Introducing Middlefield – Custom generic developer tool

https://github.com/elcaminoreal/

https://github.com/moshez/introducing-middlefield – reinventing plugins (gather), cli parsing (caparg) and dependency injection (elcaminoreal)

Moshe Zadka – https://cobordism.com

Pyninsula January 2017

Middlefield is a platform to build developer tools. In every development environment, tooling will be slightly different – so middlefield gets out of the way, letting you build your own.

There is only one built-in command: the command to self-build a Python executable that includes the base middlefield *as well as* other plugins. This executable can be distributed internally using whatever facilities exist and are beloved by local developers: an artifact repository, a homebrew channel, and so on.

Build Pex with plugins to add commands

Python EXecutable – one file with all third-party dependencies!

```
$ pip install middlefield
$ mf echo --what hello
Usage:
    self-build
$ mf self-build --output mf.pex \
  --package middlefield_echo \
  --shebang '/usr/bin/env python3'
$ ./mf.pex echo --what hello
hello
$ upload-to-internal-place mf.pex
```

Middlefield is based on the *Gather* framework, and so all middlefield plugins must be registered with Gather. This is done via `entry_points` in the `setup.py` file.

Example plugin – middlefield echo (setup.py)

```
<snip>
setuptools.setup(
entry_points={"gather":
              ["gather=middlefield_echo"]},
<snip>
```

Middlefield supports *dependencies* and *commands*. In this example, in order to make it easier to explain, we put them in separate files. Middlefield itself is agnostic to code organization issues – code organization should depend, like in this example, on making it easier for other people to read the code.

Example plugin – middlefield echo (deps.py)

```
<snip>
@middlefield.COMMANDS.dependency()
def echo_printer(_deps, _maybe_deps):
    return print
```

This dependency is simple: it does not need any dependencies of its own. In this case, it is also trivial – it does not need to do any work to get the result. However, one advantage that using dependencies gives us is the ability to override them for unit testing. In the echo plugins' unit tests (not shown here) we take advantage of that – thus avoiding the need to patch `sys.stdout`.

Example plugin – middlefield echo (command.py)

```
<snip>
@middlefield.COMMANDS.command(
parser=command('',
    what=option(type=typing.List[str],
                have_default=True)),
dependencies=['echo_printer'])
def echo(args, dependencies):
    dependencies['echo_printer'](*args['what'])
```

The echo command is a function. In the registration decorator, we get the opportunity to specify the expected command line syntax using *caparg*, also known as "Captain Arguments". Its declarative style, and ability to combine sub-commands after they are defined, are useful here.

This is it – other than some housekeeping files, this is all it takes to write a middlefield plugin, and put it in a pex file for easy distribution.

A more realistic, and interesting, plugin, is the one this talk is built with – `moshez_middlefield_beamer`. This plugin, as many future middlefield plugins, is not intended to be useful generically – it is definitely opinionated about the style of presentations. The plugin approach means that it can be used locally by the author, while other development teams can have the subcommand "beamer" do something completely different!

What's next?

- Write a plugin (internal or public)
- Distribute internally
- Talk to me – zadka.moshe@gmail.com – https://cobordism.com

Please send me any feedback, and open issues on any of the related projects.