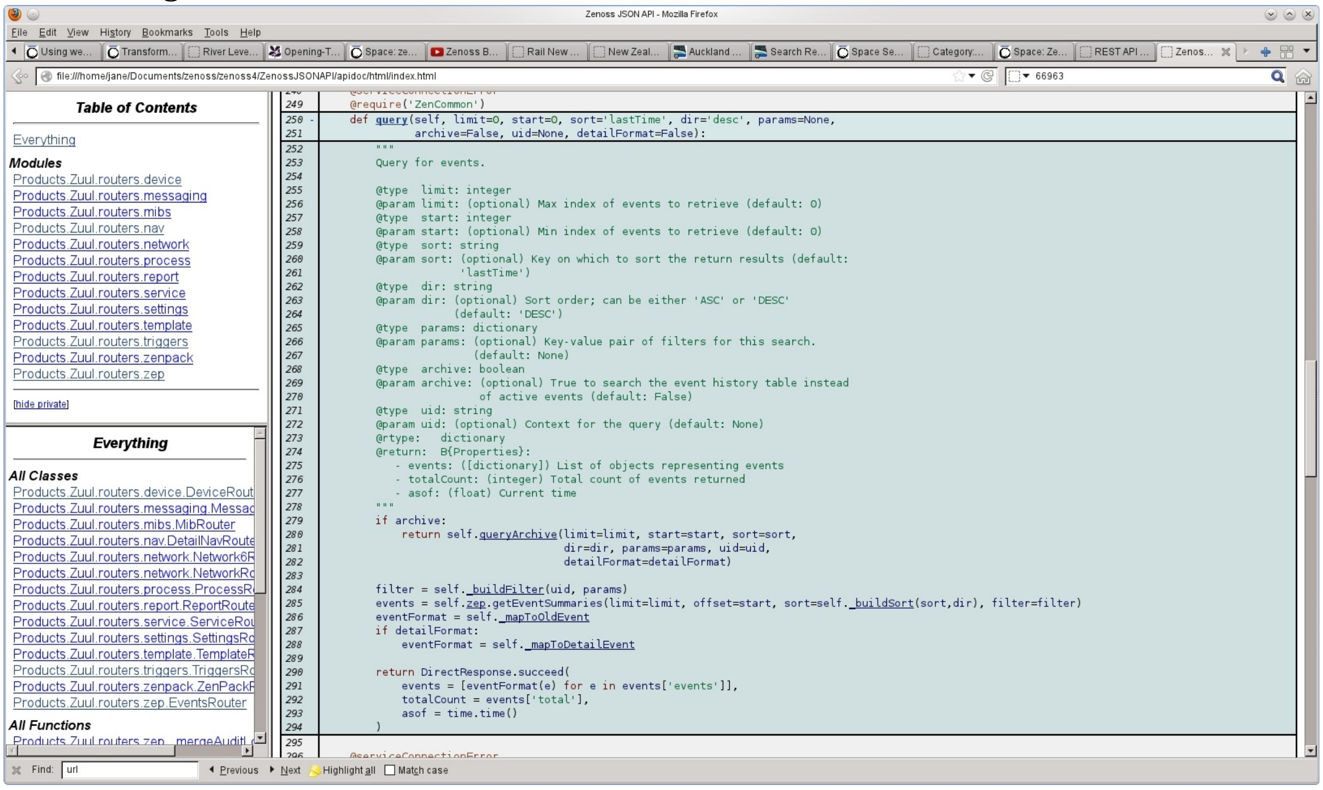
Returns: generator

Generator returning events.

Decorators:

Figure 93: JSON API - details for the query method in the EventsRouter class

At all levels of the documentation there are links to the *source code*. This should be very close to the code that you see if you inspect the file *$ZENHOME/Products/Zuul/routers* though the line numbers may not match exactly depending on the exact level of code you are running.



```
    @require('ZenCommon')
    def query(self, limit=0, start=0, sort='lastTime', dir='desc', params=None,
              archive=False, uid=None, detailFormat=False):
        """
        Query for events.

        @type  limit: integer
        @param limit: (optional) Max index of events to retrieve (default: 0)
        @type  start: integer
        @param start: (optional) Min index of events to retrieve (default: 0)
        @type  sort: string
        @param sort: (optional) Key on which to sort the return results (default:
                     'lastTime')
        @type  dir: string
        @param dir: (optional) Sort order; can be either 'ASC' or 'DESC'
                    (default: 'DESC')
        @type  params: dictionary
        @param params: (optional) Key-value pair of filters for this search.
                       (default: None)
        @type  archive: boolean
        @param archive: (optional) True to search the event history table instead
                        of active events (default: False)
        @type  uid: string
        @param uid: (optional) Context for the query (default: None)
        @rtype:   dictionary
        @return:  B{Properties}:
           - events: ([dictionary]) List of objects representing events
           - totalCount: (integer) Total count of events returned
           - asof: (float) Current time
        """
        if archive:
            return self.queryArchive(limit=limit, start=start, sort=sort,
                                     dir=dir, params=params, uid=uid,
                                     detailFormat=detailFormat)

        filter = self._buildFilter(uid, params)
        events = self.zep.getEventSummaries(limit=limit, offset=start, sort=self._buildSort(sort,dir), filter=filter)
        eventFormat = self._mapToOldEvent
        if detailFormat:
            eventFormat = self._mapToDetailEvent

        return DirectResponse.succeed(
            events = [eventFormat(e) for e in events['events']],
            totalCount = events['total'],
            asof = time.time()
        )

    @serviceConnectionError
```

Figure 94: JSON API - source code for the query method

If you inspect the *__init__* method source code for the *EventsRouter* class, you can see that the *zep* attribute is set to:

```
        self.zep = Zuul.getFacade('zep', context)
```

Each of the files in *$ZENHOME/Products/Zuul/routers* has methods that call the matching facade found under *$ZENHOME/Products/Zuul/facades*.

Think of the **routers** as a way to reach the right basic area of data - device, mibs, triggers, zep - with some top-level methods like query, _buildFilter; and think of the **facades** as more detailed access methods; so, having gained access to the events through the zep router, the facade provides createEventFilter, getEventSummaries, acknowledgeEventSummaries, and so on.

11.3 Using the JSON API

The documentation bundle includes sample code for using the JSON API from Python programs and Java programs. Further samples are available at https://gist.github.com/1901884/ that demonstrate a bash shell harness for driving the API using the *curl* utility.

Note that the Python samples both require slight bug-fixes to device.py and zep.py respectively in *$ZENHOME/Products/Zuul/routers* for the base Zenoss Core 4.2 code - see a discussion and solutions on the Zenoss User's forum at http://community.zenoss.org/message/70052#70052 . These issues appear to be fixed with Core 4.2.3.

11.3.1 Bash examples

Get the bash examples from https://gist.github.com/1901884/ (use the Download Gist link) and unpack the bundle to get *zenoss_curlExamples.sh*. Edit this file to reflect your Zenoss server parameters, if required, though the code already has a default server of *localhost*, port *8080*, user of *admin* and password of *zenoss* so it will probably work as-is if you have not changed install defaults.

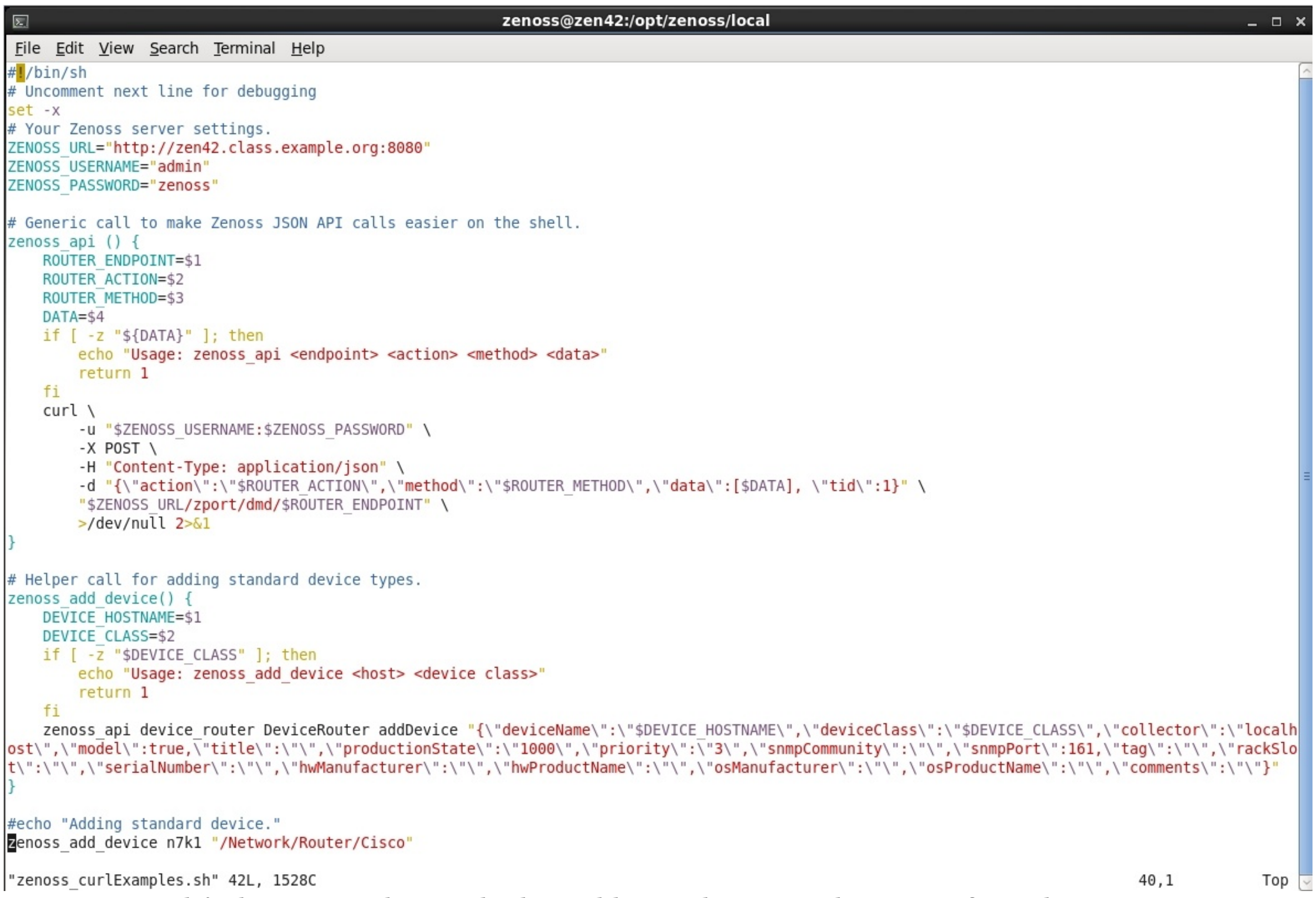
All the code to do with **services** refers to the enterprise Zenoss Resource Manager chargeable product so they can be removed. To cut the file down to a basic sample that just adds a device, n7k1, to the /Network/Router/Cisco device class, also remove the helper functions for UCS and VCS objects so that you end up with a shellscript as shown in Figure 95. Note that the device class has also been changed from the original script as the class must exist.

The single remaining body line of the script is:

```
        zenoss_add_device n7k1 "/Network/Router/Cisco"
```

calling the helper function:

```
        zenoss_add_device()
```



```
#!/bin/sh
# Uncomment next line for debugging
set -x
# Your Zenoss server settings.
ZENOSS_URL="http://zen42.class.example.org:8080"
ZENOSS_USERNAME="admin"
ZENOSS_PASSWORD="zenoss"

# Generic call to make Zenoss JSON API calls easier on the shell.
zenoss_api () {
    ROUTER_ENDPOINT=$1
    ROUTER_ACTION=$2
    ROUTER_METHOD=$3
    DATA=$4
    if [ -z "${DATA}" ]; then
        echo "Usage: zenoss_api <endpoint> <action> <method> <data>"
        return 1
    fi
    curl \
        -u "$ZENOSS_USERNAME:$ZENOSS_PASSWORD" \
        -X POST \
        -H "Content-Type: application/json" \
        -d "{\"action\":\"$ROUTER_ACTION\",\"method\":\"$ROUTER_METHOD\",\"data\":[$DATA], \"tid\":1}" \
        "$ZENOSS_URL/zport/dmd/$ROUTER_ENDPOINT" \
        >/dev/null 2>&1
}

# Helper call for adding standard device types.
zenoss_add_device() {
    DEVICE_HOSTNAME=$1
    DEVICE_CLASS=$2
    if [ -z "$DEVICE_CLASS" ]; then
        echo "Usage: zenoss_add_device <host> <device class>"
        return 1
    fi
    zenoss_api device_router DeviceRouter addDevice "{\"deviceName\":\"$DEVICE_HOSTNAME\",\"deviceClass\":\"$DEVICE_CLASS\",\"collector\":\"localh
ost\",\"model\":true,\"title\":\"\",\"productionState\":\"1000\",\"priority\":\"3\",\"snmpCommunity\":\"\",\"snmpPort\":161,\"tag\":\"\",\"rackSlo
t\":\"\",\"serialNumber\":\"\",\"hwManufacturer\":\"\",\"hwProductName\":\"\",\"osManufacturer\":\"\",\"osProductName\":\"\",\"comments\":\"\"}"
}

#echo "Adding standard device."
zenoss_add_device n7k1 "/Network/Router/Cisco"

"zenoss_curlExamples.sh" 42L, 1528C                                     40,1          Top
```

Figure 95: Modified zenoss_curlExample.sh to add a single /Network/Router/Cisco device

This function takes 2 parameters where $1 is the hostname and $2 is the device class. It then calls the *zenoss_api* function:

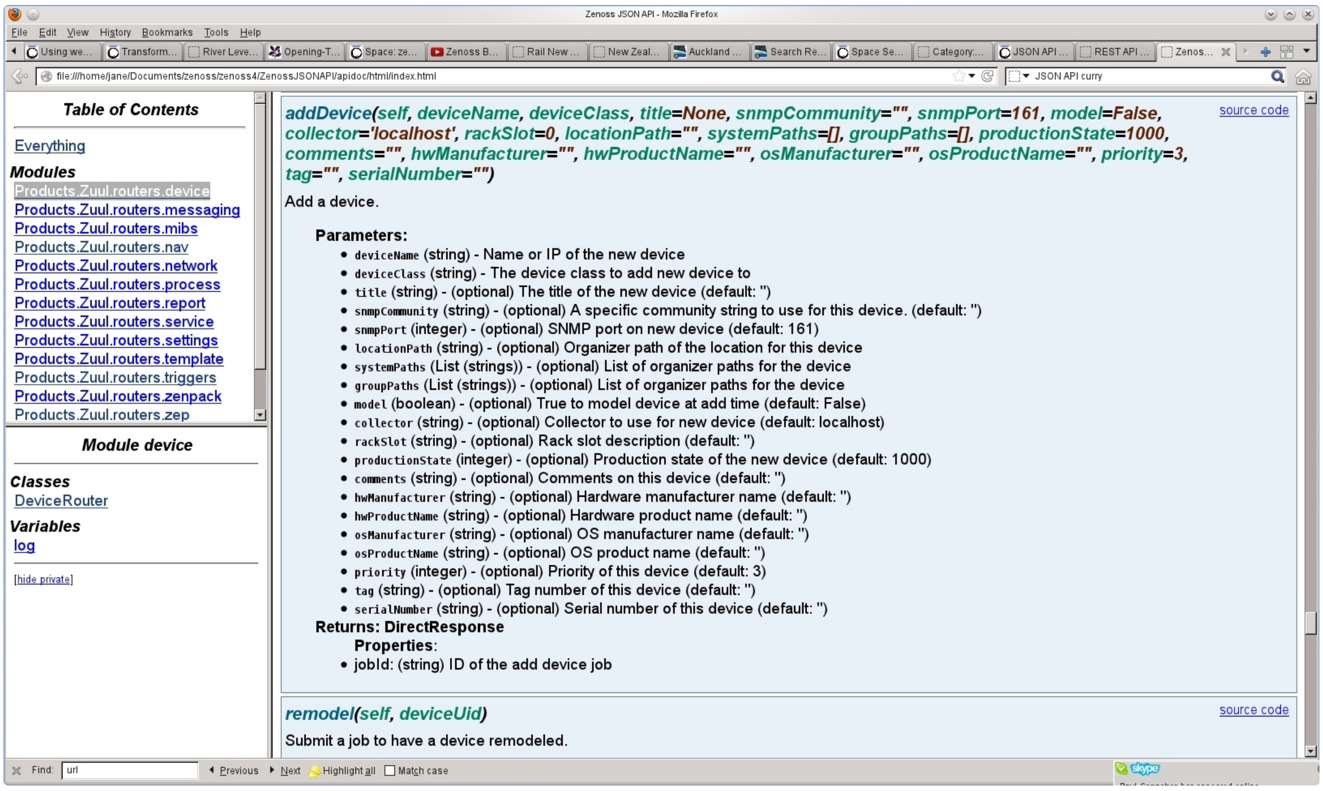
```
zenoss_api device_router DeviceRouter addDevice
"{\"deviceName\":\"$DEVICE_HOSTNAME\",\"deviceClass\":\"$DEVICE_CLASS\",\"c
ollector\":\"localhost\",\"model\":true,\"title\":\"\",\"productionState\":
\"1000\",\"priority\":\"3\",\"snmpCommunity\":\"\",\"snmpPort\":161,\"tag\"
:\"\",\"rackSlot\":\"\",\"serialNumber\":\"\",\"hwManufacturer\":\"\",\"hwP
roductName\":\"\",\"osManufacturer\":\"\",\"osProductName\":\"\",\"comments
\":\"\"}"
```

zenoss_api requires four parameters:

```
zenoss_api () {
    ROUTER_ENDPOINT=$1
    ROUTER_ACTION=$2
    ROUTER_METHOD=$3
    DATA=$4
```

where the *ROUTER_ENDPOINT* value of ***device_router*** is found from the JSON API documentation by looking at the *Available at: /zport/dmd/**device_router*** line for the module *Products.Zuul.routers.device* . The *ROUTER_ACTION* is ***DeviceRouter*** - the Class shown in the documentation; the *ROUTER_METHOD* is ***addDevice*** - the method found by exploring the DeviceRouter class; and the DATA parameter contains

<parameter name>:<parameter value> string pairs, comma-separated, with double quotes carefully escaped by backslashes.


Zenoss JSON API - Mozilla Firefox
file:///home/jane/Documents/zenoss/zenoss4/ZenossJSONAPI/apidoc/html/index.html
Table of Contents
Everything
Modules
Products.Zuul.routers.device
Products.Zuul.routers.messaging
Products.Zuul.routers.mibs
Products.Zuul.routers.nav
Products.Zuul.routers.network
Products.Zuul.routers.process
Products.Zuul.routers.report
Products.Zuul.routers.service
Products.Zuul.routers.settings
Products.Zuul.routers.template
Products.Zuul.routers.triggers
Products.Zuul.routers.zenpack
Products.Zuul.routers.zep
Module device
Classes
DeviceRouter
Variables
log
[hide private]
source code

addDevice(self, deviceName, deviceClass, title=None, snmpCommunity="", snmpPort=161, model=False, collector='localhost', rackSlot=0, locationPath="", systemPaths=[], groupPaths=[], productionState=1000, comments="", hwManufacturer="", hwProductName="", osManufacturer="", osProductName="", priority=3, tag="", serialNumber="")

Add a device.

Parameters:

- `deviceName` (string) - Name or IP of the new device
- `deviceClass` (string) - The device class to add new device to
- `title` (string) - (optional) The title of the new device (default: '')
- `snmpCommunity` (string) - (optional) A specific community string to use for this device. (default: '')
- `snmpPort` (integer) - (optional) SNMP port on new device (default: 161)
- `locationPath` (string) - (optional) Organizer path of the location for this device
- `systemPaths` (List (strings)) - (optional) List of organizer paths for the device
- `groupPaths` (List (strings)) - (optional) List of organizer paths for the device
- `model` (boolean) - (optional) True to model device at add time (default: False)
- `collector` (string) - (optional) Collector to use for new device (default: localhost)
- `rackSlot` (string) - (optional) Rack slot description (default: '')
- `productionState` (integer) - (optional) Production state of the new device (default: 1000)
- `comments` (string) - (optional) Comments on this device (default: '')
- `hwManufacturer` (string) - (optional) Hardware manufacturer name (default: '')
- `hwProductName` (string) - (optional) Hardware product name (default: '')
- `osManufacturer` (string) - (optional) OS manufacturer name (default: '')
- `osProductName` (string) - (optional) OS product name (default: '')
- `priority` (integer) - (optional) Priority of this device (default: 3)
- `tag` (string) - (optional) Tag number of this device (default: '')
- `serialNumber` (string) - (optional) Serial number of this device (default: '')

Returns: DirectResponse

Properties:

- jobId: (string) ID of the add device job

remodel(self, deviceUid)

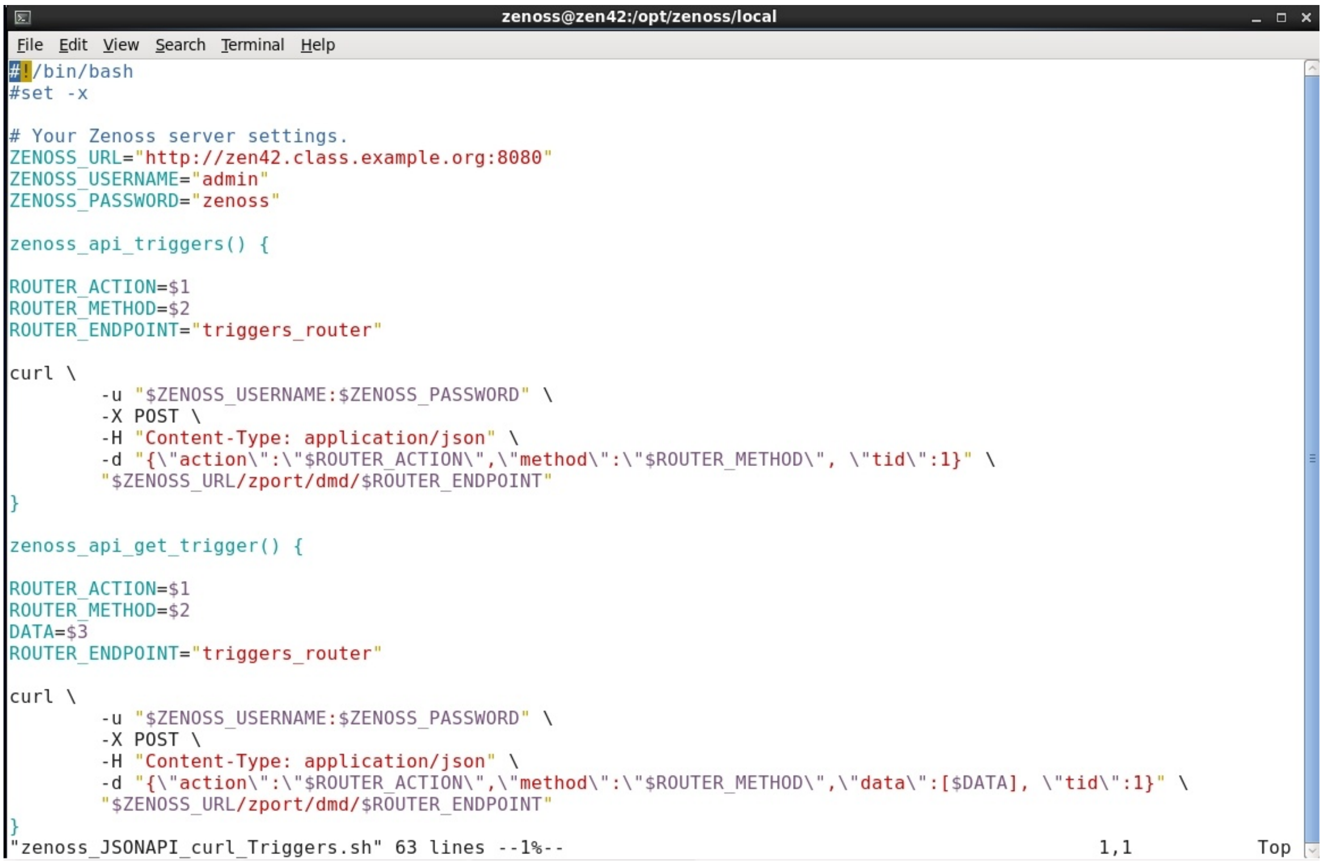
Submit a job to have a device remodeled.

Figure 96: addDevice method for the DeviceRouter class detailing input parameters

Ensure that the shellscript is executable and run it. Check that the device is added. The *set -x* line at the top of the script can be uncommented to provide debugging.

Here is a second example that explores the capabilities of the **triggers** interface.

Exploring the *triggers* module with the API documentation shows that some methods need a data parameter and some don't. This is why there are two helper functions in Figure 97.

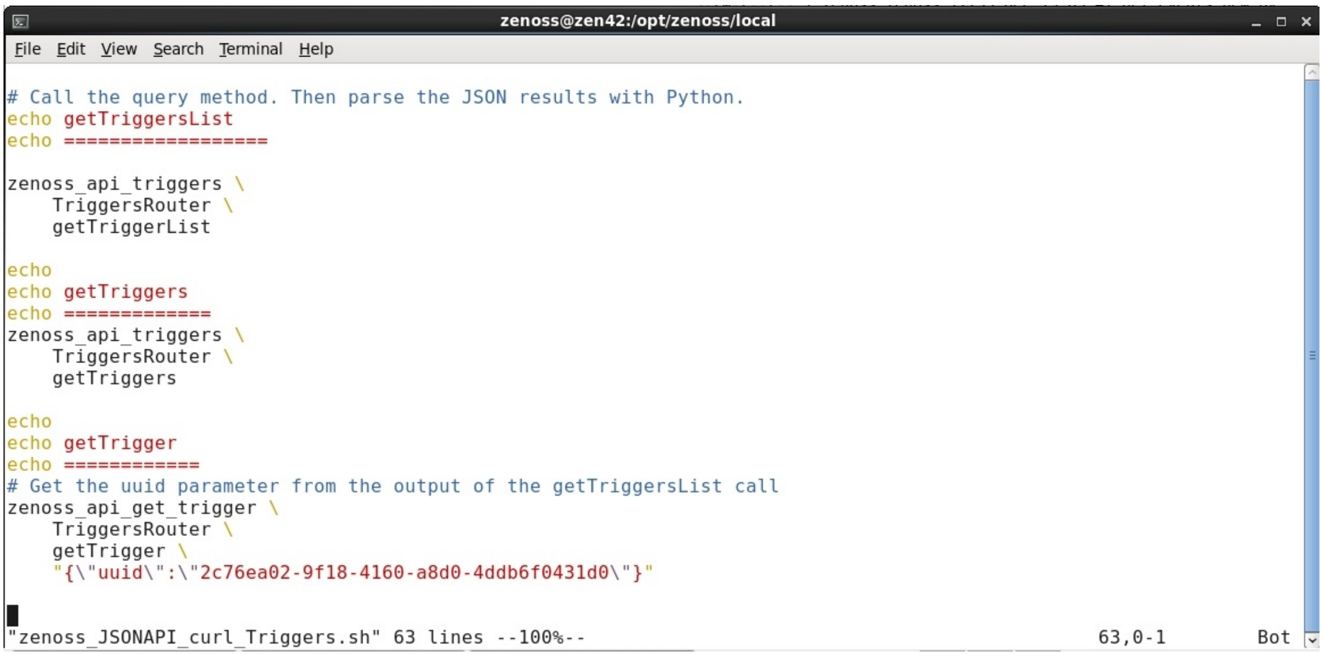


zenoss@zen42:/opt/zenoss/local

File Edit View Search Terminal Help

```
#!/bin/bash
#set -x

# Your Zenoss server settings.
ZENOSS_URL="http://zen42.class.example.org:8080"
ZENOSS_USERNAME="admin"
ZENOSS_PASSWORD="zenoss"

zenoss_api_triggers() {

ROUTER_ACTION=$1
ROUTER_METHOD=$2
ROUTER_ENDPOINT="triggers_router"

curl \
        -u "$ZENOSS_USERNAME:$ZENOSS_PASSWORD" \
        -X POST \
        -H "Content-Type: application/json" \
        -d "{\"action\":\"$ROUTER_ACTION\",\"method\":\"$ROUTER_METHOD\", \"tid\":1}" \
        "$ZENOSS_URL/zport/dmd/$ROUTER_ENDPOINT"
}

zenoss_api_get_trigger() {

ROUTER_ACTION=$1
ROUTER_METHOD=$2
DATA=$3
ROUTER_ENDPOINT="triggers_router"

curl \
        -u "$ZENOSS_USERNAME:$ZENOSS_PASSWORD" \
        -X POST \
        -H "Content-Type: application/json" \
        -d "{\"action\":\"$ROUTER_ACTION\",\"method\":\"$ROUTER_METHOD\",\"data\":[$DATA], \"tid\":1}" \
        "$ZENOSS_URL/zport/dmd/$ROUTER_ENDPOINT"
}
"zenoss_JSONAPI_curl_Triggers.sh" 63 lines --1%--                        1,1           Top
```

Figure 97: zenoss_JSONAPI_curl_triggers.sh part 1 showing 2 helper functions



zenoss@zen42:/opt/zenoss/local

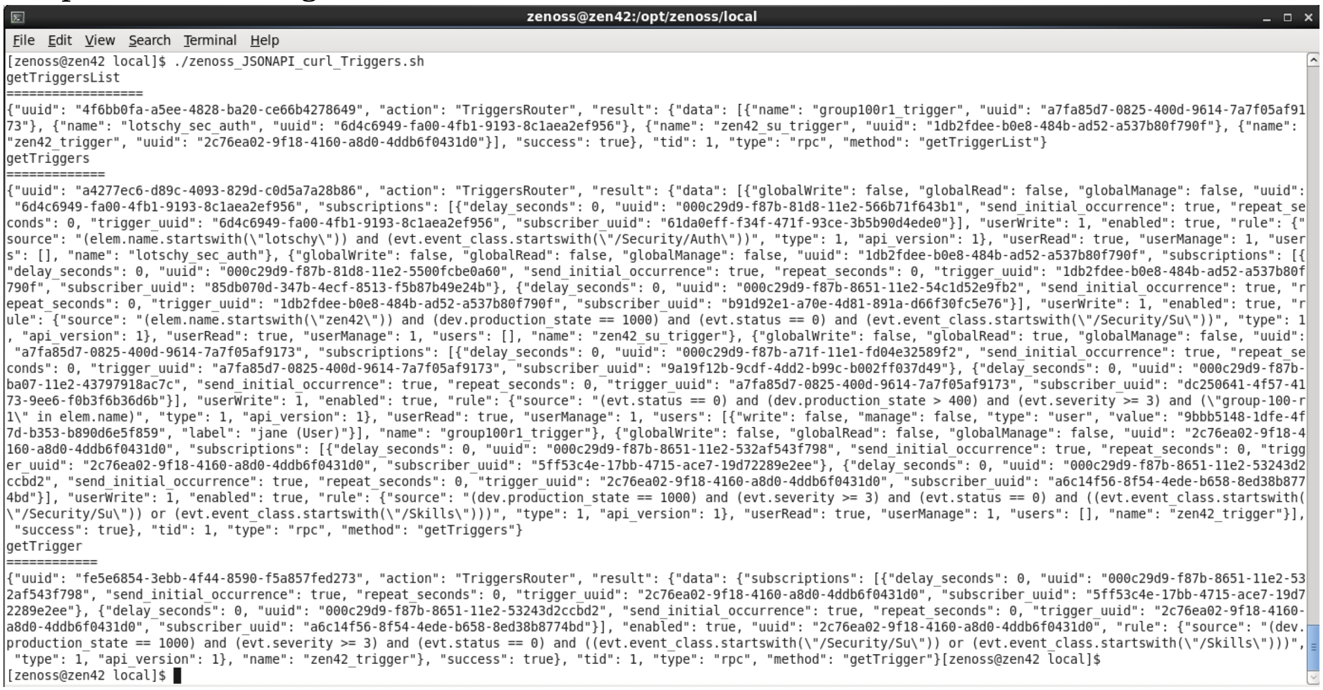
File Edit View Search Terminal Help

```
# Call the query method. Then parse the JSON results with Python.
echo getTriggersList
echo ==================

zenoss_api_triggers \
    TriggersRouter \
    getTriggerList

echo
echo getTriggers
echo =============
zenoss_api_triggers \
    TriggersRouter \
    getTriggers

echo
echo getTrigger
echo ============
# Get the uuid parameter from the output of the getTriggersList call
zenoss_api_get_trigger \
    TriggersRouter \
    getTrigger \
    "{\"uuid\":\"2c76ea02-9f18-4160-a8d0-4ddb6f0431d0\"}"

"zenoss_JSONAPI_curl_Triggers.sh" 63 lines --100%--                      63,0-1        Bot
```

Figure 98: zenoss_JSONAPI_curl_triggers.sh part 2 calling the helper functions with different methods

The main body of *zenoss_JSONAPI_curl_triggers.sh* has two calls to *zenoss_api_triggers* (with no data parameter) to produce a list of triggers and the detail for each trigger, respectively; the third call uses the second helper function with the *getTrigger* method and provides a *uuid* parameter to just get the detail of a specific trigger. The uuid was determined from the *getTriggerList* output and then hardcoded back into the script as an example.

Output looks like Figure 99.



```
[zenoss@zen42 local]$ ./zenoss_JSONAPI_curl_Triggers.sh
getTriggersList
==================
{"uuid": "4f6bb0fa-a5ee-4828-ba20-ce66b4278649", "action": "TriggersRouter", "result": {"data": [{"name": "group100r1_trigger", "uuid": "a7fa85d7-0825-400d-9614-7a7f05af91
73"}, {"name": "lotschy_sec_auth", "uuid": "6d4c6949-fa00-4fb1-9193-8c1aea2ef956"}, {"name": "zen42_su_trigger", "uuid": "1db2fdee-b0e8-484b-ad52-a537b80f790f"}, {"name":
"zen42_trigger", "uuid": "2c76ea02-9f18-4160-a8d0-4ddb6f0431d0"}], "success": true}, "tid": 1, "type": "rpc", "method": "getTriggerList"}
getTriggers
=============
{"uuid": "a4277ec6-d89c-4093-829d-c0d5a7a28b86", "action": "TriggersRouter", "result": {"data": [{"globalWrite": false, "globalRead": false, "globalManage": false, "uuid":
 "6d4c6949-fa00-4fb1-9193-8c1aea2ef956", "subscriptions": [{"delay_seconds": 0, "uuid": "000c29d9-f87b-81d8-11e2-566b71f643b1", "send_initial_occurrence": true, "repeat_se
conds": 0, "trigger_uuid": "6d4c6949-fa00-4fb1-9193-8c1aea2ef956", "subscriber_uuid": "61da0eff-f34f-471f-93ce-3b5b90d4ede0"}], "userWrite": 1, "enabled": true, "rule": {"
source": "(elem.name.startswith(\"lotschy\")) and (evt.event_class.startswith(\"/Security/Auth\"))", "type": 1, "api_version": 1}, "userRead": true, "userManage": 1, "user
s": [], "name": "lotschy_sec_auth"}, {"globalWrite": false, "globalRead": false, "globalManage": false, "uuid": "1db2fdee-b0e8-484b-ad52-a537b80f790f", "subscriptions": [{
"delay_seconds": 0, "uuid": "000c29d9-f87b-81d8-11e2-5500fcbe0a60", "send_initial_occurrence": true, "repeat_seconds": 0, "trigger_uuid": "1db2fdee-b0e8-484b-ad52-a537b80f
790f", "subscriber_uuid": "85db070d-347b-4ecf-8513-f5b87b49e24b"}, {"delay_seconds": 0, "uuid": "000c29d9-f87b-8651-11e2-54c1d52e9fb2", "send_initial_occurrence": true, "r
epeat_seconds": 0, "trigger_uuid": "1db2fdee-b0e8-484b-ad52-a537b80f790f", "subscriber_uuid": "b91d92e1-a70e-4d81-891a-d66f30fc5e76"}], "userWrite": 1, "enabled": true, "r
ule": {"source": "(elem.name.startswith(\"zen42\")) and (dev.production_state == 1000) and (evt.status == 0) and (evt.event_class.startswith(\"/Security/Su\"))", "type": 1
, "api_version": 1}, "userRead": true, "userManage": 1, "users": [], "name": "zen42_su_trigger"}, {"globalWrite": false, "globalRead": true, "globalManage": false, "uuid":
 "a7fa85d7-0825-400d-9614-7a7f05af9173", "subscriptions": [{"delay_seconds": 0, "uuid": "000c29d9-f87b-a71f-11e1-fd04e32589f2", "send_initial_occurrence": true, "repeat_se
conds": 0, "trigger_uuid": "a7fa85d7-0825-400d-9614-7a7f05af9173", "subscriber_uuid": "9a19f12b-9cdf-4dd2-b99c-b002ff037d49"}, {"delay_seconds": 0, "uuid": "000c29d9-f87b-
ba07-11e2-43797918ac7c", "send_initial_occurrence": true, "repeat_seconds": 0, "trigger_uuid": "a7fa85d7-0825-400d-9614-7a7f05af9173", "subscriber_uuid": "dc250641-4f57-41
73-9ee6-f0b3f6b36d6b"}], "userWrite": 1, "enabled": true, "rule": {"source": "(evt.status == 0) and (dev.production_state > 400) and (evt.severity >= 3) and (\"group-100-r
1\" in elem.name)", "type": 1, "api_version": 1}, "userRead": true, "userManage": 1, "users": [{"write": false, "manage": false, "type": "user", "value": "9bbb5148-1dfe-4f
7d-b353-b890d6e5f859", "label": "jane (User)"}], "name": "group100r1_trigger"}, {"globalWrite": false, "globalRead": false, "globalManage": false, "uuid": "2c76ea02-9f18-4
160-a8d0-4ddb6f0431d0", "subscriptions": [{"delay_seconds": 0, "uuid": "000c29d9-f87b-8651-11e2-532af543f798", "send_initial_occurrence": true, "repeat_seconds": 0, "trigg
er_uuid": "2c76ea02-9f18-4160-a8d0-4ddb6f0431d0", "subscriber_uuid": "5ff53c4e-17bb-4715-ace7-19d72289e2ee"}, {"delay_seconds": 0, "uuid": "000c29d9-f87b-8651-11e2-53243d2
ccbd2", "send_initial_occurrence": true, "repeat_seconds": 0, "trigger_uuid": "2c76ea02-9f18-4160-a8d0-4ddb6f0431d0", "subscriber_uuid": "a6c14f56-8f54-4ede-b658-8ed38b877
4bd"}], "userWrite": 1, "enabled": true, "rule": {"source": "(dev.production_state == 1000) and (evt.severity >= 3) and (evt.status == 0) and ((evt.event_class.startswith(
\"/Security/Su\")) or (evt.event_class.startswith(\"/Skills\")))", "type": 1, "api_version": 1}, "userRead": true, "userManage": 1, "users": [], "name": "zen42_trigger"}],
 "success": true}, "tid": 1, "type": "rpc", "method": "getTriggers"}
getTrigger
============
{"uuid": "fe5e6854-3ebb-4f44-8590-f5a857fed273", "action": "TriggersRouter", "result": {"data": {"subscriptions": [{"delay_seconds": 0, "uuid": "000c29d9-f87b-8651-11e2-53
2af543f798", "send_initial_occurrence": true, "repeat_seconds": 0, "trigger_uuid": "2c76ea02-9f18-4160-a8d0-4ddb6f0431d0", "subscriber_uuid": "5ff53c4e-17bb-4715-ace7-19d7
2289e2ee"}, {"delay_seconds": 0, "uuid": "000c29d9-f87b-8651-11e2-53243d2ccbd2", "send_initial_occurrence": true, "repeat_seconds": 0, "trigger_uuid": "2c76ea02-9f18-4160-
a8d0-4ddb6f0431d0", "subscriber_uuid": "a6c14f56-8f54-4ede-b658-8ed38b8774bd"}], "enabled": true, "uuid": "2c76ea02-9f18-4160-a8d0-4ddb6f0431d0", "rule": {"source": "(dev.
production_state == 1000) and (evt.severity >= 3) and (evt.status == 0) and ((evt.event_class.startswith(\"/Security/Su\")) or (evt.event_class.startswith(\"/Skills\")))",
 "type": 1, "api_version": 1}, "name": "zen42_trigger"}, "success": true}, "tid": 1, "type": "rpc", "method": "getTrigger"}[zenoss@zen42 local]$
[zenoss@zen42 local]$ 
```

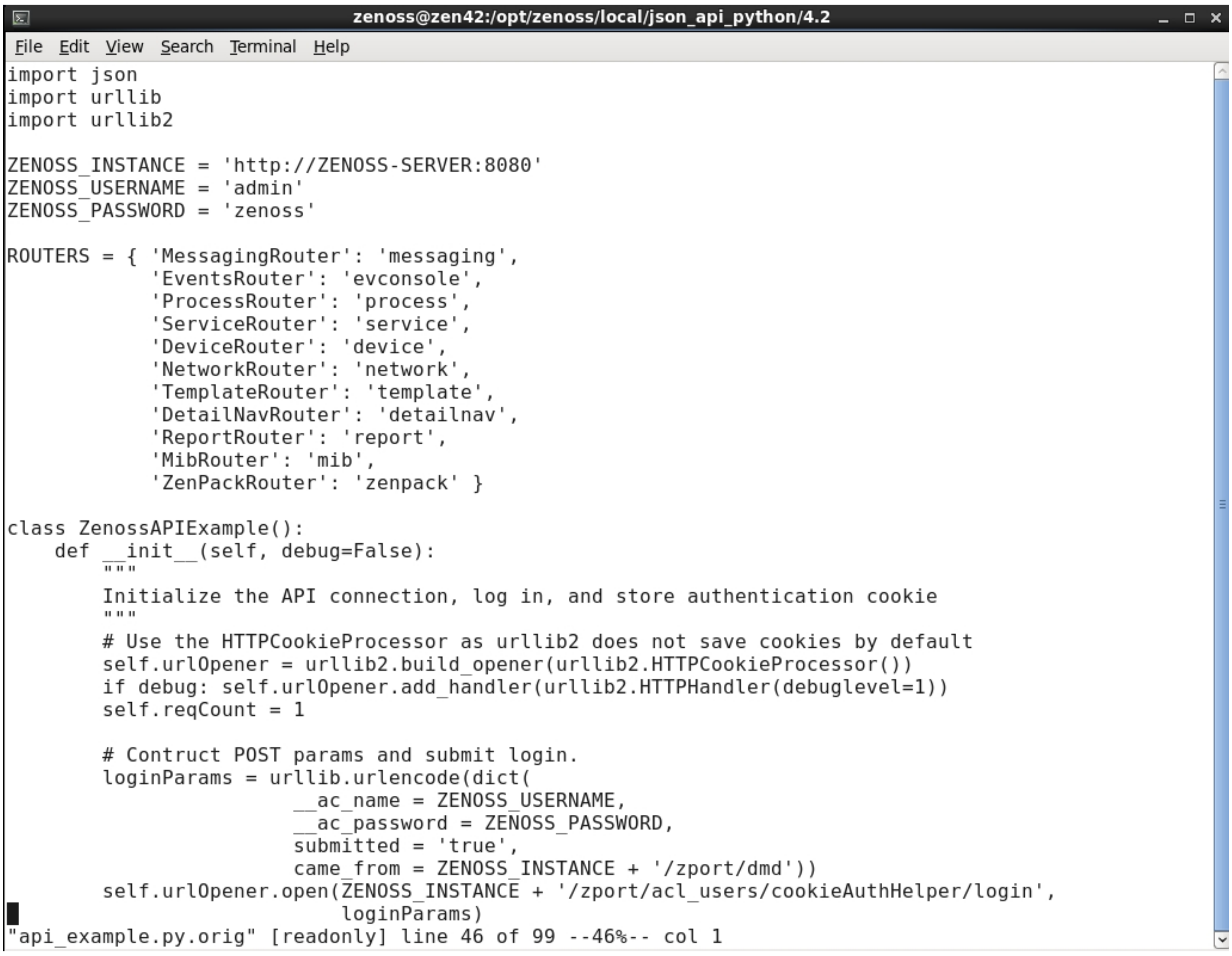
Figure 99: Output from zenoss_JSONAPI_curl_triggers.sh

Note that using the bash / curl interface with the EventsRouter class in the zep router module, is much harder as many of the methods require a dictionary as an input parameter. For this reason, it is easier to drive the events part of the JSON API from a Python harness.

11.3.2 Python examples

The JSON API documentation bundle delivers a python subdirectory with examples. Be sure to check http://community.zenoss.org/message/70052#70052 if you are seeing unexplainable errors.

api_example.py provides a generic class, *ZenossAPIExample()*, which connects to the Zenoss server.

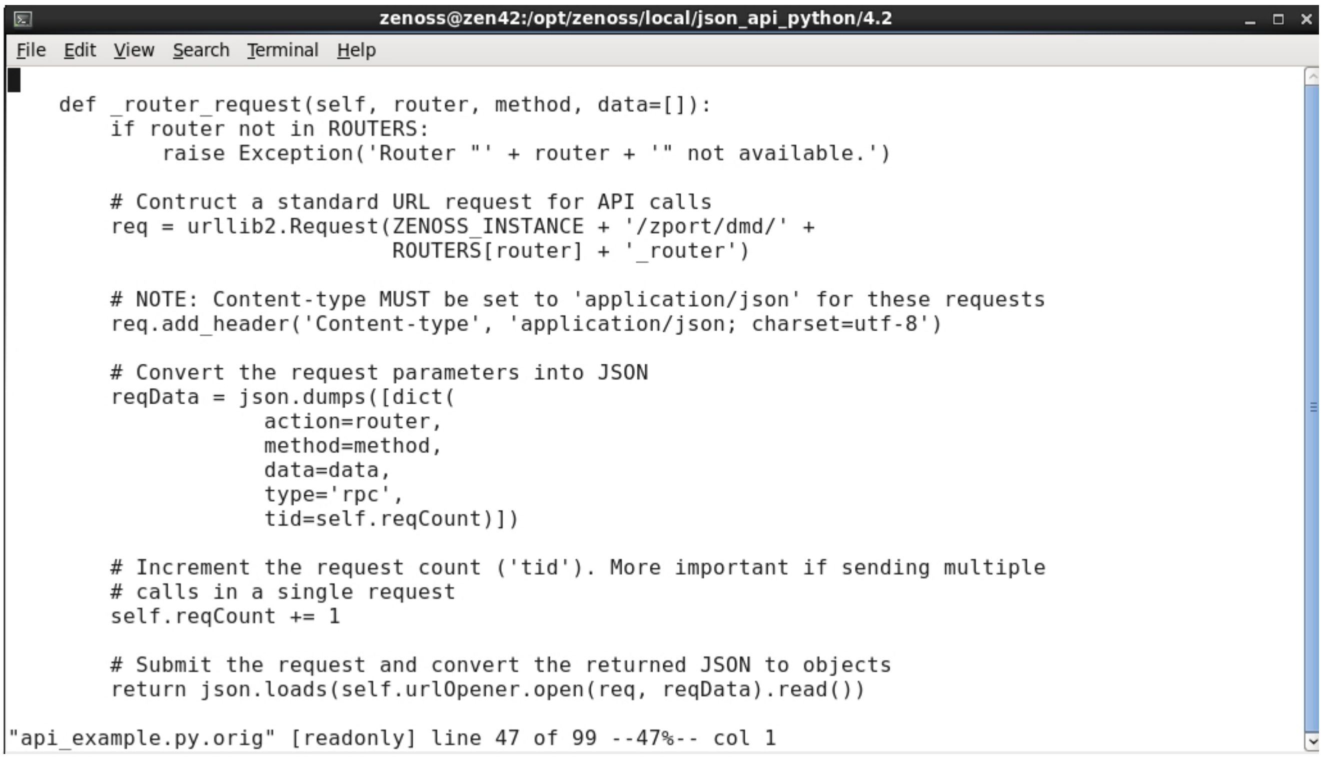

zenoss@zen42:/opt/zenoss/local/json_api_python/4.2
File Edit View Search Terminal Help

```
import json
import urllib
import urllib2

ZENOSS_INSTANCE = 'http://ZENOSS-SERVER:8080'
ZENOSS_USERNAME = 'admin'
ZENOSS_PASSWORD = 'zenoss'

ROUTERS = { 'MessagingRouter': 'messaging',
            'EventsRouter': 'evconsole',
            'ProcessRouter': 'process',
            'ServiceRouter': 'service',
            'DeviceRouter': 'device',
            'NetworkRouter': 'network',
            'TemplateRouter': 'template',
            'DetailNavRouter': 'detailnav',
            'ReportRouter': 'report',
            'MibRouter': 'mib',
            'ZenPackRouter': 'zenpack' }

class ZenossAPIExample():
    def __init__(self, debug=False):
        """
        Initialize the API connection, log in, and store authentication cookie
        """
        # Use the HTTPCookieProcessor as urllib2 does not save cookies by default
        self.urlOpener = urllib2.build_opener(urllib2.HTTPCookieProcessor())
        if debug: self.urlOpener.add_handler(urllib2.HTTPHandler(debuglevel=1))
        self.reqCount = 1

        # Contruct POST params and submit login.
        loginParams = urllib.urlencode(dict(
                        __ac_name = ZENOSS_USERNAME,
                        __ac_password = ZENOSS_PASSWORD,
                        submitted = 'true',
                        came_from = ZENOSS_INSTANCE + '/zport/dmd'))
        self.urlOpener.open(ZENOSS_INSTANCE + '/zport/acl_users/cookieAuthHelper/login',
                            loginParams)
"api_example.py.orig" [readonly] line 46 of 99 --46%-- col 1
```

Figure 100: api_example.py part 1 with connection logic and routers defined

The class has a *_router_request* method that has parameters for the router class to connect to, the method to execute and a data list that passes parameters to the method, performing the translation between Python objects and JSON, as required.

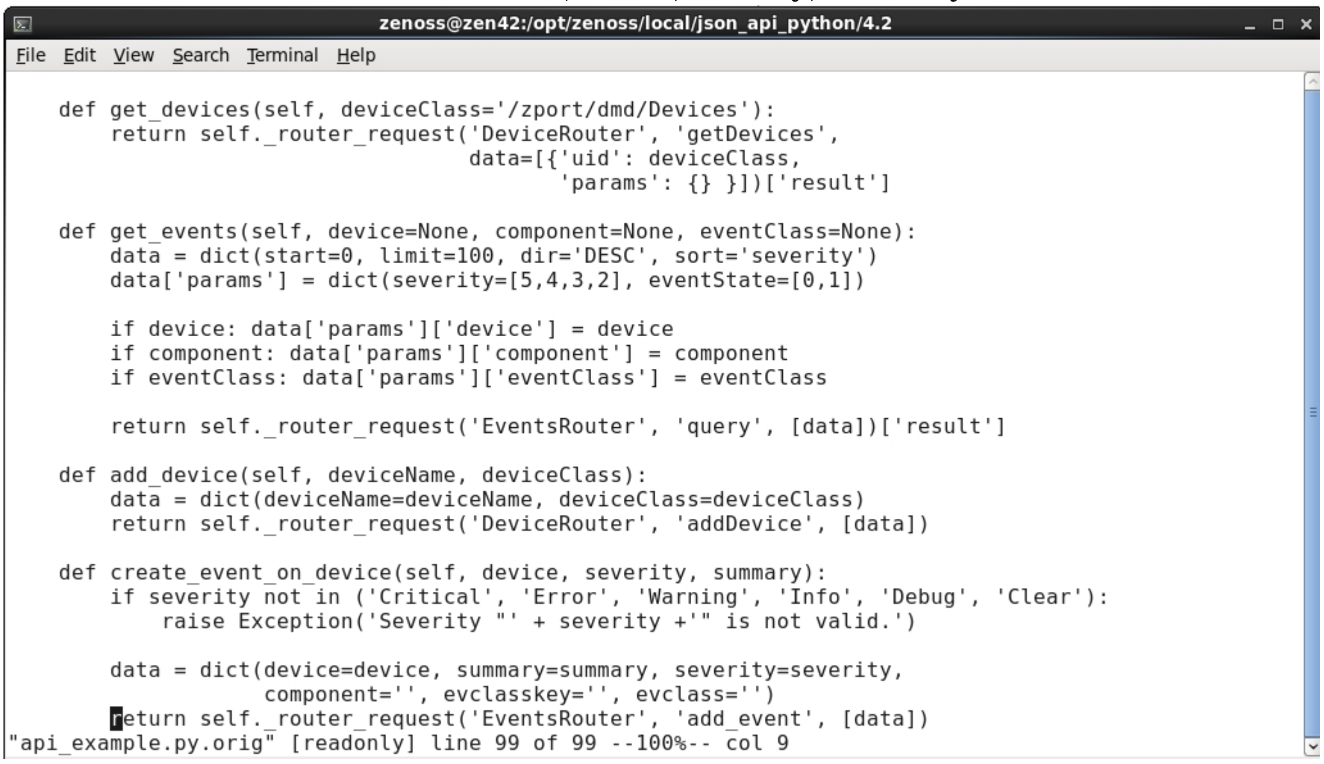
Four helper functions are also provided in *api_example.py*, each of which utilises the *_router_request* method.


zenoss@zen42:/opt/zenoss/local/json_api_python/4.2
File Edit View Search Terminal Help

```
    def _router_request(self, router, method, data=[]):
        if router not in ROUTERS:
            raise Exception('Router "' + router + '" not available.')

        # Contruct a standard URL request for API calls
        req = urllib2.Request(ZENOSS_INSTANCE + '/zport/dmd/' +
                              ROUTERS[router] + '_router')

        # NOTE: Content-type MUST be set to 'application/json' for these requests
        req.add_header('Content-type', 'application/json; charset=utf-8')

        # Convert the request parameters into JSON
        reqData = json.dumps([dict(
                    action=router,
                    method=method,
                    data=data,
                    type='rpc',
                    tid=self.reqCount)])

        # Increment the request count ('tid'). More important if sending multiple
        # calls in a single request
        self.reqCount += 1

        # Submit the request and convert the returned JSON to objects
        return json.loads(self.urlOpener.open(req, reqData).read())

"api_example.py.orig" [readonly] line 47 of 99 --47%-- col 1
```

Figure 101: api_example.py part 2 with _router_request method

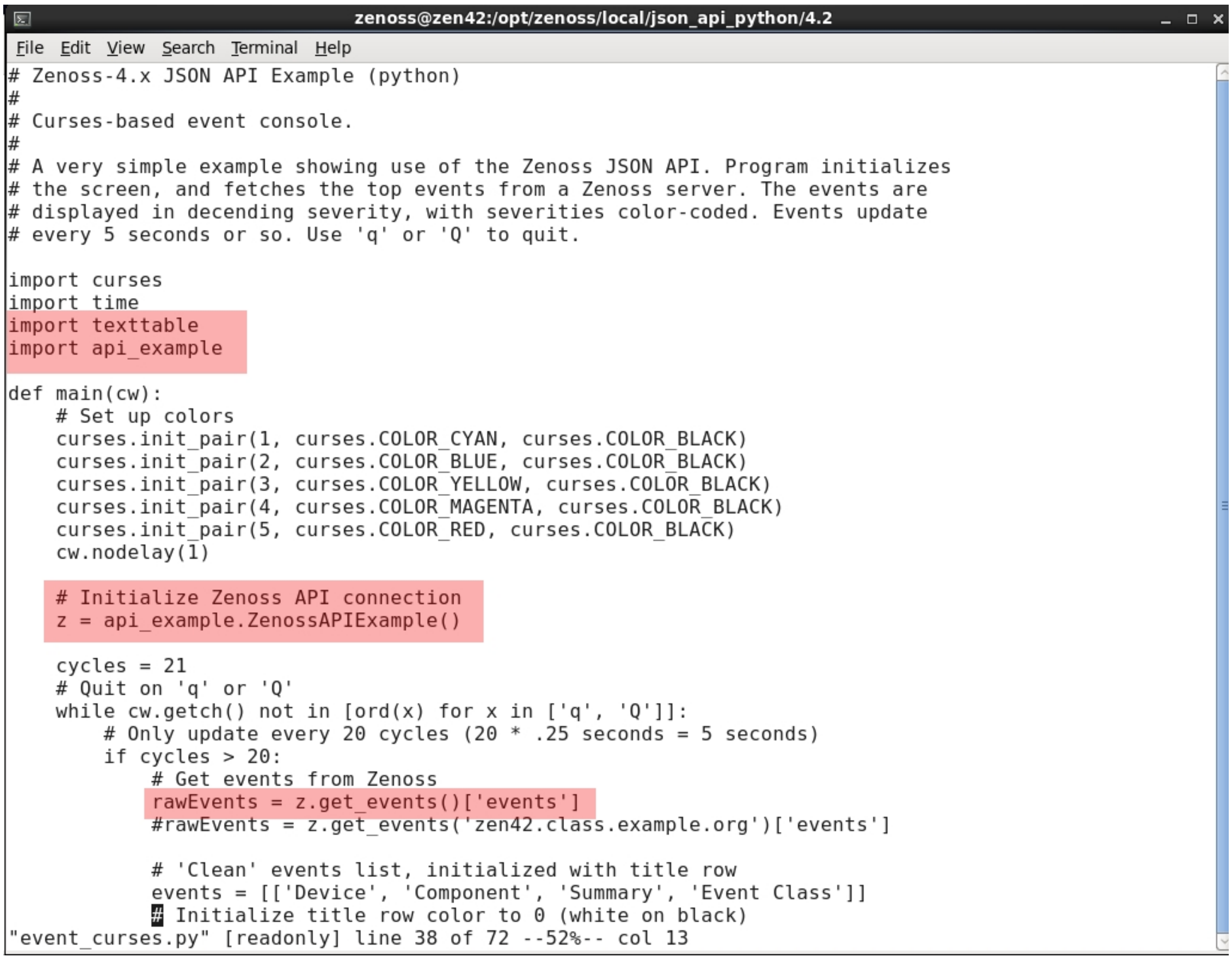
- def get_devices(self, deviceClass='/zport/dmd/Devices'):
- def get_events(self, device=None, component=None, eventClass=None):
- def add_device(self, deviceName, deviceClass):
- def create_event_on_device(self, device, severity, summary):


zenoss@zen42:/opt/zenoss/local/json_api_python/4.2
File Edit View Search Terminal Help

```
    def get_devices(self, deviceClass='/zport/dmd/Devices'):
        return self._router_request('DeviceRouter', 'getDevices',
                                    data=[{'uid': deviceClass,
                                           'params': {} }])['result']

    def get_events(self, device=None, component=None, eventClass=None):
        data = dict(start=0, limit=100, dir='DESC', sort='severity')
        data['params'] = dict(severity=[5,4,3,2], eventState=[0,1])

        if device: data['params']['device'] = device
        if component: data['params']['component'] = component
        if eventClass: data['params']['eventClass'] = eventClass

        return self._router_request('EventsRouter', 'query', [data])['result']

    def add_device(self, deviceName, deviceClass):
        data = dict(deviceName=deviceName, deviceClass=deviceClass)
        return self._router_request('DeviceRouter', 'addDevice', [data])

    def create_event_on_device(self, device, severity, summary):
        if severity not in ('Critical', 'Error', 'Warning', 'Info', 'Debug', 'Clear'):
            raise Exception('Severity "' + severity +'" is not valid.')

        data = dict(device=device, summary=summary, severity=severity,
                    component='', evclasskey='', evclass='')
        return self._router_request('EventsRouter', 'add_event', [data])
"api_example.py.orig" [readonly] line 99 of 99 --100%-- col 9
```

Figure 102: api_example.py part 3 with helper methods to access device and events objects

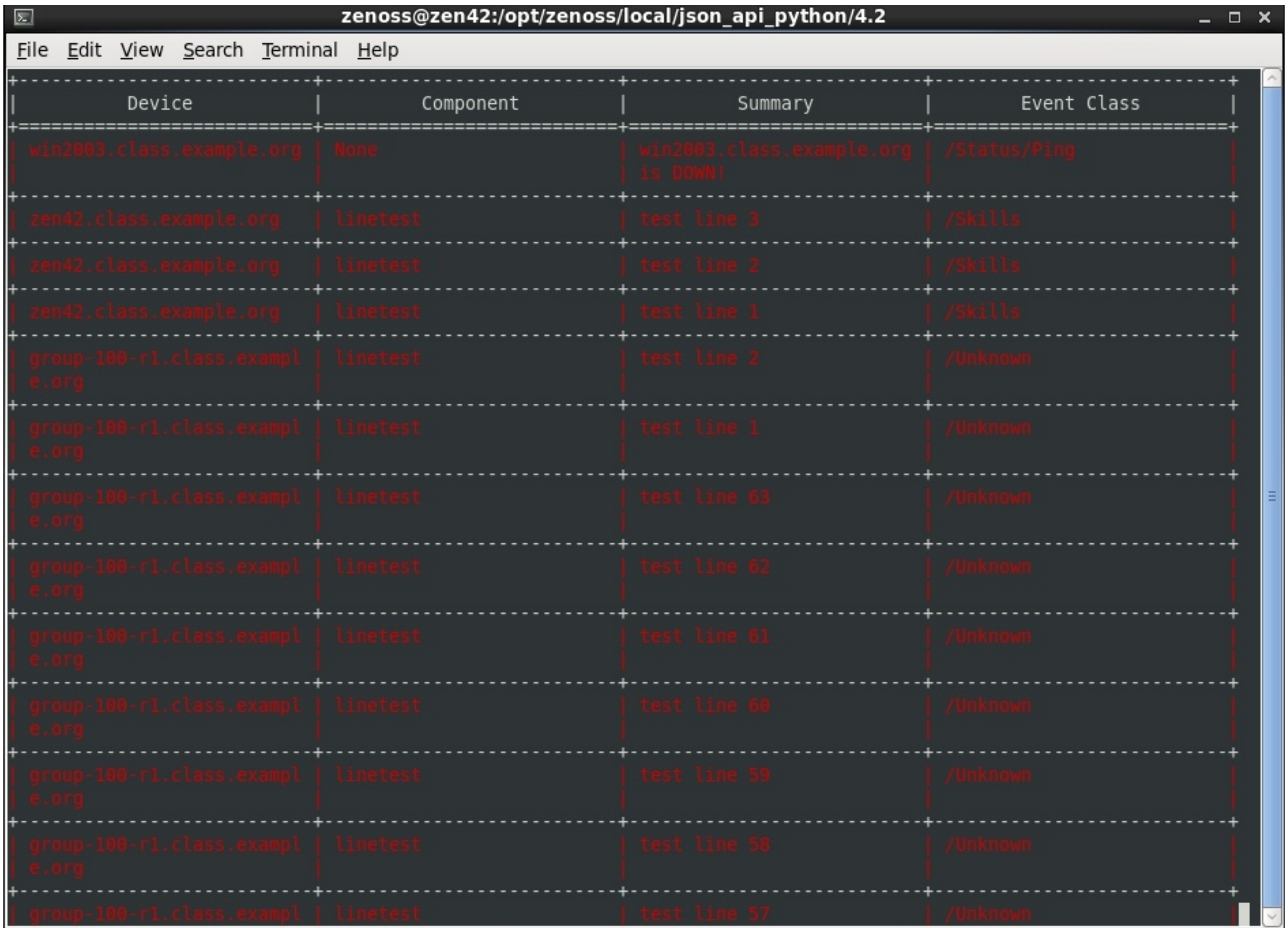
event_curses.py is an example script that imports *api_example* and uses the *get_events* method to access events in the MySQL database. The only other dependency is the import of *texttable* which is also included in the same directory (see *JSONAPIQuickstart.txt* in the top-level directory of the documentation).


zenoss@zen42:/opt/zenoss/local/json_api_python/4.2
File Edit View Search Terminal Help

```
# Zenoss-4.x JSON API Example (python)
#
# Curses-based event console.
#
# A very simple example showing use of the Zenoss JSON API. Program initializes
# the screen, and fetches the top events from a Zenoss server. The events are
# displayed in decending severity, with severities color-coded. Events update
# every 5 seconds or so. Use 'q' or 'Q' to quit.

import curses
import time
import texttable
import api_example

def main(cw):
    # Set up colors
    curses.init_pair(1, curses.COLOR_CYAN, curses.COLOR_BLACK)
    curses.init_pair(2, curses.COLOR_BLUE, curses.COLOR_BLACK)
    curses.init_pair(3, curses.COLOR_YELLOW, curses.COLOR_BLACK)
    curses.init_pair(4, curses.COLOR_MAGENTA, curses.COLOR_BLACK)
    curses.init_pair(5, curses.COLOR_RED, curses.COLOR_BLACK)
    cw.nodelay(1)

    # Initialize Zenoss API connection
    z = api_example.ZenossAPIExample()

    cycles = 21
    # Quit on 'q' or 'Q'
    while cw.getch() not in [ord(x) for x in ['q', 'Q']]:
        # Only update every 20 cycles (20 * .25 seconds = 5 seconds)
        if cycles > 20:
            # Get events from Zenoss
            rawEvents = z.get_events()['events']
            #rawEvents = z.get_events('zen42.class.example.org')['events']

            # 'Clean' events list, initialized with title row
            events = [['Device', 'Component', 'Summary', 'Event Class']]
            # Initialize title row color to 0 (white on black)
"event_curses.py" [readonly] line 38 of 72 --52%-- col 13
```

Figure 103: event_curses.py highlighting calls to the api_example functionality

When *event_curses.py* is run with *python event_curses.py*, a list of events is output to the screen with Device, Component, Summary and Event Class fields, each line being colour-coded by severity. As shipped, all New and Acknowledged status events of severity 5, 4, 3 and 2, are retrieved from the MySQL database.


zenoss@zen42:/opt/zenoss/local/json_api_python/4.2
File Edit View Search Terminal Help

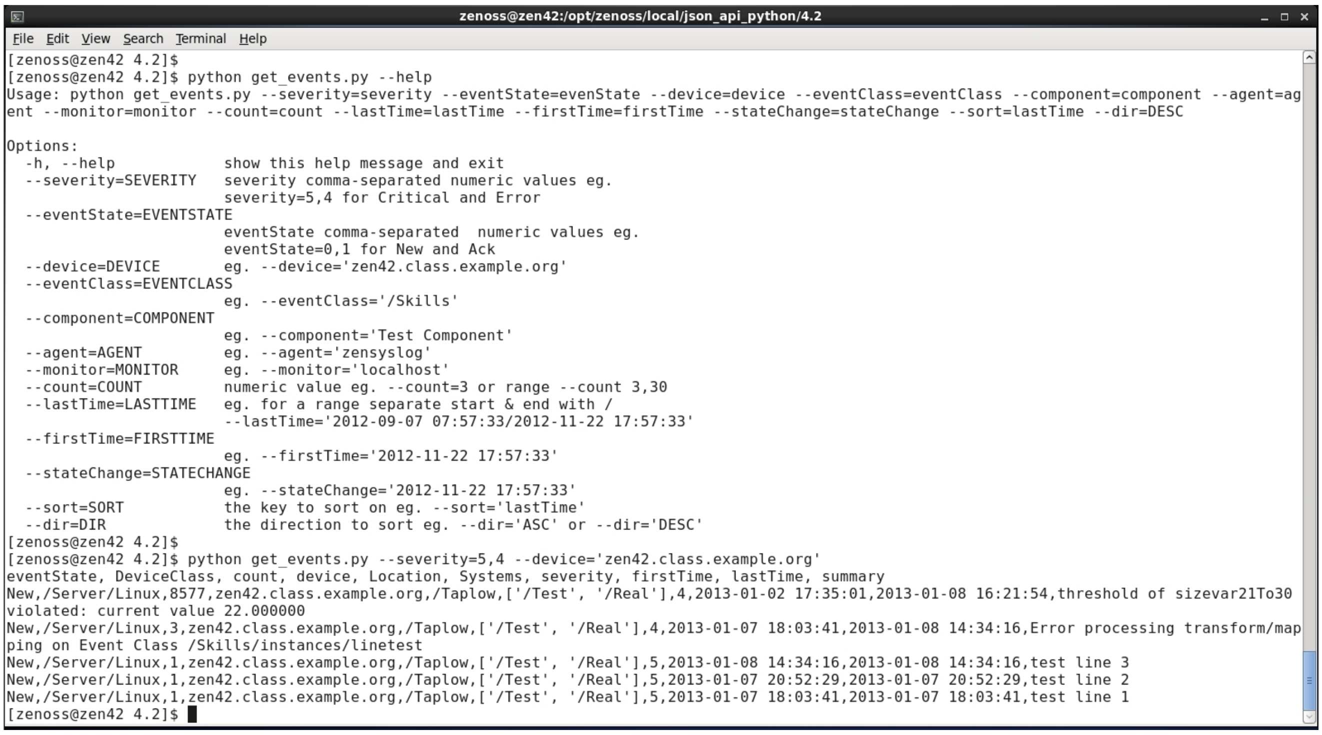
```
+---------------------------+---------------------------+---------------------------+---------------------------+
|          Device           |         Component         |          Summary          |        Event Class        |
+===========================+===========================+===========================+===========================+
| win2003.class.example.org | None                      | win2003.class.example.org | /Status/Ping              |
|                           |                           | is DOWN!                  |                           |
+---------------------------+---------------------------+---------------------------+---------------------------+
| zen42.class.example.org   | linetest                  | test line 3               | /Skills                   |
+---------------------------+---------------------------+---------------------------+---------------------------+
| zen42.class.example.org   | linetest                  | test line 2               | /Skills                   |
+---------------------------+---------------------------+---------------------------+---------------------------+
| zen42.class.example.org   | linetest                  | test line 1               | /Skills                   |
+---------------------------+---------------------------+---------------------------+---------------------------+
| group-100-r1.class.exampl | linetest                  | test line 2               | /Unknown                  |
| e.org                     |                           |                           |                           |
+---------------------------+---------------------------+---------------------------+---------------------------+
| group-100-r1.class.exampl | linetest                  | test line 1               | /Unknown                  |
| e.org                     |                           |                           |                           |
+---------------------------+---------------------------+---------------------------+---------------------------+
| group-100-r1.class.exampl | linetest                  | test line 63              | /Unknown                  |
| e.org                     |                           |                           |                           |
+---------------------------+---------------------------+---------------------------+---------------------------+
| group-100-r1.class.exampl | linetest                  | test line 62              | /Unknown                  |
| e.org                     |                           |                           |                           |
+---------------------------+---------------------------+---------------------------+---------------------------+
| group-100-r1.class.exampl | linetest                  | test line 61              | /Unknown                  |
| e.org                     |                           |                           |                           |
+---------------------------+---------------------------+---------------------------+---------------------------+
| group-100-r1.class.exampl | linetest                  | test line 60              | /Unknown                  |
| e.org                     |                           |                           |                           |
+---------------------------+---------------------------+---------------------------+---------------------------+
| group-100-r1.class.exampl | linetest                  | test line 59              | /Unknown                  |
| e.org                     |                           |                           |                           |
+---------------------------+---------------------------+---------------------------+---------------------------+
| group-100-r1.class.exampl | linetest                  | test line 58              | /Unknown                  |
| e.org                     |                           |                           |                           |
+---------------------------+---------------------------+---------------------------+---------------------------+
| group-100-r1.class.exampl | linetest                  | test line 57              | /Unknown                  |
```

Figure 104: Output of python event_curses.py

Note that if event_curses.py does not run then open a new command terminal with a default screen size and try again.

To be more selective on the event curses output, look closely at the commented out *rawEvents* = line in Figure 103. The line restricts output to just events from zen42.class.example.org.

For an extension of using the *query* method of the *EventsRouter* class, see *get_events.py* in Appendix A. It takes parameters to select the filter criteria for active events and then outputs a large number of fields. *python get_events.py --help* provides the usage.


zenoss@zen42:/opt/zenoss/local/json_api_python/4.2
File Edit View Search Terminal Help

```
[zenoss@zen42 4.2]$
[zenoss@zen42 4.2]$ python get_events.py --help
Usage: python get_events.py --severity=severity --eventState=evenState --device=device --eventClass=eventClass --component=component --agent=ag
ent --monitor=monitor --count=count --lastTime=lastTime --firstTime=firstTime --stateChange=stateChange --sort=lastTime --dir=DESC

Options:
  -h, --help            show this help message and exit
  --severity=SEVERITY   severity comma-separated numeric values eg.
                        severity=5,4 for Critical and Error
  --eventState=EVENTSTATE
                        eventState comma-separated  numeric values eg.
                        eventState=0,1 for New and Ack
  --device=DEVICE       eg. --device='zen42.class.example.org'
  --eventClass=EVENTCLASS
                        eg. --eventClass='/Skills'
  --component=COMPONENT
                        eg. --component='Test Component'
  --agent=AGENT         eg. --agent='zensyslog'
  --monitor=MONITOR     eg. --monitor='localhost'
  --count=COUNT         numeric value eg. --count=3 or range --count 3,30
  --lastTime=LASTTIME   eg. for a range separate start & end with /
                        --lastTime='2012-09-07 07:57:33/2012-11-22 17:57:33'
  --firstTime=FIRSTTIME
                        eg. --firstTime='2012-11-22 17:57:33'
  --stateChange=STATECHANGE
                        eg. --stateChange='2012-11-22 17:57:33'
  --sort=SORT           the key to sort on eg. --sort='lastTime'
  --dir=DIR             the direction to sort eg. --dir='ASC' or --dir='DESC'
[zenoss@zen42 4.2]$
[zenoss@zen42 4.2]$ python get_events.py --severity=5,4 --device='zen42.class.example.org'
eventState, DeviceClass, count, device, Location, Systems, severity, firstTime, lastTime, summary
New,/Server/Linux,8577,zen42.class.example.org,/Taplow,['/Test', '/Real'],4,2013-01-02 17:35:01,2013-01-08 16:21:54,threshold of sizevar21To30
violated: current value 22.000000
New,/Server/Linux,3,zen42.class.example.org,/Taplow,['/Test', '/Real'],4,2013-01-07 18:03:41,2013-01-08 14:34:16,Error processing transform/map
ping on Event Class /Skills/instances/linetest
New,/Server/Linux,1,zen42.class.example.org,/Taplow,['/Test', '/Real'],5,2013-01-08 14:34:16,2013-01-08 14:34:16,test line 3
New,/Server/Linux,1,zen42.class.example.org,/Taplow,['/Test', '/Real'],5,2013-01-07 20:52:29,2013-01-07 20:52:29,test line 2
New,/Server/Linux,1,zen42.class.example.org,/Taplow,['/Test', '/Real'],5,2013-01-07 18:03:41,2013-01-07 18:03:41,test line 1
[zenoss@zen42 4.2]$
```

Figure 105: get_events.py output to select active events and output to the console

12 Conclusions

Zenoss has an extensive event system capable of receiving events from Windows, syslogs and SNMP TRAPs, in addition to receiving the events generated internally by Zenoss's own discovery, availability and performance monitoring.

A large number of event classes are defined and configured when Zenoss is installed. These can be modified, removed or added to.

An event follows a fairly complex event life cycle process whereby it is mapped to an event class and then, optionally, it is transformed such that default fields of the event can be changed and user-defined fields can be created.

Event mapping for events from Windows, syslogs or SNMP, depends on the initial Zenoss parsing daemon delivering an eventClassKey field which must correspond to a defined mapping. Subsequently, a Python Rule and/or a Python Regex can be used to further distinguish between incoming events and map to different event classes.

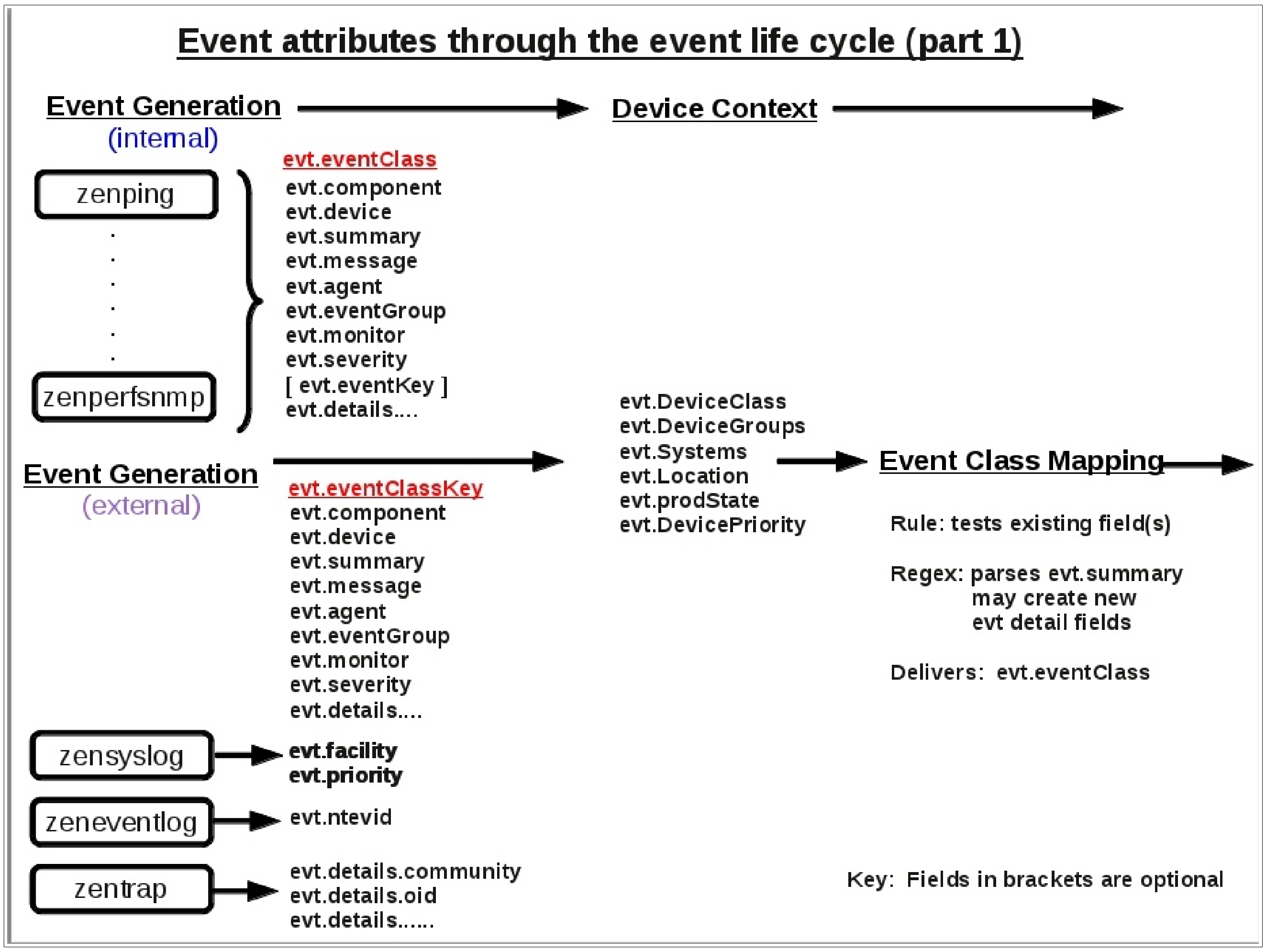


Figure 106: Event attributes through the event life cycle (part 1)

Device context is applied to an incoming event from the ZODB database; device context includes the prodState, DevicePriority, Location, DeviceClass, DeviceGroups and Systems field values. Device context provides the ability for transforms to take account of the device or device class hierarchy.

An event class includes event context – zEventAction, zEventSeverity and zEventClearClasses – which can be applied to individual subclasses of events or to class hierarchies. This means transforms can be affected by event type.

Event transforms can be simple assignment of event fields or can include complex Python programs. A good environment for testing Python is the zendmd command line utility. Transforms and/or the event context can be used to help clear events that have been resolved. Any event with a severity of Cleared will automatically clear other similar events; zEventClearClasses can be used to list extra classes that are cleared in addition.

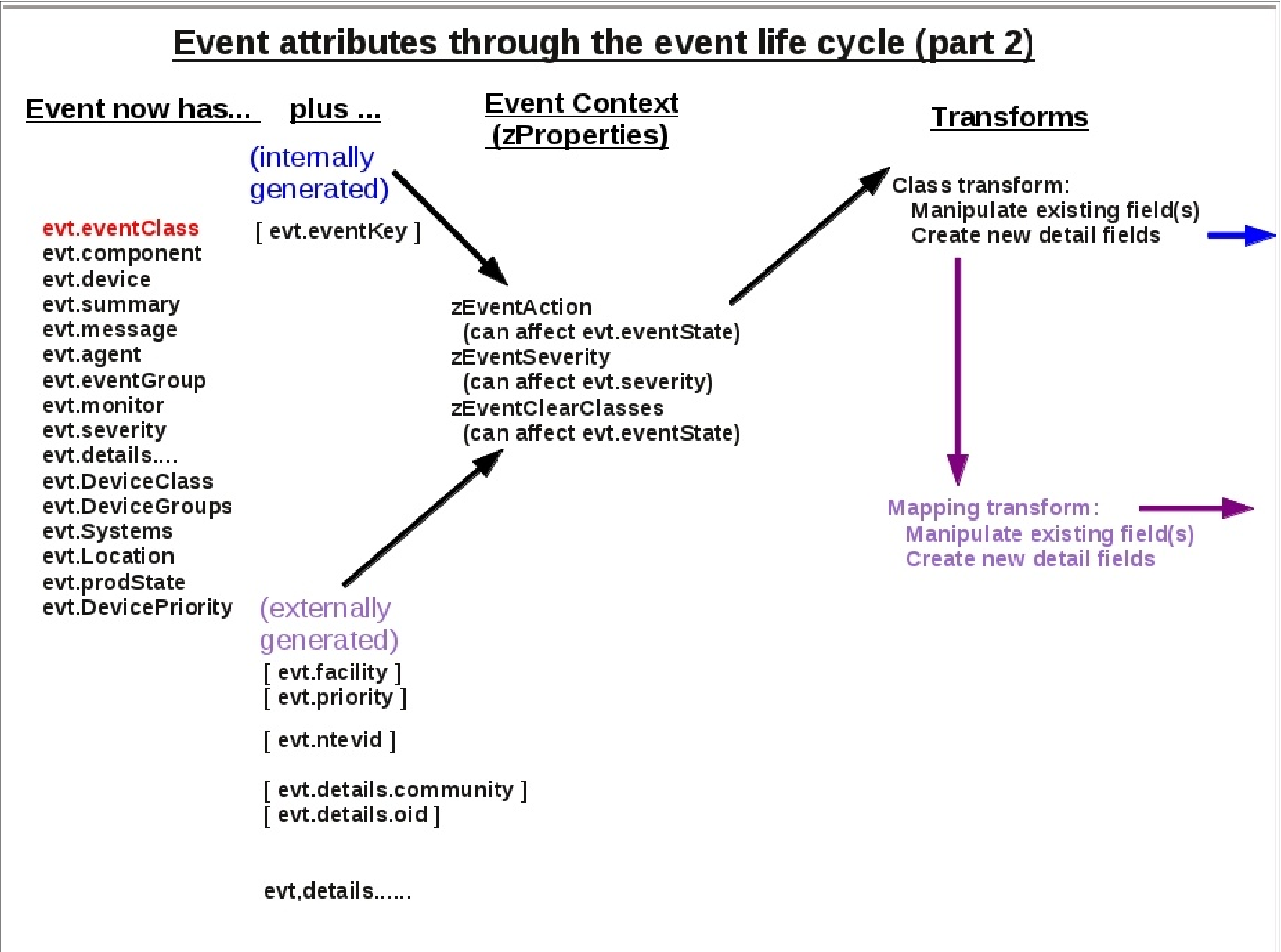


Figure 107: Event attributes through the event life cycle (part 2)

Events are saved in the MySQL zenoss_zep database in the event_summary table. Events can be Closed by users or Cleared by other events; they can also be Aged based on severity and length of time that the event has persisted. After a configurable interval, non-active events (with eventState of Closed, Cleared and Aged) are moved to the event_archive table of the database. Eventually, archived events can be deleted.

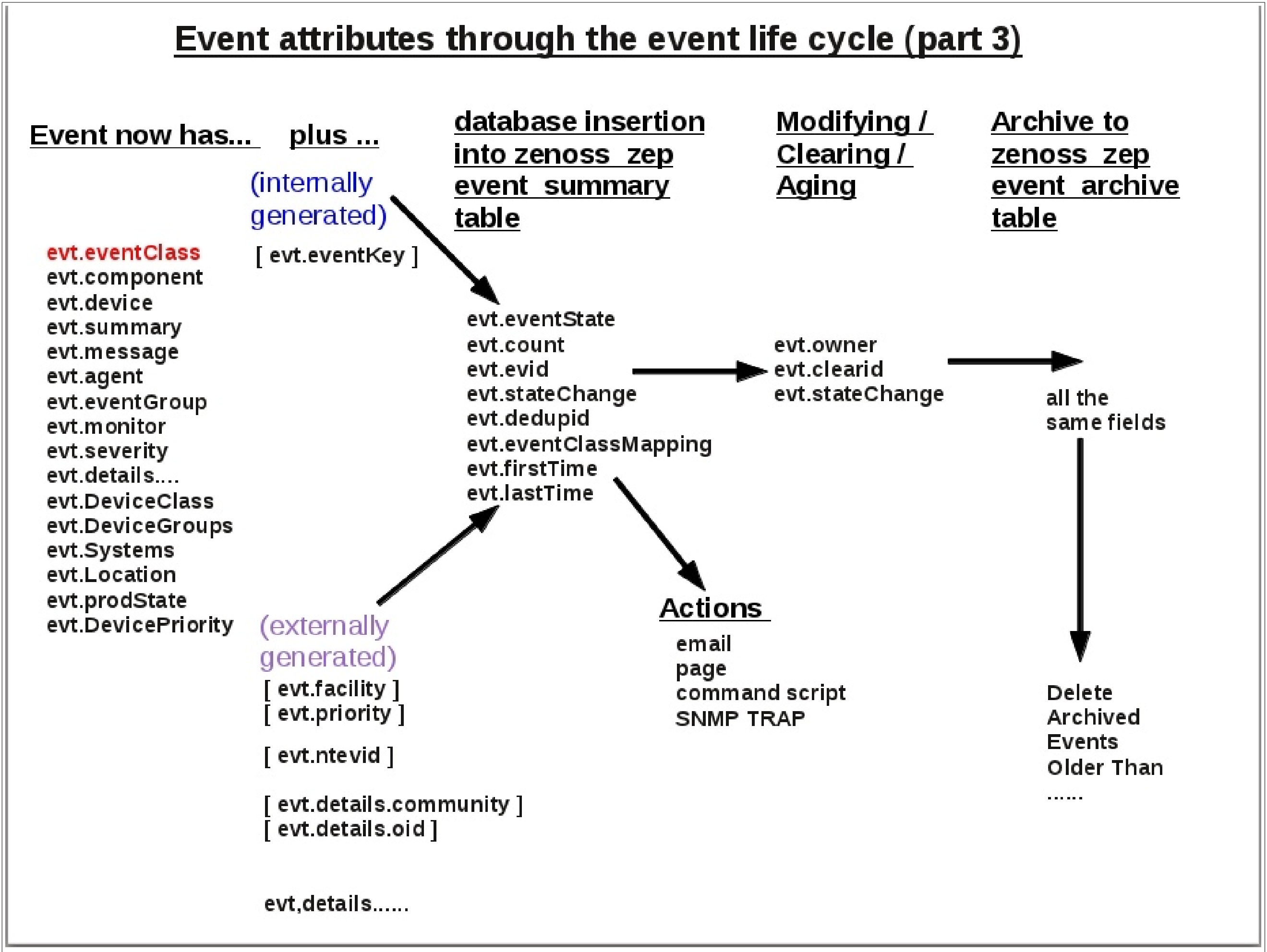


Figure 108: Event attributes through the event life cycle (part 3)

When events occur, actions can be generated either to alert users by using email or a paging system; alternatively, background actions can be configured to run a command on the Zenoss server or to generate an SNMP TRAP.

The JSON API provides a generic interface for accessing data in the Zenoss system.

As with any enterprise management system, Zenoss has the tools to configure almost any response to any event.

13 Appendix A

13.1 getevents.py

get_events.py to select active events.

```
# Zenoss-4.x JSON API Example (python)
#
# To quickly explore, execute 'python -i get_events.py
#
# >>> z = getEventsWithJSON()
# >>> events = z.get_events()
# etc.

import json
import urllib
import urllib2
from optparse import OptionParser
import pprint


#ZENOSS_INSTANCE = 'http://ZENOSS-SERVER:8080'
# Change the next line(s) to suit your environment
#
ZENOSS_INSTANCE = 'http://zen42.class.example.org:8080'
ZENOSS_USERNAME = 'admin'
ZENOSS_PASSWORD = 'zenoss'

ROUTERS = { 'MessagingRouter': 'messaging',
            'EventsRouter': 'evconsole',
            'ProcessRouter': 'process',
            'ServiceRouter': 'service',
            'DeviceRouter': 'device',
            'NetworkRouter': 'network',
            'TemplateRouter': 'template',
            'DetailNavRouter': 'detailnav',
            'ReportRouter': 'report',
            'MibRouter': 'mib',
            'ZenPackRouter': 'zenpack' }

class getEventsWithJSON():
    def __init__(self, debug=False):
        """
        Initialize the API connection, log in, and store authentication
cookie
        """
        # Use the HTTPCookieProcessor as urllib2 does not save cookies by
default
        self.urlOpener =
urllib2.build_opener(urllib2.HTTPCookieProcessor())
        if debug:
self.urlOpener.add_handler(urllib2.HTTPHandler(debuglevel=1))
        self.reqCount = 1

        # Construct POST params and submit login.
        loginParams = urllib.urlencode(dict(
                        __ac_name = ZENOSS_USERNAME,
                        __ac_password = ZENOSS_PASSWORD,
                        submitted = 'true',
                        came_from = ZENOSS_INSTANCE + '/zport/dmd'))
        self.urlOpener.open(ZENOSS_INSTANCE +
'/zport/acl_users/cookieAuthHelper/login',
                            loginParams)

    def _router_request(self, router, method, data=[]):
        if router not in ROUTERS:
            raise Exception('Router "' + router + '" not available.')

        # Construct a standard URL request for API calls
        req = urllib2.Request(ZENOSS_INSTANCE + '/zport/dmd/' +
                              ROUTERS[router] + '_router')

        # NOTE: Content-type MUST be set to 'application/json' for these
requests
        req.add_header('Content-type', 'application/json; charset=utf-8')

        # Convert the request parameters into JSON
        reqData = json.dumps([dict(
                    action=router,
                    method=method,
                    data=data,
                    type='rpc',
                    tid=self.reqCount)])

        # Increment the request count ('tid'). More important if sending
multiple
        # calls in a single request
        self.reqCount += 1

        # Submit the request and convert the returned JSON to objects
        return json.loads(self.urlOpener.open(req, reqData).read())


    def get_events(self, filter={}, sort='severity', dir='DESC'):
        """ Use EventsRouter action (Class) and query method found
        in JSON API docs on Zenoss website:

        query(self, limit=0, start=0, sort='lastTime', dir='desc',
params=None,
        archive=False, uid=None, detailFormat=False)

        Parameters:

        limit (integer) - (optional) Max index of events to retrieve
(default: 0)
        start (integer) - (optional) Min index of events to retrieve
(default: 0)
        sort (string) - (optional) Key on which to sort the return results
(default: 'lastTime')
        dir (string) - (optional) Sort order; can be either 'ASC' or 'DESC'
(default: 'DESC')
        params (dictionary) - (optional) Key-value pair of filters for this
search. (default: None)
        params are the filters to the query method and can be found in the
_buildFilter method.
            severity = params.get('severity'),
            status = [i for i in params.get('eventState', [])],
            event_class = filter(None, [params.get('eventClass')]),
            Note that the time values can be ranges where a valid range
would be
                '2012-09-07 07:57:33/2012-11-22 17:57:33'

            first_seen = params.get('firstTime') and
self._timeRange(params.get('firstTime')),
            last_seen = params.get('lastTime') and
self._timeRange(params.get('lastTime')),
            status_change = params.get('stateChange') and
self._timeRange(params.get('stateChange')),
            uuid = filterEventUuids,
            count_range = params.get('count'),
            element_title = params.get('device'),
            element_sub_title = params.get('component'),
            event_summary = params.get('summary'),
            current_user_name = params.get('ownerid'),
            agent = params.get('agent'),
            monitor = params.get('monitor'),
            fingerprint = params.get('dedupid'),
            tags = params.get('tags'),
            details = details,

        archive (boolean) - (optional) True to search the event history
table instead of active events (default: False)
        uid (string) - (optional) Context for the query (default: None)

        Returns: dictionary
        Properties:
            events: ([dictionary]) List of objects representing events
            totalCount: (integer) Total count of events returned
            asof: (float) Current time
        """

        data = dict(start=0, limit=1000)
        if sort: data['sort'] = sort
        if dir: data['dir'] = dir

        data['params'] = filter

        #print 'data[params] is %s \n' % (data['params'])
        #print 'data is %s \n' % (data)

        return self._router_request('EventsRouter', 'query', [data])
['result']

if __name__ == "__main__":
    usage = 'python %prog --severity=severity --eventState=eventState
--device=device --eventClass=eventClass --component=component --agent=agent
--monitor=monitor --count=count --lastTime=lastTime --firstTime=firstTime
--stateChange=stateChange --sort=lastTime --dir=DESC'

    parser = OptionParser(usage)
    parser.add_option("--severity", dest='severity',
                        help='severity comma-separated numeric values eg.
severity=5,4 for Critical and Error')
    parser.add_option("--eventState", dest='eventState', default='0,1',
                        help='eventState comma-separated  numeric values
eg. eventState=0,1 for New and Ack')
    parser.add_option("--device", dest='device',
                        help='eg. --device=\'zen42.class.example.org\'')
    parser.add_option("--eventClass", dest='eventClass',
                        help='eg. --eventClass=\'/Skills\'')
    parser.add_option("--component", dest='component',
                        help='eg. --component=\'Test Component\'')
    parser.add_option("--agent", dest='agent',
                        help='eg. --agent=\'zensyslog\'')
    parser.add_option("--monitor", dest='monitor',
                        help='eg. --monitor=\'localhost\'')
    parser.add_option("--count", dest='count',
                        help='numeric value eg. --count=3 or range --count
3,30')
    parser.add_option("--lastTime", dest='lastTime',
                    help='eg. for a range separate start & end with /
--lastTime=\'2012-09-07 07:57:33/2012-11-22 17:57:33\'')
    parser.add_option("--firstTime", dest='firstTime',
                    help='eg. --firstTime=\'2012-11-22 17:57:33\'')
    parser.add_option("--stateChange", dest='stateChange',
                    help='eg. --stateChange=\'2012-11-22 17:57:33\'')
    parser.add_option("--sort", dest='sort', default='lastTime',
                    help='the key to sort on eg. --sort=\'lastTime\'')
    parser.add_option("--dir", dest='dir', default='DESC',
                    help='the direction to sort eg. --dir=\'ASC\' or
--dir=\'DESC\'')

    (options, args) = parser.parse_args()

    # options is an object - we want the dictionary value of it
    # Some of the options need a little munging...

    option_dict = vars(options)
    if option_dict['severity']:
        option_dict['severity'] = option_dict['severity'].split(',')
    if option_dict['eventState']:
        option_dict['eventState'] = option_dict['eventState'].split(',')

    # count can either be a number or a range (in either list or tuple
format)
    #    (see $ZENHOME/Products/Zuul/facades/zepfacade.py -
createEventFilter method )
    #    but if this method gets a list it assumes there are 2 elements to
the list.
    #    We may get a list with a single value so convert it to a number
and the
    #    createEventFilter method can cope

    if option_dict['count']:
        option_dict['count'] = option_dict['count'].split(',')
        if len(option_dict['count']) == 1:
            option_dict['count'] = int(option_dict['count'][0])
    # option_dict includes the sort and dir keys (as we have defaulted them
in optparse)
    # These are not part of the filter string so we need to pop them out of
the dictionary
    #   to use separately.
    s = option_dict.pop('sort')
    d = option_dict.pop('dir')
    # Need to check these keys for sanity
    #   and provide sensible defaults otherwise
    dirlist=['ASC','DESC']
    if not d in dirlist:
        d='DESC'
    sortlist = ['severity', 'eventState', 'eventClass', 'firstTime',
'lastTime',
                    'stateChange', 'count', 'device', 'component', 'agent',
'monitor']
    if not s in sortlist:
        s='lastTime'

    #print 'options is %s \n' % (options)
    #print 'option_dict is %s \n' % (option_dict)

    events = getEventsWithJSON()
    #filter['evid'] = '000c29d9-f87b-8389-11e2-347cddf7a720'

    pp = pprint.PrettyPrinter(indent=4)
    fields = ['eventState', 'DeviceClass', 'count', 'device', 'Location',
'Systems', 'severity', 'firstTime', 'lastTime', 'summary']
    #fields = ['eventState', 'DeviceClass', 'count', 'device', 'Location',
'severity', 'firstTime', 'lastTime', 'summary']

    print 'eventState, DeviceClass, count, device, Location, Systems,
severity, firstTime, lastTime, summary'
    #print 'eventState, DeviceClass, count, device, Location, severity,
firstTime, lastTime, summary'
    out = events.get_events(filter=option_dict, sort=s, dir=d)
    for e in out['events']:
            #pp.pprint(e)
            outState=e['eventState']
            if e['DeviceClass']:
                outDeviceClass=e['DeviceClass'][0]['name']
            else: outDeviceClass=[]
            outcount=e['count']
            outdevice=e['device']['text']
            if e['Location']:
                outLocation=e['Location'][0]['name']
            else: outLocation=[]
            outSystems=[]
            for pos,val in enumerate(e['Systems']):
                sy=str(e['Systems'][pos]['name'])
                outSystems.append(sy)
            outseverity=e['severity']
            outfirstTime=e['firstTime']
            outlastTime=e['lastTime']
            outsummary=e['summary']
            print '%s,%s,%s,%s,%s,%s,%s,%s,%s,%s' % (outState,
outDeviceClass, outcount, outdevice, outLocation, outSystems, outseverity,
outfirstTime, outlastTime, outsummary)
            #print '%s,%s,%s,%s,%s,%s,%s,%s,%s' % (outState,
outDeviceClass, outcount, outdevice, outLocation, outseverity,
outfirstTime, outlastTime, outsummary)

    #print '\n totalCount is %d and asof is %s' % (out['totalCount'],
out['asof'])
```

13.2 zensendevent

Modified zensendevent to automatically retrieve local authentication parameters.

Zenoss Core 4.2.3 changed some security policies at installation time which results in zensendevent failing unless --auth parameters are determined and supplied explicitly.

```
#!/opt/zenoss/bin/python

__doc__ = """zensendevent

  Send events on a command line via XML-RPC or from a XML file.
This command can be put on any machine with Python installed, and
does not need Zope or Zenoss.

"""


import socket
from xmlrpclib import ServerProxy
from optparse import OptionParser
from xml.sax import make_parser, saxutils
from xml.sax.handler import ContentHandler

XML_RPC_PORT = 8081

sevconvert = {
    "critical" : 5,
    "error" : 4,
    "warn" : 3,
    "info" : 2,
    "debug" : 1,
    "clear" : 0
}


class ImportEventXML(ContentHandler):
    ignoredElements = set([
        'ZenossEvents', 'url', 'SourceComponent',
        'ReporterComponent', 'EventId',
        'clearid', 'eventClassMapping',
        'eventState', 'lastTime', 'firstTime', 'prodState',
        'EventSpecific', 'stateChange',
        ])
    evt = {}
    property = ''
    value = ''

    def __init__(self, serv):
        ContentHandler.__init__(self)
        self.sent = 0
        self.total = 0
        self.serv = serv

    def startElement(self, name, attrs):
        self.value = ''
        if name == 'ZenossEvent':
            self.evt = {}
        elif name == 'property':
            self.property = attrs['name']

    def characters(self, content):
        self.value += content

    def endElement(self, name):
        name = str(name)
        value = str(self.value)
        if name in self.ignoredElements:
            return

        elif name == 'property' and value and value != '|':
                self.evt[self.property] = value

        elif name in ['Systems', 'DeviceGroups']:
                if value and value != '|':
                    self.evt[name] = value

        elif name in ['eventClassKey', 'eventKey']:
                if value:
                    self.evt[name] = value

        elif name == 'severity':
                self.evt[name] = int(value)

        elif name == 'ZenossEvent':
            self.total += 1
            try:
                self.serv.sendEvent(self.evt)
                self.sent += 1
            except Exception, ex:
                print str(ex)
                print evt

        elif value:
            self.evt[name] = value

def sendXMLEvents(serv, xmlfile):
    infile = open(xmlfile)
    parser = make_parser()
    CH = ImportEventXML(serv)
    parser.setContentHandler(CH)
    try:
        parser.parse(infile)
    finally:
        infile.close()
    print "Sent %s of %s events" % (CH.sent, CH.total)


device = socket.getfqdn()
if device.endswith('.'): device = device[:-1]

parser = OptionParser(usage="usage: %prog [options] summary")
parser.add_option("-d", "--device", dest="device", default=device,
    help="device from which this event is sent, default: %default")
parser.add_option("-i", "--ipAddress", dest="ipAddress", default="",
    help="Ip from which this event was sent, default: %default")
parser.add_option("-y", "--eventkey", dest="eventkey", default="",
    help="eventKey to be used, default: %default")
parser.add_option("-p", "--component", dest="component", default="",
    help="component from which this event is sent, default: ''")
```

```
parser.add_option("-k", "--eventclasskey", dest="eventClassKey",
default="",
    help="eventClassKey for this event, default: ''")
parser.add_option("-s", "--severity", dest="severity", default="Warn",
    help="severity of this event: Critical, Error, Warn, Info, Debug,
Clear")
parser.add_option("-c", "--eventclass", dest="eventClass", default=None,
    help="event class for this event, default: ''")
parser.add_option("--monitor", dest="monitor", default="localhost",
    help="monitor from which this event came")
parser.add_option("--port", dest="port", default=XML_RPC_PORT,
    help="xmlrpc server port, default: %default")
parser.add_option("--server", dest="server", default="localhost",
    help="xmlrpc server, default: %default")
parser.add_option("--auth", dest="auth", default="admin:zenoss",
    help="xmlrpc server auth, default: %default")
parser.add_option("-o", "--other", dest="other", default=[],
    action='append',
    help="Specify other event_field=value arguments. Can be specified"
         " more than once.")
parser.add_option('-f', "--file", dest="input_file", default="",
    help="Import events from XML file.")
parser.add_option('-v', dest="show_event", default=False,
    action='store_true',
    help="Show the event data sent to Zenoss.")


opts, args = parser.parse_args()

# Hack by JC to get hubpasswd authentication into auth option
# Password is held in $ZENHOME/etc/hubpasswd in (almost) correct format
<user>:<password> \n

import os
# if auth is the default
if opts.auth == 'admin:zenoss':
  zenhome=os.environ['ZENHOME']
  # Try to access $ZENHOME/etc/hubpasswd and strip trailing newline
  try:
    pwfile=open(os.path.join(zenhome, 'etc', 'hubpasswd'), 'r')
    opts.auth=pwfile.read().rstrip()
    pwfile.close()
    print 'Extracting necessary user:password automatically \n'
  # If this fails then fall back to default and print message
  except:
    print 'Attempt to detect hubpasswd failed \n'

# End of JC hack

url = "http://%s@%s:%s" % (opts.auth, opts.server, opts.port)
serv = ServerProxy(url)

if opts.input_file:
    sendXMLEvents(serv, opts.input_file)
    import sys
    sys.exit(0)

evt = {}
if opts.severity.lower() in sevconvert:
    evt['severity'] = sevconvert[opts.severity.lower()]
else:
    parser.error('Unknown severity')
evt['summary'] = " ".join(args)
if not evt['summary']:
    parser.error('no summary supplied')
evt['device'] = opts.device
evt['component'] = opts.component
evt['ipAddress'] = opts.ipAddress
if opts.eventkey:
    evt['eventKey'] = opts.eventkey
if opts.eventClassKey:
    evt['eventClassKey'] = opts.eventClassKey
if opts.eventClass:
    evt['eventClass'] = opts.eventClass
evt['monitor'] = opts.monitor

for line in opts.other:
    try:
        field, value = line.split('=',1)
        evt[field] = value
    except:
        pass

if opts.show_event:
    from pprint import pprint
    pprint(evt)

serv.sendEvent(evt)
```

14 References

1. Zenoss Community site http://community.zenoss.org
2. Zenoss network, systems and application monitoring - commercial site - http://www.zenoss.com/
3. Zenoss documentation main page - http://community.zenoss.org/community/documentation
4. Zenoss Core 4 Administration Guide - http://community.zenoss.org/community/documentation/official_documentation/zenoss-guide
5. Zenoss Developer's Guide - http://community.zenoss.org/community/documentation/official_documentation/zenoss-dev-guide
6. Zenoss 4.2 JSON API documentation - http://community.zenoss.org/community/documentation/official_documentation/api
7. Samples of using the JSON API with bash and curl can be found at https://gist.github.com/1901884 .
8. Information on RelStorage and memcached - http://wiki.zenoss.org/RelStorage
9. Information on RabbitMQ - http://wiki.zenoss.org/Working_with_Queues
10. Script to reset RabbitMQ - https://gist.github.com/4192854
11. Information on AMQP - http://www.amqp.org/
12. Information on Lucene indexing - http://lucene.apache.org/core/
13. Information on JSON - http://www.json.org/
14. Discussion on modifying zensendevent utility on Zenoss wiki - http://wiki.zenoss.org/Zensendevent_in_Zenoss_4.2.3
15. Reference for Win32_NTLogEvent class event log severities - http://msdn.microsoft.com/en-gb/library/windows/desktop/aa394226%28v=vs.85%29.aspx
16. Information on Python regular expressions - http://docs.python.org/2/library/re.html , http://www.python.org/doc/2.5.2/lib/re-syntax.html and http://docs.python.org/dev/howto/regex.html
17. Information on protobufs - http://code.google.com/p/protobuf/
18. Information on the Python debugger (pdb) - http://docs.python.org/2/library/pdb.html
19. As a general Python reference, try “Learning Python” by Mark Lutz, published by O'Reilly

20. The MIB Browser ZenPack. Documentation and comments at http://community.zenoss.org/docs/DOC-10321 ; code from http://wiki.zenoss.org/ZenPack:MIB_Browser .
21. SNMP Requests For Comment (RFCs) - http://www.ietf.org/rfc.html
    - V1 – RFCs 1155, 1157, 1212, 1213, 1215
    - V2 – RFCs 2578, 2579, 2580, 3416, 3417, 3418
    - V3 – RFCs 2578-2580, 3416-18, 3411, 3412, 3413, 3414, 3415
22. SNMP Host Resources MIB, RFC s 1514 and 2790 - http://www.ietf.org/rfc.html
23. For the extension SNMP MIB from Informant, go to http://www.wtcs.org/informant/index.htm
24. For information on Zope TALES expressions, see http://docs.zope.org/zope2/zope2book/AppendixC.html
25. Datagram Syslog Client http://syslogserver.com for syslog Windows systems.
26. Raddle network emulation open source package - http://raddle.sourceforge.net/
27. “Zenoss 4 Event Management Workshop” available from Skills 1st Ltd, http://www.skills-1st.co.uk/products/courses/

Acknowledgements

A number of people have contributed information and advice to this project and I would like to thank them.

- Georges Reichs for the original “amazing architecture design” diagram
- Chet Luther for his awesome knowledge of Zenoss and his willingness to share that knowledge
- Andrew Kirch for initial proof-reading and some useful comments
- Andrew Findlay of Skills 1st for help with typesetting

About the author

Jane Curry has been a network and systems management technical consultant and trainer for 25 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 specialised 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 are 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, delivering Tivoli consultancy and training. Over the last 5 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 and IRC chat conversations and was made a Zenoss Master by Zenoss in February 2009