Taskwarrior Capsules Documentation

Release 0.3

Adam Coddington

March 08, 2015

Contents

Contents:

CHAPTER 1

About

Taskwarrior Capsules allows you to easily extend Taskwarrior functionality by allowing you to add new commands and alter the behavior of existing ones.

1.1 Installation

1. Install from Pip:

   ```
   pip install taskwarrior-capsules
   ```

Please note that you *might* need to run the above command with `sudo`.

2. Install some capsules (read: plugins) and follow their documentation.

Taskwarrior Capsules itself does not offer any meaningful functionality; to use Taskwarrior Capsules, you'll want to install some capsules.

1.2 Using Capsules

Taskwarrior Capsules wraps `task` using a separate command – `tw`, but all commands that are not recognized by Taskwarrior Capsules will be passed-through to Taskwarrior itself verbatim.

To make this clearer: to use Taskwarrior Capsules, rather than listing your tasks with `task`, use:

```
tw
```

And rather than adding a task with `task add Homework due:tomorrow priority:h`:

```
tw add Homework due:tomorrow priority:h
```

And for other Taskwarrior commands, just be sure to type `tw` instead of `task`.

1.3 Finding Capsules

Search for some capsules on github.

CHAPTER 2

Commands

Taskwarrior Capsules provides only one command on its own – `capsules` – but allows you to install 'capsules' providing additional functionality. See *Finding Capsules* for more information about how to find capsules.

2.1 `capsules <subcommand>`

You can use this command to manage your installed capsules; currently only a single subcommand is implemented: `list`.

2.1.1 Subcommands

- `list`: List installed capsules.

CHAPTER 3

Writing your own Capsules

Note: Rather than reading this document, you could perhaps have a look at one of the existing capsules. For a fairly simple example, have a look at the Capsule implementation of Taskwarrior's "context" function: taskwarrior-context-capsule.

Writing your own capsule is easy; all you really need is a single class subclassing `taskwarrior_capsules.capsule.CommandCapsule` and an entry mapping a command name to it in your new capsule's `setup.py`.

3.1 Your Capsule

```
from taskwarrior_capsules.capsule import CommandCapsule

class MyCapsule(CommandCapsule):
    """ A brief description of what your capsule does.

    The first line of this will be displayed next to the capsule's
    name when a user runs ``tw capsules list``.

    """
    # Define the minimum and maximum versions of Taskwarrior-Capsules
    # that this capsule is known to work with; note that Taskwarrior-Capsules
    # follows semver, so you can (hopefully) rely upon breaking changes
    # only occurring with major version bumps.
    MIN_VERSION = '0.3'
    MAX_VERSION = '0.9999.9999'

    # Define the minimum and maximum versions of Taskwarrior that your
    # capsule is known to work with.
    MIN_TASKWARRIOR_VERSION = '2.3'
    MAX_TASKWARRIOR_VERSION = '2.4.9999'

    # Note that if your capsule does not actually interface with
    # taskwarrior at all, you can just set the following property
    # to `False` and forgo setting the above MIN and MAX taskwarrior
    # versions.
    TASKWARRIOR_VERSION_CHECK_NECESSARY = True

    def handle(self, filter_args, extra_args, **kwargs):
        """ Do the work involved when your command is executed directly here.
```

```
        This method will be called with a number of positional
        parameters:

        * 'filter_args': Arguments appearing before the command.
        * 'extra_args': Arguments appearing after the command.

        As well as an indeterminate number of keyword arguments
        including (at the time of this writing):

        * 'command_name': The name of the command currently
          being executed.
        * 'terminal': An instance of ''blessings.Terminal'' for
          the current terminal.  You can use this for formatting
          printed text.

        """
        pass

    def preprocess(self, filter_args, extra_args, **kwargs):
        """ Do the work you'd like to do *before* *any* command is executed.

        This command receives all keyword arguments that ''handle''
        above receives.

        Please note that if you'd like to only run the preprocessor
        for specific commands, in this method you'll need to check
        that ''kwargs['command_name']'' matches the command for which
        you'd like this preprocessor executed.

        Using preprocessors, you are **required** to return a 3-tuple
        of values:

        * 'filter_args': A list of arguments to use for filtering
          when the next command is executed.  If your preprocessor
          does not need to alter 'filter_args', simply return the
          'filter_args' that were passed-in.
        * 'extra_args': A list of arguments to return *following*
          the command name.  If your preprocessor does not need to
          alter 'extra_args', simply return the 'extra_args' that
          were passed-in.
        * 'command_name': The name of the command that should be
          executed.  You can change the command name by returning
          a different command-name than was passed in.  Note that
          'command_name' is is incoming as a keyword argument; you'll
          need to either specify it in your method signature, or
          access it as ''kwargs['command_name']''.

        """
        pass

    def postprocess(self, filter_args, extra_args, **kwargs):
        """ Do the work you'd like to do *after* *any* command is executed.

        Note that this shares most characteristics with the above
        ''preprocess'' method, but receives a single extra keyword
        argument -- ''result'' -- and does **not** need to return
        anything at all.
```

```
        * 'result': The return code returned by taskwarrior
          after the command was executed.

        """
        pass
```

Warning: There are several things only gleaned at above that you should take special care about:

- When writing your capsule class, it is very important that the last argument of your `handle`, `preprocess`, and `postprocess` methods be `**kwargs`; the keyword arguments passed to those methods may change at any time even when releasing a bugfix patch.
- Be conservative when setting `MIN_VERSION`, `MAX_VERSION`, `MIN_TASKWARRIOR_VERSION`, and `MAX_TASKWARRIOR_VERSION`; when a user upgrades his or her version of Taskwarrior or Taskwarrior-Capsules to a newer version than you specify, **they'll still be able to continue using your capsule**, they'll just see a warning message indicating that your capsule is not compatible with the version of Taskwarrior or Taskwarrior Capsules in use. This can be extremely helpful information for users chasing down unexpected behaviors!
  If you absolutely need to prevent users from using a specific version of Taskwarrior or Taskwarrior Capsules, use the `self.get_taskwarrior_version` or `self.get_taskwarrior_capsules_version` methods and raise an instance of `taskwarrior_capsules.exceptions.CapsuleError` with a helpful error message explaining the incompatibility.

3.1.1 Available Methods

All Capsules inherit the following methods:

- `get_taskwarrior_version()`: Returns an instance of `verlib.NormalizedVersion` corresponding with the version of Taskwarrior currently in use.
- `get_taskwarrior_capsules_version()`: Returns an instance of `verlib.NormalizedVersion` corresponding with the version of Taskwarrior Capsules currently in use.
- `get_matching_tasks(filters)`: Returns tasks matching the specified filters; you can pass your `filter_args` directly to this method to return dictionary-like objects representing matching tasks. Each task is an instance of taskw.task.Task.
- `get_tasks_changed_since(datetime)`: Returns tasks that have been changed since the time specified by the `datetime.datetime` object passed-in.

And the following properties:

- `capsule_name`: The name of the capsule (as specified in the `setup.py` file installing it).
- `client`: An instance of `taskw.warrior.TaskWarriorShellout` allowing one to interact with Taskwarrior via an object-oriented interface. See taskw's documentation for more information.
- `configuration`: An editable dictionary-like object that stores local per-capsule configuration. If modifications are made to this object, be sure to call `.write()` to write the changes to disk. Note that configuration files are stored in `~/.taskwarrior-capsules/<capsule_name>.ini`, but that encouraging users to hand-modify the configuration file is discouraged.
- `global_configuration`: A dictionary-like object storing Taskwarrior Capsules' configuration. This file, too, is editable, but editing is discouraged.
- `meta`: An instance of `taskwarrior_capsules.capsule_meta.CapsuleMeta` storing metadata about the Taskwarrior Capsules environment.

3.2 Your `setup.py`

For registering your capsule, you'll want to make sure you've written a valid `setup.py` for installing your capsule, and used the proper entrypoints depending upon what methods you've implemented above.

- For capsules adding additional commands, you need to register your capsule using the `taskwarrior_capsules` entrypoint.
- For preprocessor capsules, you need to register your capsule using the `taskwarrior_preprocessor_capsules` entrypoint.
- For postprocessor capsules, you need to register your capsule using the `taskwarrior_postprocessor_capsules` entrypoint.

The below `setup.py` is a (fairly) minimal example of a setup file registering a new capsule executable with the command `tw example`:

```
from setuptools import setup, find_packages

setup(
    name='taskwarrior-example-capsule',  # Please follow this example for
                                         # naming your capsule so they are
                                         # easy for people to find when searching
    version='0.1',  # Your capsule's version number.  Where reasonable,
                    # we recommend that you follow semver principles.
    url='https://github.com/yourname/taskwarrior-example-capsules', # The URL at which
                                                                    # your package is hosted.
    description=(
        'This capsule does something that helps someone.'
    ),  # A brief description of what your capsule does
    author='Adam Coddington',
    author_email='me@adamcoddington.net',
    packages=find_packages(),
    entry_points={
        'taskwarrior_capsules': [
            'example = module.path.to.your.capsule:YourCapsuleClass',
        ],  # This is the most important part!
    },
)
```

Pay special attention to the `entry_points` section above! The name of the command is to the left of the = sign, and the module path to your class is to the right, using a `:` to separate the module path from your class's name.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*