dput Documentation

Release 1.0.0

Paul Tagliamonte

August 18, 2015

Contents

dput-ng is a brand-new retake of the classic Debian tool, dput. We've made some important changes, which are documented here. Please get acquainted with the documentation, in order to fully understand the changes.

The Migration guide from old-style dput might be helpful for new users.

CHAPTER 1

Motivation

Many have asked “why rewrite dput”, or “why not work with dput”?

Frankly, when it comes down to it, we were concerned with the bitrot that is present in old dput, and decided to spend our time designing dput-ng to support all of our ideas from the ground up, rather then further mangling old dput’s codebase to support them.

As far as what features, the biggest improvements in our mind are:

- Enhanced and configurable pre-upload checks baked in and enabled by default to make sure you do not accidentally upload a package which is not suitable for archives
- Support for external third party checkers
- Fragmented configuration, to allow external packages to include checks.
- Real SFTP support
- Dynamic checker behavior depending on host / profile
- Support for all checksums, Debian Changes files support (MD5, SHA1, SHA256)
- Full dcut support, including Debian Maintainer permission handling

We’re both really big fans of dput, so we’ve decided to maintain 100% compatibility with dput in dput-ng, as well as automagically reading from the old-style dput.cf conf files.

You might see some behavior change, but we believe it to be in the spirit of the original incarnation of dput. All the new features and functionality is fully disable-able, and you should be able to use dput-ng just like you were before.

dput-ng also features a lot of new features that might be of interest to Debian derivatives, such as the ability to add a new upload target (now called profiles) and unique checks, without having to fork dput. Changes which make extending dput downstream will likely be accepted in dput main. Please consider contributing.

CHAPTER 2

Documentation Index

Contents:

2.1 Reference Documentation

2.1.1 Documentation Index

Contents:

Migration guide from old-style dput

Welcome! This is a helpful starting guide for anyone looking to switch from dput to dput-ng. dput-ng features a few interesting changes, so it's worthwhile to run through this helpful starting guide.

Key points

- dput's configuration files *are* supported, and *will* override any new-style configuration file.
- Behavior of pre-upload checks *may* be different.
- dput-ng maintains backwards compatibility with the old dput's command line flags.
- dcut has a totally revamped interface, but is similar in spirit and usability of dput's dcut interface.
- This package *replaces* old style dput.

Big changes from dput

- Configuration can be defined in JSON. Configuration File Overview may be of some help.
- More and better behaved checks are enabled by default, and more are ready for use out of the box, if you so wish.
- post-upload hook and pre-upload checks (or hooks) may be written in Python, and have access to the objects which matter. For more on writing one, Writing Hooks may provide some insight.

Stability Notes

This *is not* finished. There are bits to be done, but this shows a decent amount of progress being made on the tool, and is mostly ready for limited use by technical people.

- Bug reports are extremely welcome.
- Ideas are extremely welcome.
- Contributors are extremely welcome – of all kinds (technical or otherwise) (see Contributing to dput)

Configuration File Overview

There are a few changes between dput and dput-ng's handling of configuration files. The changes can be a bit overwhelming, but stick to what's in here and it should all make great sense.

High Level Changes

Firstly, you should know dput-ng fully supports the old dput.cf style configuration file. However, it also defines its own own, JSON encoded. Settings which are specific to dput-ng, in particular hooks and profiles can only be defined in dput-ng's configuration style. It is possible to run dput-ng with old-style configuration files only, with new-style configuration files only and even with shared profiles, where both new-style and old-style dput configuration files partially define behavior of a stanza.

By default, dput-ng will look for configuration files in one of three places: `/usr/share/dput-ng/`, `/etc/dput.d/` and `~/.dput.d/`. Files in each location are additive, except in the case of a key conflict, in which case, the key is overridden by the next file. The idea here is that packages must ship defaults in `/usr/share/dput-ng`. If the system admin wishes to override the defaults on a per-host basis, the file may be overridden in `/etc/dput.d`. If a user wishes to override either of the decisions above, they may modify it in the local `~/.dput.d` directory.

Defaults (e.g. the old [DEFAULT] section) are shared (new-style location is in `profiles/DEFAULT.json`), so changing default behavior should affect the target, regardless of how it's defined. In the case of two defaults conflicting, the new-style configuration is chosen.

Order of Loading

If all possible files and directories exist, this is order of loading of files:

1. /usr/share/dput-ng/ (new-style default profiles)
2. /etc/dput.d (new-style site-wide profiles),
3. /etc/dput.cf (old-style site-wide profiles)
4. ~/.dput.d (new-style local profiles)
5. ~/.dput.cf (old-style local profiles)
6. Any file supplied via command line.

To remove a profile entirely, see operator handling below.

Theory

New-style config files have two core attributes – `class` and `name`. For a upload target, that's known as a `profile`. Technically speaking, any config file is located in `${CONFIG_DIR}/class/name.json`.

Keys can also be prefixed with one of three "operators". Operators tell dput-ng to preform an operation on the data structure when merging the layers together.

Addition:

```
# global configuration block
{
    "foo": [
        'one',
        'two'
    ]
}

# local configuration block
{
    "+foo": [
        'three'
    ]
}

# resulting data structure:
{
    "foo": [
        'one',
        'two',
        'three'
    ]
}
```

Subtraction:

```
# global configuration block
{
    "foo": [
        'one',
        'two',
        'three'
    ]
}

# local configuration block
{
    "-foo": [
        'three'
    ]
}

# resulting data structure:
{
    "foo": [
        'one',
        'two'
    ]
}
```

Assignment:

```
# It should be noted that this *IS* the same as not prefixing the block
# by an "=" operator. Please don't use this? Kay? It just uses up cycles
# and is only here to be a logical extension of the last two.

# global configuration block
{
    "foo": [
        'one',
        'two',
        'three'
    ]
}

# local configuration block
{
    "=foo": [
        'three'
    ]
}

# resulting data structure:
{
    "foo": [
        'three'
    ]
}
```

Meta

The most complex part of these files is the "meta" target. Internally, this will fetch the config file from the `metas` class with the name provided in the config's `meta` attribute. The resulting object is placed under the config.

Meta configs can declare another meta config, but will not work if it's self-referencing. Don't do that.

Practice

OK, let's look at some real config files.

I've implemented PPAs as a pure-JSON upload target. This file lives in profiles/ppa.json. It looks something like:

```
{
    "meta": "ubuntu",
    "fqdn": "ppa.launchpad.net",
    "incoming": "~%(ppa)s",
    "login": "anonymous",
    "method": "ftp"
}
```

You'll notice the old-style substring replacement is the same. While looking a bit deeper, you'll also notice that we inherit from the Ubuntu meta-class.

Overriding default hook behavior

It's idiomatic to just *extend* what you get from your parent (e.g. use the prefix operators + or −, so that you don't have to duplicate the same list over and over.

Contributing to dput

Firstly, thanks for reading! It's super cool of you to want to help!

Code + patches

This is one of the bigger areas in which hands are needed. Adding new features and refactoring the codebase is a lot of work, and the more people who want to help with such tasks, the better!

dput-ng is extremely pythonic, and it aims to be something enjoyable to hack on. We aim to be, at any time, pep8 clean, pyflakes clean, well-tested and fully documented.

If you decide to contribute, please stick to the following rules:

- Use names that make sense. In general, try to make the import as descriptive as you can. `from dput.foo import bar_function` is pretty lame, try something like `from dput.profile import load_profile`.
- When you contribute a fix, please also contribute some tests to verify what you've done. That's fine if you don't use TDD, in fact, most of us here at dput-ng HQ don't.
- docstring **all the things**. Use RST for the docstring blocks, there's a good chance it'll show up in the docs.
- Please be explicit about licensing.
- Please add your name to AUTHORS, on the first commit.
- Ask for feedback *early*

Documentation

Documentation is another huge effort that's been going on. Working to better document dput is something that's really important. Working on tutorials, reviewing old & outdated docs, or expanding on existing documentation is something that's sorely needed.

If you're also technical, documenting the internals is an ongoing effort, so any help there would be amazing.

Some rules here, too:

- Be sure to write in complete and clear English.
- Cross-reference as much as you can. It really helps.
- Include lots of examples.
- As for feedback as you go along. Also be sure to have a technical person on the dput team review your work for slight errors as you go along.
- Be explicit about licensing
- Please add yourself to AUTHORS on your first commit.

Hooks

Hooks are hugely important as well. Writing new hooks is insanely cool, and sharing them back with the dput-ng community & friends is an awesome thing to do on it's own.

Some other random guidelines we thought up:

- In general, treat your hook as self-contained and independent.
- If you feel your checker hook be in the dput main, please ensure it's properly clean, follows the code guidelines above, and finds a nice home somewhere in the dput codebase. Make sure it's below `dput.hooks`, though.
- It must be distributable under the terms of the GPL-2+ license. Permissive licenses such as Expat or BSD-3 should be fine. When in doubt, ask!

Installing Hooks

This guide will cover exactly how dput-ng handles hooks, and the proper way to install, distribute and tool with hooks. Remember, Hooks are your friend!

So, what exactly are hooks?

Well, it's pretty simple, actually – a hook is a Python importable function that takes a few arguments, which are populated with internal dput-ng objects. These objects contain such things as a way to talk to the user, the processed .changes file, and the upload target.

Hooks can run either before or after an upload, and hooks that run before the upload may halt an upload by raising a `dput.exceptions.HookException` (or a subclass of that).

Alright, you mentioned Python-importable, what exactly does that mean?

The path given is a fully qualified path to the hook. Here's an example:

```
>>> from os.path import abspath
```

The dput-ng style "fully qualified path" to that function (abspath) would be:

```
os.path.abspath
```

It's really that simple.

Note: It's also worth noting dput-ng adds a few directories to sys.path to aid with debugging and distributing trivial scripts. For each directory in `dput.core.CONFIG_LOCATIONS`, that directory plus "scripts" will be added to the system path, so (commonly) `~/.dput.d/scripts` and `/etc/dput.d/scripts` are valid Python path roots to dput-ng.

OK, let's do an example.

Let's do a simple checker – one that fails out if Arno is the maintainer:

```
def check_for_arno(changes, profile, interface):
    """
    The ``arno`` checker will explode in a firey mess if
    Arno tries to upload anything to the archive.
```

```
    This checker doesn't change it's behavior given any Profile codes.
    """
    maintainer = changes['Maintainer']
    if "arno@debian.org" in maintainer:
        raise HookException("Arno's not allowed to Upload.")
```

I've saved this file to `~/.dput.d/scripts/arno.py`. It should be noted that dput-ng can now import this file as `arno`, and the command (from inside dput-ng) `from arno import check_for_arno` will work.

Since we need to tell dput-ng about this hook, we need to drop it's def into a dput hook directory. Let's use our home directory again, even though it should be noted both `/usr/share/dput-ng/` and `/etc/dput.d/` will work as well.

I've placed `arno.json` into `~/.dput.d/hooks/arno.json`:

```
{
    "description": "Blow up if Arno's maintaining this package.",
    "path": "arno.check_for_arno",
    "pre": true
}
```

The `pre` key, or the `post` key must be present and set to a boolean. If no key is given, it assumes it's a `pre` checker. The `path` is the Python-importable path to the hook function, and `description` is for humans looking to get some information on the hook.

We can make sure it works using `dirt(1)`:

```
$ dirt info --hook arno

    The ``arno`` checker will explode in a firey mess if Arno tries to upload
    anything to the archive.

    This checker doesn't change it's behavior given any Profile codes.
```

Remember, this pulls from the docstring, so please leave docstrings!

OK. Now that dput-ng is aware of the plugin, we can add it to a profile by adding a "plus-key" to your profile choice. Let's add this to `ftp-master`, since we want to make sure Arno never uploads there.

Here's my (user-local) ftp-master config `~/.dput.d/profiles/ftp-master.json`:

```
{
    "+hooks": [
        "arno"
    ]
}
```

If you want to learn more about why this syntax works, I'd suggest checking out the Configuration File Overview documentation.

So, let's try uploading:

```
$ dput [...]
[...]
running check-debs: makes sure the upload contains a binary package
running checksum: verify checksums before uploading
running suite-mismatch: check the target distribution for common errors
running arno: Blow up if Arno's maintaining this package.
Arno's not allowed to Upload.
$ echo $?
1
```

Nice!

Writing Hooks

Note: Whether a hook runs before or after uploading a package is a matter of the JSON configuration file. Aside, they are identical.

Hooks are a fundamental part of dput-ng. Hooks make sure the package you've prepared is actually fit to upload given the target & current profile.

In general, one should implement hooks for things that the remote server would ideally check for before accepting a package. Going beyond that is OK, providing you have the user's go-ahead to do so.

Remember, this isn't some sort of magical restriction to upload, most remote servers would be happy with almost anything you put there, these are simply to help reduce the time to notice big errors.

Theory of Operation

Pre-upload Hooks are a simple function which is invoked with a few objects to help aid in the checking process & reduce code.

Pre-upload hooks will always be run before an upload, and will be given the digested .changes object, the current profile & a way to interface with the user.

Pre-upload hooks (at their core) should preform a single check (as simply as it can), and either raise a subclass of `dput.exceptions.HookException` or return normally.

Post-upload hooks work likewise. They are just simple hooks as well, that are slightly different to pre-upload hooks. Firstly, register as a hook by placing the plugin def in the `hooks` class. In the event of an error, feel free to just bail out. There's not much you can do, and throwing an error is bad form. For now. This is likely to change.

How a Hook Is Invoked

Throughout this overview, we'll be looking at the `dput.hooks.checksum.validate_checksums()` pre-upload hook. It's one of the most simple hooks, and demonstrates the concept very clearly.

To start to understand how this all works, let's take a step back and look at how `dput.hook.run_hook()` invokes the hook-function.

Basically, `run_hook` will grab all the strings in the `hooks` key of the profile. They are just that – simply strings. The hook are looked up using `dput.util.get_obj()` (which calls `dput.util.load_config()` to resolve the .json definition of the hook).

All hooks are declared in the `hooks` config class, and look something like the following:

```
{
    "description": "checksum pre-upload hook",
    "path": "dput.hooks.checksum.validate_checksums",
    "pre": true
}
```

Note: Use `"pre": true` or `"post": true` respectively to decide whether the hook should run prior or after uploading a package.

For more on this file & how it's used, check the other ref-doc on config files: Configuration File Overview

Nextly, let's take a look at the `path` key. `path` is a python-importable path to the function to invoke. Let's take a look at it a bit more closely:

```
>>> from dput.hooks.checksum import validate_checksums
>>> validate_checksums
<function validate_checksums at 0x7f9be15e1e60>
```

As you can see, we've imported the target, and it is, in fact, the function that we care about.

Now that we're clear on how we got here, let's check back with the implementation of `dput.hooks.checksum.validate_checksums()`:

```
def validate_checksums(changes, profile, interface):
```

We're passed three objects – the `changes`, `profile` and `interface`. The `changes` object is an instance of `dput.changes.Changes`, pre-loaded with the target of this upload action. `profile` is a simple dict, with the current upload profile. `interface` is a subclass of `dput.interface.AbstractInterface`, ready to be used to talk to the user, if something comes up.

What To Do When You Find an Issue

During runtime, and for any reason the checker sees fit to do so, the hook may abort the upload by raising a subclass of a `dput.exceptions.HookException`. In cases where the user aught to make the decision (lintian errors, etc), please **prompt** the user for what to do, rather then blindly raising the error. Remember, the user can't override a checker's failure except by disabling the checker. Moreover, never prompt for inputs directly. Use the `dput.interface.AbstractInterface` interface to prompt for data in a uniform way.

Don't make people disable you. Be nice.

Let's take a look at our reference implementation again:

```
def validate_checksums(changes, profile, interface):
    try:
        changes.validate_checksums(check_hash=profile["hash"])
    except ChangesFileException as e:
        raise HashValidationError(
            "Bad checksums on %s: %s" % (changes.get_filename(), e)
        )
```

As you can see, the checker verifies the hashsums, catches any Exceptions thrown by the code it uses, and raises sane error text. The Exception raised (`dput.hooks.checksum.HashValidationError`) is a subclass of the expected `dput.exceptions.HookException`.

2.2 Internal Documentation

The documentation here is to document the internal implementation of dput, for use by new contributors, old contributors, bus-factor reasons, checker hackers, and interested persons.

2.2.1 Documentation Index

Top-level modules:

Object Core

The *core* is a place where all the central objects live. This is used so that all the different modules in dput can access common constants. This helps us fake data (great for testing), and maintain sanity.

The Logger

Printing to the screen using `print()` is wrong, m'kay? Please do **not** use it under any conditions. In it's place, we have a central `logger` object, to use as all the bits of dput see fit.

The logger object is an instantiation of `dput.logger.DputLogger`, so feel free to use any if it's logging methods. In general, don't use `info` or above, unless the user *really* needs to know. Most calls should be to `debug` or `trace`.

Example usage:

```
from dput.core import logger
logger.debug("Hello, World!")
logger.warning("OH MY DEAR GOD")
```

Configuration Objects

The core contains two config directories, which are used by the config modules (as well as other, more friendly places).

All configs are in the form of a dict, the key being the path, and the value being the "weight" of the path. The higher the weight, the less important it is.

Example `dput.core.CONFIG_LOCATIONS`:

```
{
    "/usr/share/dput-ng/": 30,
    "/etc/dput.d/": 20,
    os.path.expanduser("~/.dput.d"): 10,
}
```

`dput.util.load_config()` is used to access a config from this list, and handles meta-classes, and other edge cases when loading. Please use `dput.util.load_config()` to load config files from these locations.

Example `dput.core.DPUT_CONFIG_LOCATIONS`:

```
{
    "/etc/dput.cf": 15,
    os.path.expanduser("~/.dput.cf"): 5
}
```

Both are merged into a single list, sorted by list, and used by `dput.profile.MultiConfig` to handle loading and access.

Schema Directory

This is the path to search for validictory schemas. By default, this is set to `/usr/share/dput-ng/schemas`. These are not treated as normal conf-files.

Misc. Utilities

This module contains functions that don't have a rightful home elsewhere, or are of general use.

Configuration Access

These functions are used to read a config off the filesystem, for use elsewhere in dput.

Object Loaders

These functions aid in loading a defined, dynamically imported "plugin".

Invocation

These functions aid in running things.

Changes File Implementation

This module contains code to aid in processing .changes files. Most of this code has been yanked from Jonny Lamb. Thanks, Jonny.

Facade

Abstraction

Upload Target / Profile Implementation

This contains a lot of backing code to get at profiles.

Commonly used functions

Multi Configuration Implementation

> **Warning:** This is mostly just used internally, please don't use this directly unless you know what you're doing(tm). In most cases, `dput.profile.load_profile()` and `dput.profile.profiles()` will do the trick.

Exceptions & Errors

This module contains oodles of exceptions that might be thrown or subclassed elsewhere.

Base Exceptions

Configuration File Errors

Misc Errors

Override Implementation

Hook Implementation

Hook Implementations

Documentation Index

Contents:

Lintian Checker Implementation

Archive

Checksums

Debian file routines

Distribution bits

GPG

Impatient

User Interface Implementation

User Interfaces Implementations

Documentation Index

Contents:

CLInterface Implementation

Uploader Implementation

Upload Method Implementations

Documentation Index

Contents:

FTP Implementation

HTTP Implementation

Local Uploader Implementation

SCP Implementation

SFTP Implementation

Configuration Implementation

Configuration File Implementations

Documentation Index

Contents:

Old-style dput config files

New-style dput config files

CHAPTER 3

Authors

The bulk of the work was done by Arno and Paul For a full list of contributors, please check the AUTHORS file shipped with your copy of dput-ng.