ZenPack SDK Documentation

Release 2.0.9

Zenoss, Inc.

Jan 26, 2018

Contents

The ZenPack SDK is a collection of development tools and documentation that you can use to extend Zenoss' functionality.

ZenPacks are a plugin mechanism for Zenoss. Most commonly they're used to extend Zenoss to monitor new types of targets. We developed zenpacklib to simplify the process of creating custom ZenPacks.

CHAPTER 1

What is zenpacklib?

zenpacklib is a Python library that makes building common types of ZenPacks simpler, faster, more consistent, and more accurate.

CHAPTER 2

What does zenpacklib do?

Specifically zenpacklib allows all of the following to be described in YAML, and extended in Python only if necessary.

- zProperties (a.k.a. Configuration Properties)
- Device Classes
- Monitoring Templates
- New Device and Component Types
- Relationships between Device and Component Types
- Event Classes
- Process Classes
- Device Link Providers
- Impact Triggers

It is this combination of declarative YAML and imperative Python extension that allows zenpacklib to make easy things easy and hard things possible.

CHAPTER 3

Who should use zenpacklib?

You should consider using zenpacklib if any of the following statements apply to you.

- Your ZenPack will only contain monitoring templates, but you prefer creating YAML files over creating monitoring templates by clicking around the Zenoss web interface.
- Your ZenPack needs to add zProperties.
- Your ZenPack needs to add new device or component types and relationships between them.

You should even consider using zenpacklib if you are an experience ZenPack developer and already know how to create new device and component types. You will find that the amount of boilerplate code you need to write is drastically reduced, if not elimited, by using zenpacklib. You will still have all of the power of Python to extend upon the functionality provided by zenpacklib.

If your ZenPack only consists of configuration you can create and add to a ZenPack using the Zenoss web interface, and you're more comfortable clicking through the web interface than create a YAML file, you probably should use Zenoss' built-in capabilities instead of zenpacklib.

CHAPTER 4

What about some examples?

The following example shows an example of adding new zProperties. Note the special *DEFAULTS* entry. You'll find that this is supported in many places as a way to set default properties for all other entries in a section. In this case it will set *category* to ACME Widgeter for the zWidgeterEnable and zWidgeterInterval zProperties.

```
name: ZenPacks.acme.Widgeter

zProperties:
  DEFAULTS:
    category: ACME Widgeter

  zWidgeterEnable:
    type: boolean
    default: true

  zWidgeterInterval:
    type: string
    default: 300
```

Extending upon that example we can add a device class and monitoring template complete with a datasource, threshold and graph.

```
device_classes:
  /Server/ACME/Widgeter:
    templates:
      Device:
        description: ACME Widgeter monitoring.
        targetPythonClass: ZenPacks.acme.Widgeter.Widgeter

        datasources:
          status:
            type: COMMAND
            parser: Nagios
            commandTemplate: "echo OK|available=1"

            datapoints:
```

```
            available:
              rrdtype: GAUGE
              rrdmin: 0
              rrdmax: 1

        thresholds:
          unavailable:
            dsnames: [status_available]
              eventClass: /Status
              severity: Critical
              minval: 1

        graphs:
          Availability:
            units: percent
            miny: 0
            maxy: 100

            graphpoints:
              Availability:
                dpName: status_available
                rpn: 100,*
                format: "%7.2lf%%"
                lineType: AREA
```

Finally we can add a new device type, component type and relationship between them.

```
classes:
  Widgeter:
    base: [zenpacklib.Device]
    meta_type: ACMEWidgeter

  Widget:
    base: [zenpacklib.Component]
    meta_type: ACMEWidget
    properties:
      flavor:
        label: Flavor
        type: string

class_relationships:
  - Widgeter 1:MC Widget
```

4.1 Development Environment

The process of developing a ZenPack can be made much faster and easier by starting with a good development environment. A good development environment should do the following.

- Isolate your changes from other users changes or a production systems.
- Allow you to quickly see the result of changes for faster iteration.
- Allow you to easily troubleshoot when changes don't have the desired effect.

The following recommendations provide a good starting point for anyone wanting to do ZenPack development on Zenoss 5.

4.1.1 Installing

You must have Zenoss installed to develop ZenPacks. I recommend starting by creating a dedicated Zenoss installation for your own development. Start by following the normal installation instructions available at docs.zenoss.com with the following notes.

- System requirements for development can be lower. See below.
- A single-host installation should be used for development.
- Any supported operating system can be used. This guide covers Enterprise Linux 5.
- Verify that Zenoss has been deployed and its web interface is working.

System Requirements

A development system will usually have system requirements lower than those of a production Zenoss system. This is because it likely won't be storing as much data, supporting as many web users, or even performing continual monitoring. Your development system should have at least the following resources.

- 4 CPU cores.
- 20 GB memory.
- 75 GB storage.

4.1.2 Configuring the System

We'll want to make the following changes to our development system to make ourselves more productive. Each will be detailed in the following sections.

- Add a *zenoss* user to the host that matches the same user in containers.
- Create a */z* directory on the host to share with containers.
- Configure serviced to automatically share */z* with all containers.

First make sure that you have either the Zenoss.core or Zenoss.resmgr service deployed and running, and that you're able to login to its web interface. All commands in the following sections should be run as root or through sudo unless otherwise noted.

Add a "zenoss" User

To make development easier we're going to be sharing files between the host and docker containers running on the host. We can create a *zenoss* user on the host that matches the UID (user ID) and GID (group ID) of the zenoss user in the containers to avoid having to worry about permissions problems with those shared files.

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

It'll also be useful for our *zenoss* user to be able to use sudo and docker commands. We can allow that by adding the user to the *wheel* and *docker* groups respectively.

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

Helper Aliases and Functions

A lot of the commands you'll use while developing ZenPacks must be executed inside a Zenoss container. Constantly having to attach to the Zope container, switch to the zenoss user, execute the command, and exit the container is tedious. With a few additions to our zenoss user's `.bashrc`, we can eliminate those tedious steps.

Add the following lines to the end of `/home/zenoss/.bashrc`.

```
# ZenPack development helpers.
alias zope='serviced service attach zope su zenoss -l'
alias zenhub='serviced service attach zenhub su zenoss -l'
z () { serviced service attach zope su zenoss -l -c "cd /z;$*"; }
zenbatchload () { z zenbatchload $*; }
zendisc () { z zendisc $*; }
zendmd () { z zendmd $*; }
zenmib () { z zenmib $*; }
zenmodeler () { z zenmodeler $*; }
zenpack () { z zenpack $*; }
zenpacklib () { z zenpacklib $*; }
zenpython () { z zenpython $*; }
```

Next time you login as the zenoss user, you'll have new commands available.

- *zope*: Opens the zenoss user shell in the running Zope container.
- *zenhub*: Opens the zenoss user shell in the running zenhub container.
- *z*: Run any command as zenoss user in running Zope container.
  - *zendisc*: Discovers new devices.
  - *zendmd*: Opens zendmd console.
  - *zenmib*: Import SNMP MIB files.
  - *zenmodeler*: Remodels existing devices.
  - *zenpack*: For installing and removing ZenPacks.
  - *zenpacklib*: Runs zenpacklib commands.

Authenticating as "zenoss"

You will likely want to login to the system as the *zenoss* user after getting the system configured. That way you won't have to switch (su) to the user to make sure files you create have the right permissions. I recommend either setting a password for the user, or adding your public key to the user's *authorized_keys* file to support this.

Optionally set the *zenoss* user's password:

```
passwd zenoss
```

Optionally add your SSH public key to the *zenoss* user's *authorized_keys* file to login without a password:

```
mkdir -p /home/zenoss/.ssh
chmod 700 /home/zenoss/.ssh
cat >> /home/zenoss/.ssh/authorized_keys
... paste your public key, enter, ctrl-D ...
chmod 600 /home/zenoss/.ssh/authorized_keys
chown -R zenoss:zenoss /home/zenoss/.ssh
```

Create a "/z" Directory

Now we can create a directory to share that the zenoss user on the host and in the container will be able to use. The specific path of this directory isn't particularly important, but I like using */z* because it's as short as possible.

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

Mount "/z" Into All Containers

Now we can configure serviced to automatically share (bind mount) the host's /z directory into every container it starts. This will let us use the same files on the host and in containers using the exact same path.

Edit */etc/default/serviced*. Find the existing *SERVICED_OPTS* line. It will likely be commented out (with a #) and look like the following.

```
# Arbitrary serviced daemon args
# SERVICED_OPTS=
```

Uncomment it, and add the bind mount configuration as follows.

```
# Arbitrary serviced daemon args
SERVICED_OPTS="--mount *,/z,/z"
```

You must then restart serviced.

```
systemctl restart serviced
```

Test "/z" Sharing

Now you can verify that both the host and containers can read and write files in */z*.

On the host:

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

4.1.3 Configuring Zenoss Services

There are some optional tweaks you can make to Zenoss service definitions to make development faster and easier. We'll go through the following here.

- Reducing Zope to a single instance so breakpoints can be used.
- Setting unnecessary services to not automatically launch.

Reducing Zope to a Single Instance

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

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

```
serviced service edit Zope
```

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

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

Run the following command to restart Zope and affect the change.

```
serviced service restart Zope
```

Setting Services to Manual Launch

The default Zenoss service templates are configured to launch almost all services they contain automatically. When developing ZenPacks it's usually unnecessary to have all of the collector process such as zenping running. These services are consuming memory, CPU, and may need to be restarted frequently as you're making code changes. To avoid all of that you can configure some services to not launch automatically when you start the service.

Run the following command to edit zenping's service definition to make it not automatically launch.

```
serviced service edit zenping
```

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

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

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

Here's the base list of services you should consider setting to the manual launch mode.

- zencommand
- zenjmx
- zenmail (defaults to manual)
- zenmodeler
- zenperfsnmp
- zenping
- zenpop3 (defaults to manual)
- zenprocess

- zenpython
- zenstatus
- zensyslog
- zentrap

Here are some additional services you'll find on Zenoss.resmgr only that could be set to manual.

- zenjserver
- zenpropertymonitor
- zenucsevents
- zenvsphere

You may have more or less services on your system depending on what ZenPacks are installed. The rule of thumb should be that any services under the *Collection* tree can be set to manual except for *zenhub*, *MetricShipper*, *collectorredis*, and *zminion*.

4.2 Getting Started

The first thing we'll need to do is install the ZenPackLib ZenPack into our development system. This is done in the same way as it would be in any Zenoss system.

The ZenPackLib ZenPack provides the zenpacklib command line tool, which will allow us to create ZenPacks.

Note: This tutorial assumes your system is already setup as described in *Development Environment* and *Getting Started*.

4.2.1 Installing ZenPackLib

The latest version of ZenPackLib can be downloaded from its entry in the ZenPack Catalog. The following commands show how you would download and install version 2.0.9.

Note: From here on all command should be run as the *zenoss* user on the host unless otherwise noted. If you don't login to the host as the *zenoss* user, use `su - zenoss` to get a login shell.

```
cd /tmp
wget http://wiki.zenoss.org/download/zenpacks/ZenPacks.zenoss.ZenPackLib/2.0.9/
↪ZenPacks.zenoss.ZenPackLib-2.0.9.egg
serviced service run zope zenpack-manager install ZenPacks.zenoss.ZenPackLib-2.0.9.egg
```

Executing *zenpacklib* requires a live Zenoss environment. Always executing it as the *zenoss* user in your Zope container is a good way to have the right environment setup. The following commands demonstrate how to do this.

```
serviced service attach zope # attach to zope container
su - zenoss # become zenoss user in zope container
zenpacklib --version
exit # back to root in container
exit # back to host
```

These five commands can be reduced to the following single command if you setup the helper aliases and functions your `.bashrc` recommended in *Helper Aliases and Functions*.

```
zenpacklib --version
```

4.2.2 Creating a ZenPack

There are two ways to get started with zenpacklib. You can either use it to create a new ZenPack from the command line, or you can update an existing ZenPack to use it. We'll start by creating a ZenPack from the command line.

Run the following commands to create a new ZenPack.

```
# Create ZenPacks in /z so the host and containers can access them.
cd /z
zenpacklib --create ZenPacks.acme.Widgeter
```

This will print several lines to let you know what has been created. Note that the ZenPack's source directory has been created, but it has not yet been installed.

```
Creating source directory for ZenPacks.acme.Widgeter:
  - making directory: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter
  - creating file: ZenPacks.acme.Widgeter/setup.py
  - creating file: ZenPacks.acme.Widgeter/MANIFEST.in
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/__init__.py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/__init__.py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/datasources/__init__.
↪py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/thresholds/__init__.
↪py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/parsers/__init__.py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/migrate/__init__.py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/resources/__init__.py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/modeler/__init__.py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/tests/__init__.py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/libexec/__init__.py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/modeler/plugins/__
↪init__.py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/lib/__init__.py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/__init__.py
  - creating file: ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/zenpack.yaml
```

Now let's take a look at *zenpack.yaml*. This is the file that will define a large part of what our ZenPack is.

```
name: ZenPacks.acme.Widgeter
```

Add Monitoring

Let's add a device class and a monitoring template to our ZenPack. Change *zenpack.yaml* to contain the following:

```
name: ZenPacks.acme.Widgeter

device_classes:
  /Server/ACME/Widgeter:
    zProperties:
      zDeviceTemplates:
```

```
        - WidgeterHealth

    templates:
      WidgeterHealth:
        description: ACME Widgeter monitoring.

        datasources:
          health:
            type: COMMAND
            parser: Nagios
            commandTemplate: "echo OK|percent=100"

            datapoints:
              percent:
                rrdtype: GAUGE
                rrdmin: 0
                rrdmax: 100

        thresholds:
          unhealthy:
            dsnames: [health_percent]
            eventClass: /Status
            severity: Warning
            minval: 90

        graphs:
          Health:
            units: percent
            miny: 0
            maxy: 0

            graphpoints:
              Health:
                dpName: health_percent
                format: "%7.2lf%%"
```

Check for Correctness

Now that we have a more interesting *zenpack.yaml*, let's have zenpacklib check that it's correct. This can be done using the *lint* command.

```
zenpacklib --lint ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter/zenpack.yaml
```

Lint will print information about errors it finds in the YAML file. If nothing is printed, lint thinks the YAML is correct.

4.2.3 Installing a ZenPack

Now that we've created a ZenPack called *ZenPacks.acme.Widgeter* in */z*, we can install it into our Zenoss system by running the following command.

```
z zenpack --link --install ZenPacks.acme.Widgeter
```

Zenoss must be restarted anytime a new ZenPack is installed. A full restart of the entire system can be performed by running one of the following commands depending on what distribution of Zenoss you have installed..

```
serviced service restart Zenoss.core
serviced service restart Zenoss.resmgr
```

Technically it isn't necessary to restart everything. A lot of the infrastructure services don't use ZenPack code. The following is a smaller list of services that you're likely to need to restart after installing and modifying ZenPacks during development.

- Zope
- zenhub
- zeneventd
- zenactiond
- zenjobs

The following command will quickly restart just these services.

```
echo Zope zenhub zeneventd zenactiond zenjobs | xargs -n1 serviced service restart
```

4.2.4 What Next?

You can either start with some *Tutorials* or jump right into the *YAML Reference*.

4.3 Tutorials

The following tutorials provide step-by-step instructions on using zenpacklib to extend Zenoss is common ways.

- *Monitoring an SNMP Device*

  This tutorial starts with the very basics of creating a ZenPack through Zenoss' web interface and adding configuration to it. Then it progresses to extending the model, creating a modeler plugin, monitoring components, and then to event management with SNMP traps as an example.

  This is most likely the first tutorial you should do.

- *Monitoring an HTTP API*

  In this tutorial the basics are skipped and we jump right into extending the model, modeling a custom HTTP API, and monitoring the same API using the zenpython daemon provided by the PythonCollector ZenPack.

  This is a more advanced tutorial that contains more advanced Python code.

4.3.1 Prerequisites

To follow the steps in these tutorials you will need to have access to the following:

- A Linux server with Zenoss installed on it. This should not be a Zenoss server you care about. We will break things. You can download Zenoss from the Zenoss download site.
- An SSH client to connect to your Zenoss server. PuTTY works well for Windows, ssh from the command line works well for Mac and Linux.
- These tutorials.

You may need experience in the following areas to more easily follow these tutorials.

- Zenoss: Familiarity administration and configuration.
- Linux: Ability to move around the file system, manage files and run commands.
- Programming: Any type of programming or scripting experience will help.

4.4 Monitoring an SNMP Device

The following sections will describe a common approach to monitoring an SNMP- enabled device. We'll start with the basics that can be done without writing a line of code, and then move on to more sophisticated capabilities.

For purposes of this guide we'll be building a ZenPack to support a NetBotz environmental sensor device. This device has a variety of sensors that monitor temperature, humidity, dew point, audio levels and air flow.

Note: This tutorial assumes your system is already setup as described in *Development Environment* and *Getting Started*.

4.4.1 SNMP Tools

To configure Zenoss to monitor a device using SNMP, it is necessary to understand a bit about SNMP and the specific capabilities of your device. This section will walk you through using Net-SNMP, smidump, and snmpsim to learn about SNMP and your device.

Installing Net-SNMP

In the SNMP world the client is referred to as a *manager* and the server is referred to as the *agent*. Net-SNMP is software that provides both an *agent* that's used in all sorts of devices, and many command line tools that act as *manager*. We're only going to need the command line tools, so we'll be installing the *net-snmp-utils* package.

You can install Net-SNMP's command line tools by running the following command as root.

```
yum -y install net-snmp-utils
```

Installing libsmi

`smidump` is a useful command line tool for converting MIBs to other formats. We'll be using it later in this tutorial to research what a MIB provides.

Install *smidump* by installing the *libsmi* package with the following command.

```
yum -y install libsmi
```

Installing the SNMP Simulator

When developing a ZenPack to monitor an SNMP-enabled device it can often be useful to simulate the device's SNMP agent. There are many tools available to do this. For this guide we will be using the free snmpsim because it's easy to install on our Zenoss host.

1. Run the following commands as root to install *snmpsim*:

```
yum -y groupinstall "Development Tools"
yum -y install python-devel
easy_install snmpsim
mkdir -p /usr/share/snmpsim/data
mkdir -p /var/run/snmpsim
useradd -U snmpsim
chown snmpsim:snmpsim /var/run/snmpsim
```

2. Run the following command as root to install a NetBotz recording.

```
wget https://goo.gl/OJe2vL -O /usr/share/snmpsim/data/public.snmprec
```

3. Run the following command as root to run snmpsim.

```
snmpsimd.py \
  --process-user=snmpsim \
  --process-group=snmpsim \
  --agent-udpv4-endpoint=172.17.0.1:161 \
  --daemonize
```

4. Test the simulator with the following *snmpwalk* command.

```
snmpwalk -v2c -c public 172.17.0.1 sysDescr
```

You should see the following output.

```
SNMPv2-MIB::sysDescr.0 = STRING: Linux Netbotz01 2.4.26 #1 Wed Oct 31 18:09:53␣
↪CDT 2007 ppc
```

Using snmpwalk

The tool you'll be using most often is called *snmpwalk*. All SNMP values are arranged on a tree, and snmpwalk allows you to query for all data under a given branch of that tree. See the following example that walks all values under the *system* branch.

Run the snmpwalk command.

```
snmpwalk -v2c -c public 172.17.0.1 system
```

```
SNMPv2-MIB::sysDescr.0 = STRING: Linux Netbotz01 2.4.26 #1 Wed Oct 31 18:09:53 CDT␣
↪2007 ppc
SNMPv2-MIB::sysObjectID.0 = OID: SNMPv2-SMI::enterprises.5528.100.20.10.2006
DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (7275488) 20:12:34.88
SNMPv2-MIB::sysContact.0 = STRING: unknown
SNMPv2-MIB::sysName.0 = STRING: Netbotz01
SNMPv2-MIB::sysLocation.0 = STRING: Z1 Rack02 NetBotz01
```

We can see that this NetBotz device seems to be based on Linux and that we have some more-or-less useful information about the device's name, location and administrative contact.

The second line with the *sysObjectID* has an unusual value. It's a partially decoded OID. It isn't decoded enough for us to know what it means. SNMP tools including Net-SNMP use MIB files to decode these OIDs into human readable values. In fact, we're only able to read most of the output above because Net- SNMP has a set of standard MIBs enabled by default.

Let's run that command again, but use the `-On` flag to tell snmpwalk not to decode OIDs.

```
snmpwalk -v2c -c public -On 172.17.0.1 system
```

```
.1.3.6.1.2.1.1.1.0 = STRING: Linux Netbotz01 2.4.26 #1 Wed Oct 31 18:09:53 CDT 2007␣
↪ppc
.1.3.6.1.2.1.1.2.0 = OID: .1.3.6.1.4.1.5528.100.20.10.2006
.1.3.6.1.2.1.1.3.0 = Timeticks: (7275488) 20:12:34.88
.1.3.6.1.2.1.1.4.0 = STRING: unknown
.1.3.6.1.2.1.1.5.0 = STRING: Netbotz01
.1.3.6.1.2.1.1.6.0 = STRING: Z1 Rack02 NetBotz01
```

While this data is mostly less valuable than the decoded version above, it's more useful for a single reason. We can take that `.1.3.6.1.4.1.5528.100.20.10.2006` value and search the Internet for it. It's best to remove the leading `.` and search for `1.3.6.1.4.1.5528.100.20.10.2006` instead. This should lead you to the *NETBOTZV2-MIB* which will contain the decoding information we need to learn more about this device.

Run the following command to download *NETBOTZV2-MIB.mib* into */usr/share/snmp/mibs/*.

```
wget https://goo.gl/0v4Kti -O /usr/share/snmp/mibs/NETBOTZV2-MIB.mib
```

Now we can run the original snmpwalk command again with the addition of the `-m all` option. This option tells Net-SNMP tools to use all MIBs.

```
snmpwalk -v2c -c public -m all 172.17.0.1 system
```

```
SNMPv2-MIB::sysDescr.0 = STRING: Linux Netbotz01 2.4.26 #1 Wed Oct 31 18:09:53 CDT␣
↪2007 ppc
SNMPv2-MIB::sysObjectID.0 = OID: NETBOTZV2-MIB::netBotz420ERack
DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (7275488) 20:12:34.88
SNMPv2-MIB::sysContact.0 = STRING: unknown
SNMPv2-MIB::sysName.0 = STRING: Netbotz01
SNMPv2-MIB::sysLocation.0 = STRING: Z1 Rack02 NetBotz01
```

Now we can see that the sysObjectID is NETBOTZV2-MIB::netBotz420ERack. This gives us a better idea of exactly what kind of device it is. We'll see that as we look deeper into this device that the NETBOTZV2-MIB will prove more useful.

Default Net-SNMP Options

The snmpwalk usage showed three primary command line options that we tend to use most of the time. Net-SNMP allows you to specify these in a configuration file so you don't have to type them every time. I recommend doing this.

Create `/etc/snmp/snmp.conf` and add the following lines.

```
defVersion v2c
defCommunity public
mibs ALL
```

These lines add the following equivalent command line options respectively:

- *-v2c*
- *-c public*
- *-m all*

So now we can run this command.

```
snmpwalk 172.17.0.1 sysObjectID
```

And get the same results as if we ran.

```
snmpwalk -v2c -c public -m all 172.17.0.1 sysObjectID
```

This will save you time while developing this ZenPack, and others in the future.

Decoding and Encoding OIDs

Often it can be useful to turn numeric OIDs into their human-readable equivalent, or vice-versa. The *snmptranslate* command can be used for this. See the following examples.

OID to name:

```
# snmptranslate .1.3.6.1.4.1.5528.100.20.10.2006
NETBOTZV2-MIB::netBotz420ERack
```

Name to OID:

```
# snmptranslate -On NETBOTZV2-MIB::netBotz420ERack
.1.3.6.1.4.1.5528.100.20.10.2006
```

4.4.2 Device Monitoring

This section will cover monitoring device-level metrics using SNMP. This requires no code, and you can find instructions for doing it in the normal Zenoss documentation. However, there are some extra considerations and steps required to package your configuration in a ZenPack.

Note: Commands in this section should be run on the host as the *zenoss* user unless otherwise noted.

Create a ZenPack

The first step will be to create and install a ZenPack to contain all of the NetBotz monitoring functionality we're going to build.

```
cd /z
zenpacklib --create ZenPacks.training.NetBotz
zenpack --link --install ZenPacks.training.NetBotz
```

We should also restart at least *Zope* after installing the ZenPack so that we can work with it in the web interface.

```
serviced service restart Zope
```

Create a Device Class

To support our new NetBotz environmental sensor device we will want to create a new device class. This will give us full control over how these types of devices are modeled and monitored. Use the following steps to add a new device class.

1. Navigate to the *Infrastructure* view.

2. Select the root of the *DEVICES* tree.
3. Click the + button at the bottom of the list to add a new organizer.
4. Set the *Name* to `NetBotz` then click *SUBMIT*.

   The new *NetBotz* device will now be selected. We'll want to check on some important configuration properties using the following steps.

Set Device Class Properties

1. Click the *DETAILS* button at the top of the list.
2. Select *Modeler Plugins*.

   The modeler plugins are what model information about the device. We should see a list something like the following. This list is being acquired from the root (/ or /Devices) device class.

   - zenoss.snmp.NewDeviceMap
   - zenoss.snmp.DeviceMap
   - zenoss.snmp.InterfaceMap
   - zenoss.snmp.RouteMap

   This is a good basic list that uses standard MIB-2 support that works with most SNMP-enabled devices. However, it's unlikely that we care about the routing table on our environmental sensors, so there's no reason to model it.
3. Remove *zenoss.snmp.RouteMap* from the list.
4. Click *Save*.

   Now you can see the *Path* at which our modeler plugin configuration is set has changed from */* to */NetBotz*. This allows us to know that regardless of what the user sets their default modeler plugins to in the system that NetBotz appliances will be collected using the set of modeler plugins we configure here.
5. Select *Configuration Properties* from the left navigation pane.

   There are a lot of configuration properties. You don't have to worry about understanding all of them. However, some will be critical to monitoring NetBotz appliances. We know that we're going to be using SNMP so let's make sure that it's enabled.
6. Find the *zSnmpMonitorIgnore* property and set its value to true.
7. Now set the value for *zSnmpMonitorIgnore* to false.

   The reason for flipping the value back to it's original value is the same as saving the list of modeler plugins. While the system default is to have SNMP monitoring enabled, a user could easily disable it globally and cause our NetBotz monitoring to stop working. By flipping the value, we've set it locally within our device class and will prevent changes in the global default from affecting the operation of our ZenPack.

Add Device Class to ZenPack

Now that we've setup the NetBotz device class, it's time to add it to our ZenPack using the following steps. Adding a device class to your ZenPack causes all settings in that device class to be added to the ZenPack. This includes modeler plugin configuration, configuration property values and monitoring templates.

1. Make sure that you have the NetBotz device class selected in the *Infrastructure* view.
2. Choose *Add to ZenPack* from the gear menu in the bottom-left.

3. Select your NetBotz ZenPack then click *SUBMIT*.

Add a NetBotz Device

This would be a great time to add a NetBotz device to our new */NetBotz* device class. We haven't done anything in the way of customer monitoring. It can often be helpful to see what Zenoss' default settings will return for a device before we start adding features.

You can add a the device through the web interface, or on the command line using zendisc as follows:

```
z zendisc run --deviceclass=/NetBotz --device=172.17.0.1
```

Note: I'll often use *zendisc* from the command line only because the *zenjobs* daemon must be running to add jobs from the web interface. The *zenjobs* daemon is not required to be running when adding devices using *zendisc* from the command line because it immediately adds the device instead of scheduling a job to do it.

You should now see that Zenoss was able to model some information about the device even though we haven't added any custom monitoring. For example, you should see the following on the device in the web interface.

- Overview
  - Hardware Manufacturer: NetBotz
  - Hardware Model: .1.3.6.1.4.1.5528.100.20.10.2006
  - OS Manufacturer: Unknown
  - OS Model: Linux 2.4.26
- Components - Interfaces: 2 - eth0 and lo

If we were running the *zenperfsnmp* daemon, we'd start to see that Zenoss was monitoring the uptime and interface metrics after about 10 minutes.

Configure Monitoring Templates

Before adding a monitoring template we should look to see what monitoring templates are already being used in our new device class.

Validate Existing Monitoring Templates

We created the NetBotz device class directly within the root (or /) device class. This means that we'll be inheriting the system default monitoring templates and binding. Use the following steps to validate this.

1. Select the *NetBotz* device class in the *Infrastructure* view.
2. Choose *Bind Templates* from the gear menu in the bottom-left.

   You should only see `Device (/Devices)` in the *Selected* box. Depending on what other ZenPacks you have installed in the system you may see zero or more other templates listed in the *Available* box.

   Now we investigate what this system default *Device* monitoring template does.
3. Click *CANCEL* on the *Bind Templates* dialog.
4. Click the *DETAILS* button at the top of the device class tree.

5. Select `Device (/Devices)` under *Monitoring Templates*.

   You'll see that there's a single SNMP datasource named sysUpTime. If you expand this datasource you will see that it contains a single datapoint which is also named sysUpTime. This single datapoint named the same as its containing datasource is always what you'll see for SNMP datasources. The reason for having the conceptual separation between datasources and datapoints is that other types of datasources such as COMMAND are capable of returning multiple datapoints.

   You'll note that this monitoring template has no threshold or graphs defined. This is unusual. Typically there'd be no reason to collect data that you weren't going to either threshold against or show in a graph. The *sysUpTime* datapoint is an exception because it is shown on a device's *Overview* page in the *Uptime* field and therefore doesn't need to be graphed.

Let's use `snmpwalk` to check if our NetBotz device supports *sysUpTime*. The OID listed for the *sysUpTime* datasource is `1.3.6.1.2.1.1.3.0` so we run the following command:

```
# snmpwalk 172.17.0.1 1.3.6.1.2.1.1.3.0
DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (7275488) 20:12:34.88
```

This response indicates that the NetBotz device does support the *sysUpTime* OID. This is a mandatory field for SNMP devices to support so you will be able to get it in almost all cases.

Add a Monitoring Template

Now that we've validated that the existing *Device* monitoring template will work on our NetBotz device, we'll add another monitoring template to collect additional information.

Note: We could create a local copy of the *Device* monitoring template in the NetBotz device class and add new datasources, thresholds and graphs to it. However, this prevents us from taking advantage of changes made to the system default *Device* template in the future.

Follow these steps to create and bind a new template to the NetBotz device class.

1. Navigate to *Advanced* -> *Monitoring Templates*.
2. Click the + button in the bottom-left to add a template.

1. Set the *Name* field to `NetBotzDevice`.
2. Set the *Template Path* field to */NetBotz*.
3. Click *SUBMIT*.
4. Bind this template to the *NetBotz* device class.

1. Navigate to *Infrastructure*.
2. Select the *NetBotz* device class.
3. Choose *Bind Templates* from the gear menu in the bottom-left.
4. Move *NetBotzDevice* from available to selected.
5. Click *SAVE*.

Build the Monitoring Template

Now that we've created the *NetBotzDevice* monitoring template and bound it to the *NetBotz* device class, we need to add datasources, thresholds and graphs. We don't already know what might be interesting to graph for each NetBotz device, so let's go exploring with `snmpwalk`:

```
# snmpwalk 172.17.0.1 .1.3
SNMPv2-MIB::sysDescr.0 = STRING: Linux Netbotz01 2.4.26 #1 Wed Oct 31 18:09:53 CDT␣
↪2007 ppc
SNMPv2-MIB::sysObjectID.0 = OID: NETBOTZV2-MIB::netBotz420ERack
... lots of lines removed ...
SNMPv2-MIB::snmpInTotalReqVars.0 = Counter32: 4406
... and more removed ...
```

There isn't much of interest to collect at the device level. By "device-level" I mean values that only have a single instance for the device. Typical examples of these kinds of metrics would be memory utilization or the previous sysUpTime example. With SNMP it can be easy to find these kinds of single-instance values because their OID ends in `.0` as in `SNMPv2-MIB::snmpInTotalReqVars.0`.

Note: We'll get into monitoring multi-instance values in the component monitoring section.

Since there aren't any extremely interesting single-instance values to collect, we'll collect that snmpInTotalReqVars for illustrative purposes. We'll need to know the numeric OID for this value. Use snmptranslate to find it:

```
# snmptranslate -On SNMPv2-MIB::snmpInTotalReqVars.0
.1.3.6.1.2.1.11.13.0
```

Add an SNMP Datasource

Use the steps below to add an SNMP datasource for snmpInTotalReqVars.

1. Navigate to *Advanced -> Monitoring Templates*.
2. Expand *NetBotzDevice* then select */NetBotz*.
3. Click + on the *Data Sources* pane.
    1. Set *Name* to `snmpInTotalReqVars`
    2. Set *Type* to `SNMP`
    3. Click *SUBMIT*.

    Note: Best practice is to name SNMP datasources according to the name of the OID that's being polled from the MIB.

4. Double-click to edit the *snmpInTotalReqVars* datasource.
    1. Set *OID* to `1.3.6.1.2.1.11.13.0`
    2. Click *SAVE*.

> **Warning:** A common mistake to make when setting the OID in a device-level template is to omit the trailing `.0`. The reason this is common is that if you were using the MIB as reference instead of the snmpwalk above, you'd see that the OID for SNMPv2-MIB::snmpInTotalReqVars was 1.3.6.1.2.1.11.13 instead of 1.3.6.1.2.1.11.13.0. Due to this, I always recommend using snmpwalk to verify exactly what OID you should be polling.
>
> While Zenoss will accept the OID with the leading `.`, I recommend omitting it. It isn't necessary.

We now have a choice about how we want to handle the value that comes back from polling that OID. As you can see above in the snmpwalk output, it is a *Counter32* type. This means that it starts at 0 and, in this case, increments each time an SNMP variable is requested. The most common way to handle counters like these is as a delta. It's not very interesting to know how many variables have been requested since the device last rebooted, but it might be interesting to know how many variables are requested per second.

The default type for a datapoint is *GAUGE* which would record the actual value you see in the snmpwalk output. If we'd rather monitor the rate of requests, we'd change the datapoint type to *DERIVE* using the following steps.

1. Double-click on the *snmpInTotalReqVars.snmpInTotalReqVars* datapoint.

   You may need to expand the *snmpInTotalReqVars* datasource first.

   1. Set *RRD Type* to *DERIVE*
   2. Set *RRD Minimum* to `0`
   3. Click *SAVE*.

> **Warning:** It is very important to always set the *RRD Minimum* to `0` for *DERIVE* type datapoints. If you fail to do this, you will get large negative spikes in your data anytime the device reboots or the counter resets for any other reason.
>
> The only time you wouldn't set a minimum of 0 is when the value you're monitoring can increase and decrease and you're interested in tracking rates of negative change as well as rates of positive change.

Add a Threshold

Now we can add a threshold to our monitoring template. Let's say we want to raise a warning event anytime the rate of SNMP variable requests exceeds 10 per second. This can be done with the following steps.

1. Click + on the *Thresholds* pane.

1. Set *Name* to `high SNMP variable request rate`
2. Set *Type* to *MinMaxThreshold*
3. Click *ADD*.

2. Double-click to edit the *high SNMP variable request rate* threshold.

1. Drag the *snmpInTotalReqVars* datapoint to the left box.
2. Set *Severity* to *Warning*
3. Set *Maximum Value* to `10`
4. Set *Event Class* to */Perf/Snmp*
5. Click *SAVE*.

Note: A *MinMaxThreshold* can be used to handle a variety of conditions including over a maximum value, under a minimum value, outside a defined range or within a defined range. See the regular Zenoss documentation for how to use each of these options.

Add a Graph Definition

Now we'll add a graph so the user will be able to see the trend of SNMP variable requests per second over time. This can be done with the following steps.

1. Click + on the *Graph Definitions* pane.

1. Set *Name* to `SNMP Rates`

2. Click *SUBMIT*.

2. Double-click to edit the *SNMP Rates* graph definition.

    1. Set *Units* to `requests/sec`

    2. Set *Min Y* to `0`

    3. Click *SUBMIT*.

    Note: Always set the units for your graph. Also set the minimum Y axis and maximum Y axis values if you know what the possible limits are for the data. This results in graphs that are easier to read.

    The format field should also be tweaked to best present the kind of data that is to be graphed. You can find more information on what can be used in the format field in the *RRDtool rrdgraph_graph* documentation under the *PRINT* section.

3. Select the *SNMP Rates* graph definition.

4. Choose *Manage Graph Points* from the gear menu.

1. Choose *Data Point* from the + menu.

2. Set *Data Point* to *snmpInTotalReqVars*

3. Check *Include Related Thresholds*

4. Click *SUBMIT*

5. Double-click to edit the *snmpInTotalReqVars* graph point.

    1. Set *Name* to `Variables`

    2. Click *SAVE*.

    Note: The name of a graph point is what is displayed for it in the graph legend. You should always choose something short that describes the data and makes sense in context of the units chosen above.

Test Monitoring Template

The quick way to check if we've been successful in creating and binding our monitoring template is to navigate to the NetBotz device we added to the system and verify that we see our *NetBotzDevice (/NetBotz)* monitoring template listed at the bottom of the device's left navigation pane.

Now we can test that our datasource will be collected by running the following command to do a single collection of the NetBotz device:

```
z zenperfsnmp run -v10 --device=Netbotz01
```

We can look through the output to see what zenperfsnmp does. I usually look for any lines that contain *MetricWriter*. These lines will show the collected data being published to the database. If data isn't collected, these lines won't be present. Because of this you might run the following command instead to only see lines that contain this pattern:

```
z zenperfsnmp run -v10 --device=Netbotz01 | grep "MetricWriter"
```

We should see about 18 datapoints being published. You'll see two of *eventQueueLength*, *sysUpTime*, 14 interface datapoints and our custom *snmpInTotalReqVars* in there somewhere.

Export the ZenPack

Now that we've created a ZenPack and added some configuration to it, we need to export it. Exporting a ZenPack takes all of the object's you've added to your ZenPack through the web interface and compiles them into an `objects.xml` file that gets saved into your ZenPack's source directory in the file system.

Follow these steps to export a ZenPack.

1. Navigate to *Advanced -> ZenPacks -> Your ZenPack* in the web interface.
2. Scroll to the bottom of the page to see what objects the ZenPack provides.

   All objects listed in the *ZenPack Provides* section and objects contained within them will be exported.
3. Choose *Export ZenPack* from the gear menu in the bottom-left of the screen.
4. Choose to only export and not download then click *OK*.

   You could also choose to download the ZenPack through your web browser. However, the downloaded file will be the built *egg* distribution format of the ZenPack. This means that it can be installed into other Zenoss systems, but is not suitable for further development.

This will export everything under *ZenPack Provides* to a directory within your ZenPack's source called *objects/*. No other files in your ZenPack's source directory are created or modified. You can find this file in a path such as the following.

/z/ZenPacks.acme.Widgeter/ZenPacks/acme/Widgeter

Each time you add a new object to you ZenPack within the web interface, or modify an object that's already contained within your ZenPack, you should export the ZenPack again to update objects.xml. If you're using version control on your ZenPack's source directory this would be a good time to commit the resulting changes.

> **Warning:** Exporting a ZenPack overwrites files in the *objects/* directory. For this reason it is recommended that files in this directory never be modified by hand.

4.4.3 Device Modeling

This section will cover creation of a custom *Device* subclass and modeling of device attributes.

For purposes of this example, we'll add a *temp_sensor_count* attribute to NetBotz devices. We'll walk through adding the attribute to the model, modeling it from the device, and displaying it in the overview screen for NetBotz devices.

Starting in this section we'll be working with files within the NetBotz ZenPack's directory. To keep the path names short, I'll assume the *$ZP_TOP_DIR* and *$ZP_DIR* environment variables have been set as follows.

```
export ZP_TOP_DIR=/z/ZenPacks.training.NetBotz
export ZP_DIR=$ZP_TOP_DIR/ZenPacks/training/NetBotz
```

Create the NetBotzDevice Class

A *Device* subclass should not be confused with a *device class*. In the previous section we created the /NetBotz device class from the web interface. Creating a *Device* subclass means to extend the actual Python class of a *Device* object. You'd do this to add new attributes, methods or relationships to special device types.

Use the following steps to create a *NetBotzDevice* class with a new attribute called *temp_sensor_count*.

1. Update `$ZP_DIR/zenpack.yaml` to contain following contents.

```
name: ZenPacks.training.NetBotz

classes:
  NetBotzDevice:
    base: [zenpacklib.Device]
    label: NetBotz
    properties:
      temp_sensor_count:
        type: int

device_classes:
  /NetBotz:
    zProperties:
      zPythonClass: ZenPacks.training.NetBotz.NetBotzDevice
      zSnmpMonitorIgnore: false
      zCollectorPlugins:
        - training.snmp.NetBotz
        - zenoss.snmp.NewDeviceMap
        - zenoss.snmp.DeviceMap
        - zenoss.snmp.InterfaceMap
```

   (a) The *name* field is mandatory and must match the full Python module name of your ZenPack.

   (b) The classes section is where we define extensions to the standard Zenoss model. In this case we're creating a special device type called *NetBotzDevice* because we want to add a new property called *temp_sensor_count*. See *Classes and Relationships* for more information on defining classes.

   (c) The *device_classes* section allows us to also configure the */NetBotz* device class in YAML. Note that we're configuring the same options that we already set through the web interface. You can set them either way, but once you add a device class to zenpack.yaml you'll likely find its easier to maintain all of the information in one place.

   The most important property we're setting on the */NetBotz* device class is *zPythonClass*. This is required so that the new *NetBotzDevice* class we've defined will be used for devices in this device class.

   You'll also note that we're adding *training.snmp.NetBotz* to the list of modeler plugins (*zCollectorPlugins*) even though it doesn't yet exist. This is safe to do, and we'll shortly be creating the modeler plugin.

2. Reinstall the ZenPack to have the device class changes made.

```
zenpack --link --install $ZP_TOP_DIR
```

3. Restart *Zope* process so the web interface can load our new module.

```
serviced service restart zope
```

4. Reset the Python class of our existing device.

   Run `zendmd` and execute the following snippet.

```
device = find("Netbotz01")
print device.__class__
```

   You should see <*class 'Products.ZenModel.Device.Device'*>. We see this instead of the Python class we just created because the *zPythonClass* property is only used when a new device is created in a device class, or when a device is moved into a device class with a differing *zPythonClass* value.

   So we have two options for getting our NetBotz device to use the new Python class we created. We can either delete the device and add it back, or move it to a different device class and back. Actually, there's a third option that I use most frequently to solve this problem. I move it into the same device class using *zendmd*. Execute the following snippet within *zendmd* to reset the device's Python class.

```
dmd.Devices.NetBotz.moveDevices('/NetBotz', 'Netbotz01')
commit()

device = find("Netbotz01")
print device.__class__
```

   Now you should see <*class 'ZenPacks.training.NetBotz.NetBotzDevice'*> printed. This confirms that our *Device* subclass works, and that we've configure *zPythonClass* correctly for the */NetBotz* device class.

Find Temperature Sensor Count

Before we can write a modeler plugin to populate our new *temp_sensor_count* attribute, we need to figure out how to get the information. There are a few ways we could approach this. One way is to use that NETBOTZV2-MIB as a reference to see if we can find anything about temperature sensors specifically.

Find temperature information in NETBOTZV2-MIB using the following command.

```
smidump -f identifiers /usr/share/snmp/mibs/NETBOTZV2-MIB.mib | egrep -i temp
```

You should see the following in the output:

```
NETBOTZV2-MIB tempSensorTable         table   1.3.6.1.4.1.5528.100.4.1.1
NETBOTZV2-MIB tempSensorEntry         row     1.3.6.1.4.1.5528.100.4.1.1.1
NETBOTZV2-MIB tempSensorId            column  1.3.6.1.4.1.5528.100.4.1.1.1.1
NETBOTZV2-MIB tempSensorValue         column  1.3.6.1.4.1.5528.100.4.1.1.1.2
NETBOTZV2-MIB tempSensorErrorStatus   column  1.3.6.1.4.1.5528.100.4.1.1.1.3
NETBOTZV2-MIB tempSensorLabel         column  1.3.6.1.4.1.5528.100.4.1.1.1.4
NETBOTZV2-MIB tempSensorEncId         column  1.3.6.1.4.1.5528.100.4.1.1.1.5
NETBOTZV2-MIB tempSensorPortId        column  1.3.6.1.4.1.5528.100.4.1.1.1.6
NETBOTZV2-MIB tempSensorValueStr      column  1.3.6.1.4.1.5528.100.4.1.1.1.7
NETBOTZV2-MIB tempSensorValueInt      column  1.3.6.1.4.1.5528.100.4.1.1.1.8
NETBOTZV2-MIB tempSensorValueIntF     column  1.3.6.1.4.1.5528.100.4.1.1.1.9
```

You'll also see another *node* and a bunch of *notification* entries. These are related to SNMP traps, and not relevant to what we're interested in polling right now.

What we see here is that there isn't a single OID we can request that will tell us the number of temperature sensors. We're going to have to do an *snmpwalk* of the table then count how many rows are in the response. Specifically

we want to remember the name and OID for the *row*: *tempSensorEntry*. Due to the hierarchical nature of a MIBs representation this is the most specific OID that will return the data we need.

```
snmpwalk 172.17.0.1 1.3.6.1.4.1.5528.100.4.1.1.1
```

You'll see a lot of output that starts with:

```
NETBOTZV2-MIB::tempSensorId.21604919 = STRING: nbHawkEnc_1_TEMP
NETBOTZV2-MIB::tempSensorId.1095346743 = STRING: nbHawkEnc_0_TEMP
NETBOTZV2-MIB::tempSensorId.1382714817 = STRING: nbHawkEnc_2_TEMP1
NETBOTZV2-MIB::tempSensorId.1382714818 = STRING: nbHawkEnc_2_TEMP2
NETBOTZV2-MIB::tempSensorId.1382714819 = STRING: nbHawkEnc_2_TEMP3
NETBOTZV2-MIB::tempSensorId.1382714820 = STRING: nbHawkEnc_2_TEMP4
NETBOTZV2-MIB::tempSensorId.1382714833 = STRING: nbHawkEnc_3_TEMP1
NETBOTZV2-MIB::tempSensorId.1382714834 = STRING: nbHawkEnc_3_TEMP2
NETBOTZV2-MIB::tempSensorId.1382714865 = STRING: nbHawkEnc_1_TEMP1
NETBOTZV2-MIB::tempSensorId.1382714866 = STRING: nbHawkEnc_1_TEMP2
NETBOTZV2-MIB::tempSensorId.1382714867 = STRING: nbHawkEnc_1_TEMP3
NETBOTZV2-MIB::tempSensorId.1382714868 = STRING: nbHawkEnc_1_TEMP4
NETBOTZV2-MIB::tempSensorId.2169088567 = STRING: nbHawkEnc_3_TEMP
NETBOTZV2-MIB::tempSensorId.3242830391 = STRING: nbHawkEnc_2_TEMP
```

What you're seeing above is the tempSensorId column for all 14 rows in the tempSensorTable. Continuing on you will see 14 rows for each of the other columns in the table.

Create a Modeler Plugin

The next step is to build a modeler plugin. A modeler plugin's responsibility reach out into the world, gather data, and plug it into the attributes and relationships of our model classes. In this example, this means to make the SNMP requests necessary to determine how many temperature sensors a NetBotz device has, and populate our *temp_sensor_count* attribute with the result.

Use the following steps to create our modeler plugin.

1. Make the directory that'll contain our modeler plugin.

   ```
   mkdir -p $ZP_DIR/modeler/plugins/training/snmp
   ```

   Note that we're using our ZenPack's *training* namespace, then *snmp*. This is the recommended approach to make it clear what protocol the modeler plugin will use, and to avoid our modeler plugin conflicting with one from someone else's ZenPack.

2. Create *__init__.py* or *dunder-init* files.

   ```
   touch $ZP_DIR/modeler/__init__.py
   touch $ZP_DIR/modeler/plugins/__init__.py
   touch $ZP_DIR/modeler/plugins/training/__init__.py
   touch $ZP_DIR/modeler/plugins/training/snmp/__init__.py
   ```

   These empty *__init__.py* files are mandatory if we ever expect Python to import modules from these directories.

3. Create `$ZP_DIR/modeler/plugins/training/snmp/NetBotz.py` with the following contents.

   ```
   from Products.DataCollector.plugins.CollectorPlugin import (
       SnmpPlugin, GetTableMap,
       )
   ```

```
class NetBotz(SnmpPlugin):
    snmpGetTableMaps = (
        GetTableMap(
            'tempSensorTable', '1.3.6.1.4.1.5528.100.4.1.1.1', {
                '.1': 'tempSensorId',
                }
            ),
        )

    def process(self, device, results, log):
        temp_sensors = results[1].get('tempSensorTable', {})

        return self.objectMap({
            'temp_sensor_count': len(temp_sensors.keys()),
            })
```

(a) Start by importing SnmpPlugin and GetTableMap from Zenoss. SnmpPlugin will handle all of the SNMP requests for us and present the results in a format we can easily work with. GetTableMap will be used here because we need to request an SNMP table rather than specific OIDs.

(b) Our NetBotz class extends SnmpPlugin. Note that the NetBotz class name must match the filename (module name) of the modeler plugin.

(c) By defining snmpGetTableMaps as a tuple or list on our class we can add a GetTableMap object that requests that 1.3.6.1.4.1.5528.100.4.1.1.1 row OID and specify that we only want to get the first (.1) column and name it tempSensorId.

(d) The *process* method will receive a two-element tuple containing the SNMP request results in the *request* parameter. The first elememt, *results[0]*, of this tuple would be any direct OID gets of which we didn't request any in this plugin. The second element, *results[1]* will contain a dictionary of the table results. In this case *results[1]* would look like the following.

```
{
    'tempSensorTable': {
        '21604919': 'nbHawkEnc_1_TEMP',
        '1095346743': 'nbHawkEnc_0_TEMP',
        '1382714817': 'nbHawkEnc_2_TEMP1',
        '1382714818': 'nbHawkEnc_2_TEMP2',
        '1382714819': 'nbHawkEnc_2_TEMP3',
        '1382714820': 'nbHawkEnc_2_TEMP4',
        '1382714833': 'nbHawkEnc_3_TEMP1',
        '1382714834': 'nbHawkEnc_3_TEMP2',
        '1382714865': 'nbHawkEnc_1_TEMP1',
        '1382714866': 'nbHawkEnc_1_TEMP2',
        '1382714867': 'nbHawkEnc_1_TEMP3',
        '1382714868': 'nbHawkEnc_1_TEMP4',
        '2169088567': 'nbHawkEnc_3_TEMP',
        '3242830391': 'nbHawkEnc_2_TEMP',
    },
}
```

(e) We then extract just the *tempSensorTable* results into *temp_sensors* to make the next *return* line a bit easier to understand.

(f) We then return a dictionary that sets the *temp_sensor_count* key's value to the number of keys in *temp_sensors*. Actually we return a dictionary that's been wrapped in an ObjectMap by the modeler plugin's *objectMap* utility method.

The *process* method within all modeler plugins must return one of the following types of data.

- None (makes no changes to the model)
- ObjectMap (to apply directly to the device that's being modeled)
- RelationshipMap (to apply to a relationship within the device)
- A list containing zero or more ObjectMap and/or RelationShipMap objects.

An *ObjectMap* is simply a *dict* wrapped with some meta-data. A *RelationshipMap* is a *list* wrapped with some meta-data and containing zero or more *ObjectMap* instances.

4. Restart *Zope* and *zenhub* to load the new module.

```
serviced service restart zope
serviced service restart zenhub
```

Test the Modeler Plugin

Now that we've created and enabled a basic modeler plugin, we should test it.

1. Remodel the NetBotz device.

   You can do this from the web interface, but I usually use the command line because it can be easier to work with if further debugging is necessary.

```
zenmodeler run --device=Netbotz01
```

2. Execute the following snippet in *zendmd*.

```
device = find("Netbotz01")
print device.temp_sensor_count
```

   You should see *14* printed as the number of temperature sensors.

Change the Device Overview

The next step will be to show the number of temperature sensors to users of the web interface. We'll replace the *Memory/Swap* field in the top-left box of the device overview page with the count of temperature sensors.

Follow these steps to customize the device Overview page.

1. Create a directory to store our ZenPack's JavaScript.

```
mkdir -p $ZP_DIR/resources
```

2. Create `$ZP_DIR/resources/device.js` with the following contents.

```
Ext.onReady(function() {
    var DEVICE_OVERVIEW_ID = 'deviceoverviewpanel_summary';
    Ext.ComponentMgr.onAvailable(DEVICE_OVERVIEW_ID, function(){
        var overview = Ext.getCmp(DEVICE_OVERVIEW_ID);
        overview.removeField('memory');

        overview.addField({
            name: 'temp_sensor_count',
            fieldLabel: _t('# Temperature Sensors')
        });
    });
});
```

(a) Wait for Ext to be ready.

(b) Find the overview summary panel (top-left on Overview page)

(c) Remove the *memory* field.

(d) Add our *temp_sensor_count* field.

Zenoss uses ExtJS as its JavaScript framework. You can find more in ExtJS's documentation about manipulating objects in this way.

Test the Device Overview

That's it. We can restart *Zope* and navigate to our NetBotz device's overview page in the web interface. You should see `# Temperature Sensors` label with a value of 14 at the bottom of the top-left panel.

4.4.4 Component Modeling

This section will cover creation of a custom *Component* subclass, creation of a relationship to our *NetBotDevice* class, and modeling of the components to fill the relationship.

In the *Device Modeling* section we added a *temp_sensor_count* attribute to our NetBotz devices. This isn't very useful. It would be more useful to monitor the temperature being reported by each of these sensors. So that's what we'll do. Modeling each sensor as a component allows Zenoss to automatically discover and monitor sensors regardless of how many a particular device has.

Find Temperature Sensor Attributes

In the *Device Modeling* section we used *smidump* to extract temperature sensor information from *NETBOTZV2-MIB*. This will be even more applicable as we decide what attributes and metrics are available on each sensor. Let's use *smidump* and *snmpwalk* for a refresher on what's available.

Find temperature information in NETBOTZV2-MIB using the following command.

```
smidump -f identifiers /usr/share/snmp/mibs/NETBOTZV2-MIB.mib | egrep -i temp
```

You should see the following in the output:

```
NETBOTZV2-MIB tempSensorTable        table   1.3.6.1.4.1.5528.100.4.1.1
NETBOTZV2-MIB tempSensorEntry        row     1.3.6.1.4.1.5528.100.4.1.1.1
NETBOTZV2-MIB tempSensorId           column  1.3.6.1.4.1.5528.100.4.1.1.1.1
NETBOTZV2-MIB tempSensorValue        column  1.3.6.1.4.1.5528.100.4.1.1.1.2
NETBOTZV2-MIB tempSensorErrorStatus  column  1.3.6.1.4.1.5528.100.4.1.1.1.3
NETBOTZV2-MIB tempSensorLabel        column  1.3.6.1.4.1.5528.100.4.1.1.1.4
NETBOTZV2-MIB tempSensorEncId        column  1.3.6.1.4.1.5528.100.4.1.1.1.5
NETBOTZV2-MIB tempSensorPortId       column  1.3.6.1.4.1.5528.100.4.1.1.1.6
NETBOTZV2-MIB tempSensorValueStr     column  1.3.6.1.4.1.5528.100.4.1.1.1.7
NETBOTZV2-MIB tempSensorValueInt     column  1.3.6.1.4.1.5528.100.4.1.1.1.8
NETBOTZV2-MIB tempSensorValueIntF    column  1.3.6.1.4.1.5528.100.4.1.1.1.9
```

Let's now use *snmpwalk* to see what these values look like on our NetBotz device.

```
snmpwalk 172.17.0.1 1.3.6.1.4.1.5528.100.4.1.1.1
```

You should see a lot of output that begins with the following:

```
NETBOTZV2-MIB::tempSensorId.21604919 = STRING: nbHawkEnc_1_TEMP
NETBOTZV2-MIB::tempSensorId.1095346743 = STRING: nbHawkEnc_0_TEMP
NETBOTZV2-MIB::tempSensorId.1382714817 = STRING: nbHawkEnc_2_TEMP1
NETBOTZV2-MIB::tempSensorId.1382714818 = STRING: nbHawkEnc_2_TEMP2
```

Note the *21604919* in the first response. This is the SNMP index of the first temperature sensor, or the first row in the table. I like to then restrict my snmpwalk results to only show this row with a command like the following.

```
snmpwalk 172.17.0.1 1.3.6.1.4.1.5528.100.4.1.1.1 | grep "\.21604919 ="
```

Which will show us the value of each column for that one temperature sensor:

```
NETBOTZV2-MIB::tempSensorId.21604919 = STRING: nbHawkEnc_1_TEMP
NETBOTZV2-MIB::tempSensorValue.21604919 = INTEGER: 265
NETBOTZV2-MIB::tempSensorErrorStatus.21604919 = INTEGER: normal(0)
NETBOTZV2-MIB::tempSensorLabel.21604919 = STRING: Temperature
NETBOTZV2-MIB::tempSensorEncId.21604919 = STRING: nbHawkEnc_1
NETBOTZV2-MIB::tempSensorPortId.21604919 = STRING:
NETBOTZV2-MIB::tempSensorValueStr.21604919 = STRING: 26.500000
NETBOTZV2-MIB::tempSensorValueInt.21604919 = INTEGER: 26
NETBOTZV2-MIB::tempSensorValueIntF.21604919 = INTEGER: 79
```

Now we have everything we should need to make decisions about what attributes we should model for our sensors and which would better be collected as datasources to have thresholds applied and plotted over time on graphs.

My initial thoughts would be to model the following as attributes.

- *tempSensorId*
- *tempSensorEncId* (enclosure ID)
- *tempSensorPortId*

I would then want to collect *tempSensorValueStr* as a datasource because it offers the best precision. Zenoss is capable of handling numeric strings so we don't have to collect *tempSensorValue* and divide it by 10 like other systems might.

Create a Component Subclass

Use the following steps to create a *NetBotzTemperatureSensor* class with the attributes discovered above.

1. Update `$ZP_DIR/zenpack.yaml` to include the following *NetBotzTemperatureSensor* entry in the *classes* section, and the new *class_relationships* section.

```
classes:
  NetBotzDevice:
    base: [zenpacklib.Device]
    label: NetBotz
    properties:
      temp_sensor_count:
        type: int

  NetBotzTemperatureSensor:
    base: [zenpacklib.Component]
    label: Temperature Sensor
    properties:
      enclosure:
        label: Enclosure

      port:
```

```
        label: Port

class_relationships:
  - NetBotzDevice 1:MC NetBotzTemperatureSensor
```

(a) It's important to pick class names that will be unique. The best practice is to use a short prefix based on the ZenPack's name followed by the type of thing the class represents as is being done here.

(b) Both of the new properties should be strings. Since string is the default type, we don't need to specify it. This just leaves the label.

Note: Despite noting above that we always wanted to model the *tempSensorId* attribute, we aren't adding an attribute for it here. This is because *DeviceComponent* already has both an *id* and *title* attribute that wherein we can store the value of *tempSensorId*.

(c) The *class_relationships* section is very important. We could never have any temperature sensors in the system if we didn't relate them to something else. The *1:MC* between the two class names describes the type of relationship. Specifically it says that one *NetBotzDevice* can contain many *NetBotzTemperatureSensor* objects. See *Relationships* for more information.

Test TemperatureSensor Class

With our component class defined and relationships setup we can use *zendmd* to make sure we didn't make any mistakes. Execute the following snippet in *zendmd*.

```
from ZenPacks.training.NetBotz.NetBotzTemperatureSensor import␣
↪NetBotzTemperatureSensor

sensor = NetBotzTemperatureSensor('test_sensor_01')
device = find("Netbotz01")
device.netBotzTemperatureSensors._setObject(sensor.id, sensor)
sensor = device.netBotzTemperatureSensors._getOb(sensor.id)
print sensor
print sensor.device()
```

You'll most likely get the following error when executing the above snippet:

```
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AttributeError: netBotzTemperatureSensors
```

This error is indicating that we have no *netBotzTemperatureSensors* relationship on the device object. This would seemingly make no sense because we just added it. The key here is that existing objects like the *Netbotz01* device don't automatically get new relationships. We have to either delete the device and add it again, or execute the following in *zendmd* to create the newly- defined relationship.

```
device.buildRelations()
commit()
```

Now you can go back and run the original snippet again. You should see the name of the sensor and device objects printed if everything worked as planned.

Update the Modeler Plugin

As with the *NetBotzDevice* class, the next step after creating our model class is to populate it with a modeler plugin. We could create a new modeler plugin to only capture the temperature sensor components, but we'll update the *NetBotz* modeler plugin we previously created to model the sensors instead.

1. Edit `$ZP_DIR/modeler/plugins/training/snmp/NetBotz.py` and replace its contents with the following.

```
from Products.DataCollector.plugins.CollectorPlugin import (
    SnmpPlugin, GetTableMap,
    )


class NetBotz(SnmpPlugin):
    relname = 'netBotzTemperatureSensors'
    modname = 'ZenPacks.training.NetBotz.NetBotzTemperatureSensor'

    snmpGetTableMaps = (
        GetTableMap(
            'tempSensorTable', '1.3.6.1.4.1.5528.100.4.1.1.1', {
                '.1': 'tempSensorId',
                '.5': 'tempSensorEncId',
                '.6': 'tempSensorPortId',
                }
            ),
        )

    def process(self, device, results, log):
        temp_sensors = results[1].get('tempSensorTable', {})

        rm = self.relMap()
        for snmpindex, row in temp_sensors.items():
            name = row.get('tempSensorId')
            if not name:
                log.warn('Skipping temperature sensor with no name')
                continue

            rm.append(self.objectMap({
                'id': self.prepId(name),
                'title': name,
                'snmpindex': snmpindex.strip('.'),
                'enclosure': row.get('tempSensorEncId'),
                'port': row.get('tempSensorPortId'),
                }))

        return rm
```

Let's take a closer look at how we changed the modeler plugin.

(a) We added *relname* and *modname* as class attributes.

These two settings control the meta-data that will automatically be set when the *self.relMap* and *self.objectMap* methods are called in the *process* method.

The target *relname* we should use depends on a couple of things. First, all leading uppercase letters of the class name will be converted to lowercase, i.e. *NetBotzTemperatureSensor* becomes *netBotzTemperatureSensor*. Second, the letter "s" is added to the end if it is a to-many relationship, i.e. *netBotzTemperatureSensor* becomes *netBotzTemperatureSensors*.

Setting *relname* to `netBotzTemperatureSensors` will cause the *self.relMap* call to create a *RelationshipMap* that will be applied to the *netBotzTemperatureSensors* relationship defined on the *NetBotzDevice* object.

Setting *modname* to `ZenPacks.training.NetBotz.TemperatureSensor` will cause the *self.objectMap* calls in the *process* method to create *ObjectMap* instances that will be turned into instances of our *TemperatureSensor* class.

(b) We're now requesting the *tempSensorEncId* and *tempSensorPortId* columns be returned in the SNMP table request results. We'll use these to populate their corresponding fields on the *TemperatureSensor* class.

(c) Most of the *process* method has been changed.

We're now creating a *RelationshipMap* and appending an *ObjectMap* to it for each temperature sensor in the results. We use the *self.relMap* and *self.objectMap* utility methods to make this easier.

2. Restart *Zope* and *zenhub* to load the changed module.

```
serviced service restart zope
serviced service restart zenhub
```

Test the Modeler Plugin

We already added the *training.snmp.NetBotz* modeler plugin the the */NetBotz* device class in an earlier exercise. So we only need to run *zenmodeler* to test the temperature sensor modeling updates.

1. Run `zenmodeler run --device=Netbotz01`

   We should see *Changes in configuration applied* near the end of zenmodeler's output. The changes referred to should be 14 temperature sensor objects being created and added to the device's netBotzTemperatureSensors relationship.

2. Check the *Netbotz01* device in the web interface. The temperature sensors should now be visible.

4.4.5 Component Monitoring

This section covers monitoring component metrics using SNMP. I assume that you've completed the *Component Modeling* steps and now have temperature sensor components modeled for the NetBotz device. Currently there will be no graphs for these temperature sensors.

We will add collection, thresholding and graphing for the temperature monitored by each sensor.

Find the SNMP OID

In the *Component Modeling* section we used *smidump* and *snmpwalk* to find which values would be useful to model, and which would be useful to monitor. We found *tempSensorValue* to be the best OID to use for monitoring a sensor's current temperature.

Let's use *snmpwalk* again to see what *tempSensorValue* looks like for all of the sensors on our NetBotz device.

```
snmpwalk 172.17.0.1 NETBOTZV2-MIB::tempSensorValueStr
```

This gives of the current temperature (in celsius) for each sensor:

```
NETBOTZV2-MIB::tempSensorValueStr.21604919 = STRING: 26.500000
NETBOTZV2-MIB::tempSensorValueStr.1095346743 = STRING: 27.000000
NETBOTZV2-MIB::tempSensorValueStr.1382714817 = STRING: 22.100000
NETBOTZV2-MIB::tempSensorValueStr.1382714818 = STRING: 21.100000
NETBOTZV2-MIB::tempSensorValueStr.1382714819 = STRING: 19.600000
NETBOTZV2-MIB::tempSensorValueStr.1382714820 = STRING: 19.900000
NETBOTZV2-MIB::tempSensorValueStr.1382714833 = STRING: 20.500000
NETBOTZV2-MIB::tempSensorValueStr.1382714834 = STRING: 20.100000
NETBOTZV2-MIB::tempSensorValueStr.1382714865 = STRING: 19.700000
NETBOTZV2-MIB::tempSensorValueStr.1382714866 = STRING: 20.500000
NETBOTZV2-MIB::tempSensorValueStr.1382714867 = STRING: 20.100000
NETBOTZV2-MIB::tempSensorValueStr.1382714868 = STRING: 20.000000
NETBOTZV2-MIB::tempSensorValueStr.2169088567 = STRING: 26.600000
NETBOTZV2-MIB::tempSensorValueStr.3242830391 = STRING: 27.400000
```

As we go on to add a monitoring template below, we'll need to know what OID to poll to collect this value. The key to determining this for components with an *snmpindex* attribute like *TemperatureSensor* has is to find the OID for the values above and remove the SNMP index from the end of it. Let's use *snmptranslate* to do this.

```
snmptranslate -On NETBOTZV2-MIB::tempSensorValueStr
```

This results in the following output:

```
.1.3.6.1.4.1.5528.100.4.1.1.1.7
```

This OID (minus the leading .) is what we'll need.

Add a Monitoring Template

It is important to get the monitoring template's name correct when adding a monitoring template that will be used for components. A Zenoss administrator has no control over which monitoring templates will be bound to a component like they do with monitoring templates that are bound to devices.

How do we know what to name a monitoring template that should be used to monitor our *NetBotzTemperatureSensor* components? By default the monitoring template should be named the same as the component class' *label* property with all spaces removed. In the *Component Modeling* section we set the label for the *NetBotzTemperatureSensor* class to be *Temperature Sensor*. This means out monitoring template should be named *TemperatureSensor*.

If you'd rather specify the monitoring template's name, or the names of multiple monitoring templates to use for your component class, you can do so by specifying an explicit *monitoring_templates* value for it in *zenpack.yaml*.

Perform the following steps to create our component's monitoring template.

1. Navigate to *Advanced -> Monitoring Templates*.
2. Add a template.
   (a) Click the + in the bottom-left of the template list.
   (b) Set *Name* to `TemperatureSensor`
   (c) Set *Template Path* to */NetBotz*
   (d) Click *SUBMIT*
3. Add a data source.
   (a) Click the + at the top of the *Data Sources* panel.
   (b) Set *Name* to `tempSensorValueStr`

(c) Set *Type* to *SNMP*

(d) Click *SUBMIT*

(e) Double-click to edit the *tempSensorValueStr* data source.

(f) Set *OID* to `1.3.6.1.4.1.5528.100.4.1.1.1.7`

(g) Click *SAVE*

5. Add a threshold.

   (a) Click the + at the top of the *Thresholds* panel.

   (b) Set *Name* to `high temperature`

   (c) Set *Type* to *MinMaxThreshold*

   (d) Click *ADD*

   (e) Double-click to edit the *high temperature* threshold.

   (f) Move the datapoint to the list on the right.

   (g) Set *Maximum Value* to `32`

   (h) Set *Event Class* to */Environ*

   (i) Click *SAVE*

6. Add a graph.

   (a) Click the + at the top of the *Graph Definitions* panel.

   (b) Set *Name* to `Temperature`

   (c) Click *SUBMIT*

   (d) Double-click to edit the *Temperature* graph.

   (e) Set *Units* to `degrees c.`

   (f) Click *SUBMIT*

7. Add a graph point.

   (a) Click to select the *Temperature* graph.

   (b) Choose *Manage Graph Points* from the gear menu.

   (c) Choose *Data Point* from the + menu.

   (d) Choose *tempSensorValueStr* then click *SUBMIT*

   (e) Double-click to edit the *tempSensorValueStr* graph point.

   (f) Set *Name* to `Temperature`

   (g) Set *Format* to `%7.2lf`

   (h) Click *SAVE* then *SAVE* again.

Note: You can also define monitoring templates in *zenpack.yaml* instead of creating them through the web interface. See *Monitoring Templates* for more information.

Test Monitoring Template

You can now refer back to the *Test Monitoring Template* section of *Device Monitoring* for using *zenperfsnmp* to test the data point collection aspect of your monitoring template.

You can verify that your monitoring template is getting bound to each temperature sensor properly by navigating to one of the temperature sensors in the web interface and choosing *Templates* from it's *Display* drop-down box. Furthermore, you can verify that your *Temperature* graph is shown when choosing *Graphs* from the temperature sensor's *Display* drop-down.

4.4.6 SNMP Traps

This section covers how to handle SNMP traps.

Zenoss will accept SNMP traps from your devices as soon as you configure those devices to send traps to your Zenoss server. The *zentrap* daemon will listen to the standard SNMP trap port of *162/udp* and create an event for every trap that it receives.

However, without you giving Zenoss more information about the contents of those traps, the events will contain numeric OIDs and be nearly impossible for a human to decipher.

Importing MIBs

Let's import the *NETBOTZV2-MIB* that we've been working with through these examples.

1. Copy the MIB to */z* so containers can read it.

```
cp /usr/share/snmp/mibs/NETBOTZV2-MIB.mib /z
```

2. Import the MIB file.

```
zenmib run --keepMiddleZeros NETBOTZV2-MIB.mib
```

   From which we should get the following output:

```
Found 1 MIBs to import.
Unable to find a file that defines SNMPv2-SMI
Unable to find a file that defines SNMPv2-TC
Parsed 214 nodes and 256 notifications from NETBOTZV2-MIB
Loaded MIB NETBOTZV2-MIB into the DMD
Loaded 1 MIB file(s)
```

3. Add the imported MIB to the NetBotz ZenPack.
   (a) Navigate to *Advanced -> MIBs* in the web interface.
   (b) Select *NETBOTZV2-MIB*.
   (c) Choose *Add to ZenPack* from the gear menu at the bottom of the list.
   (d) Choose the *ZenPacks.training.NetBotz* then click *SUBMIT*.

Simulating SNMP Traps

To more easily configure and test Zenoss' trap handling, it's useful to know how to simulate SNMP traps. The alternative is breaking your real devices in various ways and hoping to be able to get the device to send all of the traps you need. This isn't always possible.

Let's start by picking an SNMP trap to simulate.

1. Navigate to *Advanced -> MIBs* in the web interface.
2. Choose *NETBOTZV2-MIB* from the list of MIBs.
3. Choose *Traps* from the drop-down box in the middle of the right panel.
4. Choose *netBotzTempTooHigh* in the list of traps.

We'll now see information about this trap in the bottom-right panel. The first thing to note is the OID. This is all we need to get started.

Send a Simple Trap

Use the following steps to get your feet wet sending a basic trap.

1. Make sure the *zentrap* service is running.

   If you have stopped the zentrap service, or if you have it configured to manual launch mode, you will need to start it.

   ```
   serviced service start zentrap
   ```

2. Identify the IP address to which traps should be sent to get to zentrap.

   serviced does performs port forwarding on the serviced host to route received SNMP traps to the zentrap container. We're going to be sending simulated SNMP traps from the serviced host, and will need to know what address to send traps to so they're received by zentrap.

   Run the following command to find the address.

   ```
   sudo iptables -L FORWARD -n | grep 162
   ```

   This will output something very close to the following:

   ```
   ACCEPT     udp  --  0.0.0.0/0            172.17.0.29          udp dpt:162
   ```

   We'll be sending traps to that 172.17.0.29 address. It may be different on your system.

3. Send an SNMP trap.

   Run the following *snmptrap* command on the serviced host.

   ```
   sudo snmptrap 172.17.0.29 0 NETBOTZV2-MIB::netBotzTempTooHigh
   ```

4. Find this netBotzTempTooHigh event in web interface's event console.

   Double-click the "snmp trap netBotzTempTooHigh" event in the event console to see its details. Look for the following details.

   - eventClassKey: This should be netBotzTempTooHigh as decoded using the MIB.
   - oid: This is the original undecoded OID.

Send a Full Trap

Now that we've proved out a simple trap, we should add variable bindings or *varbinds* to the trap. If you look at the *netBotzTempTooHigh* trap in the Zenoss web interface's MIB explorer again, you'll see that there's an extensive list of *Objects* associated with the trap definition. These are variable bindings.

A variable binding allows the device sending the SNMP trap to attach additional information to the trap. In this example, one of the variable bindings for the *netBotzTempTooHigh* trap is *netBotzV2TrapSensorID*. This will give us a way to know which one of the sensors has exceeded it's high temperature threshold.

1. Run the following *snmptrap* command.

```
sudo snmptrap 172.17.0.29 0 NETBOTZV2-MIB::netBotzTempTooHigh \
    NETBOTZV2-MIB::netBotzV2TrapSensorID s 'nbHawkEnc_1_TEMP1'
```

   As you can see, this *zentrap* command starts exactly the same as in the example. We then add the following three fields.

   (a) `NETBOTZV2-MIB::netBotzV2TrapSensorID` (OID)

   (b) `s` (type)

   (c) `'nbHawkEnc_1_TEMP1'` (value)

   We can continue to add sets of these three parameters to add as many other variable bindings to the trap as we want.

   Note that the only difference between this event and the simple event is the addition of the *netBotzV2TrapSensorID* field. So now you see how Zenoss take the name/value pairs that are the SNMP trap's variable bindings and turn them into name/value pairs within the resulting event.

Mapping SNMP Trap Events

Now that we're able to create SNMP traps anytime we want, it's time to use Zenoss' event mapping system to make them more useful. The most important field on an incoming event when it comes to mapping is the *eventClassKey* field. Fortunately for us, SNMP traps get that great *eventClassKey* set that gives us a big head start.

1. Map the event.

   (a) Navigate to *Events* in the web interface.

   (b) Select the *netBotzTempTooHigh* event you just created.

   (c) Click the toolbar button that looks like a hierarchy. If you hover over it, the tooltip will say *Reclassify an event*.

   (d) Choose the */Environ* event class then click *SUBMIT*

   Now the next time a *netBotzTempTooHigh* trap is received it will be put into the */Environ* event class instead of */Unknown*.

2. Enrich the event.

   (a) Click the *Go to new mapping* link to navigate to the new mapping.

   (b) Click *Edit* in the left navigation pane.

   (c) Set *Transform* to the following:

```
evt.component = getattr(evt, 'netBotzV2TrapSensorID', '')
```

   This will use the name of the sensor as described by the *netBotzV2TrapSensorID* variable binding as the event's *component* field.

There are endless possibilities of what you could do within the transform for this event and others. This is just one practical example.

4.5 Monitoring an HTTP API

This tutorial will describe an efficient approach to monitoring data via a HTTP API. We'll start by using *zenpack.yaml* to extend the Zenoss object model. Then we'll use a Python modeler plugin to fill out the object model. Then we'll use PythonCollector to monitor for events, datapoints and even to update the model.

For purposes of this guide we'll be building a ZenPack that monitors the weather using The Weather Channel's Weather Underground API.

Note: This tutorial assumes your system is already setup as described in *Development Environment* and *Getting Started*.

4.5.1 Weather Underground API

The Weather Underground provides an API that can be used to get all sorts of data related to the weather. Before you can use most endpoints on the API you must first create an account. Fortunately you can get a *Developer* account with all of the bells and whistles for free by signing up at http://www.wunderground.com/weather/api. So go sign up and get your API key. You'll need it for the rest of this exercise.

We'll be using the following APIs for this exercise.

1. AutoComplete
2. Alerts
3. Conditions

AutoComplete API

Both the *Alerts* and *Conditions* APIs require that you query for a specific location. It can be hard to know what the name or code for a location is without doing some manual research. That's where the *AutoComplete* API comes in. You can provide a reasonable name for a location and it will return a list of possible matches along with a unique link for that location.

We'll use the AutoComplete API during modeling so that the Zenoss user can enter nearly any city or county name then let Zenoss do the work of converting that into the link that we'll subsequently use to query for weather alerts and conditions.

Here's an example query for Austin, TX:

```
http://autocomplete.wunderground.com/aq?query=Austin%2C%20TX
```

Note: "Austin%2C%20TX" is the URL encoded version of "Austin, TX". We will url encode this data when we work with it.

Here's the response to that example query for Austin, TX:

```
{
    "RESULTS": [
        {
            "c": "US",
            "l": "/q/zmw:78701.1.99999",
            "lat": "30.271158",
            "ll": "30.271158 -97.741699",
```

```
            "lon": "-97.741699",
            "name": "Austin, Texas",
            "type": "city",
            "tz": "America/Chicago",
            "tzs": "CDT",
            "zmw": "78701.1.99999"
        }
    ]
}
```


There are a few things to note about this request and response. The first is that we didn't need to use our API key. This is because the *AutoComplete* API doesn't require an API key. The second is that there's only a single result for Austin, TX. The third is the *l* value which is the unique link to Austin, TX that we can use when accessing the other API endpoints such as *Alerts* and *Conditions*.

Alerts API

The *Alerts* API provides information about severe weather alerts such as tornado warnings, flood warnings and other special weather statements. We'll be collecting these alerts to create corresponding Zenoss events. This way operators can know when severe weather may be impacting areas of concern.

Here's an example query for alerts in Austin, TX:

```
http://api.wunderground.com/api/api_key/alerts/q/zmw:78701.1.99999.
json
```

Note: Note how the URL ends with /alerts/<link>.json using the *l* link value from the *AutoComplete* query for Austin, TX above.

Here's the relevant portion of the response to an alerts query. Of course Austin doesn't have severe weather so we'll be looking at Des Moines alerts instead:


```
{
    "alerts": [
        {
            "date": "1:07 PM CDT on June 16, 2014",
            "date_epoch": "1402942020",
            "description": "Severe Thunderstorm Warning",
            "expires": "2:15 PM CDT on June 16, 2014",
            "expires_epoch": "1402946100",
            "message": "\nThe National Weather Service in Des Moines has issued␣
↪a\n\n* Severe Thunderstorm Warning for...\n southern Crawford County in west␣
↪central Iowa...\n western Carroll County in west central Iowa...\n northwestern␣
↪Audubon County in west central Iowa...\n\n* until 215 PM CDT\n\n* at 107 PM CDT...a␣
↪severe thunderstorm was located 6 miles southwest\n of Earling...or 22 miles␣
↪southwest of Denison...moving northeast at\n 25 mph.\n\n Hazard...half dollar size␣
↪hail. \n\n Source...radar indicated. \n\n Impact...damage to vehicles is expected.␣
↪\n\n* Locations impacted include...\n Denison...Manning...Dunlap...Manilla...Dow␣
↪City...Arcadia...Vail...\n Templeton...Westside...Halbur...Arion...gray...Buck␣
↪Grove...\n Aspinwall...Denison Municipal Airport and Manning Municipal\n Airport.
↪\n\nPrecautionary/preparedness actions...\n\nA Tornado Watch remains in effect for␣
↪the warned area. Tornadoes can\ndevelop quickly from severe thunderstorms. Although␣
↪a tornado is not\nimmediately likely...if one is spotted...act quickly and move to␣
↪a\nplace of safety inside a sturdy structure...such as a basement or\nsmall␣
↪interior room.\n\nFor your protection move to an interior room on the lowest floor␣
↪of a\nbuilding.\n\nTo report severe weather contact your nearest law enforcement␣
↪agency.\nThey will send your report to the National Weather Service office in\nDes␣
↪Moines .\n\n\nA Tornado Watch remains in effect until 800 PM CDT Monday evening␣
↪for\nnorthwest Iowa.\n\nLat...Lon 4219 9506 4176 [illegible] 9564 4192 9567 4195 9568\ntime...Mot...loc 1807z 236deg 24kt 4172 9552 \n\nHail...1.
↪25in\nwind...<50mph\n\n\nRev\n\n\n",
```

```
            "phenomena": "SV",
            "significance": "W",
            "type": "WRN",
            "tz_long": "America/Chicago",
            "tz_short": "CDT"
        }
    ]
}
```

It's easy to imagine turning this alert into a Zenoss event. We'll see how to do this a bit later. The *Alerts* API documentation has a link to a document that describes what the *phenomena*, *significance*, and *type* values represent.

Conditions API

The *Conditions* API provides information about current weather conditions for a given location. The *Conditions* API is used in exactly the same way as the *Alerts* API, and accepts the same *link* to specify the location. There's a lot of numeric data that would be useful to graph and threshold as Zenoss datapoints.

Here's an example query for conditions in Austin, TX:

```
http://api.wunderground.com/api/api_key/conditions/q/zmw:78701.1.
99999.json
```

Here's the relevant portion of the response to a conditions query:

```
{
    "current_observation": {
        "UV": "1",
        "dewpoint_c": 11,
        "dewpoint_f": 51,
        "dewpoint_string": "51 F (11 C)",
        "display_location": {
            "city": "San Francisco",
            "country": "US",
            "country_iso3166": "US",
            "elevation": "47.00000000",
            "full": "San Francisco, CA",
            "latitude": "37.77500916",
            "longitude": "-122.41825867",
            "magic": "1",
            "state": "CA",
            "state_name": "California",
            "wmo": "99999",
            "zip": "94101"
        },
        "estimated": {},
        "feelslike_c": "13.9",
        "feelslike_f": "57.0",
        "feelslike_string": "57.0 F (13.9 C)",
        "forecast_url": "http://www.wunderground.com/US/CA/San_Francisco.html",
        "heat_index_c": "NA",
        "heat_index_f": "NA",
        "heat_index_string": "NA",
        "history_url": "http://www.wunderground.com/weatherstation/WXDailyHistory.asp?
↪ID=KCASANFR58",
        "icon": "partlycloudy",
        "icon_url": "http://icons.wxug.com/i/c/k/partlycloudy.gif",
        "image": {
```

```
            "link": "http://www.wunderground.com",
            "title": "Weather Underground",
            "url": "http://icons.wxug.com/graphics/wu2/logo_130x80.png"
        },
        "local_epoch": "1402931138",
        "local_time_rfc822": "Mon, 16 Jun 2014 08:05:38 -0700",
        "local_tz_long": "America/Los_Angeles",
        "local_tz_offset": "-0700",
        "local_tz_short": "PDT",
        "nowcast": "",
        "ob_url": "http://www.wunderground.com/cgi-bin/findweather/getForecast?
↪query=37.773285,-122.417725",
        "observation_epoch": "1402931132",
        "observation_location": {
            "city": "SOMA - Near Van Ness, San Francisco",
            "country": "US",
            "country_iso3166": "US",
            "elevation": "49 ft",
            "full": "SOMA - Near Van Ness, San Francisco, California",
            "latitude": "37.773285",
            "longitude": "-122.417725",
            "state": "California"
        },
        "observation_time": "Last Updated on June 16, 8:05 AM PDT",
        "observation_time_rfc822": "Mon, 16 Jun 2014 08:05:32 -0700",
        "precip_1hr_in": "0.00",
        "precip_1hr_metric": " 0",
        "precip_1hr_string": "0.00 in ( 0 mm)",
        "precip_today_in": "0.00",
        "precip_today_metric": "0",
        "precip_today_string": "0.00 in (0 mm)",
        "pressure_in": "29.89",
        "pressure_mb": "1012",
        "pressure_trend": "+",
        "relative_humidity": "81%",
        "solarradiation": "--",
        "station_id": "KCASANFR58",
        "temp_c": 13.9,
        "temp_f": 57.0,
        "temperature_string": "57.0 F (13.9 C)",
        "visibility_km": "16.1",
        "visibility_mi": "10.0",
        "weather": "Scattered Clouds",
        "wind_degrees": 238,
        "wind_dir": "WSW",
        "wind_gust_kph": 0,
        "wind_gust_mph": 0,
        "wind_kph": 4.8,
        "wind_mph": 3.0,
        "wind_string": "From the WSW at 3.0 MPH",
        "windchill_c": "NA",
        "windchill_f": "NA",
        "windchill_string": "NA"
    }
}
```

4.5.2 Create the ZenPack

The first thing we'll need to do is create the Weather Underground ZenPack. We'll use zenpacklib to create this ZenPack from the command line using the following steps. These commands should be run as the *zenoss* user.

```
cd /z
zenpacklib --create ZenPacks.training.WeatherUnderground
```

You should see output similar to the following. Most importantly that *zenpack.yaml* file is being created.

```
Creating source directory for ZenPacks.training.WeatherUnderground:
  - making directory: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground
  - creating file: ZenPacks.training.WeatherUnderground/setup.py
  - creating file: ZenPacks.training.WeatherUnderground/MANIFEST.in
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/datasources/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/thresholds/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/parsers/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/migrate/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/resources/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/modeler/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/tests/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/libexec/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/modeler/plugins/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/lib/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/__init__.py
  - creating file: ZenPacks.training.WeatherUnderground/ZenPacks/training/
↪WeatherUnderground/zenpack.yaml
```

Define zProperties and Classes

The *zenpack.yaml* that's created within the ZenPack source directory above contains only the absolute minimum to be a valid YAML file. Let's take a look at its current contents.

1. First let's set a couple of environment variables to reduce some typing.

   ```
   export ZP_TOP_DIR=/z/ZenPacks.training.WeatherUnderground
   export ZP_DIR=$ZP_TOP_DIR/ZenPacks/training/WeatherUnderground
   ```

2. Now let's look at the contents of zenpack.yaml.

   ```
   cd $ZP_DIR
   cat zenpack.yaml
   ```

You should only see the following line.

```
name: ZenPacks.training.WeatherUnderground
```

3. Replace the contents of zenpack.yaml with the following.

```
name: ZenPacks.training.WeatherUnderground

zProperties:
  DEFAULTS:
    category: Weather Underground

  zWundergroundAPIKey: {}
  zWundergroundLocations:
    type: lines
    default:
      - Austin, TX
      - San Jose, CA
      - Annapolis, MD

classes:
  WundergroundDevice:
    base: [zenpacklib.Device]
    label: Weather Underground API

  WundergroundLocation:
    base: [zenpacklib.Component]
    label: Location

    properties:
      country_code:
        label: Country Code

      timezone:
        label: Time Zone

      api_link:
        label: API Link
        grid_display: False

class_relationships:
  - WundergroundDevice 1:MC WundergroundLocation
```

You can see this YAML defines the following important aspects of our ZenPack.

1. The *name* field is mandatory. It must match the name of the ZenPack's source directory.
2. The *zProperties* field contains configuration properties we want the ZenPack to add to the Zenoss system when it is installed.

   Note that *DEFAULTS* is not added as configuration property. It is a special value that will cause it's properties to be added as the default for all of the other listed zProperties. Specifically in this case it will cause the *category* of *zWundergroundAPIKey* and *zWundergroundLocations* to be set to `Weather Underground`. This is a convenience to avoid having to repeatedly type the category for each added property.

   The *zWundergroundAPIKey* zProperty has an empty dictionary (`{}`). This is because we want it to be a *string* type with an empty default value. These happen to be the defaults so they don't need to be specified.

The *zWundergroundLocations* property uses the *lines* type which allows the user to specify multiple lines of text. Each line will be turned into an element in a list which you can see is also how the default value is specified. The idea here is that unless the user configures otherwise, we will default to monitoring weather alerts and conditions for Austin, TX, San Jose, CA, and Annapolis, MD.

3. The *classes* field contains each of the object classes we want the ZenPack to add.

   In this case we're adding *WundergroundDevice* which because *base* is set to *Device* will be a subclass or specialization of the standard Zenoss device type. We're also adding *WundergroundLocation* which because *base* is set to *Component* will be a subclass of the standard component type.

   The *label* for each is simply the human-friendly name that will be used to refer to the resulting objects when they're seen in the Zenoss web interface.

   The *properties* for *WundergroundLocation* are extra bits of data we want to model from the API and show to the user in the web interface. *order* will be used to show the properties in the defined order, and setting *grid_display* to false for *api_link* will allow it be shown in the details panel of the component, but not in the component grid.

4. *class_relationships* uses a simple syntax to define a relationship between *WundergroundDevice* and *WundergroundLocation*. Specifically it is saying that each (1) *WundergroundDevice* object can contain many (MC) *WundergroundLocation* objects.

Install the ZenPack

Creating the ZenPack with zenpacklib doesn't install the ZenPack for you. So you must now install the ZenPack in developer (–link) mode.

1. Run the following command to install the ZenPack in developer mode.

```
zenpack --link --install $ZP_TOP_DIR
```

4.5.3 Create a Modeler Plugin

Now that we've created a *WundergroundLocation* component type, we need to create a modeler plugin to create locations in the database. We're dealing with a custom HTTP API, so we'll want to base our modeler plugin on the *PythonPlugin* class. This gives us full control of both the collection and processing of the modeling data.

The modeler plugin will pass each location the user has specified in the *zWundergroundLocations* property to Weather Underground's *AutoComplete* API to retrieve some basic information about the location, and very importantly the *l* (*link*) that uniquely identifies the location. The *link* will later be used to monitor the alerts and conditions for the location.

Use the following steps to create our modeler plugin.

1. Make the directory that will contain our modeler plugin.

```
mkdir -p $ZP_DIR/modeler/plugins/WeatherUnderground
```

2. Create `__init__.py` or *dunder-init* files.

```
touch $ZP_DIR/modeler/__init__.py
touch $ZP_DIR/modeler/plugins/__init__.py
touch $ZP_DIR/modeler/plugins/WeatherUnderground/__init__.py
```

   These empty `__init__.py` files are mandatory if we ever expect Python to import modules from these directories.

3. Create `$ZP_DIR/modeler/plugins/WeatherUnderground/Locations.py` with the following contents.

```
"""Models locations using the Weather Underground API."""

# stdlib Imports
import json
import urllib

# Twisted Imports
from twisted.internet.defer import inlineCallbacks, returnValue
from twisted.web.client import getPage

# Zenoss Imports
from Products.DataCollector.plugins.CollectorPlugin import PythonPlugin


class Locations(PythonPlugin):

    """Weather Underground locations modeler plugin."""

    relname = 'wundergroundLocations'
    modname = 'ZenPacks.training.WeatherUnderground.WundergroundLocation'

    requiredProperties = (
        'zWundergroundAPIKey',
        'zWundergroundLocations',
        )

    deviceProperties = PythonPlugin.deviceProperties + requiredProperties

    @inlineCallbacks
    def collect(self, device, log):
        """Asynchronously collect data from device. Return a deferred."""
        log.info("%s: collecting data", device.id)

        apikey = getattr(device, 'zWundergroundAPIKey', None)
        if not apikey:
            log.error(
                "%s: %s not set. Get one from http://www.wunderground.com/weather/
↪api",
                device.id,
                'zWundergroundAPIKey')

            returnValue(None)

        locations = getattr(device, 'zWundergroundLocations', None)
        if not locations:
            log.error(
                "%s: %s not set.",
                device.id,
                'zWundergroundLocations')

            returnValue(None)

        rm = self.relMap()

        for location in locations:
            try:
```

```
                response = yield getPage(
                    'http://autocomplete.wunderground.com/aq?query={query}'
                    .format(query=urllib.quote(location)))

                response = json.loads(response)
            except Exception, e:
                log.error(
                    "%s: %s", device.id, e)

                returnValue(None)

            for result in response['RESULTS']:
                rm.append(self.objectMap({
                    'id': self.prepId(result['zmw']),
                    'title': result['name'],
                    'api_link': result['l'],
                    'country_code': result['c'],
                    'timezone': result['tzs'],
                    }))

        returnValue(rm)

    def process(self, device, results, log):
        """Process results. Return iterable of datamaps or None."""
        return results
```

While it looks like there's quite a bit of code in this modeler plugin, a lot of that is the kind of error handling you'd want to do in a real modeler plugin. Let's walk through some of the highlights.

(a) Imports

We import the standard *json* module because the Weather Underground API returns json-encoded responses that we'll want to convert to Python data structures.

We import *inlineCallBacks* and *returnValue* because the *PythonPlugin.collect* method should return a *Deferred* so that it can be executed asynchronously by zenmodeler. You don't need to use *inlineCallbacks*, but I find it to be a nice way to make Twisted's asynchronous callback-based code look more procedural and be easier to understand. I recommend Dave Peticolas' excellent Twisted Introduction for learning more about Twisted. *inlineCallback* is covered in part 17.

We also import Twisted's *getPage* function. This is an extremely easy to use function for asynchronously fetching a URL.

We import *PythonPlugin* because it will be the base class for our modeler plugin class. It's the best choice for modeling data from HTTP APIs.

(b) *Locations* Class

Remember that your modeler plugin's class name must match the filename or Zenoss won't be able to load it. So because we named the file `Locations.py` we must name the class `Locations`.

(c) *relname* and *modname* Properties

These should be defined in this way for modeler plugins that fill a single relationship like we're doing in this case. It states that this modeler plugin creates objects in the device's *wundergroundLocations* relationship, and that it creates objects of the *ZenPacks.training.WeatherUnderground.WundergroundLocation* type within this relationship.

Where does relname come from? It comes from the `WundergroundDevice 1:MC WundergroundLocation` relationship we defined in `zenpack.yaml`. Because it's a *to-many*

relationship to the *WundergroundLocation* type, *zenpacklib* will name the relationship by lowercasing the first letter and adding an "s" to the end to make it plural.

Where does modname come from? It will be <name-of-zenpack>.<name-of- class>. So because we defined the *WundergroundLocation* class in `zenpack.yaml` , and the ZenPack's name is *ZenPacks.training.WeatherUnderground*, the modname will be *ZenPacks.training.WeatherUnderground.WundergroundLocation*.

(d) *deviceProperties* Properties

The class' *deviceProperties* property provides a way to get additional device properties available to your modeler plugin's *collect* and *process* methods. The default properties that will be available for a *PythonPlugin* are: *id*, *manageIp*, *_snmpLastCollection*, *_snmpStatus*, and *zCollectorClientTimeout*. Our modeler plugin will also need to know what values the user has set for *zWundergroundAPIKey* and *zWundergroundLocations*. So we add those to the defaults.

(e) *collect* Method

The *collect* method is something *PythonPlugin* has, but other base modeler plugin types like *SnmpPlugin* don't. This is because you must write the code to collect the data to be processed, and that's exactly what you should do in the *collect* method.

While the *collect* method can return either normal results or a *Deferred*, it is highly recommend to return a *Deferred* to keep zenmodeler from blocking while your *collect* method executes. In this example we've decorated the method with `@inlineCallbacks` and have returned out data at the end with `returnValue(rm)`. This causes it to return a *Deferred*. By decorating the method with `@inlineCallbacks` we're able to make an asynchronous request to the Weather Underground API with `response = yield getPage(...)`.

The first thing we do in the *collect* method is log an informational message to let the user know what we're doing. This log will appear in `zenmodeler.log`, or on the console if we run *zenmodeler* in the foreground, or in the web interface when the user manually remodels the device.

Next we make sure that the user has configured a value for *zWundergroundAPIKey*. This isn't strictly necessary here because the modeler plugin only uses Weather Underground's *AutoComplete* API which doesn't require an API key. I put this check here because I didn't want to get into a situation where the locations modeled successfully, but then failed to collect because an API key wasn't set.

Next we make sure that the user as configured at least one location in *zWundergroundLocations*. This is mandatory because this controls what locations will be modeled.

Next we create *rm* which is a common convention we use in modeler plugins and stands for *RelationshipMap*. Because we set the *relname* and *modname* class properties this will create a *RelationshipMap* with it's *relname* and *modname* set to the same.

Now we iterate through each location making a call to the *AutoComplete* API for each. For each matching location in the response we will append an *ObjectMap* to *rm* with some key properties set.

- *id* is mandatory and should be set to a value unique to all components on the device. If you look back the example *AutoComplete* response you'll see that the *zmw* property is useful for this purpose. Note that *prepId* should always be used for *id*. It will make any string safe to use as a Zenoss *id*.
- *title* will default to the value of *id* if it isn't set. It's usually a good idea to explicitly set it as we're doing here. It should be a human-friendly label for the component. The location's *name* is a good candidate for this. It will look something like "Austin, Texas".
- *api_link* is a property we defined for the *WundergroundLocation* class in `zenpack.yaml`. This is where we'll store the returned *link* or *l* property. This will be important for monitoring the alerts and conditions of the location later on.

- *country_code* is another property we defined. It's purely informational and will simply be shown to the user when they're viewing the location in the web interface.
- *timezeone* is another property we defined just for informational purposes.

(f) *process* Method

The *process* method is usually where you take the data in the *results* argument and process it into DataMaps to return. However, in the case of *PythonPlugin* modeler plugins, the data returned from the *collect* method will be passed into *process* as the *results* argument. In this case that is already completely processed data. So we just return it.

4. Restart Zenoss.

After adding a new modeler plugin you must restart Zenoss. During development like this, it would be enough to just restart Zope and zenhub with the following commands.

```
serviced service restart zope
serviced service restart zenhub
```

That's it. The modeler plugin has been created. Now we just need to do some Zenoss configuration to allow us to use it.

4.5.4 Add a Device Class

To support adding our special *WundergroundDevice* devices that we defined in `zenpack.yaml` to Zenoss we must create a new device class. This will give us control of the *zPythonClass* configuration property that defines what type of devices will be created. It will also allow us to control what modeler plugins and monitoring templates will be used.

Use the following steps to add the device class.

1. Add the following content to the end of `$ZP_DIR/zenpack.yaml`.

```
device_classes:
  /WeatherUnderground:
    zProperties:
      zPythonClass: ZenPacks.training.WeatherUnderground.WundergroundDevice
      zPingMonitorIgnore: true
      zSnmpMonitorIgnore: true
      zCollectorPlugins:
        - WeatherUnderground.Locations
```

Let's take a look at what we're doing here.

1. First we're saying the device class is going to be */WeatherUnderground*. We add it at the top level because it doesn't fall into one of the existing categories like /Server or /Network.
2. Next we set *zPythonClass* to *ZenPacks.training.WeatherUnderground.WundergroundDevice*. The zPythonClass property controls what type of device will be created in this device class. Note that the value for this is the name of the ZenPack followed by the name of the class we created in the above *classes* section.
3. We then set both *zPingMonitorIgnore* and *zSnmpMonitorIgnore* to true to prevent any ping or SNMP monitoring Zenoss would perform on the device by default. Neither of these make sense since we're dealing with an HTTP API, not a traditional device.
4. Finally we set *zCollectorPlugins* to contain the name of the modeler plugin we created in the previous section. Note that *zCollectorPlugins* is a *lines* property, meaning it accepts multiple values in a list format.

2. Reinstall the ZenPack to create the device class.

```
zenpack --link --install $ZP_TOP_DIR
```

Add a Device

Now would be a good time to add a device to the new device class. There are many ways to add a device to Zenoss. Either of the following approaches can be easily done from the command line.

Using *zendisc*

Using *zendisc* is the easiest way to add device from the command line. However, it only lets you specify the device class and the device's address.

Run the following command to add *wunderground.com*.

```
zendisc run --deviceclass=/WeatherUnderground --device=wunderground.com
```

You should see output similar to the following.

```
INFO zen.ZenModeler: Collecting for device wunderground.com
INFO zen.ZenModeler: No WMI plugins found for wunderground.com
INFO zen.ZenModeler: Python collection device wunderground.com
INFO zen.ZenModeler: plugins: WeatherUnderground.Locations
INFO zen.PythonClient: wunderground.com: collecting data
ERROR zen.PythonClient: wunderground.com: zWundergroundAPIKey not set. Get one from␣
↪http://www.wunderground.com/weather/api
INFO zen.PythonClient: Python client finished collection for wunderground.com
WARNING zen.ZenModeler: The plugin WeatherUnderground.Locations returned no results.
INFO zen.ZenModeler: No change in configuration detected
INFO zen.ZenModeler: No command plugins found for wunderground.com
INFO zen.ZenModeler: SNMP monitoring off for wunderground.com
INFO zen.ZenModeler: No portscan plugins found for wunderground.com
INFO zen.ZenModeler: Scan time: 0.02 seconds
INFO zen.ZenModeler: Daemon ZenModeler shutting down
```

Note: The error about *zWundergroundAPIKey* not being set is expected because we haven't set it. The solution is to go to the *wunderground.com* device in the web interface and add your API key to the *zWundergroundAPIKey* configuration property. After adding the API key you should remodel the device.

Using *zenbatchload*

Another good way to add a device to Zenoss from the command line is *zenbatchload*. Using *zenbatchload* also allows us to set configuration properties such as *zWundergroundAPIKey* as the device is added.

Create a `/z/wunderground.zenbatchload` file with the following contents.

```
/Devices/WeatherUnderground
wunderground.com zWundergroundAPIKey='<your-api-key>', zWundergroundLocations=[
↪'Austin, TX', 'Des Moines, IA']
```

Before you remodel the device, you need to remove the existing device, or its stored state will prevent remodeling. Find your wunderground.com device in the device list. Select it, and click the Remove Devices button (has a Do Not Enter icon).

Now run the following command to load from that file.

```
zenbatchload wunderground.zenbatchload
```

You should now be able to see a list of locations on the *wunderground.com* device!

4.5.5 Datasource Plugin (Events)

Now that we have one or more locations modeled on our *wunderground.com* device, we'll want to start monitoring each location. Using *PythonCollector* we have the ability to create events, record datapoints and even update the model. We'll start with an example that creates events from weather alert data.

The idea will be that we'll create events for locations that have outstanding weather alerts such as tornado warnings. We'll try to capture severity data so tornado warnings are higher severity events than something like a frost advisory.

Using PythonCollector

Before using a Python plugin in our ZenPack, we must make sure we install the PythonCollector ZenPack, and make it a requirement for our ZenPack.

The *PythonCollector* ZenPack adds the capability to write high performance datasources in Python. They will be collected by the *zenpython* daemon that comes with the *PythonCollector* ZenPack. I'd recommend reading the PythonCollector Documentation for more information.

Installing PythonCollector

The first thing we'll need to do is to make sure the *PythonCollector* ZenPack is installed on our system. If it isn't, follow these instructions to install it.

1. Download the latest release from the PythonCollector page.
2. Run the following command to install the ZenPack:

   ```
   zenpack --install ZenPacks.zenoss.PythonCollector-<version>.egg
   ```

3. Restart Zenoss.

Add PythonCollector Dependency

Since we're going to be using *PythonCollector* capabilities in our ZenPack we must now update our ZenPack to define the dependency.

Follow these instructions to define the dependency.

1. Navigate to *Advanced* -> *Settings* -> *ZenPacks*.
2. Click into the *ZenPacks.training.WeatherUnderground* ZenPack.
3. Check *ZenPacks.zenoss.PythonCollector* in the list of dependencies.
4. Click *Save*.
5. Export the ZenPack.

Create the Alerts Plugin

Follow these steps to create the *Alerts* data source plugin:

1. Create `$ZP_DIR/dsplugins.py` with the following contents.

```
"""Monitors current conditions using the Weather Underground API."""

# Logging
import logging
LOG = logging.getLogger('zen.WeatherUnderground')

# stdlib Imports
import json
import time

# Twisted Imports
from twisted.internet.defer import inlineCallbacks, returnValue
from twisted.web.client import getPage

# PythonCollector Imports
from ZenPacks.zenoss.PythonCollector.datasources.PythonDataSource import (
    PythonDataSourcePlugin,
    )


class Alerts(PythonDataSourcePlugin):

    """Weather Underground alerts data source plugin."""

    @classmethod
    def config_key(cls, datasource, context):
        return (
            context.device().id,
            datasource.getCycleTime(context),
            context.id,
            'wunderground-alerts',
            )

    @classmethod
    def params(cls, datasource, context):
        return {
            'api_key': context.zWundergroundAPIKey,
            'api_link': context.api_link,
            'location_name': context.title,
            }

    @inlineCallbacks
    def collect(self, config):
        data = self.new_data()

        for datasource in config.datasources:
            try:
                response = yield getPage(
                    'http://api.wunderground.com/api/{api_key}/alerts{api_link}.
↪json'
                    .format(
                        api_key=datasource.params['api_key'],
                        api_link=datasource.params['api_link']))
```

```
            response = json.loads(response)
        except Exception:
            LOG.exception(
                "%s: failed to get alerts data for %s",
                config.id,
                datasource.location_name)

            continue

        for alert in response['alerts']:
            severity = None

            if int(alert['expires_epoch']) <= time.time():
                severity = 0
            elif alert['significance'] in ('W', 'A'):
                severity = 3
            else:
                severity = 2

            data['events'].append({
                'device': config.id,
                'component': datasource.component,
                'severity': severity,
                'eventKey': 'wu-alert-{}'.format(alert['type']),
                'eventClassKey': 'wu-alert',

                'summary': alert['description'],
                'message': alert['message'],

                'wu-description': alert['description'],
                'wu-date': alert['date'],
                'wu-expires': alert['expires'],
                'wu-phenomena': alert['phenomena'],
                'wu-significance': alert['significance'],
                'wu-type': alert['type'],
                })

    returnValue(data)
```

Let's walk through this code to explain what is being done.

(a) Logging

The first thing we do is import *logging* and create *LOG* as our logger. It's important that the name of the logger in the `logging.getLogger()` begins with `zen.`. You will not see your logs otherwise.

The stdlib and Twisted imports are almost identical to what we used in the modeler plugin, and they're used for the same purposes.

Finally we import *PythonDataSourcePlugin* from the *PythonCollector* ZenPack. This is the class our data source plugin will extend, and basically allows us to write code that will be executed by the *zenpython* collector daemon.

(b) *Alerts* Class

Unlike our modeler plugin, there's no need to make the plugin class' name the same as the filename. As we'll see later when we're setting up the monitoring template that will use this plugin, there's no specific name for the file or the class required because we configure where to find the plugin in the datasource configuration within the monitoring template.

(c) *config_key* Class Method

The *config_key* method must have the `@classmethod` decorator. It is passed *datasource*, and *context*. The *datasource* argument will be the actual datasource that the user configures in the monitoring templates section of the web interface. It has properties such as *eventClass*, *severity*, and as you can see a *getCycleTime()* method that returns the interval at which it should be polled. The *context* argument will be the object to which the monitoring template and datasource is bound. In our case this will be a location object such as Austin, TX.

The purpose of the *config_key* method is to split monitoring configuration into tasks that will be executed by the zenpython daemon. The zenpython daemon will create one task for each unique value returned from *config_key*. It should be used to optimize the way data is collected. In some cases it is possible to make a single query to an API to get back data for many components. In these cases it would be wise to remove `context.id` from the config_key so we get one task for all components.

In our case, the Weather Underground API must be queried once per location so it makes more sense to put `context.id` in the config_key so we get one task per location.

The value returned by *config_key* will be used when *zenpython* logs. So adding something like *wundergrround-alerts* to the end makes it easy to see logs related to collecting alerts in the log file.

The *config_key* method will only be executed by *zenhub*. So you must restart *zenhub* if you make changes to the *config_key* method. This also means that if there's an exception in the *config_key* method it will appear in the *zenhub* log, not *zenpython*.

(d) *params* Class Method

The *params* method must have the `@classmethod` decorator. It is passed the same *datasource* and *context* arguments as *config_key*.

The purpose of the *params* method is to copy information from the Zenoss database into the *config.datasources[*]* that will be passed as an argument to the *collect* method. Since the *collect* method is run by *zenpython* it won't have direct access to the database, so it relies on the *params* method to provide it with any information it will need to collect.

In our case you can see that we're copying the context's *zWundergroundAPIKey*, *api_link* and *title* properties. All of these will be used in the *collect* method.

Just like the *config_key* method, *params* will only be executed by *zenhub*. So be sure to restart *zenhub* if you make changes, and look in the *zenhub* log for errors.

(e) *collect* Method

The *collect* method does all of the real work. It will be called once per cycletime. It gets passed a *config* argument which for the most part has two useful properties: *config.id* and *config.datasources*. *config.id* will be the device's id, and *config.datasources* is a list of the datasources that need to be collected.

You'll see in the collect method that each datasource in *config.datasources* has some useful properties. *datasource.component* will be the id of the component against which the datasource is run, or blank in the case of a device-level monitoring template. *datasource.params* contains whatever the *params* method returned.

Within the body of the collect method we see that we create a new *data* variable using `data = self.new_data()`. *data* is a place where we stick all of the collected events, values and maps. *data* looks like the following:

```
data = {
    'events': [],
    'values': defaultdict(<type 'dict'>, {}),
    'maps': [],
}
```

Next we iterate over every configured datasource. For each one we make a call to Weather Underground's *Alerts* API, then iterate over each alert in the response creating an event for each.

The following standard fields are being set for every event. You should read Zenoss' event management documentation if the purpose of any of these fields is not clear. I highly recommend setting all of these fields to an appropriate value for any event you send into Zenoss to improve the ability of Zenoss and Zenoss' operators to manage the events.

- *device*: Mandatory. The device id related to the event.
- *component*: Optional. The component id related to the event.
- *severity*: Mandatory. The severity for the event.
- *eventKey*: Optional. A further uniqueness key for the event. Used for de-duplication and clearing.
- *eventClassKey*: Optional. An identifier for the *type* of event. Used during event class mapping.
- *summary*: Mandatory: A (hopefully) short summary of the event. Truncated to 128 characters.
- *message*: Optional: A longer text description of the event. Not truncated.

You will also see many *wu-** fields being added to the event. Zenoss allows arbitrary fields on events so it can be a good practice to add any further information you get about the event in this way. It can make understanding and troubleshooting the resulting event easier.

Finally we return data with all of events we appended to it. *zenpython* will take care of getting the events sent from this point.

2. Restart Zenoss.

   After adding a new datasource plugin you must restart Zenoss. While developing it's enough to just restart *zenhub* with the following command.

```
serviced service restart zenhub
```

That's it. The datasource plugin has been created. Now we just need to do some Zenoss configuration to allow us to use it.

Configure Monitoring Templates

Rather than use the web interface to manually create a monitoring template, we'll specify it in our *zenpack.yaml* instead.

1. Edit *$ZP_DIR/zenpack.yaml* and add the *templates* section below to the existing */WeatherUnderground'* device class.

```
device_classes:
  /WeatherUnderground:
    templates:
      Location:
        description: Location weather monitoring using the Weather Underground␣
↪API.
        targetPythonClass: ZenPacks.training.WeatherUnderground.
↪WundergroundLocation

        datasources:
          alerts:
            type: Python
```

```
            plugin_classname: ZenPacks.training.WeatherUnderground.dsplugins.
↪Alerts
            cycletime: "600"
```

At least some of this should be self-explanatory. The YAML vocabulary has been designed to be as intuitive and concise as possible. Let's walk through it.

(a) The highest-level element (based on indentation) is */WeatherUnderground/Location*. This means to create a *Location* monitoring template in the */WeatherUnderground* device class.

Note: The monitoring template must be called `Location` because that is the *label* for the *WundergroundLocation* class to which we want the template bound.

(b) The *description* is for documentation purposes and should describe the purpose of the monitoring template.

(c) The *targetPythonClass* is a hint to what type of object the template is meant to be bound to. Currently this is only used to determine if users should be allowed to manually bind the template to device classes or devices. Providing a valid component type like we've done prevents users from making this mistake.

(d) Next we have *datasources* with a single *alerts* datasource defined.

The *alerts* datasource only has three properties:

- *type*: This is what makes *zenpython* collect the data.
- *plugin_classname*: This is the fully-qualified class name for the *PythonDataSource* plugin we created that will be responsible for collecting the datasource.
- *cycletime*: The interval in seconds at which this datasource should be collected.

2. Reinstall the ZenPack to add the monitoring templates.

   Some sections of *zenpack.yaml* such as zProperties and templates only get created when the ZenPack is installed.

   Run the usual command to reinstall the ZenPack in development mode.

```
zenpack --link --install $ZP_TOP_DIR
```

3. Navigate to *Advanced -> Monitoring Templates* in the web interface to verify that the *Location* monitoring template has been created as defined.

Test Monitoring Weather Alerts

Testing this is a bit tricky since we'll have to be monitoring a location that currently has an active weather alert. Fortunately there's an easy way to find one of these locations.

Follow these steps to test weather alert monitoring:

1. Go to the following URL for the current severe weather map of the United States.

   http://www.wunderground.com/severe.asp

2. Click on one of the colored areas. Orange and red are more exciting. This will take you to the text of the warning. It should reference city or county names.

3. Update *zWundergroundLocations* on the *wunderground.com* device to add one of the cities or counties that has an active weather alert. For example, "Buffalo, South Dakota".

4. Remodel the *wunderground.com* device then verify that the new location is modeled.

5. Run the following command to collect from *wunderground.com*.

```
zenpython run -v10 --device=wunderground.com
```

There will be a lot of output from this command, but we're mainly looking for an event to be sent for the weather alert. It will look similar to the following output:

```
DEBUG zen.zenpython: Queued event (total of 1) {'rcvtime': 1403112635.631883, 'wu-
↪type': u'FIR', 'wu-significance': u'W', 'eventClassKey': 'wu-alert', 'wu-expires
↪': u'8:00 PM MDT on June 18, 2014', 'component': '80901.1.99999', 'monitor':
↪'localhost', 'agent': 'zenpython', 'summary': u'Fire Weather Warning', 'wu-date
↪': u'3:39 am MDT on June 18, 2014', 'manager': 'zendev.damsel.loc', 'eventKey':
↪'wu-alert-FIR', 'wu-phenomena': u'FW', 'wu-description': u'Fire Weather Warning
↪', 'device': 'wunderground.com', 'message': u'\n...Red flag warning remains in␣
↪effect from noon today to 8 PM MDT\nthis evening for gusty winds...low relative␣
↪humidity and dry fuels for\nfire weather zones 222...226 and 227...\n\n*␣
↪affected area...fire weather zones 222...226 and 227.\n\n* Winds...southwest 10␣
↪to 20 mph with gusts up to 35 mph.\n\n* Relative humidity...as low as 13␣
↪percent.\n\n* Impacts...extreme fire behavior will be possible if a fire \n␣
↪starts. \n\nPrecautionary/preparedness actions...\n\nA red flag warning means␣
↪that critical fire weather conditions\nare either occurring now...or will␣
↪shortly. A combination of\nstrong winds...low relative humidity...and warm␣
↪temperatures can\ncontribute to extreme fire behavior.\n\n\n\n\n', 'device_guid
↪': 'f59e7e4d-be5d-4b86-b005-7357ce58f79c', 'severity': 3}
```

You should now be able to confirm that this event was created in the Zenoss event console.

4.5.6 Datasource Plugin (Data Points)

We've already created a data source plugin that creates Zenoss events for weather alerts. Now we want to use the Weather Underground *Conditions* API to monitor current weather conditions for each location. The purpose of this is to illustrate that these Python data source plugins can also be used to collect datapoints.

Create Conditions Data Source Plugin

Follow these steps to create the *Conditions* data source plugin:

1. Add the following contents to the end of `$ZP_DIR/dsplugins.py`.

```
class Conditions(PythonDataSourcePlugin):

    """Weather Underground conditions data source plugin."""

    @classmethod
    def config_key(cls, datasource, context):
        return (
            context.device().id,
            datasource.getCycleTime(context),
            context.id,
            'wunderground-conditions',
            )

    @classmethod
    def params(cls, datasource, context):
        return {
            'api_key': context.zWundergroundAPIKey,
```

```
            'api_link': context.api_link,
            'location_name': context.title,
            }

    @inlineCallbacks
    def collect(self, config):
        data = self.new_data()

        for datasource in config.datasources:
            try:
                response = yield getPage(
                    'http://api.wunderground.com/api/{api_key}/conditions{api_
↪link}.json'
                    .format(
                        api_key=datasource.params['api_key'],
                        api_link=datasource.params['api_link']))

                response = json.loads(response)
            except Exception:
                LOG.exception(
                    "%s: failed to get conditions data for %s",
                    config.id,
                    datasource.location_name)

                continue

            current_observation = response['current_observation']
            for datapoint_id in (x.id for x in datasource.points):
                if datapoint_id not in current_observation:
                    continue

                try:
                    value = current_observation[datapoint_id]
                    if isinstance(value, basestring):
                        value = value.strip(' %')

                    value = float(value)
                except (TypeError, ValueError):
                    # Sometimes values are NA or not available.
                    continue

                dpname = '_'.join((datasource.datasource, datapoint_id))
                data['values'][datasource.component][dpname] = (value, 'N')

        returnValue(data)
```

Most of the *Conditions* plugin is almost identical to the *Alerts* plugin so I won't repeat what can be read back in that section. The main difference starts at the `current_observation = response['current_observation']` line of the *collect* method.

It grabs the *current_observation* data from the response then iterates over every datapoint configured on the datasource. This is a nice approach because it allows for some user-flexibility in what datapoints are captured from the *Conditions* API. If the API made *temp_c* and *temp_f* available, we could choose to collect *temp_c* just by adding a datapoint by that name.

The following line is the most important in terms of explaining how to have your plugin return datapoint values.

```
data['values'][datasource.component][dpname] = (value, 'N')
```

We just stick `(value, 'N')` into the component's datapoint dictionary. The `'N'` is the timestamp at which the value occurred. If you know the time it should be specified as the integer UNIX timestamp. Use `'N'` if you don't know. This will use the current time.

2. Restart Zenoss.

   After adding a new datasource plugin you must restart Zenoss. While developing it's enough to just restart *zenhub* with the following command.

```
serviced service restart zenhub
```

That's it. The datasource plugin has been created. Now we just need to do some Zenoss configuration to allow us to use it.

Add Conditions to Monitoring Template

To use this new plugin we'll add a new datasource and corresponding graphs to the existing *Location* monitoring template defined in *zenpack.yaml*.

Follow these steps to update the monitoring template:

1. Update *$ZP_DIR/zenpack.yaml* to add the *conditions* entry within the existing *datasources* section.

```
device_classes:
  /WeatherUnderground:
    templates:
      Location:
        description: Location weather monitoring using the Weather Underground␣
↪API.
        targetPythonClass: ZenPacks.training.WeatherUnderground.
↪WundergroundLocation

        datasources:
          conditions:
            type: Python
            plugin_classname: ZenPacks.training.WeatherUnderground.dsplugins.
↪Conditions
            cycletime: "600"

            datapoints:
              temp_c: GAUGE
              feelslike_c: GAUGE
              heat_index_c: GAUGE
              windchill_c: GAUGE
              dewpoint_c: GAUGE
              relative_humidity: GAUGE
              pressure_mb: GAUGE
              precip_1hr_metric: GAUGE
              UV: GAUGE
              wind_kph: GAUGE
              wind_gust_kph: GAUGE
              visibility_km: GAUGE

        graphs:
          Temperatures:
            units: "degrees C."
```

```
    graphpoints:
      Temperature:
        dpName: conditions_temp_c
        format: "%7.2lf"

      Feels Like:
        dpName: conditions_feelslike_c
        format: "%7.2lf"

      Heat Index:
        dpName: conditions_heat_index_c
        format: "%7.2lf"

      Wind Chill:
        dpName: conditions_windchilltemp_c
        format: "%7.2lf"

      Dewpoint:
        dpName: conditions_dewpoint_c
        format: "%7.2lf"

  Relative Humidity:
    units: percent
    miny: 0
    maxy: 100

    graphpoints:
      Relative Humidity:
        dpName: conditions_relative_humidity
        format: "%7.2lf%%"

  Pressure:
    units: millibars
    miny: 0

    graphpoints:
      Pressure:
        dpName: conditions_pressure_mb
        format: "%7.0lf"

  Precipitation:
    units: centimeters
    miny: 0

    graphpoints:
      1 Hour:
        dpName: conditions_precip_1hr_metric
        format: "%7.2lf"

  UV Index:
    units: UV index
    miny: 0
    maxy: 12

    graphpoints:
      UV Index:
        dpName: conditions_UV
```

```
            format: "%7.0lf"

        Wind Speed:
          units: kph
          miny: 0

          graphpoints:
            Sustained:
              dpName: conditions_wind_kph
              format: "%7.2lf"

            Gust:
              dpName: conditions_wind_gust_kph
              format: "%7.2lf"

        Visibility:
          units: kilometers
          miny: 0

          graphpoints:
            Visibility:
              dpName: conditions_visibility_km
              format: "%7.2lf"
```

You can refer to *Monitoring Templates* for more information on creating monitoring templates in YAML.

2. Reinstall the ZenPack to update the monitoring template.

```
zenpack --link --install $ZP_TOP_DIR
```

3. Navigate to *Advanced -> Monitoring Templates* in the web interface to verify that the *Location* monitoring template has been updated with the *conditions* datasource and corresponding graphs.

Test Monitoring Weather Conditions

Follow these steps to test weather condition monitoring:

1. Run the following command to collect from *wunderground.com*.

```
zenpython run -v10 --device=wunderground.com
```

There will be a lot of output from this command, but we're mainly looking for at least one datapoint being written. If one works, it's likely that they all work. Look for a line similar to the following:

```
DEBUG zen.MetricWriter: publishing metric wunderground.com/conditions_temp_c 14.1␣
↪1452024379
```

4.5.7 Datasource Plugin (Modeling)

The final capability of Python data source plugins is to make changes to the Zenoss model. This allows a data source to make changes to the model in the same way that *zenmodeler* does. Having this capability in a data source allows modeling more frequently than the normal 12 hour *zenmodeler* interval.

To demonstrate this through an exercise, we'll extend the existing *Conditions* plugin to capture the what the *Conditions* API calls *weather* which is some text that looks like "Scattered Clouds" or "Sunny". We'll then show this value for each location in the web interface.

Note: Model updates are much more expensive operations than creating events or collecting datapoints. It is better to perform as much modeling as possible using modeler plugins on their typical 12 hour interval, and perform only the absolutely necessary smaller model updates more frequently using a PythonDataSourcePlugin. Too much modeling activity can result in the degradation of a Zenoss' systems overall performance.

Add Modeling to Conditions Data Source Plugin

Follow these steps to add modeling to the *Conditions* data source plugin:

1. Edit `$ZP_DIR/zenpack.yaml`.

   Add the following *weather* property to the *WundergroundLocation* class between the existing *timezone* and *api_link* properties.

   ```
   weather:
     label: Weather
   ```

2. Edit `$ZP_DIR/dsplugins.py`.

   Add the following needed import to the top of `dsplugins.py`.

   ```
   from Products.DataCollector.plugins.DataMaps import ObjectMap
   ```

   Add the following code to the *Conditions* class' *collect* method right above the `returnValue(data)` line indented one level further. The `returnValue(data)` line is included in the following update to show where the new code should be placed.

   ```
       data['maps'].append(
           ObjectMap({
               'relname': 'wundergroundLocations',
               'modname': 'ZenPacks.training.WeatherUnderground.WundergroundLocation
   ↪',
               'id': datasource.component,
               'weather': current_observation['weather'],
               }))

   returnValue(data)  # existing line
   ```

   The *maps* concept here is exactly the same as it is in modeler plugins. `data['maps']` can contain anything that a modeler plugin's *process* method can return.

2. Don't update the *Location* monitoring template.

   We're adding capability to a datasource that's already configured. No updates are required to the monitoring template.

3. Restart Zenoss.

   If we had only updated the *collect* method of the *Conditions* plugin we would only need to restart *zenpython*. However, because we added the new *weather* property to the *WundergroundLocation* class, we must restart nearly everything, so it's simpler to restart everything.

Test Modeling Current Weather

Follow these steps to test weather condition monitoring:

1. Run the following command to collect from *wunderground.com*.

```
zenpython run -v10 --device=wunderground.com
```

   There will be a lot of output from this command, but we're looking for the following line which indicates that our maps were applied:

```
DEBUG zen.python: wunderground.com 600 21401.1.99999 wunderground-conditions␣
↪sending 1 datamaps
```

2. Navigate to the *Locations* on the *wunderground.com* device and verify that each location shows something in its *Weather* column.

4.6 Troubleshooting

4.6.1 Using the Python Debugger

One of the most powerful tools when debugging the Python portions of a ZenPack is the Python debugger (*pdb*). With *pdb* you can set breakpoints in your code. When the breakpoints are hit, you get a *(pdb)* prompt that has full access to examine the stack and any local or global variables.

To set a breakpoint in your code you add the following line.

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

As with any code change, you must restart the Zenoss process that executes the code in question.

Pickling data

ZenPackLib v2.0 also adds a decorator, *writeDataToFile*, that can be used to save real-world results that your plugins will be processing. This data can then be used to determine why a plugin is not behaving as expected or to create your own unit tests.

In order to use this decorator, import it from the ZenPackLib zenpack:

```
from ZenPacks.zenoss.ZenPackLib.lib.helpers.utils import writeDataToFile
```

Then use as a decorator for your plugin's process function. *writeDataToFile* is generic and can be used on any python function or class method. It does not pickle file or logger objects. You can also specify keywords which, when matched against an object's attributes, will cause an object not to be pickled.

```
class MyPlugin(PythonPlugin):
    @writeDataToFile(keywords=['zCommandPassword', 'windows_password'])
    def process(self, device, results, log):
        '''Perform device specific processing on modeler plugin results'''
        rm = self.relMap()
        # Add data to relationship map
        rm.attr1 = results.attr1
        rm.attr2 = results.attr2
        return rm
```

The save functionality is disabled unless you use the *ZPL_DUMP_DATA* environment variable. Be sure to only use in limited runs or you will end up with a large number of pickle files.

```
$ export ZPL_DUMP_DATA=1; zenmodeler run -d mydevice; unset ZPL_DUMP_DATA
```

The pickle file(s) will be written to your */tmp* folder using the class name and function name with current timestamp. Using the definition from above, the file name would be *MyPlugin_process_XXXXXX.pickle* where *XXXXXX* is the time at which the data was processed. Assuming *device* has either a zCommandPassword or windows_password attribute, the *self*, *device*, and *log* objects will not be pickled.

Known Issues

- When dumping existing event classes using the zenpacklib tool with *–dump-event-classes* option, some transforms and/or explanations may show as either unformatted text within double quotes or as formatted text within single quotes. This is due to how the python yaml package handles strings. Either of these two formats are acceptable when used in zenpack.yaml.
- ZenPacks using earlier verisons of ZenPackLib logged template changes to the console during installation. These messages might have disturbed some users due to their wording and logging as "ERROR" status. These have been revised and now log as informational, but the old format will be displayed when upgrading from a pre-ZenPacklib 2.0 ZenPack to one using the latest version. Subsequent installs will use the newer format.

4.7 Command Line Reference

While most of zenpacklib's functionality is as a Python module to be used as a library for helping build ZenPacks, *zenpacklib* also acts as a command line script to perform some useful actions.

The *zenpacklib* script can be run from the command line with:

```
`$ZENHOME/bin/zenpacklib` (usually `/opt/zenoss/bin/zenpacklib`)
```

Running the command alone or with *–help* will return the following (truncated):

```
Usage: zenpacklib.py [options] [FILENAME|ZENPACK|DEVICE]

ZenPack Conversion:
 -t, --dump-templates
                     export existing monitoring templates to YAML
 -e, --dump-event-classes
                     export existing event classes to YAML
 -r, --dump-process-classes
                     export existing process classes to YAML

ZenPack Development:
 -c, --create        Create a new ZenPack source directory
 -l, --lint          check zenpack.yaml syntax for errors
 -o, --optimize      optimize zenpack.yaml format and DEFAULTS
 -d, --diagram       print YUML (http://yuml.me/) class diagram source
                     based on zenpack.yaml
 -p, --paths         print possible facet paths for a given device and
                     whether currently filtered.
```

The following commands are supported:

- *-c, –create*: Create a new zenpacklib-enabled ZenPack source directory.
- *-l, –lint*: Provides syntax and correctness on a YAML file.
- *-d, –diagram*: Export yUML (yuml.me) class diagram from a YAML file.

- *-t, –dump-templates*: Export existing monitoring templates to YAML.
- *-e, –dump-event-classes*: Export existing event classes and mappings to YAML.
- *-r, –dump-process-classes*: Export existing process classes to YAML.
- *-p, –paths*: Using the specified device, print a report of paths between objects.
- *-o, –optimize*: Optimize the layout of an existing zenpack.yaml file
- *–version*: Print zenpacklib version.

4.7.1 create

The *—create* switch will create a source directory for a zenpacklib-enabled ZenPack. This will include a setup.py, MANIFEST.in, the namespace and module directories, and a zenpack.yaml in the module directory. It will also make a copy of zenpacklib.py inside the module directory. This ZenPack will be ready to be installed immediately though it will contain no functionality yet.

Example usage:

```
zenpacklib --create ZenPacks.example.MyNewPack
```

Running the above command would result in the following output.

```
Creating source directory for ZenPacks.test.ZPLTest2:
  - making directory: ZenPacks.test.ZPLTest2/ZenPacks/test/ZPLTest2
  - creating file: ZenPacks.test.ZPLTest2/setup.py
  - creating file: ZenPacks.test.ZPLTest2/MAINFEST.in
  - creating file: ZenPacks.test.ZPLTest2/ZenPacks/__init__.py
  - creating file: ZenPacks.test.ZPLTest2/ZenPacks/test/__init__.py
  - creating file: ZenPacks.test.ZPLTest2/ZenPacks/test/ZPLTest2/__init__.py
  - creating file: ZenPacks.test.ZPLTest2/ZenPacks/test/ZPLTest2/zenpack.yaml
```

4.7.2 lint

The *—lint* switch will check the provided YAML file for correctness. It checks that the provided file is syntactically-valid YAML, and it will also perform many others checks that validate that the contained entries, fields and their values are valid.

The following example shows an example of using an unrecognized parameter in a monitoring template.

```
zenpacklib --lint zenpack.yaml
zenpack.yaml:47:9: Unrecognized parameter 'targetPythnoClass' found while processing␣
↪RRDTemplateSpec
```

Note: *lint* will provide no output if the provided YAML file is found to be correct.

4.7.3 diagram

The *—diagram* switch will use *Classes and Relationships* in the provided YAML file to output the source for a yUML (http://yuml.me) class diagram. For ZenPacks with a non-trivial class model this can provide a useful view of the model.

Using this example *zenpack.yaml*:

```
name: ZenPacks.example.NetBotz

classes:
  NetBotzDevice:
    base: [zenpacklib.Device]

  NetBotzEnclosure:
    base: [zenpacklib.Component]

  NetBotzSensor:
    base: [zenpacklib.Component]

class_relationships:
  - NetBotzDevice 1:MC NetBotzEnclosure
  - NetBotzDevice 1:MC NetBotzSensor
  - NetBotzEnclosure 1:M NetBotzSensor
```

Then running the following command..

```
zenpacklib --diagram zenpack.yaml
```

Would result in the following yUML class diagram source. You can now paste this into http://yuml.me to see what it looks like.

```
# Classes
[NetBotzDevice]
[NetBotzEnclosure]
[NetBotzSensor]

# Inheritence
[Device]^-[NetBotzDevice]
[Component]^-[NetBotzEnclosure]
[Component]^-[NetBotzSensor]

# Containing Relationships
[NetBotzDevice]++netBotzEnclosures-netBotzDevice[NetBotzEnclosure]
[NetBotzDevice]++netBotzSensors-netBotzDevice[NetBotzSensor]

# Non-Containing Relationships
[NetBotzEnclosure]netBotzSensors-.-netBotzEnclosure++[NetBotzSensor]
```

4.7.4 paths

The *—paths* switch shows the paths between defined component classes in the zenpack, using the device name you have specified as a sample. To obtain useful results, ensure that the device has at least one component of each type you are interested in.

The paths shown are those used to control which devices will show up in the bottom grid of the zenoss UI when a component is selected and a target class is selected from the filter dropdown.

The default behavior is to show component of that type that are directly related to the selected component through 1:M or 1:MC relationships, but additional objects that are indirectly related can be added through the use of the 'extra_paths' configuration directive. *—paths* is primarily intended as a debugging tool during the development of extra_paths patterns to verify that they are having the intended effect.

Example usage:

```
zenpacklib --paths mydevice
```

4.7.5 dump-templates

The *—dump-templates* switch is designed to export monitoring templates already loaded into your Zenoss instance and associated with a ZenPack. It will export them to the YAML format required for *zenpack.yaml*. It is up to you to merge that YAML with your existing *zenpack.yaml*. file.

Example usage:

```
zenpacklib --dump-templates ZenPacks.example.BetterAlreadyBeInstalled
```

4.7.6 dump-event-classes

The *—dump-event-classes* switch is designed to export event class organizers and mappings already loaded into your Zenoss instance and associated with a ZenPack. It will export them to the YAML format required for *zenpack.yaml*. It is up to you to merge that YAML with your existing *zenpack.yaml*. file.

Only event classes sourced from the ZenPack's XML will be exported. Any event classes sourced from the ZenPack's YAML will not be exported. If the YAML for these event classes is desired, it should be copied from the ZenPack's existing YAML.

Example usage:

```
zenpacklib --dump-event-classes ZenPacks.example.BetterAlreadyBeInstalled
```

Note: When dumping existing event classes using the zenpacklib tool with the *–dump-event-classes* option, some transforms and/or explanations may show as either unformatted text within double quotes or as formatted text within single quotes. This is due to how the python yaml package handles strings. Either of these two formats are acceptable when used in zenpack.yaml.

4.7.7 dump-process-classes

The *—dump-process-classes* switch is designed to export process class organizers and classes already loaded into your Zenoss instance and associated with a ZenPack. It will export them to the YAML format required for *zenpack.yaml*. It is up to you to merge that YAML with your existing *zenpack.yaml*. file.

Only process class organizers sourced from the ZenPack's XML will be exported. Any process class organizers sourced from the ZenPack's YAML will not be exported. If the YAML for these process class organizers is desired, it should be copied from the ZenPack's existing YAML.

Example usage:

```
zenpacklib --dump-process-classes ZenPacks.example.BetterAlreadyBeInstalled
```

4.7.8 optimize

The *—optimize* switch (experimental) is designed to examine your *zenpack.yaml* file and rearrange it for brevity and use of DEFAULTS where detected. Once optimized, the command compares the original YAML file to the optimized

version to ensure that the same objects are created. The change detection, however, is still being improved and may output false warnings. It is recommended to double-check the optimized YAML output.

Example usage:

```
zenpacklib --optimize zenpack.yaml
```

4.7.9 version

The —*version* switch prints the zenpacklib version.

Example usage:

```
zenpacklib --version
```

4.8 YAML Reference

A ZenPack's functionality can be described via a *zenpack.yaml* file. The following sections describe the syntax and capabilities of this YAML file.

4.8.1 ZenPack

The ZenPack YAML file, *zenpack.yaml*, contains the specification for a ZenPack. It must at at least contain a *name* field. It may optionally contain one each of *zProperties*, *device_classes*, *classes*, and *class_relationships* fields.

The following example shows an example of a *zenpack.yaml* file with examples of every supported field.

```
 name: ZenPacks.acme.Widgeter

 zProperties:
   zWidgeterEnable: {}

 device_classes:
   /Server/ACME/Widgeter: {}

 classes:
   ACMEWidgeter:
     base: [zenpacklib.Device]

   ACMEWidget:
     base: [zenpacklib.Component]

 class_relationships:
   - Widgeter 1:MC Widget

link_providers:
  Virtual Machine:
    link_class: ZenPacks.example.XenServer
    catalog: device
    device_class: /Server/XenServer
    queries: [vm_id:manageIp]
  XenServer:
    global_search: True
    queries: [manageIp:vm_id]
```

```
event_classes:
  /Status/Acme:
    remove: false
    description: Acme event class
    mappings:
      Widget:
        eventClassKey: WidgetEvent
        sequence:  10
        remove: true
        transform: "if evt.message.find('Error reading value for') >= 0:\n\
          \   evt._action = 'drop'"

process_class_organizers:
  Widget:
    description: Organizer for Widget process classes
    process_classes:
      widget:
        description: Widget process class
        includeRegex: sbin\/widget
        excludeRegex: "\\b(vim|tail|grep|tar|cat|bash)\\b"
        replaceRegex: .*
        replacement: Widget
```

See the following for more information on each of these fields.

- *zProperties*
- *Device Classes*
- *Classes and Relationships*
- *Device Link Providers*
- *Event Classes*
- *Process Classes*

ZenPack Fields

The following fields are valid for a ZenPack entry.

name

> **Description** Name (e.g. ZenPacks.acme.Widgeter). Must begin with "ZenPacks.".
>
> **Required** Yes
>
> **Type** string
>
> **Default Value** None

zProperties

> **Description** zProperties added by the ZenPack
>
> **Required** No
>
> **Type** map<name, *zProperty*>
>
> **Default Value** {} *(empty map)*

device_classes

Description Device classes added by the ZenPack.

Required No

Type map<path, *Device Class*>

Default Value {} *(empty map)*

classes

Description Classes for device and component types added by this ZenPack.

Required No

Type map<name, *Class*>

Default Value {} *(empty map)*

class_relationships

Description Relationships between classes.

Required No

Type list<*Class Relationship*>

Default Value [] *(empty list)*

link_providers

Description Device Link Providers.

Required No

Type list<*Link Provider*>

Default Value [] *(empty list)*

event_classes

Description Event Class organizers and mappings

Required No

Type list<*Event Class*>

Default Value [] *(empty list)*

process_class_organizers

Description Process Class organizers and mappings

Required No

Type list<*Process Class*>

Default Value [] *(empty list)*

4.8.2 zProperties

zProperties are one part of Zenoss' hierarchical configuration system. They are configuration properties that can be specified on any device class including the root /Devices class, and on any individual device.

zProperty Inheritance

The most-specific value for a zProperty within the hierarchy will be used for any given device. For instance, given a device *linux1* in the /Server/Linux device class. The value for zSnmpMonitorIgnore will be checked first on the linux1 device. If it is not set locally on the device, the /Server/Linux device class will then be checked. If not set there, /Server will be checked. Finally the value at / (or /Devices) will be checked as a final resort. Since all zProperties must have a default values that is set at the root device class, there will always be a value for the zProperty. Even if it is an empty string.

Adding zProperties

To add a zProperty to your ZenPack you must include a *zProperties* section in your YAML file. The following example shows an example of adding two zProperties.

```
zProperties:
  zWidgeterEnable:
    category: ACME Widgeter
    type: boolean
    default: true

  zWidgeterInterval:
    category: ACME Widgeter
    type: string
    default: 300
```

Each of these zProperty entries specifies a *category*, *type* and *default*. These are the only valid fields of the a zProperty entry. However, each of these fields has a default value that will be used if the field isn't explicitly specified. For example, the default value for *type* is string. So the above example could be shortened slightly by omitting the explicit *type* on zWidgeterInterval.

```
zProperties:
  zWidgeterEnable:
    category: ACME Widgeter
    type: boolean
    default: true

  zWidgeterInterval:
    category: ACME Widgeter
    default: 300
```

There is a special zProperty entry named *DEFAULTS* that can be used to further shorten definitions in cases where you're adding many zProperties. The following example shows how *DEFAULTS* can be used to replace the duplicated *category* property.

```
zProperties:
  DEFAULTS:
    category: ACME Widgeter

  zWidgeterEnable:
    type: boolean
    default: true

  zWidgeterInterval:
    default: 300
```

Each zProperty listed in *zProperties* will be created when the ZenPack is installed, and removed when the ZenPack is removed.

Note: When changing the default or category for a zProperty in the yaml, it changes in the zenoss system. Removing a zProperty from yaml will not remove it from the zenoss system. To remove it completely, you must write a migrate script to remove it.

---

zProperty Fields

The following fields are valid for a zProperty entry.

name

Description Name (e.g. zWidgeterEnable). Must be begin with a lowercase "z".

Required Yes

Type string

Default Value *implied from key in zProperties map*

type

Description Type of property. Valid types:

- *boolean*
- *float*
- *int*
- *lines*
- *long*
- *password*
- *string*

Required No

Type string

Default Value string

default

Description Default value for property. Default value depends on the type:

- boolean: *false*
- lines: *[]*
- password: *""* (empty string)
- string: *""* (empty string)
- all others: *null* (None)

Required No

Type *varies*

Default Value *varies*

category

Description Category name. (e.g. ACME Widgeter). Used to group related zProperties in the UI.

Required No

Type string

Default Value "" (empty string)

Zenoss specific zProperties

When changing modeler bindings using the zDeviceTemplates property, this will take effect on your ZenPack. Any previously defined bindings will be replaced. The same applies to the device level template bindings using the zCollectorPlugins property.

Note: Beginning with ZenPackLib 2.0, this behavior has changed. zProperties will no longer be overwritten if a target device class already exists (i.e. during an upgrade or if the YAML affects a preexisting class such as /Devices/Server. Instead, a warning will be displayed to the user during installation, and the target zProperty will be left alone.

4.8.3 Device Classes

Device classes are the functional categorization mechanism for Zenoss devices. Everything about how a device is modeled and monitored is controlled by its device class unless the device class configuration is overridden specifically for the device.

Example device classes that are a default part of every Zenoss system:

- /Discovered
- /Network
- /Server

Device classes are one of the most common items a ZenPack would add to Zenoss.

Adding Device Classes

To add a device class to your ZenPack you must include a *device_classes* section to your YAML file. The following example shows an example of adding a device class.

```
device_classes:
  /Server/ACME/Widgeter:
    remove: true
```

The *remove* field controls whether the device class will be removed from Zenoss if the ZenPack is removed. It defaults to false. In this example we set it to true because we do want our custom device class removed if the ZenPack that supports it is removed.

Each device class listed in *device_classes* will be created when the ZenPack is installed. The device classes will be created recursively if necessary. Meaning that if the /Server or /Server/ACME device classes don't already exist, they will be automatically created.

Setting zProperties

You can also set zProperty values for each device class. These values will be set each time the ZenPack is installed.

```
device_classes:
  /Server/ACME/Widgeter:
    remove: true
    zProperties:
      zWidgeterEnable: true
      zWidgeterInterval: 60
```

Note: As of version 2.0, zProperties will not be set on existing device classes during ZenPack installation. Developers wishing to do so should handle these cases via migrate scripts that run during the installation process.

The referenced zProperties must already exist in the Zenoss system, or be added by your ZenPack via a global *zProperties* entry.

Adding Monitoring Templates

Within each device class entry you can add monitoring templates. See the following example for adding a simple monitoring template with a single COMMAND datasource.

```
device_classes:
  /Server/ACME/Widgeter:
    zProperties:
      zDeviceTemplates:
        - Device

    templates:
      Device:
        description: ACME Widgeter monitoring.

        datasources:
          phony:
            type: COMMAND
            parser: Nagios
            commandTemplate: "echo OK|percent=100"

            datapoints:
              percent: {}

        graphs:
          Phoniness:
            units: percent
            miny: 0
            maxy: 100

            graphpoints:
              Phoniness:
                dpName: phony_percent
                format: "%7.2lf%%"
                lineType: AREA
```

This *Device* monitoring template will be added to the /Server/ACME/Widgeter device class each time the ZenPack is installed. This doesn't explicitly bind the monitoring template to the device class. To do that you need to set

zDeviceTemplates as shown in the example.

See *Monitoring Templates* for more information on creating monitoring templates.

Device Class Fields

The following fields are valid for a device class entry.

path

Description Path (e.g. /Server/ACME/Widgeter). Must begin with "/".

Required Yes

Type string

Default Value *(implied from key in device_classes map)*

create

Description Should the device class be created when the ZenPack is installed?

Required No

Type boolean

Default Value true

remove

Description Should the device class be removed when the ZenPack is removed?

Required No

Type boolean

Default Value false

zProperties

Description zProperty values to set on the device class.

Required No

Type map<name, value>

Default Value {} *(empty map)*

templates

Description Monitoring templates to add to the device class.

Required No

Type map<name, *Monitoring Template*>

Default Value {} *(empty map)*

description

Description Description used for devtype entry in device multi-add dialog

Required No

Type string

Default Value None

protocol

Description Protocol used for devtype entry in device multi-add dialog

Required No

Type string

Default Value None

4.8.4 Monitoring Templates

Monitoring templates are containers for monitoring configuration. Specifically datasources, thresholds and graphs. A monitoring template must be created to perform periodic collection of data, associate thresholds with that data, or define how that data should be graphed.

Location and Binding

Two important concepts in understanding how monitoring templates are used are location and binding. Location is the device class in which a monitoring template is contained. Binding is the device class, device or component to which a monitoring template is bound.

A monitoring template's location is important because it restricts to which devices a the template may be bound. Assume you have a device named *widgeter1* in the /Server/ACME/Widgeter device class that as a monitoring template named *WidgeterHealth* bound. Zenoss will attempt to find a monitoring template named *WidgeterHealth* in the following places in the following order.

1. On the *widgeter1* device.
2. In the /Server/ACME/Widgeter device class.
3. In the /Server/ACME device class.
4. In the /Server device class.
5. In the / device class.

The first template that matches by name will be used for the device. No template will be bound if no matching template is found in any of these locations.

It is because of this search up the hierarchy that allows the monitoring template's location to be used to restrict to which devices it can be bound. For example, by locating our monitoring template in the /Server/ACME device class we make it available to be bound for all devices in /Server/ACME and /Server/ACME/Widgeter, but we also make unavailable to be bound in other device classes such as /Server or /Network/Cisco.

After deciding on the right location for a monitoring template should then decide where it should be bound. Remember that to cause the template to be used it must be bound. This is done by adding the template's name to the *zDeviceTemplates* zProperty of a device class. See the following example that shows how to bind the *WidgeterHealth* monitoring template to the /Server/ACME/Widgeter device class.

```
name: ZenPacks.acme.Widgeter

device_classes:
  /Server/ACME/Widgeter:
    zProperties:
      zDeviceTemplates:
        - WidgeterHealth

    templates:
      WidgeterHealth: {}
```

Note that zDeviceTemplates didn't have to be declared in the ZenPack's zProperties field because it's a standard Zenoss zProperty.

Note: Binding templates using zDeviceTemplates is only applicable for monitoring templates that should be bound to devices. See *Classes and Relationships* for information on how monitoring templates are bound to components.

Alternatives to YAML

It's possible to create monitoring templates and add them to a ZenPack entirely through the Zenoss web interface. If you don't have complex or many monitoring templates to create and prefer to click through the web interface, you may choose to create your monitoring templates this way instead of through the *zenpack.yaml* file.

There are some advantages to defining monitoring templates in YAML.

- Using text-editor features such as search can be an easier way to make changes than clicking through the web interface.
- Having monitoring templates defined in the same document as the zProperties they use, and the device classes they're bound to can be easier to understand.
- Changes made to monitoring templates in YAML are much more diff-friendly than the same changes made through the web interface then exported to objects.xml. For those keeping ZenPack source in version control this can make changes clearer. For the same reason it can also be of benefit when multiple authors are working on the same ZenPack.

See *Command Line Reference* for information on the *dump_templates* option if you're interested in exporting monitoring templates already created in the web interface to YAML.

Adding Monitoring Templates

To add a monitoring template to *zenpack.yaml* you must first add the device class where it is to be located. Then within this device class entry you must add a templates field. The following example shows a *WidgeterHealth* monitoring template being added to the /Server/ACME/Widgeter device class. It also shows that template being bound to the device class by setting zDeviceTemplates.

```
name: ZenPacks.acme.Widgeter

device_classes:
  /Server/ACME/Widgeter:
    zProperties:
      zDeviceTemplates:
        - WidgeterHealth

    templates:
      WidgeterHealth:
        description: ACME Widgeter monitoring.

        datasources:
          health:
            type: COMMAND
            parser: Nagios
            commandTemplate: "echo OK|percent=100"

            datapoints:
              percent:
```

```
                rrdtype: GAUGE
                rrdmin: 0
                rrdmax: 100

        thresholds:
          unhealthy:
            dsnames: [health_percent]
            eventClass: /Status
            severity: Warning
            minval: 90

        graphs:
          Health:
            units: percent
            miny: 0
            maxy: 0

            graphpoints:
              Health:
                dpName: health_percent
                format: "%7.2lf%%"
```

Many different entry types are shown in the above example. See the references below for more information on each.

Monitoring Template Fields

The following fields are valid for a monitoring template entry.

name

Description Name (e.g. WidgeterHealth). Must be a valid Zenoss object ID.

Required Yes

Type string

Default Value *(implied from key in templates map)*

description

Description Description of the templates purpose and function.

Required No

Type string

Default Value "" *(empty string)*

targetPythonClass

Description Python module name (e.g. ZenPacks.acme.Widgeter.Widgeter) to which this template is intended to be bound.

Required No

Type string

Default Value "" (empty string is equivalent to Products.ZenModel.Device)

datasources

Description Datasources to add to the template.

Required No

Type map<name, *Datasource*>

Default Value {} *(empty map)*

thresholds

Description Thresholds to add to the template.

Required No

Type map<name, *Threshold*>

Default Value {} *(empty map)*

graphs

Description Graphs to add to the template.

Required No

Type map<name, *Graph*>

Default Value {} *(empty map)*

Note: ZenPackLib also allows for defining a replacement or additional template by adding "-replacement" or "-additional" to the end of the template name. For example, a defined *Device-replacement* template will replace the existing Device template on a device class. A defined *Device-addition* template will be applied in addition to the existing Device template on a device class.

Datasource Fields

The following fields are valid for a datasource entry.

name

Description Name (e.g. health). Must be a valid Zenoss object ID.

Required Yes

Type string

Default Value *(implied from key in datasources map)*

type

Description Type of datasource. See *Datasource Types*.

Required Yes

Type string *(must be a valid source type)*

Default Value None. Must be specified.

enabled

Description Should the datasource be enabled by default?

Required No

Type boolean

Default Value true

component

Description Value for the *component* field on events generated by the datasource. Accepts TALES expressions.

Required No

Type string

Default Value "" *(empty string)* – can vary depending on type.

eventClass

Description Value for the *eventClass* field on events generated by the datasource.

Required No

Type string

Default Value "" *(empty string)* – can vary depending on type.

eventKey

Description Value for the *eventKey* field on events generated by the datasource.

Required No

Type string

Default Value "" *(empty string)* – can vary depending on type.

severity

Description Value for the *severity* field on events generated by the datasource.

Required No

Type integer

Default Value 3 *(0=Clear, 1=Debug, 2=Info, 3=Warning, 4=Error, 5=Critical)* – can vary depending on type.

cycletime

Description How often the datasource will be executed in seconds.

Required No

Type integer – can vary depending on type.

Default Value 300 – can vary depending on type.

datapoints

Description Datapoints to add to the datasource.

Required No

Type map<name, *Datapoint*>

Default Value {} *(empty map)*

Datasources also allow other ad-hoc options to be added not referenced in the above list. This is because datasources are an extensible type in Zenoss, and depending on the value of *type*, other fields may be valid.

Datasource Types

The following datasource types are valid on any Zenoss system. They are the default types that are part of the platform. This list is not exhaustive as datasources types are commonly added by ZenPacks.

SNMP

Description Performs an SNMP GET operation using the *oid* field.

Availability Zenoss Platform

Additional Fields

oid

Description The SNMP OID to get.

Required Yes

Type string

Default Value "" *(empty string)*

COMMAND

Description Runs command in *commandTemplate* field.

Availability Zenoss Platform

Additional Fields

commandTemplate

Description The command to run.

Required Yes

Type string

Default Value "" *(empty string)*

usessh:

Description Run command on bound device using SSH, or run it on the Zenoss collector server?

Required No

Type boolean

Default Value false

parser:

Description Parser used to parse output from command.

Required No

Type string *(must be a valid parser name)*

Default Value Nagios

PING

Description Pings (ICMP echo-request) an IP address.

Availability Zenoss Platform

Additional Fields

cycleTime

Description How many seconds between ping attempts. (note capitalization)

Required No

Type integer

Default Value 60

attempts:

Description How many ping attempts to perform each cycle.

Required No

Type integer

Default Value 2

sampleSize

Description How many echo requests to send with each attempt.

Required No

Type integer

Default Value 1

Built-In

Description No collection. Assumes associated data will be populated by an external mechanism.

Availability Zenoss Platform

Additional Fields None

Custom Datasource and Datapoint Types

Some datasource (and datapoint) types are provided by a particular ZenPack and only available if that ZenPack is installed. These types often have unique paramters that control their function. ZenPackLib allows the specification of these parameters, but the degree of documentation for each varies. As a result, designing YAML templates using these requires a bit of investigation. The available properties depend on the datasource or datapoint type being used. Currently, examination of the related source code is a good way to investigate them, but an alternative is given below.

The following exmaple demonstrates how to create a YAML template that relies on the ZenPacks.zenoss.CalculatedPerformance ZenPack. Please note that the datasource properties used are not documented below, since they are provided by the CalculatedPerformance ZenPack.

First, we want to determine a list of available parameters, and we can use ZenDMD to display them as follows:

```
# This is the reference class and its properties are documented here.
from Products.ZenModel.RRDDataSource import RRDDataSource as Reference
# replace the import path and class with the class you are interested in
from ZenPacks.zenoss.CalculatedPerformance.datasources.AggregatingDataSource \
   import AggregatingDataSource as Comparison
# this prints out the list of non-standard properties and their types
props = [p for p in Comparison._properties if p not in Reference._properties]
print '\n'.join(['{} ({})'.format(p['id'], p['type']) for p in props])
```

In this case, we should see the following output:

```
targetMethod (string)
targetDataSource (string)
targetDataPoint (string)
targetRRA (string)
targetAsRate (boolean)
debug (boolean)
```

An example tempalte using the CalculatedPerformance datasources might resemble the following:

```
name: ZenPacks.zenoss.ZenPackLib
device_classes:
  /Device:
    templates:
      ExampleCalculatedPerformanceTemplate:
        datasources:
          # standard SNMP datasources
          memAvailReal:
            type: SNMP
            oid: 1.3.6.1.4.1.2021.4.6.0
            datapoints:
              memAvailReal: GAUGE
          memAvailSwap:
            type: SNMP
            oid: 1.3.6.1.4.1.2021.4.4.0
            datapoints:
              memAvailSwap: GAUGE
          # CalculatedPerformance datasources
          totalAvailableMemory
            type: Calculated Performance
            # "expression" paramter is unique to the
            # CalculatedPerformance datasource
            expression: memAvailReal + memAvailSwap
            datapoints:
              totalAvailableMemory: GAUGE
          # Aggregated Datasource
          agg_out_octets:
            # These are standard parameters
            type: Datapoint Aggregator
            # The following parameters are "extra" parameters,
            # attributes of the "Datapoint Aggregator" datasource
            targetDataSource: ethernetcmascd_64
            targetDataPoint: ifHCOutOctets
            targetMethod: os.interfaces
            # AggregatingDataPoint is subclassed from RRDDataPoint and
            # has the unique "operation" paramter
            datapoints:
              aggifHCOutOctets:
                operation: sum
```

Further experimentation, though, is required to determine workable values for these properties, and creating templates manually using the Zenoss GUI is a good way to do so.

Datapoint Fields

The following fields are valid for a datapoint entry.

name

Description Name (e.g. percent). Must be a valid Zenoss object ID.

Required Yes

Type string

Default Value *(implied from key in datapoints map)*

description

Description Description of the datapoint's purpose and function.

Required No

Type string

Default Value "" *(empty string)*

rrdtype

Description Type of datapoint. Must be GAUGE or DERIVE.

Required No

Type string *(must be either GAUGE or DERIVE)*

Default Value GAUGE

rrdmin

Description Minimum allowable value that can be written to the datapoint. Any lower values will be ignored.

Required No

Type int

Default Value None *(no lower-bound on acceptable values)*

rrdmax

Description Maximum allowable value that can be written to the datapoint. Any higher values will be ignored.

Required No

Type int

Default Value None *(no upper-bound on acceptable values)*

aliases

Description Analytics aliases for the datapoint with optional RPN calculation. Learn more about Reverse Polish Notiation

Required No

Type map<name, formula>

Default Value {} *(empty map)*

Example 1 aliases: { datapointName: '1024,*' }

Example 2 aliases: datapointName

Datapoints also allow other ad-hoc options to be added not referenced in the above list. This is because datapoints are an extensible type in Zenoss, and depending on the value of the datasource's *type*, other fields may be valid.

YAML datapoint specification also supports the use of an alternate "shorthand" notation for brevity. Shorthand notation follows a pattern of *RRDTYPE_MIN_X_MAX_X* where RRDTYPE is one of "GAUGE, DERIVE, COUNTER, RAW", and the "MIN_X"/"MAX_X" parameters are optional.

For example, DERIVE, DERIVE_MIN_0, and DERIVE_MIN_0_MAX_100 are all valid shorthand notation.

Threshold Fields

The following fields are valid for a threshold entry.

name

Description Name (e.g. unhealthy). Must be a valid Zenoss object ID.

Required Yes

Type string

Default Value *(implied from key in thresholds map)*

type

Description Type of threshold. See *Threshold Types*.

Required No

Type string *(must be a valid threshold type)*

Default Value MinMaxThreshold

enabled

Description Should the threshold be enabled by default?

Required No

Type boolean

Default Value true

dsnames

Description List of *datasource_datapoint* combinations to threshold.

Required No

Type list

Default Value [] *(empty list)*

Example dsnames: ['status_status']

eventClass

Description Value for the *eventClass* field on events generated by the threshold.

Required No

Type string

Default Value /Perf/Snmp – can vary depending on type.

severity

Description Value for the *severity* field on events generated by the threshold.

Required No

Type int

Default Value 3 *(0=Clear, 1=Debug, 2=Info, 3=Warning, 4=Error, 5=Critical)* – can vary depending on type.

escalateCount:

Description Event count after which severity increases

Required No

Type int

Default Value None

Thresholds also allow other ad-hoc options to be added not referenced in the above list. This is because thresholds are an extensible type in Zenoss, and depending on the value of the threshold's *type*, other fields may be valid.

Threshold Types

The following threshold types are valid on any Zenoss system. They are the default types that are part of the platform. This list is not exhaustive as additional threshold types can be added by ZenPacks.

MinMaxThreshold:

Description Creates an event if values are below or above specified limits.

Availability Zenoss Platform

Additional Fields

minval

Description The minimum allowable value. Values below this will raise an event.

Required No

Type string – Must evaluate to a number. Accepts Python expressions.

Default Value None *(no lower-bound on allowable values)*

maxval

Description The maximum allowable value. Values above this will raise an event.

Required No

Type string – Must evaluate to a number. Accepts Python expressions.

Default Value None *(no upper-bound on allowable values)*

ValueChangeThreshold

Description Creates an event if the value is different than last time it was checked.

Availability Zenoss Platform

Additional Fields None

Graph Fields

The following fields are valid for a graph entry.

name

Description Name (e.g. Health). Must be a valid Zenoss object ID.

Required Yes

Type string

Default Value *(implied from key in graphs map)*

units

Description Units displayed on graph. Used as the y-axis label.

Required No

Type string

Default Value None

miny

Description Value for bottom of y-axis.

Required No

Type integer

Default Value -1 *(-1 causes the minimum y-axis to conform to the plotted data)*

maxy

Description Value for top of y-axis.

Required No

Type integer

Default Value -1 *(-1 causes the maximum y-axis to conform to the plotted data)*

log

Description Should the y-axis be a logarithmic scale?

Required No

Type boolean

Default Value false

base

Description Is the plotted data in base 1024 like storage or memory size?

Required No

Type boolean

Default Value false

hasSummary

Description Should the graph legend be shown?

Required No

Type boolean

Default Value true

height

Description The graph's height in pixels.

Required No

Type integer

Default Value 100

width

Description The graph's width in pixels.

Required No

Type integer

Default Value 500

graphpoints

Description Graphpoints to add to the graph.

Required No

Type map<name, *Graphpoint*>

Default Value {} *(empty map)*

comments

Description List of comments to display in the graph's legend.

Required No

Type list<string>

Default Value [] *(empty list)*

Graphpoint Fields

The following fields are valid for a graphpoint entry.

name

Description Name (e.g. Health). Must be a valid Zenoss object ID.

Required Yes

Type string

Default Value *(implied from key in templates map)*

legend

Description Label to be shown for this graphpoint in the legend. The name field will be used if legend is not set.

Required No

Type string

Default Value None

dpName

Description *datasource_datapoint* combination to plot.

Required Yes

Type string

Default Value None

Example dpName: 'status_status'

lineType

Description How to plot the data: "LINE", "AREA" or "DONTDRAW".

Required No

Type string

Default Value LINE

lineWidth

Description How thick the line should be for the line type.

Required No

Type integer

Default Value 1

stacked

Description Should this graphpoint be stacked (added) to the last? Ideally both area "AREA" types.

Required No

Type boolean

Default Value false

color

Description Color for the line. Specified as RRGGBB (e.g. 1f77b4).

Required No

Type string

Default Value Cycles through a preset list depending on graphpoint's sequence.

colorindex

Description Color index for the line. Can be used instead of color to specify the color sequence number rather than the specific color.

Required No

Type integer

Default Value None

format

Description String format for this graphpoint in the legend (e.g. %7.2lf%s). The format option follows the RRDTool PRINT Format

Required No

Type string

Default Value "%5.2lf%s"

cFunc

Description Consolidation function. One of AVERAGE, MIN, MAX, LAST.

Required No

Type string

Default Value AVERAGE

limit

Description Maximum permitted value. Value larger than this will be nulled. Not used if negative.

Required No

Type integer

Default Value -1

rpn

Description RPN (Reverse Polish Notation) calculation to apply to datapoint. Learn more about Reverse Polish Notiation

Required No

Type string

Default Value None

includeThresholds

Description Should thresholds associated with *dpName* be automatically added to the graph?

Required No

Type boolean

Default Value false

thresholdLegends

Description Mapping of threshold id to legend (string) and color (RRGGBB)

Required No

Type map

Default Value None

Example thresholdLegends: {threshold_id: {legend: Legend, color: OO1122}}

4.8.5 Classes and Relationships

Classes and relationships form the model that forms the basis for everything Zenoss does. Classes are things like *Device*, *FileSystem*, *IpInterface* and *OSProcess*. Relationships state things like a *Device* contains many *FileSystems*. You will need to extend this model when the standard classes and relationships don't adequately represent the model of a target your ZenPack is attempting to monitor. For example, a XenServer ZenPack needs to represent concepts like pools, storage repositories and virtual machines.

Standard Classes

The standard classes exist on all Zenoss systems. If these are the only types of things you care to model and monitor then you may not need to create your own classes or relationships.

- Device
  - DeviceHW (hw)
    - CPU (hw/cpus)

        * ExpansionCard (hw/cards)
        * Fan (hw/fans)
        * HardDisk (hw/harddisks)
        * PowerSupply (hw/powersupplies)
        * TemperatureSensor (hw/temperaturesensors)
    - OperatingSystem (os)
        * FileSystem (os/filesystems)
        * IpInterface (os/interfaces)
        * IpRouteEntry (os/routes)
        * OSProcess (os/processes)
        * IpService (os/ipservices)
        * WinService (os/winservices)

Classes

If you need more than the standard classes provide, you will need to extend one of the following base classes provided by zenpacklib.

- zenpacklib.Device
- zenpacklib.Component
    - zenpacklib.HWComponent
        * zenpacklib.CPU
        * zenpacklib.ExpansionCard
        * zenpacklib.Fan
        * zenpacklib.HardDisk
        * zenpacklib.PowerSupply
        * zenpacklib.TemperatureSensor
    - zenpacklib.OSComponent
        * zenpacklib.FileSystem
        * zenpacklib.IpInterface
        * zenpacklib.IpRouteEntry
        * zenpacklib.OSProcess
        * zenpacklib.Service
            · zenpacklib.IpService
            · zenpacklib.WinService

You use *zenpacklib.Device* to create new device types of which instances will appear on the *Infrastructure* screen. You use *zenpacklib.Component* to create new component types of which instances will appear under *Components* on a device's left navigation pane. Frequently when ZenPacks need to add new classes, they will add a single new device type with many new components types. For example, a XenServer ZenPack would add a new device type

called *Endpoint* which represents the XenAPI management interface. That *Endpoint* device type would have many components of types such as *Pool*, *StorageRepository* and *VirtualMachine*.

The other supported classes are proxies for their platform equivalents, and are to be used when you want to extend one of the platform component types rather than creating a totally new component type.

Relationships

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

zenpacklib supports the following types of relationships.

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

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

A simplified version of XenServer's classes and relationships provides for a good example. The following list of relationship states the following: An endpoint contains zero or more pools, each pool contains zero or more storage repositories and virtual machines, and each storage repository is related to zero or more virtual machines.

- Endpoint 1:MC Pool
- Pool 1:MC StorageRepository
- Pool 1:MC VirtualMachine
- StorageRepository M:M VirtualMachine

Adding Classes and Relationships

To add classes and relationships to *zenpack.yaml* you add entries to the top-level *classes* and *class_relationships* fields. The following example shows a XenServer *Endpoint* device type along with *Pool*, *StorageRepository*, and *VirtualMachine* component types.

```
name: ZenPacks.example.XenServer

classes:
  DEFAULTS:
    base: [zenpacklib.Component]

  XenServerEndpoint:
    base: [zenpacklib.Device]
    label: Endpoint

  XenServerPool:
```

```
    label: Pool

    properties:
      ha_enabled:
        type: boolean
        label: HA Enabled
        short_label: HA

      ha_allow_overcommit:
        type: boolean
        label: HA Allow Overcommit
        short_label: Overcommit

  XenServerStorageRepository:
    label: Storage Repository

    properties:
      physical_size:
        type: int
        label: Physical Size
        short_label: Size

  XenServerVirtualMachine:
    label: Virtual Machine

    properties:
      vcpus_at_startup:
        type: int
        label: vCPUs at Startup
        short_label: vCPUs

class_relationships:
  - XenServerEndpoint 1:MC XenServerPool
  - XenServerPool 1:MC XenServerStorageRepository
  - XenServerPool 1:MC XenServerVirtualMachine
  - XenServerStorageRepository M:M XenServerVirtualMachine
```

Note: DEFAULTS can be used in classes just like in zProperties to avoid repetitively setting the same field for many entries. Note specifically how XenServerPool, XenServerStorageRepository and XenServerVirtualMachine will inherit the default while XenServerEndpoint overrides it.

Classes and their properties allow for a wide range of control. See the following section for details.

Extending ZenPackLib Classes

Occasionally, you may wish to add your own custom methods to your YAML-defined classes or otherwise extend their functionality beyond ZenPackLib's current capabilities. Doing so requires creating a Python file that imports and overrides the class you wish to modify, and this is relatively straightforward.

Suppose we have a component class called "BasicComponent", and we want to provide a method called "hello world" that, when called, will return the string "Hello World" and display it in the component grid.

Our YAML file looks like this:

```
name: ZenPacks.zenoss.BasicZenPack
class_relationships:
- BasicDevice 1:MC BasicComponent
classes:
  BasicDevice:
    base: [zenpacklib.Device]
  BasicComponent:
    base: [zenpacklib.Component]
    properties:
      hello_world:
        # this will appear as the column header
        # in the component grid
        label: Hello World
        # this should be displayed in the component grid
        grid_display: true
        # tells ZenPackLib that this isn't a typical
        # property like a string, integer, boolean, etc...
        api_only: true
        # this is the type of property
        api_backendtype: method
```

First, the ZenPack's init file:

```
$ZPDIR/ZenPacks.zenoss.BasicZenPack/ZenPacks/zenoss/BasicZenPack/__init__.py
```

should contain the following lines:

```
from ZenPacks.zenoss.ZenPackLib import zenpacklib
CFG = zenpacklib.load_yaml()
schema = CFG.zenpack_module.schema
```

Note: For better performance, specify the explicit path(s) to your yaml file. e.g. *CFG = zenpacklib.load_yaml([os.path.join(os.path.dirname(__file__), "zenpack.yaml")])*

Next, we create the file:

```
$ZPDIR/ZenPacks.zenoss.BasicZenPack/ZenPacks/zenoss/BasicZenPack/BasicComponent.py
```

and it should contain the lines:

```
from . import schema

class BasicComponent(schema.BasicComponent):
    """Class override for BasisComponent"""
```

From here, we proceed to add our "hello_world" method to obtain:

```
from . import schema

class BasicComponent(schema.BasicComponent):
    """Class override for BasisComponent"""
    def hello_world(self):
        return 'Hello World!'
```

And we're done.

The "Hello World" column will now display in the component grid, and the string "Hello World!" will be printed in each row of component output.

We can also override ZenPackLib's built-in methods, but must be careful doing so to avoid undesirable results. Supposing that our YAML specifies some monitoring templates (not defined here) for BasicComponent, and for some reason we want to randomly choose which ones are displayed in the GUI. To do so, we need to override the "getRRDTemplates" method.

Our YAML file is modified:

```
name: ZenPacks.zenoss.BasicZenPack
class_relationships:
- BasicDevice 1:MC BasicComponent
classes:
  BasicDevice:
    base: [zenpacklib.Device]
  BasicComponent:
    base: [zenpacklib.Component]
    properties:
      hello_world:
        label: Hello World
        api_only: true
        api_backendtype: method
        grid_display: true
    monitoring_templates: [ThisTemplate, ThatTemplate]
```

And we further modify our BaseComponent.py as follows:

```
import random
from . import schema

class BasicComponent(schema.BasicComponent):
    """Class override for BasisComponent"""
    def hello_world(self):
        return 'Hello World!'

    def getRRDTemplates(self):
        """ Safely override the ZenPackLib
            getRRDTemplates method, returning
            randomly chosen templates. """
        templates = []
        # make sure we call the base method when we override it
        for template in super(BasicComponent, self).getRRDTemplates():
            # rolling the dice
            if bool(random.randint(0,1)):
                templates.append(template)
        return templates
```

The key point to remember here is the call to:

```
super(BasicComponent, self).getRRDTemplates()
```

which instructs Python to use the original method before we modify its output. Similar care must be excercised when overriding built-in methods and properties, assuming a safer method cannot be found.

Support for multiple YAML files (Version 2.0)

For particularly complex ZenPacks the YAML file can grow to be quite large, potentially making management cumbersome. To address this concern, ZenPackLib now supports splitting the zenpack.yaml files into multiple files. The following conditions should be observed when using multiple files:

- The YAML files should have a .yaml extension.
- The "load_yaml" method will detect and load yaml files automatically. This behavior can be overridden by calling load_yaml(yaml_doc=[doc1, doc2]). In this case the full file paths will need to be specified:

```
import os
files = ['file1.yaml', 'file2.yaml']
YAML_DOCS = [os.path.join(os.path.dirname(__file__), f) for f in files]
from ZenPacks.zenoss.ZenPackLib import zenpacklib
CFG = zenpacklib.load_yaml(yaml_doc=YAML_DOCS)
schema = CFG.zenpack_module.schema
```

- The 'name' parameter (ZenPack name), if used in multiple files, should be identical between them
- If a given YAML section (device_classes, classes, device_classes, etc) is split between files, then each file should give the complete path to the defined objects. The following is valid:

```
# File 1
name: ZenPacks.zenoss.BasicZenPack
class_relationships:
- BaseComponent 1:MC AuxComponent
classes:
  BasicDevice:
    base: [zenpacklib.Device]
    monitoring_templates: [BasicDevice]
  BasicComponent:
    base: [zenpacklib.Component]
    monitoring_templates: [BasicComponent]
```

```
# File 2
class_relationships:
- BaseDevice 1:MC BaseComponent
classes:
  SubComponent:
    base: [BasicComponent]
    monitoring_templates: [SubComponent]
  AuxComponent:
    base: [SubComponent]
    monitoring_templates: [AuxComponent]
```

- Using conflicting parameters (like setting different DEFAULTS for the same entity in different files) will likely lead to undesirable results.

Class Fields

The following fields are valid for a class entry.

name

> **Description** Name (e.g. XenServerEndpoint). Must be a valid Python class name.
>
> **Required** Yes

Type string

Default Value *(implied from key in classes map)*

base

Description List of base classes to extend. See *Classes*

Required No

Type list<classname>

Default Value [zenpacklib.Component]

meta_type

Description Globally unique name for the class.

Required No

Type string

Default Value *(same as name)*

label

Description Human-friendly label for the class.

Required No

Type string

Default Value *(same as meta_type)*

plural_label

Description Plural form of label.

Required No

Type string

Default Value *(same as label with an "s" suffix)*

short_label

Description Short form of label. Used as a column header or where space is limited.

Required No

Type string

Default Value *(same as label)*

plural_short_label

Description Plural form of short_label.

Required No

Type string

Default Value *(same as short_label with an "s" suffix)*

icon

Description Filename (in resources/) for icon.

Required No

Type string

Default Value *(same as name with a ".png" suffix in resources/icon/)*

label_width

Description Width of label text in pixels.

Required No

Type integer

Default Value 80

plural_label_width

Description Width of plural_label text in pixels.

Required No

Type integer

Default Value *(same as label_width + 7)*

content_width

Description Expected width of object's title in pixels.

Required No

Type integer

Default Value *(same as label_width)*

auto_expand_column

Description Column (property) to auto-expand in component grid.

Required No

Type string

Default Value name

initial_sort_column

Description Column (property) to initially sort in component grid.

Required No

Type string

Default Value name

order

Description Order to display this class among other classes. (1-100)

Required No

Type integer

Default Value 100

Note: The *order* parameter takes any integer value between 1 and 100. However, it's behavior depends somewhat depending on whether it applies to a Class, a Property, or a Relationship. For a relationship, order behavior can further depend on the type of relationship.

There is an overall clustering for like items in the GUI component grid, following this order:

1. Containing Components

2. Properties
3. Contained Components

with container relationships listed before visible properties and finally any containing relationships.

Earlier (pre-2.0) versions of ZenPackLib accepted float arguments for order. However, ZenPackLib now "normalizes" these values behind the scenes to integers between 1 and 100.

---

filter_display

Description Will related components be filterable by components of this type?

Required No

Type boolean

Default Value true

filter_hide_from

Description Classes for which this class should not show in the filter dropdown.

Required No

Type list<classname>

Default Value [] *(empty list)*

monitoring_templates

Description List of monitoring template names to bind to components of this type.

Required No

Type list<string>

Default Value [*(label with spaces removed)*]

properties

Description Properties for this class.

Required No

Type map<name, *Class Property*>

Default Value {} *(empty map)*

relationships

Description Relationship overrides for this class.

Required No

Type map<name, *Relationship Override*>

Default Value {} *(empty map)*

impacts

Description Relationship or method names that when called return a list of objects that objects of this class could impact.

Required No

Type list<*relationship_or_method_name*>

Default Value [] *(empty list)*

impacted_by

Description Relationship or method names that when called return a list of objects that could impact objects of this class.

Required No

Type list<*relationship_or_method_name*>

Default Value [] *(empty list)*

impact_triggers

Description Impact trigger policy definitions for this class.

Required No

Type map<name, *Impact Trigger*>

Default Value {} *(empty map)*

dynamicview_views

Description Names of Dynamic Views objects of this class can appear in.

Required No

Type list<*dynamicview_view_name*>

Default Value [service_view]

dynamicview_group

Description Dynamic View group name for objects of this class. Can be overridden by implementing getDynamicViewGroup() method on class.

Required No

Type string

Default Value *(same as plural_short_label)*

dynamicview_weight

Description Dynamic View weight for objects of this class. Higher numbers are further to the right. Can be overridden by implementing getDynamicViewGroup() method on class.

Required No

Type float or int

Default 1000 + order * 10

dynamicview_relations

Description Map of Dynamic View relationships for this class and the relationship or method names that when called populate them.

Required No

Type map<relationship_name, list<*relationship_or_method_name*>>

Default Value {} *(empty map)*

extra_paths

Description By default, components are indexed based upon paths that include objects they have a direct relationship to. This option allows additional paths to be specified (this can be useful when indirect containment is used)

Required No

Type list<list<regexp>>

Default Value [] *(empty list)*

Example 1 ['resourcePool', 'owner'] # from cluster or standalone

Example 2 ['resourcePool', '(parentResourcePool)+'] # from all parent resource pools, recursively.

Note: Each item in extra_paths is expressed as a tuple of regular expression patterns that are matched in order against the actual relationship path structure as it is traversed and built up get_facets.

To facilitate matching, we construct a compiled set of regular expressions that can be matched against the entire path string, from root to leaf.

So:

('orgComponent', '(parentOrg)+')

is transformed into a "pattern stream", which is a list of regexps that can be applied incrementally as we traverse the possible paths:

(re.compile(^orgComponent), re.compile(^orgComponent/(parentOrg)+), re.compile(^orgComponent/(parentOrg)+/?$')

Once traversal embarks upon a stream, these patterns are matched in order as the traversal proceeds, with the first one to fail causing recursion to stop. When the final one is matched, then the objects on that relation are matched. Note that the final one may match multiple times if recursive relationships are in play.

Class Property Fields

The following fields are valid for a class property entry.

name

Description Name (e.g. ha_enabled). Must be a valid Python variable name.

Required Yes

Type string

Default Value *(implied from key in properties map)*

type

Description Type of property: *string*, *int*, *float*, *boolean*, *lines*, *password* or *entity*. All types are strictly enforced except for *entity*.

Required No

Type string

Default Value string

default

Description Default value for property.

Required No

Type *(varies depending on type)*

Default Value None

label

Description Human-friendly label for the property.

Required No

Type string

Default Value *(same as name)*

short_label

Description Short form of label. Used as a column header where space is limited.

Required No

Type string

Default Value *(same as label)*

label_width

Description Width of label text in pixels.

Required No

Type integer

Default Value 80

content_width

Description Expected width of property's value in pixels.

Required No

Type integer

Default Value *(same as label_width)*

display

Description Should this property be shown as a column and in details?

Required No

Type boolean

Default Value true

details_display

Description Should this property be shown in details?

Required No

Type boolean

Default Value true

grid_display

Description Should this property be shown as a column?

Required No

Type boolean

Default Value true

order

Description Order to display this property among other properties. (1-100)

Required No

Type integer

Default Value 100

editable

Description Should this property be editable in details?

Required No

Type boolean

Default Value false

renderer

Description JavaScript renderer for property value.

Required No

Type string

Default Value None *(renders value as-is)*

api_only

Description Should this property be for the API only? The property or method (according to api_backendtype) must be manually implemented if this is set to true.

Required No

Type boolean

Default Value false

api_backendtype

Description Implementation style for the property if *api_only* is true. Must be *property* or *method*.

Required No

Type string

Default Value property

enum

Description Enumeration map for property. Set to something like {1: 'OK', 2: 'ERROR'} for an int-type property to provide text representations for property values.

Required No

Type map<value, representation>

Default Value {} *(empty map)*

datapoint

Description *datasource_datapoint* value to use as the value for this property. Useful for displaying the most recent collected datapoint value in the grid or details as any modeled property would be.

Required No

Type string

Default Value None

datapoint_default

Description Default value for property if *datapoint* is set, but no data exists.

Required No

Type string, integer or float

Default Value None

datapoint_cached

Description Should the value for datapoint be cached for a limited time? Can improve UI performance.

Required No

Type boolean

Default Value true

index_type

Description Type of indexing for the property: *field* or *keyword*.

Required No

Type string

Default Value None *(no indexing)*

index_scope

Description Scope of index: *device* or *global*. Only applies if *index_type* is set.

Required No

Type string

Default Value device

Relationship Override Fields

The following fields are valid for a relationship override entry.

name

Description Name (e.g. xenServerPools). Must match a relationship name defined in *class_relationships*.

Required Yes

Type string

Default Value *(implied from key in relationships map)*

label

Description Human-friendly label for the relationship.

Required No

Type string

Default Value *(label of class to which the relationship refers)*

short_label

Description Short form of label. Used as a column header where space is limited.

Required No

Type string

Default Value *(same as label or referred class' short_label)*

label_width

Description Width of label text in pixels.

Required No

Type integer

Default Value *(same as referred class' label width)*

content_width

Description Expected width of relationship's value in pixels. To-Many relationships are shown simply as a count and will have a shorter width. To-One relationships show a link to the object and will require a width long enough to accommodate the object's title.

Required No

Type integer

Default Value *(varies depending on relationship type)*

display

Description Should this relationship be shown as a column and in details?

Required No

Type boolean

Default Value true

details_display

Description Should this relationship be shown in details?

Required No

Type boolean

Default Value true

grid_display

Description Should this relationship be shown as a column?

Required No

Type boolean

Default Value true

order

Description Order to display this relationship among other relationships and properties. (1-100)

Required No

Type integer

Default Value 100

renderer

Description JavaScript renderer for relationship value.

Required No

Type string

Default Value None

render_with_type

Description Should related object be rendered with it's type? Only applies to To-One relationships.

Required No

Type boolean

Default Value false

Impact Trigger Fields

The following fields are valid for an Impact trigger entry.

name

Description Name (e.g. avail_pct_5). Must be a valid Python variable name.

Required Yes

Type string

Default Value *(implied from key in properties map)*

policy

Description Type of policy, one of: AVAILABILITY, PERFORMANCE, CAPACITY

Required Yes

Type string

Default Value AVAILABILITY

trigger:

Description Type of trigger, one of: policyPercentageTrigger, policyThresholdTrigger, or negativeThresholdTrigger

Required Yes

Type string

Default Value policyPercentageTrigger

threshold:

Description Numerical boundary for the trigger

Required Yes

Type int

Default Value 50

state:

Description State of this object when trigger criteria met (see note)

Required Yes

Type str

Default Value UNKNOWN

dependent_state:

Description State of dependent objects meeting trigger criteria (see note)

Required Yes

Type str

Default Value UNKNOWN

Note: Valid values for both **state** and **dependent_state** depend on the choice of **policy** parameter:

- **AVAILABILITY**: DOWN, UP, DEGRADED, ATRISK, or UNKNOWN
- **PERFORMANCE**: UNACCEPTABLE, DEGRADED, ACCEPTABLE, or UNKNOWN
- **CAPACITY**: UNACCEPTABLE, REDUCED, ACCEPTABLE, or UNKNOWN

4.8.6 Device Link Providers

A device link provider is a subscriber interface that gives a hook for adding context-specific html links (for example, the device links on the Device Details page). Zenpacklib provides a simple class to search the global catalog, device class catalog, or a device local catalog that match a specific query.

```
name: ZenPacks.zenoss.BasicZenPack

link_providers:
  Virtual Machine:
    link_class: ZenPacks.example.XenServer
    catalog: device
    device_class: /Server/XenServer
    queries: [vm_id:manageIp]
  XenServer:
    global_search: True
    queries: [manageIp:vm_id]
```

To search the global catalog, set *global_search* to true. If the catalog you wish to search is in a specific device class, set *global_search* to false and specify the class name with *device_class*. To search a catalog on the local device, simply set *global_search* to false and do not set *device_class*. You can use one or more queries to match up devices/components. In the above example, the XenServer provider will search the global device catalog and match any device's *manageIp* attribute with the current device's *vm_id* attribute.

Device Link Provider Fields

The following fields are valid for a device link provider entry.

link_title:

Description Title which will appear on the overview page of the type of device or component.

Required Yes

Type string

Default Value *(implied from key in relationships map)*

global_search:

Description Search the global catalog?

Required No

Type boolean

Default Value false

link_class:

Description Python class on which this provider will apply. You must supply the full python class name. For example, 'ZenPacks.example.XenServer.Device.Device'.

Required No

Type string

Default Value 'Products.ZenModel.Device.Device'

device_class:

Description Device class containing the catalog which to search. You must supply the full device class name. For example, '/Server/XenServer'.

Required No

Type string

Default Value None

catalog:

Description Catalog name on which to search for linked devices/components

Required No

Type string

Default Value 'device'

queries:

Description Queries to use to match a linked device/component. Each query must be in *remote:local* format, meaning that the catalog search will match a remote attribute with a local attribute. The local search term could also be actual text. Examples: *id:manageIp* will match the remote id attribute with the local device's manageIp attribute, and *meta_type:ClusterDevice* will match the meta_type on a remote device to "ClusterDevice".

Required Yes

Type list<string>

Default Value None

4.8.7 Event Classes

Event Classes are used to group together specific types of events. This can be useful for situations where an event should be dropped or text should be altered to be more human readable. This is typically done through a python transform.

To define a class, supply the path to the class or classes. Then, for each event class, supply the appropriate properties for the class. These include the option to remove the event class during ZenPack installation/uninstallation, description, and a transform. You can also define mappings to apply to events based on a key and supply an explanation and/or resolution to an issue.

Note: When you define an event class and/or mapping which already exists, any settings defined in your ZenPack will overwrite existing settings.

The following example shows an example of a *zenpack.yaml* file with an example of a definition of an event class.

```
name: ZenPacks.acme.Events

event_classes:
  /Status/Acme:
    remove: false
    description: Acme event class
    mappings:
      Widget:
        eventClassKey: WidgetEvent
        sequence:  10
        remove: true
        transform: |-
          if evt.message.find('Error reading value for') >= 0:
              evt._action = 'drop'
```

Note: When assigning values to multi-line fields such as **transform** or **example**, the best way to preserve whitespace and readability is to use the

transform: |-

if evt.message.find('Error reading value for') >= 0: evt._action = 'drop'

style convention demonstrated. As this example demonstrates, the indented line following the **"|-"** style character becomes the first line of the transform, with subsequent whitespace preserved.

Event Class Fields

The following fields are valid for an event class entry.

path

Description Path to the Event Class (e.g. /Status/Acme). Must begin with "/".

Required Yes

Type string

Default Value None

description

Description Description of the event class

Required No

Type string

Default Value None

remove

Description Remove the event class during ZenPack removal? This will only apply to a ZenPack that has created the event class. Any existing event classes not created by the ZenPack will not be removed. Any event classes created by the platform will also never be removed.

Required No

Type boolean

Default Value False

transform

Description A python expression for transformation.

Required No

Type string (multiline)

Default Value None

mappings

Description Event class mappings

Required No

Type map<name, *Event Class Mapping*>

Default Value None

Event Class Mapping Fields

The following fields are valid for an event class mapping entry.

name

Description Name of the event class mapping (e.g. WidgetDown).

Required Yes

Type string

Default Value None

eventClassKey

Description Event class key

Required No

Type string

Default Value None

explanation

Description Textual description for matches of this event class mapping. Use in conjunction with the Resolution field.

Required No

Type string (multiline)

Default Value None

resolution

Description Use the Resolution field to enter resolution instructions for clearing the event.

Required No

Type string (multiline)

Default Value None

sequence

Description Define the match priority. Lower is a higher priority.

Required No

Type integer

Default Value None

rule

Description A python expression to match an event.

Required No

Type string

Default Value None

regex

Description A regular expression to match an event.

Required No

Type string

Default Value None

transform

Description A python expression for transformation.

Required No

Type string (multiline)

Default Value None

example

Description Debugging string to use in the regular expression ui testing.

Required No

Type string (multiline)

Default Value None

remove

Description Remove the Mapping when the ZenPack is removed.

Required No

Type boolean

Default Value None

4.8.8 Process Classes

Process Classes are used to define sets of similar running processes using a regular expression. You can then monitor various aspects of the running processes, such as cpu and memory usage, with a datasource.

To define a class, supply the Process Class Organizer under which the Process Class will reside. Optionally add a description of the organizer. Then, for each Process Class, supply the processes to include/exclude, description,

and replacement text. You can also optionally override specific zProperties of a process class, such as zMonitor or zFailSeverity. See the *Process Class Fields* section below.

The following example shows an example of a *zenpack.yaml* file with an example of a definition of a process class.

```
name: ZenPacks.acme.Processes

process_class_organizers:
  Widget:
    description: Organizer for Widget process classes
    process_classes:
      widget:
        description: Widget process class
        includeRegex: sbin\/widget
        excludeRegex: "\\b(vim|tail|grep|tar|cat|bash)\\b"
        replaceRegex: .*
        replacement: Widget
```

Note: When you define a process class organizer and/or class which already exists, any settings defined in your ZenPack will overwrite existing settings.

Process Class Organizer Fields

The following fields are valid for a process class organizer entry.

name

Description Name (e.g. Widget or "Widget/ACME").

Required Yes

Type string

Default Value None

description

Description Description of the Process Class Organizer

Required No

Type string

Default Value None

Process Class Fields

The following fields are valid for a process class entry.

name

Description Name of the process class (e.g. widget).

Required Yes

Type string

Default Value None

description

Description Description of the Process Class Organizer

Required No

Type string

Default Value None

includeRegex

Description Include processes matching this regular expression

Required No

Type string

Default Value Name of the process class

excludeRegex

Description Exclude processes matching this regular expression

Required No

Type string

Default Value None

replaceRegex

Description Replace command line text matching this regular expression

Required No

Type string

Default Value None

replacement

Description Text which will replace the command line text that matches replaceRegex

Required No

Type string

Default Value None

monitor

Description Enable monitoring? Overrides parent process class organizer setting.

Required No

Type boolean

Default Value None

alert_on_restart

Description Send event on restart? Overrides parent process class organizer setting.

Required No

Type boolean

Default Value None

fail_severity

Description

Failure event severity. Overrides parent process class organizer setting. Valid values:

- 0=Clear
- 1=Debug
- 2=Info
- 3=Warning
- 4=Error
- 5=Critical)

Required No

Type integer

Default Value None

modeler_lock

Description

Lock process components. Overrides parent process class organizer setting. Valid values:

- 0: Unlocked
- 1: Lock from Deletes
- 2: Lock from Updates

Required No

Type integer

Default Value None

send_event_when_blocked

Description Send and event when action is blocked? Overrides parent class organizer setting.

Required No

Type boolean

Default Value None

4.9 Compatibility

Starting with version 2.0, zenpacklib.py will ship as a separately installed ZenPack. This change offers several advantages over the earlier distribution method along with many new features and fixes. Existing ZenPacks based on earlier versions of zenpacklib.py should coexist peacefully with those based on the newer version, and eventual migration to version 2.0 should be relatively painless. Future versions of Zenoss-provided ZenPacks will use the newer ZenPackLib version as they are developed and released.

4.9.1 Migrating ZenPacks to 2.0

For the most part, migrating to ZenPackLib 2.0 should be straightforward and requires minimal changes to your ZenPack. These largely involve changing import statements where appropriate and removing the older zenpacklib.py files

Note: ZenPacks based on ZenPackLib 2.0 will need to have a dependency set to prevent potential issues when installing or removing them. If ZenPackLib 2.0 is not installed, a dependent ZenPack should refuse to install until the dependency is met. Similarly, ZenPackLib 2.0 should refuse removal if dependent ZenPacks are still installed. To achieve this, make sure that the INSTALL_REQUIRES variable in the setup.py file contains the following:

INSTALL_REQUIRES = ['ZenPacks.zenoss.ZenPackLib']

Please note that "INSTALL_REQUIRES" may already contain entries, and these should be preserved if they exist.

This can also be configured in the GUI if the dependent ZenPack is installed in develop mode.

The __init__.py file will need its import statements changed.

```
from . import zenpacklib
```

changes to:

```
from ZenPacks.zenoss.ZenPackLib import zenpacklib
```

while:

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

remains unchanged unless some of the new logging capabilities are desired such as:

```
CFG = zenpacklib.load_yaml(verbose=True, level=10)
```

In addition, the statement (if it exists):

```
from . import schema
```

should be changed to:

```
schema = CFG.zenpack_module.schema
```

or added if it does not exist.

Note: Care should also be taken to delete the zenpacklib.py and zenpacklib.pyc files in the ZenPack's source directory, since leaving them in place may cause unforseen behavior.

Note: Import statements should also be checked throughout any class overrides or other python files, since the statements will fail if they refer to the older zenpacklib.py.

Note: The tag *!ZenPackSpec* is not necessary and should be removed from your yaml definitions.

4.9.2 Version 2.0 Logging

Logging has been substantially enhanced for ZenPackLib version 2.0 and provides numerous features to aid during development or troubleshooting. Logging can now be controlled on a per-ZenPack basis by supplying additional paramters to the "load_yaml()" method call in the ZenPack's __init__.py.file:

```
CFG = zenpacklib.load_yaml(verbose=True, level=10)
```

In this example, logging verbosity is enabled with at the DEBUG level.

Every class in ZenPackLib has a "LOG" attribute that can be called within any class override files you may have. For example, given the file BasicComponent.py class extension, logging features would be accessed as follows:

```
from . import schema

class BasicComponent(schema.BasicComponent):
    """Class override for BasisComponent"""
    def hello_world(self):
        self.LOG.info("You called hello_world")
        return 'Hello World!'
```

Note: Log messages generated within the new logging framework are written to the Zope logger (event.log) and can be viewed there. Logging used within class extension files will follow the verbosity and level parameters provided to the "load_yaml" method.

Please note that additional Zope configuration may be required to see log messages, since Zope configuration determines what is accepted for writing to its event log. For example, if Zope logging is set to "warn", then any "info" or "debug" messages will not be logged regardless of the load_yaml parameters used. Zope logging in this case must be set to "info" for ZPL "info", "warning", and "critical" logging.

4.9.3 Older Versions of zenpacklib.py

Note: The following applies to pre-2.0 versions of zenpacklib.py only. Starting with version 2.0, zenpacklib.py will ship as a separately installed ZenPack designed for use by dependent ZenPacks

Distributing *zenpacklib.py* with each ZenPack allows different ZenPacks in the same Zenoss system to use different versions of zenpacklib. This can make things simpler for the ZenPack author as they know which version of zenpacklib will be used. It will be the one that's shipped with the ZenPack.

This approach does have the drawback of potentially forcing ZenPacks to be updated to include a new version of zenpacklib to support a new version of Zenoss. Care will be taken to make each zenpacklib version compatible with as many versions of Zenoss as possible. Furthermore, care will be taken to make future versions of Zenoss compatible with existing zenpacklib versions within reason.

The following table describes which versions of Zenoss are supported by different versions of zenpacklib.

zenpacklib Version	Zenoss Versions
1.1	4.2 *, 5.0, 5.1, 5.2
1.0	4.2 *, 5.0, 5.1, 5.2

Compatibility only considers <major>.<minor> versions of both zenpacklib and Zenoss. Maintenance or patch releases of each are always considered compatible.

4.9.4 Determining Version

Note: Beginning with version 2.0, you can check the zenpacklib version with either:

zenpacklib –version

from the command line, or by navigating to:

Advanced -> Settings -> ZenPacks

in the Zenoss GUI

You can check which version of zenpacklib you're using in two ways. The first is by using the *version* command line option.

```
python zenpacklib.py version
```

If you have ZenPack code that needs the version it can also be accessed from Python code that has imported *zenpacklib* module through the module's *__version__* property.

```
from . import zenpacklib
zenpacklib.__version__
```

4.9.5 PyYAML Requirement

Note: Beginning with version 2.0, the ZenPacks.zenoss.ZenPackLib ZenPack will refuse to install unless PyYAML is already installed

---

zenpacklib requires that PyYAML be installed in the Zenoss system. PyYAML was not a standard part of a Zenoss system until Zenoss 5. To use zenpacklib, or to use a ZenPack built with zenpacklib on a Zenoss 4.2 system you must first make sure that PyYAML is installed.

Note: PyYAML has been added to Zenoss 4.2.5 as of SP457, and Zenoss 4.2.4 as of SP776.

Checking for PyYAML

On your main Zenoss 4.2 server run the following command to check for PyYAML.

```
su - zenoss -c "python -c 'import yaml;print yaml.version'"
```

You will see the version of PyYAML if it installed.

```
3.11
```

You will see the following error if PyYAML is not installed.

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named yaml
```

Installing PyYAML

Run the following command to install PyYAML if it isn't already installed.

```
su - zenoss -c "easy_install PyYAML"
```

It's normal for the *easy_install* command to print many errors and warnings even when it successfully installs. Run the first command to verify it's installed when complete.

If your Zenoss system is distributed to multiple servers for hubs, collectors, or any other reason you will need to update those hubs and collectors after installing PyYAML to make sure it also gets installed on them.

4.10 Changes

4.10.1 Version 2.0

Backwards Incompatible Changes

- Any installed ZenPacks using older versions of zenpacklib.py will continue to function unchanged.
- Using version 2.0 is slightly different. The __init__.py file import statements should now contain the following:

```
from ZenPacks.zenoss.ZenPackLib import zenpacklib
CFG = zenpacklib.load_yaml()
schema = CFG.zenpack_module.schema
```

Note: For better performance, specify the explicit path(s) to your yaml file. e.g. *CFG = zenpacklib.load_yaml([os.path.join(os.path.dirname(__file__), "zenpack.yaml")])*

- zProperties will not be updated automatically on existing device classes. These should be handled on a case basis by using migrate scripts.

Release 2.0.9

Fixes

- Fix incorrect removal of organizers such as /Status event class (ZPS-2660)

Release 2.0.8

Fixes

- Improve component grid loading performance. (ZPS-2033)
- Fix potential POSKeyError when modeling devices. (ZPS-2371)
- Improved support for multi-line text in YAML (ZPS-444)
- Improved component path reporters for mixin platform proxy classes (ZPS-1262)
- Fix zenpacklib.TestCase when Impact >= 5.2.2 is installed (ZPS-2011)

Release 2.0.7

Fixes

- Fix all the zenpacklib dump options. (ZPS-1601)

- Implement template replacement and addition on device level. (ZPS-1704)

Release 2.0.6

Fixes

- Fix mishandling of 0/clear severity (ZPS-1454)
- Fix attempts to load non-YAML files (ZPS-1483)
- Use appropriate sequence for graph points (ZPS-1361)
- Fix GUI ZenPack export of objects.xml (ZPS-1589)
- Fix datapoint alias shorthand export handling (ZPS-1589)

Release 2.0.5

Fixes

- Fix version reported by "zenpacklib –version". (ZPS-1145)
- Template backups use YYYYMMDDHHMM format instead of unix timestamp.
- Fix failure to back up customized templates during upgrade from pre-2.0 ZenPacks. (ZPS-1195)
- Fix failure to back up customized templates during upgrade. (ZPS-1176)

Release 2.0.4

Fixes

- Fix for missing Dynamic View on some components (ZPS-703)
- Fix for failure to create device classes in uncommon case (ZPS-1012)
- Fix event class mappings with mismatched id and eventClassKey (ZPS-1016)

Release 2.0.3

Fixes

- Preserve ordering when loading multiple YAML files (ZPS-921)
- Fix setting of zProperty values when loading multiple YAML files. (ZPS-925)

Release 2.0.2

Fixes

- Only create a monitoring template if it changes or does not exist (ZPS-570)
- Ensure display of ZPL classes such as OSProcess in GUI elements (ZPS-572, ZPS-651)

Release 2.0.1

Fixes

- Ensure all datapoint attributes export to YAML (ZEN-26593)
- Ensure subsquent installations complete if ZP install fails (ZPS-627)

Release 2.0.0

Features

- zenpacklib is now an installable ZenPack
- Added Event Class definitions (ZEN-24903)
- Support multiple YAML file loading
- Support directory loading for YAML
- Support log verbosity per ZenPack
- Centralized, per-derived ZenPack logging
- Improved template change detection during install
- Improved type handling of yaml loaded/dumped data
- Support centralized use of older monolithic zenpacklib.py
- Added –optimize parameter to zenpacklib
- Dramatically enhanced unit testing
- Support for using enum proprty with datapoint properties (string/int mapping)
- Ability to call /opt/zenoss/bin/zenpacklib
- Added ZPLCommand to handle running zenpacklib with arguments
- Separated zenpacklib.py classes into module files
- Ability to use ZenPack-provided zenpacklib module
- Added support for Process Class definitions
- Deprecated support for python-based “yaml” specifications
- Support for threshold graphpoint legend and color (ZEN-24904)
- Ability to specify an initial sort column on a component grid
- Performance enhancments for grid display of metrics (ZEN-23870)
- Support for Device Link Providers
- Added troubleshooting aid for easily saving function data(writeDataToFile)
- Avoid setting zProperties on existing device class (ZPS-137)

Fixes

- Fix handling of boolean datasource options (ZEN-25315)
- Merge Detail View groups into ‘Overview’ group (ZEN-24759)
- Ensure that component detail pane honors relation “details_display” (ZEN-24762)
- Update ZenPackLib (ZP) Unit tests (ZEN-24599)

- Ensure that subcomponent nav JS uses relationship label if provided (ZEN-24305)
- Ensure ability to set label or a subclass on an inherited relationship (ZEN-24303)
- Ensure inherited relationship name overrides displayed in details pane (ZEN-24302)
- Ensure extra_paths is working (ZEN-24268)
- Ensure that 'extra_params' get applied to template-related objects (ZEN-24083)
- Improved handling of "custom columns exceed 750 pixels" warnings (ZEN-24022)
- Avoid patching _relations on ZPL-derived subclasse (ZEN-24018)
- Incorrect display of nested custom-named relations (ZEN-23995)
- Fix missing relations (ZEN-23968)
- Fix maximum recursion depth exceeded traceback in get_facets (ZEN-23840)
- Allow specifying properties on an inherited relationship (ZEN-23763)
- Zenpacklib logging more helpful and less scary (ZEN-23621)
- Batch buildRelations() commits during ZenPack installs (ZEN-22655)
- Support adding devtypes (ZEN-22366)
- Improve ImportError logging in class files (ZEN-22927)
- Ensure non-cached datapoints return current value (ZEN-22288)
- Fix issue when setting datapoint_cached to False (ZEN-22287)
- Set all component property details to correct Python type (ZEN-22057)
- Honor relationship label containing component overrides in component (ZEN-21966)
- Prevent attempts to process relationships not in class_relationships (ZEN-21927)
- Ensure component display properties honored (ZEN-19798)
- Support setting datapoint alias as string (ZEN-19486)
- Check datapoint consistency in template graph points and thresholds (ZEN-19461)
- Check/warn against reserved keyword use (ZEN-19460)
- getRRDTemplateName can return label of base class (ZEN-19025)
- Ensure catalog creation respects spec property indexes (ZEN-18269)
- Ensure device classes can be removed properly (ZEN-18134)
- Ensure that datapoint alias keys do not exceed 31 chars (ZEN-17950)
- Log obscure error with ill-defined relationships (ZEN-16701)
- Fix handling of !ZenPackSpec tag in yaml definitions

4.10.2 Version 1.1

Release 1.1.0

Features

- Add dynamicview_weight class field.

- Add overridable getDynamicViewGroup method to generated classes.
- Class icons beginning with / will be treated as absolute URL paths.
- Improve performance of entity properties in component grids.
- Simplify what device status means to critical event(s) in /Status.
- Improve grid performance with streamlined info adapters
- Add base class proxies for all platform component classes.

Fixes

- Fix tracebacks caused by property datapoint_cached. (ZEN-22287)
- Fix 'display' property to honor initialized values. (ZEN-19798)
- Fix wrong template displayed for subclassed component (ZEN-19025)
- Fix inheritance for displayed relationship properties (ZEN-23763)
- Fix traceback in get_facets (maximum recursion depth exceeded) (ZEN-23840)
- Ensure that 'extra_params' get applied to template-related objects (ZEN-24083)
- Fix for lost relationships on ZPL-derived subclasses (ZEN-24018)
- Fix for extra_paths failures (ZEN-24268)
- Fix to gracefully handle unknown relationship properties (ZEN-21927)
- Ensure that inherited relationship names are used (ZEN-24302)
- Ensure that inherited relationship names are displayed consistently (ZEN-24303)
- Ensure that subcomponent nav JS uses relationship label if given (ZEN-24305)
- Fix for setting of zProperty values before zProperty exists
- Fix "unexpected keyword default" message
- Fix support for extending platform component classes. (ZEN-25559)

Documentation

- Fix YAML reference for dynamicview_group class field.
- Fix documentation of default value for dynamicview_views.
- Document new component class proxies such as IpInterface and FileSystem.

4.10.3 Version 1.0

Release 1.0.13

Fixes

- Honor graph and graphpoint ordering in zenpack.yaml. (ZEN-23590)

Release 1.0.12

Fixes

- Fix tracebacks due to stale catalog entries. (ZEN-22592)
- Fix hidden zenpacklib errors due to unitialized logging.
- Prevent setting values on undefined zProperties.
- Drastically reduce catalog creation time.

Documentation

- Add missing types to zProperty documentation.

Release 1.0.11

Fixes

- Only show Dynamic View for components that support it. (ZEN-22391)
- Fix created __init__.py to work with zenpacklib.TestCase. (ZEN-22387)

Release 1.0.10

Fixes

- Fix display of nested component container-of-container. (ZEN-21897)

Documentation

- Fix graphpoint lineType documentation.

Release 1.0.9

Fixes

- Fix non-containing setters with standard device types. (ZEN-21747)
- Fix filtering of YAML templates in ZenPack export. (ZEN-21697)
- Prevent backups of unchanged monitoring templates. (ZEN-21719)

Release 1.0.8

Fixes

- Fix various dump_templates issues. (ZEN-18824)

Release 1.0.7

Fixes

- Fix dynamicview_relations type issue.

Release 1.0.6

Fixes

- Make YAML-defined JMX datasources work. (ZEN-21467)

Release 1.0.5

Fixes

- Fix KeyError on install after adding device class. (ZEN-21461)

Release 1.0.4

Features

- TestCase: Automatically load ZenPack's configure.zcml if it exists.
- Default to checkbox renderer for boolean properties. (ZEN-19585)

Fixes

- TestCase: Fix transaction error without DynamicView or Impact installed.
- Fix entity grid renderer to make it possible to click links into a new tab. (ZEN-19922)
- Fix enum property type. (ZEN-20769)

Release 1.0.3

Fixes

- Fix testing of SNMP datasources by converting OIDs to string.
- Fix for inherited relationships and properties not appearing in UI.

Release 1.0.2

Fixes

- Log YAML errors more concisely instead of full traceback. (ZEN-17681)
- Fix "[Object]" details panel display for custom renderers. (ZEN-17732)
- Fix handling of nested device class remove field.
- Fix KeyError when removing non-existent device class.
- Fix handling of datapoint rrdtype. (ZEN-18188)

Release 1.0.1

Features

- Add Class.extra_paths for controlling object path indexing.
- Add Class.filter_hide_from option.

Fixes

- Fix handling of class _properties and _relationships.
- Prefix ExtJS components to avoid conflicting zenpacklib versions.
- Fix handling of Class property types.
- Fix py_to_yaml for ZenPacks that subclass ZenPack.
- Remove superfluous YAML type hints from py_to_yaml conversion.
- Fix "Unable to find TEMPLATE_ID" installation error.
- Base component status on events in /Status event class.
- Fix removal of objects when PyYAML isn't installed.

Release 1.0.0

Features

- Added ability to define ZenPack with YAML.
- Added support for model classes and relationships.
- Added support for zProperties.
- Added support for device classes.
- Added support for monitoring templates.
- Added *create* command for creating ZenPacks from the command line.
- Added *lint* command to check YAML for correctness.
- Added *class_diagram* command to create yUML class diagram from YAML.
- Added *dump_templates* command to export monitoring templates to YAML.
- Added *py_to_yaml* command to convert old Python specs to YAML.
- Added *version* command to print zenpacklib's version.

Documentation

- Added first pass at documentation (http://zenpacklib.zenoss.com/).

4.11 License

zenpacklib is by default licensed under the terms of the *GNU General Public License*. The terms of GPLv2 prohibit the redistribution of zenpacklib as part of any software that is not also open source. See *Commercial License* if you'd like to use zenpacklib in a closed-source ZenPack.

4.11.1 GNU General Public License

zenpacklib is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

zenpacklib is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
                       51 Franklin St, Fifth Floor, Boston, MA  02110-1301 ␣
↪USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Library General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.

  We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.

  Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software.  If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

  Finally, any free program is threatened constantly by software
patents.  We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary.  To prevent this, we have made it clear that any
```

```
patent must be licensed for everyone's free use or not licensed at all.

  The precise terms and conditions for copying, distribution and
modification follow.

                    GNU GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License.  The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
language.  (Hereinafter, translation is included without limitation in
the term "modification".)  Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running the Program is not restricted, and the output from the Program
is covered only if its contents constitute a work based on the
Program (independent of having been made by running the Program).
Whether that is true depends on what the Program does.

  1. You may copy and distribute verbatim copies of the Program's
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

  2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices
    stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third
    parties under the terms of this License.

    c) If the modified program normally reads commands interactively
    when run, you must cause it, when started running for such
    interactive use in the most ordinary way, to print or display an
    announcement including an appropriate copyright notice and a
    notice that there is no warranty (or else, saying that you provide
```

```
    a warranty) and that users may redistribute the program under
    these conditions, and telling the user how to view a copy of this
    License.  (Exception: if the Program itself is interactive but
    does not normally print such an announcement, your work based on
    the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable
    source code, which must be distributed under the terms of Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
    an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it.  For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
control compilation and installation of the executable.  However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
```

```
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
compelled to copy the source along with the object code.

  4. You may not copy, modify, sublicense, or distribute the Program
except as expressly provided under this License.  Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
this License will not have their licenses terminated so long as such
parties remain in full compliance.

  5. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Program or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Program (or any work based on the
Program), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Program or works based on it.

  6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the
original licensor to copy, distribute or modify the Program subject to
these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to
this License.

  7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Program at all.  For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
```

patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices. Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded. In such case, this License incorporates
the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time. Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number. If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation. If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.

  10. If you wish to incorporate parts of the Program into other free
programs whose distribution conditions are different, write to the author
to ask for permission. For software which is copyrighted by the Free
Software Foundation, write to the Free Software Foundation; we sometimes
make exceptions for this. Our decision will be guided by the two goals
of preserving the free status of all derivatives of our free software and
of promoting the sharing and reuse of software generally.

NO WARRANTY

  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
REPAIR OR CORRECTION.

```
  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

                     END OF TERMS AND CONDITIONS
```

4.11.2 Commercial License

Zenoss, Inc. can provide zenpacklib under an appropriate commercial license if you are interested in developing a ZenPack, but don't want to release it as open source software. Contact partner@zenoss.com for more information.

Click *Next* for information on setting up a development environment.