PKGBUILDer 3.0

Chris *Kwpolska* Warrick

2013-03-19

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

Code listings were (more or less) heavily modified before inclusion. Comments that are only in this document are marked with # (as opposed to #).

This document plans many improvements. Thus, I decided to name the new version 3.0, because 2.2 doesn't feel right given the scale of those improvements.

Contents

Part I
Exceptions

Chapter 1

The current system

It is ugly. There is one exception: PBError. It takes messages. You know, *text*. To display for *humans*, not *machines*. For example, like this:

Listing 1 Current error messages.

```
raise PBError(_('AUR: HTTP Error {}').format(req.status_code))
raise PBError(_('download: 0 bytes downloaded')
```

That's uninformative. What does the error code mean, exactly? Not everybody has the HTTP status codes memorized (and *nobody* memorizes the more obscure ones, which shouldn't appear in PKGBUILDer at all[1]). Also, what does the AUR part mean, exactly? The place *in the code* where this message was produced. In our case, it is `pkgbuilder.aur.AUR().jsonreq()` and `pkgbuilder.build.Build().download()`.

§1.1 But wait, there's more!

The exceptions output are currently handled in *three* places:

a) `main.main()` (see Listing 2);

b) `build.Build().auto_build()` (Listing 4 on the next page; this is the ugliest code in PKGBUILDer);

c) `build.Build().build_runner()` (Listing 3 on the following page).

Listing 2 `main.main()`

```
def main(source='AUTO', quit=True):
    """Main routine of PKGBUILDer."""
    try:
        # 200 (yes, exactly 200!) lines of logic
    except requests.exceptions.ConnectionError as inst:
        DS.fancy_error(str(inst))
        # TRANSLATORS: do not translate the word 'requests'.
        DS.fancy_error(_('PKGBUILDer (or the requests library) had '
                         'problems with fulfilling an HTTP request.'))
        exit(1)
        # snip the exact same thing thrice, only with different exceptions
    except PBError as inst:
        DS.fancy_error(str(inst))
        exit(1)

    DS.log.info('Quitting.')  # A very lonely line.
```

[1] Error codes that are likely to appear and be unhandled in PKGBUILDer: 403, 404, 500, 501, 503. Status codes that are handled by the awesome *Requests* library include 200, 301, 302.

Listing 3 `build.Build().build_runner()`.

```
def build_runner(self, pkgname, performdepcheck=True,
                 pkginstall=True):
    # docstring goes here
    try:
        # snip 79 lines of logic
            if aurbuild != []:
                return [72337, aurbuild]
        # snip 43 lines
    except PBError as inst:
        DS.fancy_error(str(inst))
        return [72789, None]
    except IOError as inst:
        DS.fancy_error(str(inst))
        return [72101, None]
```

Listing 4 `build.Build().auto_build()`, the ugliest code in PKGBUILDer.

```
def auto_build(self, pkgname, performdepcheck=True, pkginstall=True):
    # docstring goes here
    build_result = self.build_runner(pkgname, performdepcheck, pkginstall)
    os.chdir('../')
    try:
        if build_result[0] == 0:
            DS.fancy_msg(_('The build function reported a proper build.'))
        elif build_result[0] >= 0 and build_result[0] < 72000:  # PBxxx.
            raise PBError(_('makepkg (or someone else) failed and '
                            'returned {}.').format(build_result[0]))
            exit(build_result[0])
        elif build_result[0] == 72789:  # PBSUX.
            raise PBError(_('PKGBUILDer had a problem.'))
            exit(1)
        elif build_result[0] == 72101:  # I/O error.
            raise PBError(_('There was an input/output error.'))
            exit(1)
        elif build_result[0] == 72337:  # PBDEP.
            # insert magic and recurrency here

        return build_result
    except PBError as inst:
        DS.fancy_error(str(inst))
```

§1.2 Recap: why is it so *evil*?

The main problems are:

1. *One* exception class that has only a *human–only* (or even *Chris–only!*) message;
2. Weird return codes (`build.Build().auto_build()` and his friend `build.Build().build_runner()`);
3. Repetitiveness and general ugliness;
4. If anyone uses PKGBUILDer as a library in his code (eg. aurqt, which hadn't had any problems *yet* and to which this part the document applies), they hate my `PBErrors`.

Chapter 2

Exceptions 2.0

§2.1 How to fix it?

Well, just the reverse of the list in section 1.2 on the preceding page and get:

1. One base class (which, in order to break backwards compatibility for various reasons, *won't be named* `PBError`), multiple subclasses with appropriate class members;
2. Replace the return codes with a true `try`…`except`;
3. Make it look pretty and drop all the repeats.

Easy, wasn't it? Even better, it is not too hard to fix it. It requires time and thinking.

§2.2 Proposed subclasses

Keep in mind that this isn't the finished list, and it might be expanded. Also, as a general rule, we hate people doing `from` imports and make them `import pkgbuilder.exceptions`.

1. Base class: **PBException**
   (a) AurError:
      - `r.results` if `r.type == 'error'`.

   (b) MakepkgError:
      - Carry non-zero return codes from `makepkg`;
      - Handled rather quickly, not to break anything.

   (c) NetworkError:
      - Error codes listed in footnote 1 on page 7;
      - `requests.exceptions`.

   (d) PackageError:
      - When we can't find a package or something regarding one is broken, somehow.

   (e) SanityError:
      - When stuff goes apeshit.
2. IOError — handled in `main.main()` (now there are `build.Build()` handlers!).

§2.2.1 But will those subclasses be?

I plan to put them in a new module, named (obviously) `exceptions`. It will contain all the exceptions listed above, and anyone who needs them will import them. This practice is inspired by *Requests* by Kenneth Reitz.

Part II
Improved OOP

Chapter 3

What and why

Our current OOP is bad. I plan to create a `Package` class, containing the obvious things, in an even nicer format. Bonus points for handling AUR output as `pkg.__dict__.update()`. Other classes, on the contrary, don't make sense.

More details will be available soon.

Part III

Force `--safeupgrade` for pkgbuilder

Chapter 4

Rationale

When pacman developers release a new version, you are asked to install it before any other upgrades. PKGBUILDer should do the same, but with one major change: using the `--safeupgrade` option, introduced in commit `0f91814e51` (merged in `72dda04c25`) and shipped with 2.1.6.2.

§4.1 How to pull it off?

Steal the question from pacman localization files and ask it when we find a PKGBUILDer upgrade. When the user agrees, we need to run the `--safeupgrade` routine, which currently sits in `main.main()`. It should be moved to `upgrade.pb_failsafe()` or something like that.

Part IV

`cower -d` implementation

Chapter 5

What does it do?

That's probably the easiest improvement: add an option to run `build.Build.build_runner()`, stopping right before `os.chdir('./{}/'.format(pkg['Name']))`. Also, we will split the first few package determination lines to another function while we are at it.

§5.1 Implementation

There is one major problem: `-dDw` are already used. We would need to find a better abbreviation.