Charm Tools Documentation

Cory Johns, Marco Ceppi, Kapil Thangavelu

Feb 25, 2019

Reference

The *charm* command includes several subcommands used to build, maintain, and release Juju Charms, which are Open Source encapsulated operations logic for managing software in the cloud or bare-metal servers using cloud-like APIs.

Installation is easy with snaps:

```
snap install --classic charm
```

Reference for the various available commands can be found below, or via the command-line with:

```
charm help
```

CHAPTER 1

Available Commands

The following subcommands are available and can be invoked as `charm <command>` (for example, `charm build`). Details for each command, including the supported options and parameters, can be output with either `charm help <command>` or `charm <command> --help`.

Command	Description
`add`	add icon, readme, or tests to a charm
`attach`	upload a file as a resource for a charm
`attach-plan`	associates the charm with the plan
`build`	build a charm from layers and interfaces
`create`	create a new charm
`grant`	grant charm or bundle permissions
`help`	Show help on a command or other topic.
`layers`	Show a colored breakdown of what layers each file came from
`list`	list charms for the given users.
`list-plans`	list plans
`list-resources`	display the resources for a charm in the charm store
`login`	login to the charm store
`logout`	logout from the charm store
`proof`	perform static analysis on a charm or bundle
`pull`	download a charm or bundle from the charm store
`pull-resource`	pull a charm resource to the local machine
`push`	push a charm or bundle into the charm store
`push-plan`	push new plan
`push-term`	create new Terms and Conditions document (revision)
`release`	release a charm or bundle
`release-term`	releases the given terms document
`resume-plan`	resumes plan for specified charms
`revoke`	revoke charm or bundle permissions
`set`	set charm or bundle extra-info, home page or bugs URL
`show`	print information on a charm or bundle

Continued on next page

Table 1 – continued from previous page

Command	Description
`show-plan`	show plan details
`show-plan-revisions`	show all revision of a plan
`show-term`	shows the specified term
`suspend-plan`	suspends plan for specified charms
`terms`	list terms owned by the current user
`terms-used`	list terms required by current user's charms
`version`	display tooling version information
`whoami`	display jaas user id and group membership

CHAPTER 2

Build Tactics

When building charms, multiple layers are brought together in an ordered, depth-first recursive fashion. The individual files of each layer are merged according to a list of merge tactics. These tactics determine whether the file from a higher layer will replace or be merged with the copy from the lower layer, with the details of how the merge happens being implemented by the tactic. Each file is tested against each tactic in a specific order (as determined by the `DEFAULT_TACTICS` list), with the first one to match being applied to the file and all other tactics disregarded.

2.1 Built-in Tactics

ActionsYAML	Tactic for processing and combining the `actions.yaml` file from each layer.
ConfigYAML	Tactic for processing and combining the `config.yaml` file from each layer.
CopyTactic	Tactic to copy a file without modification or merging.
CopyrightTactic	Tactic to combine the copyright info from all layers into a final machine-readable format.
DistYAML	Tactic for processing and combining the `dist.yaml` file from each layer.
DynamicHookBind	Base class for process hooks dynamically generated from the hook template.
ExactMatch	Mixin to match a file with an exact name.
ExcludeTactic	Tactic to handle per-layer excludes.
IgnoreTactic	Tactic to handle per-layer ignores.
InstallerTactic	Tactic to process any `.pypi` files and install Python packages directly into the charm's `lib/` directory.
InterfaceBind	Tactic to copy the hook template into place for all relation hooks.
InterfaceCopy	Tactic to process a relation endpoint using an interface layer.

Continued on next page

Table 1 – continued from previous page

JSONTactic	Base class for tactics dealing with JSON data.
LayerYAML	Tactic for processing and combining the `layer.yaml` file from each layer.
ManifestTactic	Tactic to avoid copying a build manifest file from a base layer.
MetadataYAML	Tactic for processing and combining the `metadata.yaml` file from each layer.
ResourcesYAML	Tactic for processing and combining the `resources.yaml` file from each layer.
SerializedTactic	Base class for tactics which deal with serialized data, such as YAML or JSON.
StandardHooksBind	Tactic to copy the hook template into place for all standard hooks.
StorageBind	Tactic to copy the hook template into place for all storage hooks.
Tactic	Base class for all tactics.
VersionTactic	Tactic to generate the `version` file with VCS revision info to be displayed in `juju status`.
WheelhouseTactic	Tactic to process the `wheelhouse.txt` file and build a source-only wheelhouse of Python packages in the charm's `wheelhouse/` directory.
YAMLTactic	Base class for tactics dealing with YAML data.
extend_with_default	Extend a jsonschema validator to propagate default values prior to validating.
load_tactic	Load a tactic from the current layer using a dotted path.

class charmtools.build.tactics.**ActionsYAML**(**args*, ***kwargs*)
Tactic for processing and combining the `actions.yaml` file from each layer.

class charmtools.build.tactics.**ConfigYAML**(**args*, ***kwargs*)
Tactic for processing and combining the `config.yaml` file from each layer.

class charmtools.build.tactics.**CopyTactic**(*entity*, *target*, *layer*, *next_config*)
Tactic to copy a file without modification or merging.

The last version of the file "wins" (e.g., from the charm layer).

This is the final fallback tactic if nothing else matches.

class charmtools.build.tactics.**CopyrightTactic**(**args*, ***kwargs*)
Tactic to combine the copyright info from all layers into a final machine-readable format.

class charmtools.build.tactics.**DistYAML**(**args*, ***kwargs*)
Tactic for processing and combining the `dist.yaml` file from each layer.

class charmtools.build.tactics.**DynamicHookBind**(*name*, *owner*, *target*, *config*, *output_files*, *template_file*)
Base class for process hooks dynamically generated from the hook template.

This tactic is not used directly, but serves as a base for the type-specific dynamic hook tactics, like *StandardHooksBind*, or *InterfaceBind*.

HOOKS = []
List of all hooks to populate.

sign()
Sign all hook files generated by this tactic.

class charmtools.build.tactics.**ExactMatch**
Mixin to match a file with an exact name.

FILENAME = None
The filename to be matched

classmethod trigger(*entity*, *target*, *layer*, *next_config*)
Match if the current entity's filename is what we're looking for.

class charmtools.build.tactics.**ExcludeTactic**(*entity*, *target*, *layer*, *next_config*)
Tactic to handle per-layer excludes.

If a given layer's `layer.yaml` has an `exclude` list, then any file or directory included in that list that is provided by the current layer will be ignored, though any matching file or directory provided by base layers or any higher level layers will be included.

The `exclude` list uses the same format as a `.gitignore` file.

class charmtools.build.tactics.**IgnoreTactic**(*entity*, *target*, *layer*, *next_config*)
Tactic to handle per-layer ignores.

If a given layer's `layer.yaml` has an `ignore` list, then any file or directory included in that list that is provided by base layers will be ignored, though any matching file or directory provided by the current or any higher level layers will be included.

The `ignore` list uses the same format as a `.gitignore` file.

class charmtools.build.tactics.**InstallerTactic**(*entity*, *target*, *layer*, *next_config*)
Tactic to process any `.pypi` files and install Python packages directly into the charm's `lib/` directory.

This is used in Kubernetes type charms due to the lack of a proper install or bootstrap phase.

class charmtools.build.tactics.**InterfaceBind**(*name*, *owner*, *target*, *config*, *output_files*, *template_file*)
Tactic to copy the hook template into place for all relation hooks.

This tactic is not part of the normal set of tactics that are matched against files. Instead, it is manually called to fill in the set of relation hooks needed by this charm.

class charmtools.build.tactics.**InterfaceCopy**(*interface*, *relation_name*, *role*, *target*, *config*)
Tactic to process a relation endpoint using an interface layer.

This tactic is not part of the normal set of tactics that are matched against files. Instead, it is manually called for each relation endpoint that has a corresponding interface layer.

class charmtools.build.tactics.**JSONTactic**(**args*, ***kwargs*)
Base class for tactics dealing with JSON data.

dump(*data*)
Serialize and write the data to the file.

Must be impelemented by a subclass.

load(*fn*)
Load and deserialize the data from the file.

Must be impelemented by a subclass.

class charmtools.build.tactics.**LayerYAML**(**args*, ***kwargs*)
Tactic for processing and combining the `layer.yaml` file from each layer.

The input `layer.yaml` files can contain the following sections:

- `includes` This is the heart of layering. Layers and interface layers referenced in this list value are pulled in during charm build and combined with each other to produce the final layer.
- `config`, `metadata`, `dist`, or `resources` These objects can contain a `deletes` object to list keys that should be deleted from the resulting `<section>.yaml`.
- `defines` This object can contain a jsonschema used to defined and validate options passed to this layer from another layer. The options and schema will be namespaced by the current layer name. For example, layer "foo" defining `bar: {type: string}` will accept `options: {foo: {bar: "foo"}}` in the final `layer.yaml`.
- `options` This object can contain option name/value sections for other layers. For example, if the current layer includes the previously referenced "foo" layer, it could include `foo: {bar: "foo"}` in its `options` section.

class charmtools.build.tactics.**ManifestTactic**(*entity*, *target*, *layer*, *next_config*)
Tactic to avoid copying a build manifest file from a base layer.

class charmtools.build.tactics.**MetadataYAML**(**args*, ***kwargs*)
Tactic for processing and combining the `metadata.yaml` file from each layer.

class charmtools.build.tactics.**ResourcesYAML**(**args*, ***kwargs*)
Tactic for processing and combining the `resources.yaml` file from each layer.

class charmtools.build.tactics.**SerializedTactic**(**args*, ***kwargs*)
Base class for tactics which deal with serialized data, such as YAML or JSON.

apply_edits()
Apply any edits defined in the final `layer.yaml` file to the data.

An example edit definition:

```
metadata:
  deletes:
    - requires.http
```

combine(*existing*)
Merge the deserialized data from two layers using `deepmerge`.

dump(*data*)
Serialize and write the data to the file.

Must be impelemented by a subclass.

load(*fn*)
Load and deserialize the data from the file.

Must be impelemented by a subclass.

process()
Now that the tactics for the current entity have been combined for all layers, process the entity to produce the final output file.

Must be implemented by a subclass.

read()
Read and cache the data into memory, using `self.load()`.

class charmtools.build.tactics.**StandardHooksBind**(*name*, *owner*, *target*, *config*, *output_files*, *template_file*)
Tactic to copy the hook template into place for all standard hooks.

This tactic is not part of the normal set of tactics that are matched against files. Instead, it is manually called to fill in the standard set of hook implementations.

class charmtools.build.tactics.**StorageBind**(*name*, *owner*, *target*, *config*, *output_files*, *template_file*)

Tactic to copy the hook template into place for all storage hooks.

This tactic is not part of the normal set of tactics that are matched against files. Instead, it is manually called to fill in the set of storage hooks needed by this charm.

class charmtools.build.tactics.**Tactic**(*entity*, *target*, *layer*, *next_config*)

Base class for all tactics.

Subclasses must implement at least `trigger` and `process`, and probably also want to implement `combine`.

combine(*existing*)

Produce a tactic informed by the existing tactic for an entry.

This is when a rule in a higher level charm overrode something in one of its bases for example.

Should be implemented by a subclass if any sort of merging behavior is desired.

config

Return the combined config from the layer above this (if any), this, and all lower layers.

Note that it includes one layer higher so that the tactic can make decisions based on the upcoming layer.

current

Alias for `Tactic.layer`

entity

The current entity (a.k.a. file) being processed.

classmethod get(*entity*, *target*, *layer*, *next_config*, *current_config*, *existing_tactic*)

Factory method to get an instance of the correct Tactic to handle the given entity.

layer

The current layer under consideration

layer_name

Name of the current layer being processed.

lint()

Test the resulting file to ensure that it is valid.

Return `True` if valid. If invalid, return `False` or raise a `BuildError`

Should be implemented by a subclass.

process()

Now that the tactics for the current entity have been combined for all layers, process the entity to produce the final output file.

Must be implemented by a subclass.

read()

Read the contents of the file to be processed.

Can be implemented by a subclass. By default, returns `None`.

relpath

The path to the file relative to the layer.

sign()

Return signature in the form `{relpath: (origin layer, SHA256)}`

Can be overridden by a subclass, but the default implementation will usually be fine.

target
The target (final) layer.

target_file
The location where the processed file will be written to.

classmethod trigger(*entity*, *target*, *layer*, *next_config*)
Determine whether the rule should apply to a given entity (file).

Generally, this should check the entity name, but could conceivably also inspect the contents of the file.

Must be implemented by a subclass or the tactic will never match.

class charmtools.build.tactics.**VersionTactic**(*charm*, *target*, *layer*, *next_config*)
Tactic to generate the `version` file with VCS revision info to be displayed in `juju status`.

This tactic is not part of the normal set of tactics that are matched against files. Instead, it is manually called to generate the `version` file.

class charmtools.build.tactics.**WheelhouseTactic**(**args*, ***kwargs*)
Tactic to process the `wheelhouse.txt` file and build a source-only wheelhouse of Python packages in the charm's `wheelhouse/` directory.

class charmtools.build.tactics.**YAMLTactic**(**args*, ***kwargs*)
Base class for tactics dealing with YAML data.

Tries to ensure that the order of keys is preserved.

dump(*data*)
Serialize and write the data to the file.

Must be impelemented by a subclass.

load(*fn*)
Load and deserialize the data from the file.

Must be impelemented by a subclass.

charmtools.build.tactics.**extend_with_default**(*validator_class*)
Extend a jsonschema validator to propagate default values prior to validating.

Used internally to ensure validation of layer options supports default values.

charmtools.build.tactics.**load_tactic**(*dpath*, *basedir*)
Load a tactic from the current layer using a dotted path.

The final element in the path should be a *Tactic* subclass.

2.2 Custom Tactics

A charm or layer can also define one or more custom tactics in its `layer.yaml` file. The file can contain a top-level `tactics` key, whose value is a list of dotted Python module names, relative to the layer's base directory. For example, a layer could include this in its `layer.yaml`:

```
tactics:
  - tactics.my_layer.README Tactic
```

This would cause the build command to look for a module `tactics/my_layer.py` with a class of `README Tactic` in it, which must inherit from *Tactic*.

Custom tactics are tested before the built-in tactics, so they can override the behavior of built-in tactics if desired. Care should be taken if doing this because changing the behavior of built-in tactics can end up breaking other layers or charms.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

S

T

V

W

Y