tool Documentation

Release 0.8.0

Andy Mikhaylenko

August 19, 2016

Contents

Tool is a lightweight framework for building modular configurable applications. It is intended to:

- be as unobtrusive as possible. No conventions except for APIs. The modules are structured following their own logic;
- store configuration in ordinary Python structures, impose no limits on them and support YAML input out of the box;
- encourage modularity by providing a simple but flexible layered API for extensions with support for dependencies;
- stick to existing standards and APIs;
- combine existing components: Argh (argparse) for commands, PyDispatcher for signals;
- let user choose non-critical components but ship with batteries included: Werkzeug for request handling and routing (yes, even this is pluggable!), Jinja and Mako for templates, Doqu for modeling and more specialized extensions, repoze.who for authentication and so on;
- keep even some core functionality (such as routing and serving) as plugins so the framework can be used for CLI, web or CLI+web purposes without adding weight when it is not needed; moreover, the user can swap almost every component without breaking other components.

Contents:

CHAPTER 1

Tutorial

1.1 The simple echo script

This example shows how to create a simple echo application with and without Tool. First off, let's design the command-line interface:

```
$ ./app.py echo "hello there!"
hello there!
```

That's what we need. THe interface is extremely simple.

1.1.1 The naïve approach

The straightforward implementation (with barebones Python) only takes three lines of code:

```
import sys

if __name__=='__main__':
    return u'You said {0}'.format(sys.argv[1])
```

It works!

However, the naïve approach is not scalable. When you have more than one command and more than one argument, the code dealing with `sys.argv` becomes bloated, overcomplicated and unmaintainable. You'll need many *if/elif/.../else* branches, you'll need to provide helpful error messaes and documentation. So we need a parser.

1.1.2 Using parser

...This is why getopt was introduced in early versions of Python and later replaced by more powerful optparse which was in turn recently replaced with argparse. We'll use the latter:

```
import argparse
import sys

parser = argparse.ArgumentParser()
parser.add_argument('text')

if __name__=='__main__':
    args = parser.parse_args(sys.argv[1:])
    return u'You said {0}'.format(args.text)
```

This approach has several issues:

- the whole application is a single command; you cannot just plug in a function as a subcommand with its own arguments.
- imperative approach to *defining* arguments makes it hard to separate them from the dispatcher; therefore the application cannot be truly modular.

1.1.3 Using parser with subcommands

We should at least try to solve the subcommands problem. *Argparse*, unlike *getopt* and *optparse*, directly supports the concept of subcommands. It creates a subparser for each command so there can be a tree of nested commands, e.g.:

```
$ ./app.py blog add "hello"
added #1
$ ./app.py blog ls
* hello
```

This is basically a namespace "blog" with two functions acting as subcommands with each accepting its own set of arguments. A possible implementation:

```
import argparse

def blog_list(args):
    ... do something ...

def blog_add(args):
    id = add_to_database(args.text)
    print 'added #'+id

parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers()

blog_parser = subparsers.add_parser('blog')
blog_list_parser = blog_parser.add_parser('ls')
blog_list_parser.set_defaults(function=blog_list)
blog_add_parser = blog_parser.add_parser('add')
blog_add_parser.set_defaults(function=blog_add)
blog_add_parser.add_argument('text')

if __name__=='__main__':
    args = parser.parse_args()
    print args.function(args)
```

The problems:

- still imperative;
- a lot of boilerplate code related to commands.

1.1.4 Using parser: a cleaner way

Fortunately, there's an excellent wrapper for *argparse* named argh. It enables lazy declaration of commands and removes a lot of boilerplate code while still allowing you to access the *ArgumentParser*:

```
from argh import *

def blog_list(args):
    ... do something ...

@arg('text')
def blog_add(args):
    id = add_to_database(args.text)
    print 'added #'+id

# ...not necessarily in the same module:

parser = ArghParser()
parser.add_commands([blog_list, blog_add], namespace='blog')

if __name__=='__main__':
    parser.dispatch()
```

The difference is huge: the script now uses declarative approach and therefore command declarations can be safely decoupled from the dispatcher.

Now it would be great to have a means to assemble the commands in a uniform way.

1.1.5 The Tool application

A Tool application is an `tool.Application` managed by a script. Let's go straight to an example:

```
from tool import Application

app = Application()

if __name__=='__main__':
    app.dispatch()
```

The method `tool.application.Application.dispatch()` does the same as `parser.dispatch()` in the previous section. the application object contains an *ArghParser* instance as `app.cli_parser`. So you can add and run the commands this way:

```
app = Application()
app.cli_parser.add_commands([blog_list, blod_add])
if __name__=='__main__':
    app.dispatch()
```

But wait, why do we need this new abstraction level if it doesn't do anything what the argument parser itself can do? Well, it does. It is *extensible*. You can configure the application object so that it loads certain extensions and *they* contribute commands.

The configuration is just a dictionary with optional nested structures. We will use YAML as it is much more readable than Python in terms of defining data structures:

```
extensions:
    blog.setup: null
```

Save this to *conf.yaml* and let the application know about the configuration:

```
app = Application('conf.yaml')
```

This is equivalent to:

```
app = Application({'extensions': {'blog.setup': None}})
```

Now the application will load the module `blog`, find a function `setup` in it and run it with two arguments: `app` (the application object itself) and `conf` (the *extension settings*, in our case they are just set to *None*).

Let's now write the `blog` module:

```
def blog_list(args):
    ...do somthing...

def setup(app, conf):
    app.cli_parser.add_commands([blog_list])
```

Then try running your management script:

```
$ ./app.py
```

...and you will see usage information with `blog-list` command in it! Now run the command:

```
$ ./app.py blog-list
```

The function `blog_list` has been called and the result printed. Easy!

So to write pluggable applications with Tool you need to simply add a function that accepts the application object and then deal with its API. The function will be called automatically if it is mentioned in the *extensions* section of the configuration. And you can configure the extension itself by providing some data instead of *null* (or *None*), e.g.:

```
extensions:
    blog.setup:
        theme: green
```

...and the setup function will be:

```
def setup(app, conf):
    assert 'theme' in conf, 'You must specify the theme!'
    return conf
```

The returned value (the environment) will be stored in the application object and can be later accessed this way:

```
from tool import app

env = app.get_extension('blog.setup')
theme = env['theme']
```

Why not just `get_extension('blog')`? Because a single extension can provide multiple configuration functions (or even classes) for different usage patterns. If you want a single name for all such functions if your extension or even across multiple extensions (to make them swappable) you can use the "feature" concept:

```
def setup_cli(app):
    pass
setup_cli.features = 'blog'

def setup_web(app):
    pass
setup_web.features = 'blog'
```

Or with some syntax sugar:

```
from tool.plugins import features

@features('blog')
def setup_cli(app):
    pass
```

```
@features('blog')
def setup_web(app):
    pass
```

Note that *setup_cli* and *setup_web* are mutually exclusive as they implement the same feature. A third, mixed CLI/web function can be introduced to offer both interfaces. Of course the web interface will have more dependencies that the CLI one, so you can make sure that they are also configured:

```
from tool.plugins import requires

@requires('sqlobject.setup', 'werkzeug_routing.setup')
def setup_web(app):
    pass
```

The ORM and routing extensions also can be swappable (e.g. Autumn, SQLAlchemy or Storm may be used instead of SQLObject), so it is safer to reference them by feature name:

```
@requires('{orm}', '{routing}')
def setup_web(app):
    pass
```

The application will gather feature names from all configured extensions and resolve them to actual paths to the configuring functions.

Note: The same API cannot be guaranteed across extensions that implement the same feature. This is a problem yet to be resolved so the "feature" concept may be changed in the future.

1.1.6 Local settings

Often you need to supply default project settings along with the code but have certain values overloaded on your development box and keep passwords in a separate config on the production machine. The common solution is having two copies of settings: default and local. But how do we do it?

The simplest way would be to update the dictionary with another one from a module which is excluded from versioning:

```
from local_conf import extra_settings

settings = {'foo': 123}
settings.update(extra_settings)

app = Application(settings)
```

This however does not work because the settings dictionary is multi-level. It is usually desired to *merge* the dictionaries instead of replacing the whole first-level branch. Tool ships with a convenience function for proper merging:

```
from tool.conf import merge_dicts

settings = merge_dicts(settings, extra_settings)
app = Application(settings)
```

Or as simple as this:

```
app = Application(settings, extra_settings)
```

Of course both "base" and "local" settings can be not only *dict* instances but also strings, so this will work too:

```
app = Application('conf.yaml', 'local.yaml')
```

1.2 Blog

Having understood the basics, let’s try something practical.

A blog requires some database and a means to expose the records via web interface.

Note: TODO.

Glossary

TODO

http://docs.repoze.org/bfg/1.2/glossary.html#term-application-registry (as inspiration)

Application a WSGI application

Application manager Manager for the *application*. Provides *routing* and supports *middleware*. Implemented by `tool.application.ApplicationManager`.

CLI Command Line Interface. Implemented by `tool.cli`.

Middleware ordinary WSGI middleware. It doesn't matter whether you wrap the *application* into the middleware using the *application manager* or not. However, introspection is much easier if you use the manager.

Routing the process of finding the right callable for given request. The callable is then returned by the *application*. Tool uses Werkzeug for routing.

CHAPTER 3

Questions

Feel free to ask on the mailing list.

CHAPTER 4

Similar projects

- Cement:
  - has a more complex and much more restrictive API than that of *Tool*. In fact, *Tool* also has a complex API for extensions but it is optional.
  - is based on outdated *optparse* and therefore is bound to implement some features already present in *argparse* (e.g. nested commands and some help-related stuff).
  - depends on *ConfigObj* and stores the settings in ugly *ini files* while *Tool* allows the configuration to be stored in any format (favouring the clean YAML) including simple Python structures.
  - stores the environment is stored in a module-level variable while Tool stores it in extension objects within the application.
  - has the notion of “namespaces” similar to *Tool*‘s “features”.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index