

datasources. In order to write such code it is necessary to **really** understand the constructs that zenpacklib simplifies for you.

### 2.3.3 Standard Zenoss documentation

Each version of the Zenoss product Administration Guide has included basic ZenPack information, including the Zenoss 5 Administration Guide (see chapter 14). Find official Zenoss documentation at [https://www.zenoss.com/resources/documentation?field=zsd\\_core\\_value\\_selective=Core](https://www.zenoss.com/resources/documentation?field=zsd_core_value_selective=Core).

Zenoss 3 provided a Zenoss Developer's Guide, published in 2010, with much more in-depth information about coding ZenPacks but, although much of this information is still relevant, much is also out of date. This document appears to have disappeared from Zenoss's own websites but I found it in October 2015 at [http://docs.huihoo.com/zenoss/3/Zenoss\\_Developers\\_Guide\\_08-102010-3.0-v01.pdf](http://docs.huihoo.com/zenoss/3/Zenoss_Developers_Guide_08-102010-3.0-v01.pdf).

### 2.3.4 Community ZenPack documentation

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

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

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

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

## 3 The mechanics of building a ZenPack

### 3.1 ZenPack development environment



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

#### 3.1.1 Zenoss 4 and earlier

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

When developing a complex ZenPack that involves writing code, it is likely that some or all of the Zenoss daemons will need to be restarted.

Generally only a small core of daemons is required for testing - you don't need *zenjmx* running if you are testing a ZenPack based on Python code!

To ensure a minimal set of Zenoss daemons, two files are required in *\$ZENHOME/etc*:

- *DAEMONS\_TXT\_ONLY* must exist, probably zero-length
- *daemons.txt* with one daemon per line

Typically, *DAEMONS\_TXT\_ONLY* (note all in capitals) is created with the *touch* command.

Note that if you are trying to hide this file, to revert to having all daemons operational, do not rename it to something starting *DAEMONS\_TXT\_ONLY*; use *hide\_DAEMONS\_TXT\_ONLY*. At least some versions of Zenoss **would still find and use** a file called *DAEMONS\_TXT\_ONLY\_hide*.

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

- *zeneventserver*
- *zopectl*
- *zeneventd*
- *zenhub*
- *zenjobs*

Zenoss Service Dynamics also requires *zencatalogservice*.

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

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

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

The “-v10” provides full debugging output to stdout.

### 3.1.2 Zenoss 5

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

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

Control Center provides a snapshot mechanism to provide a **snapshot** backup facility before performing ZenPack installation.

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

```
serviced service snapshot Zenoss.core
```
  - c. The serviced command returns the ID of the new snapshot on completion.
5. Restart the *Zenoss* service.

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

#### 3.1.2.1 *zenoss* user

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

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

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

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

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

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

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

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

```

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

```

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

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

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

```
passwd zenoss
```



Note that the *zenoss* user on the base host does **not** share the same home directory as that in the containers. This includes differing *.bashrc* scripts and differing *.ssh* directories. A consequence is that any direct ssh communications tests that will

update the *zenoss* user's *.ssh/known\_hosts* file **must** be conducted from within a container.

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

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

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

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

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

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

The configuration must be reloaded and the service restarted:

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

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

<i>/z/zenpacks</i>	for ZenPack development
<i>/z/packages</i>	for Operating System packages
<i>/z/scripts</i>	for local scripts

### 3.1.2.3 Configuring the service for a development minimum

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

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

```
serviced service edit Zope
```

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

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

Restart Zope with:

```
serviced service restart Zope
```

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

```
serviced service edit zenping
```

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

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

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

Good candidates for setting to manual launch are:

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

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

- zenjserver
- zenpropertymonitor
- zenucsevents
- zenvsphere

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

#### 3.1.2.4 Useful references for managing a Zenoss 5 environment

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

Some useful Knowledge Base articles are emerging which provide assistance:

- “Introduction to Zenoss Control Center” - <https://support.zenoss.com/hc/en-us/articles/206278353-Introduction-to-Zenoss-Control-Center>
- “Virtualization and Docker Containerization for Poets” - <https://support.zenoss.com/hc/en-us/articles/202254069-Virtualization-and-Docker-Containerization-for-Poets>
- “How to add tools or scripts into a Resource Manager 5.x Docker Container” - <https://support.zenoss.com/hc/en-us/articles/207610516-How-to-add-tools-or-scripts-into-a-Resource-Manager-5-x-Docker-Container>
- “How to troubleshoot Resource Manager 5.x services that fail to start” - <https://support.zenoss.com/hc/en-us/articles/207348996-How-to-troubleshoot-Resource-Manager-5-x-services-that-fail-to-start>
- “How to Recover Control Center from Hardware Failure” - <https://support.zenoss.com/hc/en-us/articles/204643769-How-to-Recover-Control-Center-from-Hardware-Failure>

## 3.2 ZenPack creation

The method for creating a ZenPack varies depending on the version of Zenoss. Zenoss 4 and earlier provides a GUI menu but no command-line method. Zenoss 5 offers a command-line interface but no GUI.

zenpacklib provides a method common to Zenoss 4 and 5 but only creates a minimal ZenPack directory structure.

### 3.2.1 What's in a name?

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

Pack	Package	Author	Version	Egg
<a href="#">ZenPacks.ShaneScott.ipSLA</a>	ShaneScott	Shane William Scott	3.5.4	Yes
<a href="#">ZenPacks.SymbioticSystemDesign.BaseMIBs</a>	SymbioticSystemDesign	Manuel Deschambault	1.0.0	Yes
<a href="#">ZenPacks.community.DirFile</a>	community	Jane Curry - jane.curry@skills-1st.co.uk	1.0.2	Yes
<a href="#">ZenPacks.community.LogMatch</a>	community	Jane Curry - jane.curry@skills-1st.co.uk	1.0.1	Yes
<a href="#">ZenPacks.community.dummy</a>	community	Jane Curry	1.0.0	Yes
<a href="#">ZenPacks.zenoss.ApacheMonitor</a>	zenoss	Zenoss	2.1.4	Yes
<a href="#">ZenPacks.zenoss.Dashboard</a>	zenoss	Zenoss	1.0.6	Yes
<a href="#">ZenPacks.zenoss.DellMonitor</a>	zenoss	Zenoss	2.2.0	Yes
<a href="#">ZenPacks.zenoss.DeviceSearch</a>	zenoss	Zenoss	1.2.1	Yes
<a href="#">ZenPacks.zenoss.DigMonitor</a>	zenoss	Zenoss	1.1.0	Yes
<a href="#">ZenPacks.zenoss.DnsMonitor</a>	zenoss	Zenoss	2.1.0	Yes
<a href="#">ZenPacks.zenoss.FinMonitor</a>	zenoss	Zenoss	1.1.0	Yes

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

It is perfectly possible to have names with more than three segments; *ZenPacks.zenoss.Microsoft.Windows* is an example.



ZenPack names **must** be unique.



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

### 3.2.2 ZenPack directory hierarchy

When creating a new ZenPack, the only initial requirement is the name. Once created, you can then specify other parameters, like Zenoss version dependency or other co-requisite ZenPacks. You should also specify an author, a version and a license for this ZenPack. It is good practice to supply contact details in the author field; for example Jane Curry - [jane.curry@skills-1st.co.uk](mailto:jane.curry@skills-1st.co.uk). These details can be configured using the GUI or by editing the *setup.py* file in the top-level ZenPack directory.





The screenshot shows the ZenPackManager interface. The top navigation bar includes 'DASHBOARD', 'EVENTS', 'INFRASTRUCTURE', 'REPORTS', and 'ADVANCED'. The user 'jane' is logged in. The left sidebar has 'Settings' selected. The main content area is titled 'ZenPackManager > ZenPacks.community.dummy'. It contains two sections: 'Metadata' and 'Dependencies'.

**Metadata**

Name	ZenPacks.community.dummy
Version	1.0.0
Author	Jane Curry - jane.curry@skills-1st.co.uk (ZenossDev)
License	GPLv2

There is a 'Save' button below the metadata fields.

**Dependencies**

Required?	Name	Version(s)
<input type="checkbox"/>	Zenoss	>=3.2
<input type="checkbox"/>	ZenPacks.Eseye.ActiveMQ	
<input type="checkbox"/>	ZenPacks.Markit.MarkitDatabase	
<input type="checkbox"/>	ZenPacks.Markit.RigHost	
<input type="checkbox"/>	ZenPacks.Nova.IbmDb2	
<input type="checkbox"/>	ZenPacks.Nova.WinServiceSNMP	
<input type="checkbox"/>	ZenPacks.Nova.Windows.SNMPPerfMonitor	
<input type="checkbox"/>	ZenPacks.SCC.ShowGraphPortlet	
<input type="checkbox"/>	ZenPacks.ShaneScott.ipSLA	
<input type="checkbox"/>	ZenPacks.SteelHouseLabs.EventForwarder	

Figure 6: Creation details for a ZenPack

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

- `ZenPacks.community.dummy`
- `ZenPacks.community.dummy/ZenPacks`
- `ZenPacks.community.dummy/ZenPacks/community`
- `ZenPacks.community.dummy/ZenPacks/community/dummy`



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

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

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

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

```
zenoss@zen42:/opt/zenoss/ZenPacks/ZenPacks.community.dummy
File Edit View Search Terminal Help
[zenoss@zen42 ZenPacks.community.dummy]$ pwd
/opt/zenoss/ZenPacks/ZenPacks.community.dummy
[zenoss@zen42 ZenPacks.community.dummy]$ cat setup.py
#####
# These variables are overwritten by Zenoss when the ZenPack is exported
# or saved. Do not modify them directly here.
# NB: PACKAGES is deprecated
NAME = "ZenPacks.community.dummy"
VERSION = "1.0.0"
AUTHOR = "Jane Curry"
LICENSE = "GPLv2"
NAMESPACE PACKAGES = ['ZenPacks', 'ZenPacks.community']
PACKAGES = ['ZenPacks', 'ZenPacks.community', 'ZenPacks.community.dummy']
INSTALL_REQUIRES = []
COMPAT_ZENOSS_VERS = ">=3.2"
PREV_ZENPACK_NAME = ""
# STOP REPLACEMENTS
#####
# Zenoss will not overwrite any changes you make below here.

import os
from subprocess import Popen, PIPE
from setuptools import setup, find_packages

# Run "make build" if a GNUmakefile is present.
if os.path.isfile('GNUmakefile'):
    print 'GNUmakefile found. Running "make build" ..'
    p = Popen('make build', stdout=PIPE, stderr=PIPE, shell=True)
    print p.communicate()[0]
    if p.returncode != 0:
        raise Exception("make build" exited with an error: %s' % p.returncode)

setup(
    # This ZenPack metadata should usually be edited with the Zenoss
    # ZenPack edit page. Whenever the edit page is submitted it will
    # overwrite the values below (the ones it knows about) with new values.
    name=NAME,
    version=VERSION,
    author=AUTHOR,
```

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

Each of the directories, other than the top-level, will have a largely-empty **`__init__.py`** file that is required but won't need any modification for simple ZenPacks. Note that it is **essential** that all directories and subdirectories below the top-level that contain python code, must have an **`__init__.py`** even if it is a zero-length file.

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

### ● Important Points

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

Check [https://zenosslabs.readthedocs.org/en/latest/zenpack\\_standards\\_guide.html](https://zenosslabs.readthedocs.org/en/latest/zenpack_standards_guide.html) for a good overview of standard ZenPack files and directories and some best practices.

```
zenoss@zen42:/opt/zenoss/ZenPacks/ZenPacks.community.dummy/ZenPacks/community/dummy$ pwd
/opt/zenoss/ZenPacks/ZenPacks.community.dummy/ZenPacks/community/dummy
zenoss@zen42 dummy$ ls -l
total 124
-rw-r--r-- 1 zenoss zenoss 843 Mar 11 2014 analytics.py
drwxr-xr-x 2 zenoss zenoss 4096 Oct 14 2014 bin
drwxr-xr-x 3 zenoss zenoss 4096 Oct 23 09:18 browser
-rw-r--r-- 1 zenoss zenoss 7111 Mar 11 2014 configure.zcml
drwxr-xr-x 2 zenoss zenoss 4096 Oct 14 2014 daemons
drwxr-xr-x 2 zenoss zenoss 4096 Oct 14 2014 datasources
-rw-r--r-- 1 zenoss zenoss 1327 Mar 11 2014 dynamicview.py
-rw-r--r-- 1 zenoss zenoss 692 Mar 11 2014 events.py
-rw-r--r-- 1 zenoss zenoss 1354 Mar 11 2014 ExampleComponent.py
-rw-r--r-- 1 zenoss zenoss 1070 Mar 11 2014 ExampleDevice.py
-rw-r--r-- 1 zenoss zenoss 3665 Mar 11 2014 impact.py
-rw-r--r-- 1 zenoss zenoss 1956 Mar 11 2014 info.py
-rw-r--r-- 1 zenoss zenoss 1764 Mar 11 2014 __init__.py
-rw-r--r-- 1 zenoss zenoss 169 Oct 23 09:18 __init__.pyc
-rw-r--r-- 1 zenoss zenoss 1837 Mar 11 2014 interfaces.py
drwxr-xr-x 2 zenoss zenoss 4096 Oct 14 2014 lib
drwxr-xr-x 2 zenoss zenoss 4096 Oct 14 2014 libexec
-rw-r--r-- 1 zenoss zenoss 18092 Oct 23 09:18 LICENSE.txt
drwxr-xr-x 2 zenoss zenoss 4096 Oct 23 09:18 migrate
drwxr-xr-x 3 zenoss zenoss 4096 Oct 14 2014 modeler
drwxr-xr-x 2 zenoss zenoss 4096 Oct 14 2014 objects
drwxr-xr-x 4 zenoss zenoss 4096 Mar 11 2014 reports
drwxr-xr-x 2 zenoss zenoss 4096 Oct 14 2014 services
drwxr-xr-x 2 zenoss zenoss 4096 Oct 14 2014 tests
-rw-r--r-- 1 zenoss zenoss 3629 Mar 11 2014 zenexample.py
drwxr-xr-x 2 zenoss zenoss 4096 Oct 14 2014 zep
zenoss@zen42 dummy$
```

Figure 8: Default files and subdirectories in ZenPack base directory

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

- analytics.py For integration with Service Dynamics Analytics
- bin Any binaries the ZenPack creates
- browser Code for driving the GUI
- configure.zcml “Glue” code to link info & interface information to GUI code
- daemons If the ZenPack creates a new daemon, code goes here
- datasources For new datasource code
- dynamicview.py For integration with Service Dynamics dynamic views
- events.py For integration with Service Dynamics Impact

- `ExampleComponent.py` Sample code for a new component
- `ExampleDevice.py` Sample code for a new device object class
- `info.py` Defines mapping between object attributes and interface classes for display in the GUI
- `__init__.py` Mandatory. Can modify almost anything - or nothing!
- `interfaces.py` Defines what attributes should be displayed in GUI & how
- `lib` For new libraries required by the ZenPack
- `libexec` For scripts delivered and used by the ZenPack
- `LICENSE.txt` The text of the selected license
- `migrate` Code to help migration between Zenoss versions
- `modeler` Directory hierarchy for modeler plugins
- `objects` Contains `objects.xml` - objects added to ZenPack from the GUI and possibly other objects
- `reports` For reports created by the ZenPack
- `services` Provides configuration services for custom daemons
- `tests` test scripts
- `zenexample.py` Sample daemon code
- `zep` To provide custom triggers, notifications & event fields

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

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

### 3.2.3 ZenPack creation for Zenoss 4 and earlier

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

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

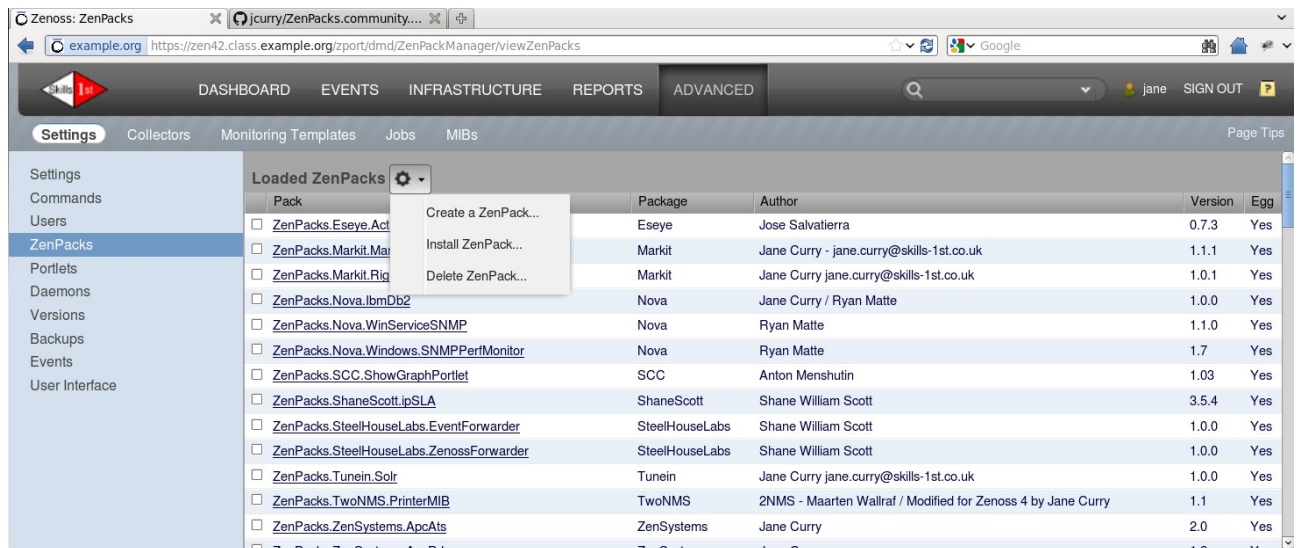


Figure 9: ZenPacks option from the ADVANCED -> Settings menu

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

- Create a ZenPack
- Install ZenPack
- Delete ZenPack

### 3.2.4 Zenoss 5 ZenPack creation

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

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

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

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

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

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

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

### 3.2.5 ZenPack creation using zenpacklib

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

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

See section 8.4 for more information.

The drawback with this method is that the directory hierarchy is only created down to the base directory,

`/opt/zenoss/ZenPacks/ZenPacks.community.dummy/ZenPacks/community/dummy.`



No subdirectories or sample files are created. For this reason, it is usually better practice to avoid zenpacklib for the actual ZenPack creation.

### 3.3 Exporting ZenPacks

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



**Note** that the export process also creates the *objects/objects.xml* file. If you were planning to create a tar bundle of your ZenPack as backup or to ship elsewhere, do ensure that an export has been performed, otherwise the *objects.xml* will not contain any GUI additions since the last export, even though those objects may exist in the running environment.

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

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

eg. *Device\_Server\_Linux\_UserGroup\_rrdTemplates\_test.xml*

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

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

The options presented are:

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

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

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

### 3.4 Installing ZenPacks



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



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




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



After installation, some Zenoss daemons need to be recycled; this depends on the ZenPack and the version of Zenoss.

- The safe (but slow) option is to recycle all daemons.
- With Zenoss Service Dynamics 4.x you may have to recycle all daemons. *zenhub* and *zenwebserver* are an absolute minimum.
- With Zenoss 5 (Core and SD) you have to restart the *Zenoss.core* or *Zenoss.resmgr* application and all its child services.
- As a minimum on Zenoss Core prior to version 5, you need to recycle *zenhub* and *zopectl*. Adding or changing any object (device, component) will necessitate a full recycle. **Adding** a new element (datasource, parser, report, modeler) will necessitate a full recycle. **Changing** an existing datasource, modeler or report you will probably get away with just restarting *zenhub* and *zopectl*. If you have changed a datasource then you will also need to recycle the daemon that runs the datasource eg. *zenpython*.
- Where significant development work will take place, it is worth investing time to confirm what minimum daemons / containers need to be restarted for a code change to be fully implemented. This can be a huge time saver. Certainly a test environment should be available with a single localhost collector.
- The documentation provided with the ZenPack should state what needs recycling.

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

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

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

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

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

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

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

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

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

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

Note that some developers deliberately remove the Python version nomenclature from their egg file, especially if it is not Python-version-specific. Great care must be taken if files are renamed as this can cause major issues if done injudiciously, including making the ZODB inconsistent and rendering the ZenPack neither installable nor uninstallable.

### 3.4.1 Installing ZenPacks on Zenoss 4

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

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

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

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





### 3.4.2 Installing ZenPacks on Zenoss 5

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

The `.bashrc` file for the `zenoss` user sets up a common directory, `/z`, that is shared between the base host and the various containers and creates command aliases that set the current directory to `/z`. As a local standard, this document maintains ZenPack code in `/z/zenpacks`. **Note** that `/z/zenpacks` must be local (not mounted) and must be readable, writeable, and executable by all users. `.bashrc` defines that both `zenpack` and `zenpacklib` commands will be executed in the context of the current directory being `/z/zenpacks`.

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

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

The Zenoss service must be restarted with:

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

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

TODO: If a link-install is performed, the `.egg-link` file resides in `/opt/serviced/var/volumes/dtvdokjjhx6b9kkoehxu839p/var-zenpacks/ZenPacks ?? /exports/serviced_var_volumes/dtvdokjjhx6b9kkoehxu839p/var-zenpacks/ZenPacks ??` and points to the top-level directory of the ZenPack.

## 3.5 Removing ZenPacks

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

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

Zenoss 5 only provides the command-line option.

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



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