Charm Tools Documentation

Cory Johns, Marco Ceppi, Kapil Thangavelu

Jan 27, 2021

Reference

1 Available Commands **3**

2 Build Tactics **5**
2.1 Built-in Tactics . . . 5
2.2 Custom Tactics . . . 11

3 Reproducible Charms **13**
3.1 Creating the lock file . . . 13
3.2 Rebuilding the charm from the lock file . . . 14
3.3 Other useful information . . . 14

4 Contributing **15**

5 Changelog **17**
5.1 charm-tools 2.8.1 + charmstore-client 2.5.0 . . . 17
5.2 charm-tools 2.8.0 + charmstore-client 2.5.0 . . . 17
5.3 charm-tools 2.7.8 + charmstore-client 2.4.0+git-13-547c6f2 . . . 18
5.4 charm-tools 2.7.7 + charmstore-client 2.4.0+git-13-547c6f2 . . . 18
5.5 charm-tools 2.7.6 + charmstore-client 2.4.0+git-13-547c6f2 . . . 18
5.6 charm-tools 2.7.5 + charmstore-client 2.4.0+git-13-547c6f2 . . . 18
5.7 charm-tools 2.7.5 + charmstore-client 2.4.0+git-13-547c6f2 . . . 18
5.8 charm-tools 2.7.4 + charmstore-client 2.4.0+git-13-547c6f2 . . . 19
5.9 charm-tools 2.7.3 + charmstore-client 2.4.0+git-13-547c6f2 . . . 19
5.10 charm-tools 2.7.2 + charmstore-client 2.4.0+git-3-cbbf887 . . . 19
5.11 charm-tools 2.7.1 + charmstore-client 2.4.0 . . . 19
5.12 charm-tools 2.7.0 + charmstore-client 2.4.0 . . . 20
5.13 charm-tools 2.6.1 + charmstore-client 2.4.0 . . . 20
5.14 charm-tools 2.6.0 + charmstore-client 2.4.0 . . . 20

6 Indices and tables **23**

Python Module Index **25**

Index **27**

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.
- `defines` This object can contain a jsonschema used to define and validate options passed to this layer from another layer. The options and schema will be namespaced by the current layer name.
- `options` This object can contain option name/value sections for other layers.
- `config`, `metadata`, `dist`, or `resources` These objects can contain a `deletes` object to list keys that should be deleted from the resulting `<section>.yaml`.

Example, layer `foo` might define this `layer.yaml` file:

```
includes:
  - layer:basic
  - interface:foo
defines:
  foo-opt:
    type: string
    default: 'foo-default'
options:
  basic:
    use_venv: true
```

And layer `bar` might define this `layer.yaml` file:

```
includes:
  - layer:foo
options:
  foo-opt: 'bar-value'
metadata:
  deletes:
    - 'requires.foo-relation'
```

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.

read()
Read the contents of the file to be processed.

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

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

Reproducible Charms

When building charms, multiple layers are brought together in an ordered, depth-first recursive fashion. The individual files of each layer are merged, and then python modules are brought in according to `wheelhouse.txt` files that may exist in each layer.

Layers (and Interfaces) are typically Git repositories, and by default the default branch (usually called `master`) of the repository is fetched and used.

Also, although the top level Python modules can be pinned in the `wheelhouse.txt` files, any dependent modules are fetched as their latest versions. This makes re-building a charm with the same layers and modules tricky, which may be required for stable charms. It is possible, by populating layer and interface directories directly, and by pinning every Python module in a `wheelhouse.txt` override file that is passed using the `--wheelhouse-overrides` option to the `charm build` command.

An alternative strategy is to use a new feature of the `charm build` command which can generate a lock file that contains all of the layers and Python modules, and their versions. This can then, for subsequent builds, be used to fetch the same layer versions and Python modules to re-create the charm.

As the lock file is a `JSON` file, it can be manually edited to change a layer version if a new version of a stable charm is needed, or a python module can be changed.

Additionally, it is possible to track a branch in the repository for a layer so that a stable (or feature) branch can be maintained and then charms rebuilt from that branch.

The new options for this feature are:

- `--write-lock-file`
- `--use-lock-file-branches`
- `--ignore-lock-file`

3.1 Creating the lock file

To create a lock file, the option `--write-lock-file` is passed to the `charm build` command. This option *automatically* ignores the existing lock file, and rebuilds the charm using the latest versions of the layers and the

versions of the modules as determined in the various `wheelhouse.txt` files.

Python module versions are also recorded. If a VCS repository is used for the python module, then any branch specified is also recorded, along with the commit.

At the end of the build, the lock file is written with all of the layer and Python module information.

The lock file is installed *in* the base layer directory so that it can be committed into the VCS and used for subsequent builds.

The name of the lock file is `build.lock`.

3.2 Rebuilding the charm from the lock file

If a lock file (`build.lock`) is available in the top layer, then it will be used to control the versions of the layers and modules *by default*. i.e. the presence of the lock file controls the build.

Three options are available which can influence the build when a lock file is present:

- `--ignore-lock-file`
- `--use-lock-file-branches`
- `--wheelhouse-overrides`

If the `--ignore-lock-file` option is used, then the charm is built as though there is no lock file.

If the `--use-lock-file-branches` is used, then, for VCS items (layers, interfaces, and Python modules specified using a VCS string), then the branch (if there was one) will be used, rather than the commit version. This can be used to track a branch in a layer or Python module.

Note: if `--wheelhouse-overrides` is used, then that wheelhouse will override the lock file. i.e. the lock file overrides the layers' `wheelhouse.txt` file, and then the `--wheelhouse-overrides` then can override the lock-file. This is intentional to allow the build to perform specific overrides as needed.

3.3 Other useful information

This is the first iteration of 'reproducible charms'. As such, only Git is supported as the VCS for the layers, and Git and Bazaar for Python modules. A future iteration may support more VCS systems.

Only the top layer is inspected for a `build.lock` file. Any other layers are considered inputs and their `build.lock` files are ignored (if they are present).

Also, regardless of the `wheelhouse.txt` layers, the lock file will override any changes that may be introduced in stable branches, if they are bing tracked using `--use-lock-file-branches`. This may lead to unexpected behaviour.

CHAPTER 4

Contributing

The charm command is created from the combination of two repositorires:

- charm-tools handles local operations, such as building and linting charms
- charmstore-client handles interactions with the charm store, such as pushing or pulling charms, or managing access controls

Bugs should be filed against the appropriate repository for the most efficient handling.

CHAPTER 5

Changelog

5.1 charm-tools 2.8.1 + charmstore-client 2.5.0

Wednesday January 27 2021

charm-tools

- Add option to create .charm file (#592)
- Add 'docs' to known metadata fields (#591)
- Add reproducible charm build feature (#585)
- Fix exception rendering "already promulgated" error (#590)
- Align setup.py to requirements.txt (#589)
- Fix TypeError from linter on X.Y min-juju-version (#588)
- Make output_dir the same as build_dir (#564)

5.2 charm-tools 2.8.0 + charmstore-client 2.5.0

Tuesday November 10 2020

charm-tools

- Fix snap build for updated charmstore-client (#587)
- Store rev when pull-source on a subdir layer (#583)
- Add revision info to output of pull-source (#582)
- Add –branch option to pull-source (#581)
- Raise more useful BuildError on missing pkg name (#579)
- Deprecate Operator charm template (#578)

charmstore-client

- Update dependencies
- Make charm-push support archives

5.3 charm-tools 2.7.8 + charmstore-client 2.4.0+git-13-547c6f2

Tuesday July 21 2020

charm-tools

- Normalize package names when processing wheelhouse (#576)

5.4 charm-tools 2.7.7 + charmstore-client 2.4.0+git-13-547c6f2

Monday July 20 2020

charm-tools

- Fix handling of comments in wheelhouse (#574)

5.5 charm-tools 2.7.6 + charmstore-client 2.4.0+git-13-547c6f2

Thursday July 16 2020

charm-tools

- Switch to requirements-parser for wheelhouse (#572)

5.6 charm-tools 2.7.5 + charmstore-client 2.4.0+git-13-547c6f2

Thursday June 25 2020

charm-tools

- Process wheelhouse.txt holistically rather than per-layer (#569)
- Handle invalid config file more gracefully (#567)
- Default to charming category of the Juju Discourse (#565)

5.7 charm-tools 2.7.5 + charmstore-client 2.4.0+git-13-547c6f2

Thursday June 25 2020

charm-tools

- Process wheelhouse.txt holistically rather than per-layer (#569)
- Handle invalid config file more gracefully (#567)
- Default to charming category of the Juju Discourse (#565)

5.8 charm-tools 2.7.4 + charmstore-client 2.4.0+git-13-547c6f2

Thursday March 26 2020

charm-tools

- Add workaround for user site package conflicts (#561)
- Add Build Snap action so PRs have snap to test easily (#562)

5.9 charm-tools 2.7.3 + charmstore-client 2.4.0+git-13-547c6f2

Saturday Feb 29 2020

charm-tools

- Add Operator charm template (#557)
- Add OpenStack templates to requirements (#558)
- Fix 471 (#556)
- Add functions support; (#555)
- Allow boolean config options to have null default (#554)

charmstore-client

- fix dependencies
- cmd/charm: allow users with domains in ACLs
- Updated charmstore and charmrepo dependency.
- charm whoami: return an error when the user is not logged in
- Update dependencies
- Fix dependency files

5.10 charm-tools 2.7.2 + charmstore-client 2.4.0+git-3-cbbf887

Tuesday October 8 2019

charm-tools

- Add opendev.org https and git fetcher (#553)

charmstore-client

- Disallow release in promulgated namespace

5.11 charm-tools 2.7.1 + charmstore-client 2.4.0

Tuesday September 24 2019

charm-tools

- Fix maintainer validation not handling unicode (#550)

- Fix snap builds on other arches (#548)
- Change deployment.type optional (for k8s charms) (#547)
- Move daemonset to deployment.type (for k8s charms) (#546)

5.12 charm-tools 2.7.0 + charmstore-client 2.4.0

Wednesday September 18 2019

charm-tools

- Fix charm-build conflict when building concurrently (#545)
- Rename README files with markdown extension (#543)
- Update charm.1 manpage (#522)
- Feature/add deployment field2metadata (#544)
- fix charm build help message (#542)
- Cleanup cached layers / interfaces after build (#540)
- edge case for setting charm_ver (#538)

5.13 charm-tools 2.6.1 + charmstore-client 2.4.0

Thursday July 11 2019

charm-tools

- Remove bad URL from PR template (#537)
- Update pypi release target to work with newer tox (#530)
- requirements.txt: update version limit for requests (#535) (#536)
- Fix config key regexp to allow short config keys. (#534)

5.14 charm-tools 2.6.0 + charmstore-client 2.4.0

Thursday June 6 2019

charm-tools

- Honor ignores / excludes when checking for post-build changes (#529)
- Resolve vergit runtime dependency (#527)
- Upgrade to use py3.7 on Travis (#523)
- Fix installing from git without vergit installed (#520)
- Fix installation dependency on vergit (#519)
- Gracefully handle JSON decode errors from layer index (#516)
- Add support for layer-index and fallback-layer-index (#515)
- Ensure setuptools for charmstore-client build (#509)

- Refactor version handling in snap to work with core18 (#508)
- Make series required (#499)
- Add setuptools to requirements.txt (#498)
- Fix charm-layer handling of old format build-manifest (#496)
- Fix nested build dir check in Python2 (#494)
- Improve docs for LayerYAML tactic (#493)
- Add promulgate and unpromulgate commands (#491)
- Fix and improve charm-layers (#492)
- Fix checking of build dir nested under source dir (#490)
- Add basic documentation (#489)
- Allow *build* folders in the charm (#486)
- Fix CHARM_HIDE_METRICS environment variable (#483)
- Address security alerts from GitHub (#484)
- Use shutil.copytree instead of path.rename (#482)

charmstore-client

- Remove the temporary file
- update charmrepo dependency
- update charm dependency
- internal/ingest: set permissions correctly
- cmd/charm-ingest: use –hardlimit not –softlimit
- cmd/charm-ingest: expose disk limits
- make tests pass
- internal/ingest: transfer resources
- cmd/charm-ingest: Add a basic ingest command
- internal/ingest: resolve resources in whitelist
- internal/ingest: expose public ingest API.
- cmd/charm-ingest: Add the basics of whitelist parsing
- restore go-cmp dependency version
- Move cmd/ingest to internal/ingest
- cmd/ingest: fix comment from previous review
- cmd/ingest: run tests against real charmstore servers
- cmd/ingest: core ingestion logic
- cmd/charm/charmcmd: add some basic tests for show command
- cmd/charm/charmcmd: improve output in *charm show* for unpublished charms
- cmd/ingest: new ingest command
- cmd/charm/charmcmd: improve incompatible registry version error

- Update usage of docker to oci-image resource type.
- Reviews.
- cmd/charmcmd: Better yaml output for resources.
- cmd/charmcmd: Allow multiple users in list.
- all: use quicktest for tests

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

charmtools.build.tactics, 6

Index

A

ActionsYAML (*class in charmtools.build.tactics*), 6
apply_edits() (*charmtools.build.tactics.SerializedTactic method*), 8

C

charmtools.build.tactics (*module*), 6
combine() (*charmtools.build.tactics.SerializedTactic method*), 8
combine() (*charmtools.build.tactics.Tactic method*), 9
config (*charmtools.build.tactics.Tactic attribute*), 9
ConfigYAML (*class in charmtools.build.tactics*), 6
CopyrightTactic (*class in charmtools.build.tactics*), 6
CopyTactic (*class in charmtools.build.tactics*), 6
current (*charmtools.build.tactics.Tactic attribute*), 9

D

DistYAML (*class in charmtools.build.tactics*), 6
dump() (*charmtools.build.tactics.JSONTactic method*), 7
dump() (*charmtools.build.tactics.SerializedTactic method*), 8
dump() (*charmtools.build.tactics.YAMLTactic method*), 10
DynamicHookBind (*class in charmtools.build.tactics*), 6

E

entity (*charmtools.build.tactics.Tactic attribute*), 9
ExactMatch (*class in charmtools.build.tactics*), 6
ExcludeTactic (*class in charmtools.build.tactics*), 7
extend_with_default() (*in module charmtools.build.tactics*), 10

F

FILENAME (*charmtools.build.tactics.ExactMatch attribute*), 7

G

get() (*charmtools.build.tactics.Tactic class method*), 9

H

HOOKS (*charmtools.build.tactics.DynamicHookBind attribute*), 6

I

IgnoreTactic (*class in charmtools.build.tactics*), 7
InstallerTactic (*class in charmtools.build.tactics*), 7
InterfaceBind (*class in charmtools.build.tactics*), 7
InterfaceCopy (*class in charmtools.build.tactics*), 7

J

JSONTactic (*class in charmtools.build.tactics*), 7

L

layer (*charmtools.build.tactics.Tactic attribute*), 9
layer_name (*charmtools.build.tactics.Tactic attribute*), 9
LayerYAML (*class in charmtools.build.tactics*), 7
lint() (*charmtools.build.tactics.Tactic method*), 9
load() (*charmtools.build.tactics.JSONTactic method*), 7
load() (*charmtools.build.tactics.SerializedTactic method*), 9
load() (*charmtools.build.tactics.YAMLTactic method*), 10
load_tactic() (*in module charmtools.build.tactics*), 11

M

ManifestTactic (*class in charmtools.build.tactics*), 8
MetadataYAML (*class in charmtools.build.tactics*), 8

P

process() (*charmtools.build.tactics.SerializedTactic method*), 9

process() (*charmtools.build.tactics.Tactic method*), 10

R

read() (*charmtools.build.tactics.SerializedTactic method*), 9
read() (*charmtools.build.tactics.Tactic method*), 10
read() (*charmtools.build.tactics.WheelhouseTactic method*), 10
relpath (*charmtools.build.tactics.Tactic attribute*), 10
ResourcesYAML (*class in charmtools.build.tactics*), 8

S

SerializedTactic (*class in charmtools.build.tactics*), 8
sign() (*charmtools.build.tactics.DynamicHookBind method*), 6
sign() (*charmtools.build.tactics.Tactic method*), 10
StandardHooksBind (*class in charmtools.build.tactics*), 9
StorageBind (*class in charmtools.build.tactics*), 9

T

Tactic (*class in charmtools.build.tactics*), 9
target (*charmtools.build.tactics.Tactic attribute*), 10
target_file (*charmtools.build.tactics.Tactic attribute*), 10
trigger() (*charmtools.build.tactics.ExactMatch class method*), 7
trigger() (*charmtools.build.tactics.Tactic class method*), 10

V

VersionTactic (*class in charmtools.build.tactics*), 10

W

WheelhouseTactic (*class in charmtools.build.tactics*), 10

Y

YAMLTactic (*class in charmtools.build.tactics*), 10