debexpo Documentation

Release 3.0dev

Jonny Lamb

February 23, 2018

Contents

debexpo (Debian package exposition) is a web application that allows you to run a user-friendly public Debian package repository with social interaction that you may be used to from social networking sites. debexpo is the basis for the mentors.debian.net site but may be useful for others as well. It is basically a Pylons application that can be deployed on any server. You do not need anything more than Pylons, a little harddisk space and a database like PostgreSQL, MySQL or Sqlite.

CHAPTER 1

Setting up debexpo

1.1 Installing and setting up debexpo

debexpo is easy to set up on your own. Simply follow the instructions below.

There are three solutions:

1. Install all dependencies on your system as root.
2. Install dependencies and debexpo in an isolated environment using virtualenv and virtualenvwrapper.
3. Use VirtualBox and Vagrant. If you choose this method, follow the instructions under *Using Vagrant*.

1.1.1 Getting debexpo

You can clone the git repository:

```
git clone git://git.debian.org/debexpo/debexpo.git
```

1.1.2 Dependencies needed for both methods

Whatever method you choose, these packages are required:

```
sudo apt-get install python-apt python-debian iso-codes
```

If you want to run qa plugins, you will need *lintian* and *dpkg-dev*:

```
sudo apt-get install lintian dpkg-dev
```

1.1.3 Installing on Debian Squeeze or Wheezy as root

You need to install the required packages. Using *apt*, you should execute:

```
sudo apt-get install python-setuptools python-sphinx python-pylons python-sqlalchemy␣
↪python-soappy python-nose python-pybabel
```

python-nose is optional if you don't want to run the test suite.

You also need python-soaplib (version >= 0.8.2).

Using pip:

```
sudo pip install soaplib
```

1.1.4 Installing in a virtualenv

Using this method, you will create a virtual Python environment in which you can install the dependencies for debexpo without altering your system (i.e., without requiring root). In addition, this will also let you isolate debexpo's requirements, in the event an application installed globally might require a conflicting version of a library, or vice versa.

Virtualenv setup

Skip this section if you already have a working virtualenv setup.

Install *virtualenvwrapper*:

```
sudo apt-get install virtualenvwrapper
```

Dependencies

To install *lxml* from sources, you will need *gcc*, *libxml2*, *libxslt1.1* and *python-dev*, *python-apt* depends on *libapt-pkg-dev*:

```
sudo apt-get install gcc libxml2 libxml2-dev libxslt1.1 libxslt1-dev python-dev␣
↪libapt-pkg-dev
```

Debexpo installation

First, create a new virtualenv for debexpo, and enter it:

```
mkvirtualenv expo
workon expo
```

Note: If you get a "command not found" error for "mkvirtualenv", run the following in your shell:

```
/etc/bash_completion.d/virtualenvwrapper
```

Note that now, whenever you run "python", you run an interpreter that is sandboxed to the "virtualenv" in question. You can test this by typing:

```
which python
```

and you will see it is not /usr/bin/python! Additionally, your shell prompt should have a little prefix before the prompt that looks like:

```
(expo)
```

You can now install debexpo. This will download and install all required libraries:

```
python setup.py develop
```

If for any reason you need to exit the virtualenv, you may enter *deactivate* to exit the virtualenv.

1.1.5 Editing your configuration

Now edit *development.ini* to match your configuration.

1.1.6 Setting up the application

Execute the following commands to setup the application:

```
paster setup-app development.ini
python setup.py compile_catalog
```

1.1.7 Using Vagrant

1. Install VirtualBox.
2. Install Vagrant.
3. In the checked-out debexpo repository on the host machine, run:

   ```
   vagrant up --provision
   vagrant ssh
   ```

4. You'll now be in a shell session on your Vagrant-configured VirtualBox.
5. Run:

   ```
   cd debexpo
   . venv/bin/activate
   ```

6. You now have a setup virtualenv with all the dependencies for debexpo installed. Follow the rest of the instructions to run debexpo.

1.1.8 Running debexpo

Using paste's built-in webserver

Simply execute:

```
paster serve development.ini
```

and visit http://localhost:5000/ in your web browser.

Using Apache

(Canonical instructions for getting Pylons apps working under Apache are here.)

1. Install *apache2*, *mod-fastcgi* and *flup*:

```
sudo apt-get install python-flup apache2 libapache2-mod-fastcgi
```

2. Edit the `server:main` section of your *debexpo.ini* so it reads something like this:

```
[server:main]
use = egg:PasteScript#flup_fcgi_thread
host = 0.0.0.0
port = 6500
```

3. Add the following to your config:

```
<IfModule mod_fastcgi.c>
  FastCgiIpcDir /tmp
  FastCgiExternalServer /some/path/to/debexpo.fcgi -host localhost:6500
</IfModule>
```

Note: Parts of this may conflict with your */etc/apache2/conf-available/fastcgi.conf*. */some/path/to/debexpo/fcgi* need not physically exist on the webserver.

1.2 Config file

These are configuration options that go in the ini file that configures debexpo. Every option should be present otherwise debexpo will fail somewhere. A sane default is in the distributed ini file.

1.2.1 debexpo.upload.incoming

This variable specifies the incoming directory. Newly uploaded files will be installed into this directory. Therefore, it should be writeable by the webserver.

1.2.2 debexpo.repository

This variable specifies the repository directory, where uploaded files are stored. The directory structure is easy – files belonging to a package are stored in a subdirectory of this directory, with name of the source package name. For example, If this is set to `/home/myexpo/files` then the package 'cream' would have its files stored in `/home/myexpo/files/cream/`. The directory does not have a Sources.gz file (no "apt-get source") but source packages can be downloaded via "dget . . . dsc".

1.2.3 debexpo.importer

This variable specifies the path to the importer script, distributed in `bin/debexpo-importer`. Therefore, this option is typically `%(here)s/bin/debexpo-importer`.

1.2.4 debexpo.handle_debian

This variable specifies whether debexpo should handle the `/debian/` directory. This can be set to false and let Apache handle this directory.

1.2.5 debexpo.sitename

Name of the site repository. This is used as the title of the web pages.

1.2.6 debexpo.tagline

Tag-line of the repository. This is used under the main title of the web pages.

1.2.7 debexpo.logo

Site logo of the repository to display at the top of the web pages.

1.2.8 debexpo.email

Email address of site support.

1.2.9 debexpo.debian_specific

Toggle whether to show Debian-specific contents of the site. Values are `true` or `false`.

1.2.10 debexpo.plugins.post_upload

Which post-upload plugins to run, in what order. Separate each plugin with a space.

1.2.11 debexpo.plugins.qa

Which QA plugins to run, in what order. Separate each plugin with a space.

1.2.12 debexpo.plugins.post_upload_to_debian

Which plugins to run when the package is uploaded to Debian, in what order. Separate each plugin with a space.

1.2.13 debexpo.plugins.post_successful_upload

Which plugins to run when a package is successfully uploaded to the repository, in what order. Separate each plugin with a space.

1.2.14 debexpo.plugindir

Directory to add to path to put user-defined plugins in.

1.2.15 debexpo.debian_mirror

Location of the most convenient Debian mirror.

1.2.16 debexpo.changes_list

Email to send package accepts to.

1.2.17 debexpo.server

Server root URL debexpo is running on, including protocol and excluding trailing slash. For example `http://localhost:5000`.

1.2.18 debexpo.frontpage

Path to file to include which contains HTML for the front page. This defaults to `%(here)s/debexpo/public/frontpage.html`.

1.2.19 debexpo.gpg_path

Path to the GnuPG binary. This defaults to `/usr/bin/gpg`.

CHAPTER 2

Using debexpo

2.1 Uploading

Uploading to a debexpo repository is easy. You must use dput as this is the only tool that can upload via HTTP (at the time of writing). (Former versions of debexpo used HTTP uploads with authentication which repeatedly failed due to dput bugs which were in fact urllib2 API changes.)

2.1.1 Setting up dput

Once you have debexpo and dput installed and set up, add an entry like the following to your `~/.dput.cf`:

```
[debexpo]
fqdn = localhost:5000
incoming = /upload/email@address/yourpassword
method = http
allow_unsigned_uploads = 0
```

You should change the *email@address* and *yourpassword* entries with the email address and password you use to login. And you may have to change the *fqdn* to suit your setup.

2.1.2 Uploading the package

Now you should execute:

```
dput debexpo package_version_source.changes
```

You will get an output like this:

```
% dput -f debexpo odccm_0.11.1-17_source.changes
Uploading to debexpo (via http to localhost:5000): ...
```

At this point your upload will run and you should see the logs flying by showing the status of the upload. The **.changes* file will get uploaded last

2.2 Plugins

debexpo has many "plugins" for different purposes. Some are to make sure packages are in a good condition for the archive, some make sure the packages can even be imported as they might be damaged in the upload and some simply find information about the package, such as the programming language used.

Here is a list of the plugins installed by standard with debexpo:

2.2.1 buildsystem

This plugin looks at the package and by looking at the package's *Build-Depends*, it tries to work out what build system the package is using. The possible options are:

- CDBS
- debhelper
- debhelper v7
- unknown

This is an informational QA plugin and should only be run in that stage, between upload and successful importing.

2.2.2 changeslist

This plugin emails the `debexpo.changes_list` email address with an email on every package upload in exactly the same format as the debian-devel-changes mailing list.

This is a post-successful-upload plugin and should only be run in that stage, after the package has successfully been imported into the archive.

2.2.3 checkfiles

This plugin checks whether all the files referenced in the *changes* file are present. It also checks each file's md5sum to make sure it matches the md5sum given in the *changes* file.

If any part of this plugin fail, the whole upload should fail as this is a critical error.

This is a post-upload plugin and should only be run in that stage, straight after the package has been uploaded onto the system, but before any package manipulation.

2.2.4 closedbugs

This plugin checks on the Debian BTS whether bugs that are reported to be closed in the package upload do actually belong to the package being uploaded.

This is an information QA plugin and should only be run in that stage, between upload and successful importing.

2.2.5 controlfields

This plugin looks for additional `debian/control` fields, such as *Vcs-Browser* and *Homepage*.

This is an information QA plugin and should only be run in that stage, between upload and successful importing.

2.2.6 debianqa

This plugin tests a number of things with the uploaded package against information in Debian:

- whether the package is an NMU
- whether the package is already in Debian
- whether the package maintainer is the Debian maintainer
- whether the package introduces a new maintainer
- whether the package closes any wnpp bugs
- finds information about any ITPs closed
- finds previous sponsors of the package

This is an information QA plugin and should only be run in that stage, between upload and successful importing.

2.2.7 diffclean

This package looks at the package's *diff.gz* and makes sure that it is clean. This means that it does not include any changes to files outside of the *debian* directory as this is considered bad practice.

This is an information QA plugin and should only be run in that stage, between upload and successful importing.

2.2.8 getorigtarball

This package looks whether the package is missing an *original tarball* and if so, it tries to download the appropriate file from the Debian archives. You can set your favourite Debian mirror with the `debexpo.debian_mirror` config option.

This is a post-upload plugin and should only be run in that stage, straight after the package has been uploaded onto the system, but before any package manipulation.

2.2.9 gpgsigned

This plugin checks to see whether the *changes* and *dsc* files have been GPG signed.

This is an information QA plugin and should only be run in that stage, between upload and successful importing.

2.2.10 lintian

This plugin runs lintian on the package.

This is an information QA plugin and should only be run in that stage, between upload and successful importing.

2.2.11 maintaineremail

This plugin looks to see whether the email of the uploader of the package is the same as the email of the maintainer of the package.

This is an information QA plugin and should only be run in that stage, between upload and successful importing.

2.2.12 native

This plugin looks to see whether the package is a native package.

This is an information QA plugin and should only be run in that stage, between upload and successful importing.

2.2.13 notupoader

This plugin checks to make sure that the uploader of the package is the owner of any subsequent package uploads of the same name.

If the plugin finds that there has been a previous upload of the package, and the previous uploader is different from the new uploader, the import will stop.

This is a post-upload plugin and should only be run in that stage, straight after the package has been uploaded onto the system, but before any package manipulation.

2.2.14 removepackage

This plugin removes a package and all of its associated comments, metrics and information from the database.

This is a post-upload-to-debian plugin that should only be run after the package has been uploaded to Debian.

2.2.15 watchfile

This plugin checks to see whether the package has a watch file. If it does, then the plugin will check the watch file to make sure it works. If it does work, then it will report back on any new upstream versions.

This is an information QA plugin and should only be run in that stage, between upload and successful importing.

2.3 SOAP documentation

debexpo repositories can be accessed by using SOAP using its *soap* controller. Its methods are described below:

`uploader`(*email*)
: Returns an array of packages given an uploader's email address.

  email is the email address you are querying.

`section`(*name*)
: Returns an array of packages given a section name.

  name is the name of the section you are querying.

`maintainer`(*email*)
: Returns an array of packages given a maintainer's email address.

  email is the email address you are querying.

packages()
Returns an array of all packages.

package(*name*, *version*)
Returns details on a specific package and version.

name is the package name you are querying.

version is the version name you are querying.

2.3.1 Example client

Using SOAPpy:

```
import SOAPpy
server = SOAPpy.SOAPProxy("http://localhost:5000/soap")
print server.section(name='utils')
```

And the output:

```
<SOAPpy.Types.structType retval at 141282572>: {'stringArray': <SOAPpy.Types.
↪structType stringArray at 141279660>: {'string': ['odccm', '0.11.1-17', 'Jonny Lamb
↪<jonny@debian.org>', 'odccm - Daemon to keep a connection to Windows Mobile device',
↪ 'http://localhost:5000/package/odccm']}}
```

CHAPTER 3

Development documentation

3.1 Building the software

If you like to build the software you can get the Git repository from debexpo.workaround.org using:

```
git clone git://debexpo.workaround.org/debexpo.git
```

Then simply enter into the debexpo directory and execute `make build`:

```
cd debexpo
make build
```

It is easier in some situations to leave debexpo in its source directory and run it from there. However, if you wish to have it installed, create a virtualenv environment:

```
aptitude install python-virtualenv
virtualenv .
source bin/activate
```

Then you can safely:

```
make install
```

to install the package in your encapsulated environment.

If you attempt to install the package without virtualenv then setuptools (the Python software management system) will install the files into the system-wide directories `/usr/lib/pythonX.Y/site-packages/`. Setuptools is not good at removing files again and it is generally a bad idea to mix setuptools-installed packages with Debian packages. So you should know what you are doing.

3.2 Writing plugins

Writing plugins for debexpo is easy. You only need to know a bit of python and you're away! A sample QA plugin is shown here:

```
import logging

from debexpo.lib import constants
from debexpo.lib.base import *
from debexpo.plugins import BasePlugin

log = logging.getLogger(__name__)

class YInNamePlugin(BasePlugin):

    def test_y_in_name(self):
        log.debug('Checking whether the name has a letter Y in its name')

        package_name = self.changes['Source']

        if 'y' in package_name:
            self.passed('y-in-name', None, constants.PLUGIN_SEVERITY_INFO)
        else:
            self.failed('no-y-in-name', None, constants.PLUGIN_SEVERITY_INFO)

plugin = YInNamePlugin

outcomes = {
    'y-in-name' : { 'name' : 'Package has the letter Y in its name' },
    'no-y-in-name' : { 'name' : 'Package has no letter Y in its name' },
}
```

This plugin looks at a package and looks whether the package name has the letter Y in it. It has two outcomes. It will pass the plugin if a letter Y is present. It will fail the test, albeit with a low priority, if a letter Y is not present.

3.2.1 A walk through

Lines 1-7:

```
import logging

from debexpo.lib import constants
from debexpo.lib.base import *
from debexpo.plugins import BasePlugin

log = logging.getLogger(__name__)
```

These are just imports of the logger, debexpo constants, and other classes that you need not worry about. These imports and statements should always be present in a plugin.

Line 9:

```
class YInNamePlugin(BasePlugin):
```

This starts the definition of the plugin, which must extend on the BasePlugin class. The name of this class doesn't matter, as you will see in a bit.

Line 11:

```
def test_y_in_name(self):
```

This starts the actual plugin. All methods in the plugin starting with the name "*test*" will be run. This allows you to have as many tests in each plugin as you wish. You may also have other methods that, as long as they do not start with the word "*test*" will not be run automatically.

Line 12:

```
log.debug('Checking whether the name has a letter Y in its name')
```

This is a simple logging statement. This should be used well and frequently if necessary. It uses the standard python logging module.

Line 14:

```
package_name = self.changes['Source']
```

This gets the name of the package by getting the *Source* field from the *changes* file. Most plugin-running locations will have a *self.changes* `debexpo.lib.changes.Changes` object that can be used and inspected in the plugin.

Line 17:

```
self.passed('y-in-name', None, constants.PLUGIN_SEVERITY_INFO)
```

This records a pass for the plugin. The `passed` and `failed` methods both have three arguments:

class debexpo.plugins.**BasePlugin**(***kw*)

The class all other plugins should extend.

failed(*outcome*, *data*, *severity*)

Adds a PluginResult for a failed test to the result list.

outcome Outcome tag of the test.

data Resulting data from the plugin, like more details about the process.

severity Severity of the result.

info(*outcome*, *data*)

Adds a PluginResult for an info test to the result list.

outcome Outcome tag of the test.

data Resulting data from the plugin, like more detail about the process.

passed(*outcome*, *data*, *severity*)

Adds a PluginResult for a passed test to the result list.

outcome Outcome tag of the test.

data Resulting data from the plugin, like more details about the process.

severity Severity of the result.

As you can see, there is another method called `info`. This is for outcomes that do not mean there was a success, and similarly do not mean there was a failure. The YInNamePlugin is actually a good example of where the `info` method should be used. It defaults the severity to "info".

The *outcome* first variable of the functions should be a string relating to the key of the `outcomes` dictionary at the bottom of the file. The value of each key in this dictionary should be another dictionary with at least one key/value pair: *name*: This should be an English string as to what the outcome actually means.

Line 21:

```
plugin = YInNamePlugin
```

This shows that the name of the plugin really does not matter. As long as the `plugin` variable points to a class based on BasePlugin, it is fine.

3.2.2 Other plugins

This is a very brief outline of a simple QA plugin. However, there are different stages of plugin execution. The *Plugins* page tells more about stock plugins, what they do, and when they should run. You should use these plugins as a reference for future plugins.

If you get stuck, do not hesitate to pop by the mailing list.

3.3 Writing cronjobs

Writing cronjobs works similar to plugins. The invocation and arguments passed are different though. A minimal cronjob looks like this:

```
import datetime
from debexpo.cronjobs import BaseCronjob

class ImportUpload(BaseCronjob):
    def setup(self):
        pass

    def teardown(self):
        pass

    def invoke(self):
        self.log.debug("Hello World")

cronjob = ImportUpload
schedule = datetime.timedelta(seconds = 10)
```

3.3.1 The architecture

A cronjob should be subclassed from BaseCronjob. That ensure API compliant invocation. A worker thread runs your jobs cyclically, persistence is guaranteed for the object runtime. Technically, you `must` define two objects in your module.

Invocation:: cronjob = ImportUpload schedule = datetime.timedelta(seconds = 10)

The 'cronjob' attribute is an object reference which should be instantiated upon cronjob invocation. The 'schedule' attribute defines how often your cronjob should invoke your worker method. This must be a datetime.timedelta object. This is a soft guarantee. The Worker thread will guarantee you not to run the job more often than you specified, but it will not invoke it precisely for every delta. Your cronjob will not be invoked if another cronjob is still pending or running once your delta expires. Additionally the Worker thread does not execute the worker queue more often than every *debexpo.cronjob_delay* milliseconds.

3.3.2 The constructor

Please don't override the base class constructor, it will call your *setup* method if you need any setup. The following attributes will be instantiated for your cronjob:

`self.parent`

This is a reference to the worker object, which instantiated your cronjob. You can call any method in from there, if necessary.

`self.config`

This is an instantiated configuration object. You can access every Debexpo configuration setting from it.

`self.log`

An instantiated log object, use it to display messages within the worker thread

3.3.3 The destructor

Similarly to the constructor, don't override the destructor. Use the *teardown* method instead

3.3.4 The worker method invocation

Implement the *invoke* method as your working horse. It will be called regularly and run your stuff. The Worker is designed as single threaded application, which means your method will block the entire cron job architecture. Don't do anything which shall not be run synchronously.

3.4 Coding standards

Parts of this document are copied from netconf's coding standards.

3.4.1 Python

- Python code adheres to PEP-8 coding guidelines.
- Write all code for Python 2.5; thus, all features up until and including Python 2.5 may be used.
- If the choice is between a Python 2.5 way of implementing something, and a pre-2.5 way, the former should be taken.
- Existing pre-2.5 constructs which have been deprecated by Python 2.5 must be reimplemented accordingly.
- Existing pre-2.5 constructs which can merely be expressed more concisely with Python 2.5 can be migrated, and probably should be.
- Use the `docs/py.template` file as a start to all Python files. Alter author and copyright information if needed.
- Use Python unicode strings – `u'foo'` instead of `'foo'`.

- Private Python functions' names should start with an underscore.

3.4.2 General

- Use UTF-8 everywhere.
- Limit line width to 100 characters.
- Everything is in English (including comments, variable names, etc.)
- All text that will be shown to users **must** be localized using the Pylons framework
- Create tests using the Pylons framework for all functions or features that it is feasible for
- When showing potentially long lists of things use the *paginate* module.
- Use the *logging* module to log activity, and use the three severity levels.

3.4.3 Directory structure

- Functions that deal with the database models should go into `model/`.
- Functions that provide general functionality go into `lib/helpers/`.

3.4.4 Mako Templates

- Start all templates with `# -*- coding: utf-8 -*-` on the first line.
- Use correct, validated, XHTML.
- Try to use the webhelpers where possible.
- Indent XHTML properly – 4 spaces per level, as in the Python code.

3.5 Contributing

Help is always welcome in debexpo. There are also a number of ways to contribute:

- Fixing bugs
- Testing the software and filing bugs
- Filing wishlist bugs
- Implementing new features
- Writing new plugins

3.5.1 Getting the source

You can clone the Git repository using:

```
git clone git://git.debian.org/debexpo/debexpo.git
```

3.5.2 How we use branches

Contributions to Debexpo should be based on the "master" branch. This is the default when you clone the repository.

We recommend rebasing your work so that it is based on the latest "origin/master" just before you submit the changes for review.

We also use the branch name *live* to indicate what is running on the main site.

3.5.3 Where to send patches

You should send patches or any other feedback and information to the debexpo-devel mailing list.

We also welcome Git branches.

3.6 API documentation

3.6.1 debexpo.lib.changes

3.6.2 debexpo.lib.email

3.6.3 debexpo.lib.plugins

3.6.4 debexpo.lib.repository

3.6.5 debexpo.lib.utils

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index