Craft Parts

Release 1.13.0

Canonical Ltd.

Sep 15, 2022

GETTING STARTED

Craft Parts provides a mechanism to obtain data from different sources, process it in various ways, and prepare a filesystem subtree suitable for deployment. The components used in its project specification are called *parts*, which can be independently downloaded, built and installed, and also depend on each other in order to assemble the subtree containing the final artifacts.

CHAPTER

ONE

ADDING PARTS PROCESSING TO AN APPLICATION

1.1 A simple example

To add parts processing to an application, instantiate the `LifecycleManager` class passing the parts dictionary. Plan actions for the `PRIME` target step, and execute them:

```
import yaml
from craft_parts import LifecycleManager, Step

parts_yaml = """
parts:
  hello:
    plugin: autotools
    source: https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
    prime:
      - usr/local/bin/hello
      - usr/local/share/man/*
"""

parts = yaml.safe_load(parts_yaml)

lcm = LifecycleManager(parts, application_name="example", cache_dir=".")
actions = lcm.plan(Step.PRIME)
with lcm.action_executor() as aex:
    aex.execute(actions)
```

When executed, the lifecycle manager will download the tarball we specified, unpack it, run its configuration script, compile the source code, install the resulting artifacts, and extract only the files we want to deploy. The final result is a subtree containing the files we wanted to prime:

```
prime
prime/usr
prime/usr/local
prime/usr/local/bin
prime/usr/local/bin/hello
prime/usr/local/share
prime/usr/local/share/man
prime/usr/local/share/man/man1
prime/usr/local/share/man/man1/hello.1
```

1.2 Learning more

- The `LifecycleManager` class offers many options to configure parts processing.
- The CLI tool source code is a good reference for a real world usage of the lifecycle manager.
- Parts are similar to those used in Snapcraft, including some of its V2 plugins. See the Snapcraft parts documentation for details.

CHAPTER

TWO

THE CRAFTCTL TOOL

Craft-parts installs the `craftctl` utility executable. It is intended to be invoked from user-defined scriptlets in parts to call the built-in handler for a given step or to manipulate application-defined variables.

2.1 Calling default step handlers

Use `craftctl default` from within a overriden step scriptlet to execute the built-in handler for the step being processed:

```
import yaml
from craft_parts import LifecycleManager, Step

parts_yaml = """
parts:
  hello:
    plugin: autotools
    source: https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
    override-build: |
      echo "Running the build step"
      craftctl default
"""

parts = yaml.safe_load(parts_yaml)

lcm = LifecycleManager(parts, application_name="example", cache_dir=".")
actions = lcm.plan(Step.PRIME)
with lcm.action_executor() as aex:
    aex.execute(actions)
```

This example will result in the message being displayed, and the part source being built:

```
+ echo 'Running the build step'
Running the build step
+ craftctl default
+ '[' '!' -f ./configure ']'
+ '[' '!' -f ./configure ']'
+ '[' '!' -f ./configure ']'
+ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
```

(continues on next page)

(continued from previous page)

```
checking for a thread-safe mkdir -p... /bin/mkdir -p
...
```

2.2 Using application variables

The application can define project variables that can be read and written during execution of user-defined scriptlets by using `craftctl get` and `craftctl set`. Valid variables and their initial values must be specified when creating the `LifecycleManager`, and the variable value must be consumed by the application after the parts lifecycle execution is finished:

```
import yaml
from craft_parts import LifecycleManager, Step

parts_yaml = """
parts:
  foo:
    plugin: nil
    override-pull: |
      echo "Running the pull step"
      craftctl set version="2"
"""

parts = yaml.safe_load(parts_yaml)

lcm = LifecycleManager(
  parts,
  application_name="example",
  cache_dir="."
  project_vars={"version": "1"}
)
actions = lcm.plan(Step.PRIME)
with lcm.action_executor() as aex:
    aex.execute(actions)

version = lf.project_info.get_project_variable("version")
print(f"Version is version")
```

Execution of this example results in:

```
+ echo 'Running the pull step'
Running the pull step
+ craftctl set version=2
Version is 2
```

Note that project variables are not intended for use in logic construction during parts processing, and each variable must not be set more than once. Variable setting can also be restricted to a specific part if `project_vars_part_name` is passed to `LifecycleManager`.

CHAPTER

THREE

USING CRAFT-PARTS FROM THE COMMAND LINE

3.1 The CLI tool

Parts processing can be also executed directly from the command line by invoking the module's main entry point. This can be useful for debugging, or to experiment different configuration options:

```
$ python3 -mcraft_parts pull
Execute: Pull foo
$ python3 -mcraft_parts --dry-run --show-skipped
Skip pull foo (already ran)
Build foo
Stage foo
Prime foo
```

By default, parts will be read from a file called `parts.yaml`. Run the tool with `--help` for a list of valid arguments.

CHAPTER

FOUR

THE LIFECYCLE MANAGER

The lifecycle manager holds information about the parts specification and the project state, and coordinates planning and execution of actions required to run steps for a given set of parts.

CHAPTER

FIVE

PARTS AND STEPS

Parts and steps are the basic data types craft-parts will work with. Together, they define the lifecycle of a project (i.e. how to process each step of each part in order to obtain the final primed result).

5.1 Parts

When the `LifecycleManager` is invoked, parts are defined in a dictionary under the `parts` key. If the dictionary contains other keys, they will be ignored.

5.2 Steps

Steps are used to establish plan targets and in informational data structures such as `StepInfo`. They are defined by the `Step` enumeration, containing entries for the lifecycle steps `PULL`, `OVERLAY`, `BUILD`, `STAGE`, and `PRIME`.

5.2.1 Step execution environment

Craft-parts defines the following environment for use during step processing and execution of user-defined scriptlets:

- `CRAFT_ARCH_TRIPLET`: The the machine-vendor-os platform triplet definition.
- `CRAFT_TARGET_ARCH`: The architecture we're building for.
- `CRAFT_PARALLEL_BUILD_COUNT`: The maximum number of concurrent build jobs to execute.
- `CRAFT_PROJECT_DIR`: The path to the current project's subtree in the filesystem.
- `CRAFT_PART_NAME`: The name of the part currently being processed.
- `CRAFT_PART_SRC`: The path to the part source directory. This is where sources are located after the `PULL` step.
- `CRAFT_PART_SRC_WORK`: The path to the part source subdirectory, if any. Defaults to the part source directory.
- `CRAFT_PART_BUILD`: The path to the part build directory. This is where parts are built during the `BUILD` step.
- `CRAFT_PART_BUILD_WORK`: The path to the part build subdirectory in case of out-of-tree builds. Defaults to the part source directory.
- `CRAFT_PART_INSTALL`: The path to the part install directory. This is where built artifacts are installed after the `BUILD` step.
- `CRAFT_OVERLAY`: The path to the part's layer directory during the `OVERLAY` step if overlays are enabled.
- `CRAFT_STAGE`: The path to the project's staging directory. This is where installed artifacts are migrated after the `STAGE` step.

- `CRAFT_PRIME`: The path to the final primed payload directory after the `PRIME` step.

CHAPTER

SIX

ACTIONS

Actions are the execution units needed to move the project state to a given step through the parts lifecycle. The action behavior is determined by its action type.

CHAPTER

SEVEN

PROJECT INFORMATION

Project parameters are provided to callback functions by passing an instance of the `StepInfo` class. It consolidates properties from classes `PartInfo`, `ProjectInfo` and `ProjectDirs`, including custom application-specific parameters passed as keyword arguments when instantiating `LifecycleManager`.

CHAPTER

EIGHT

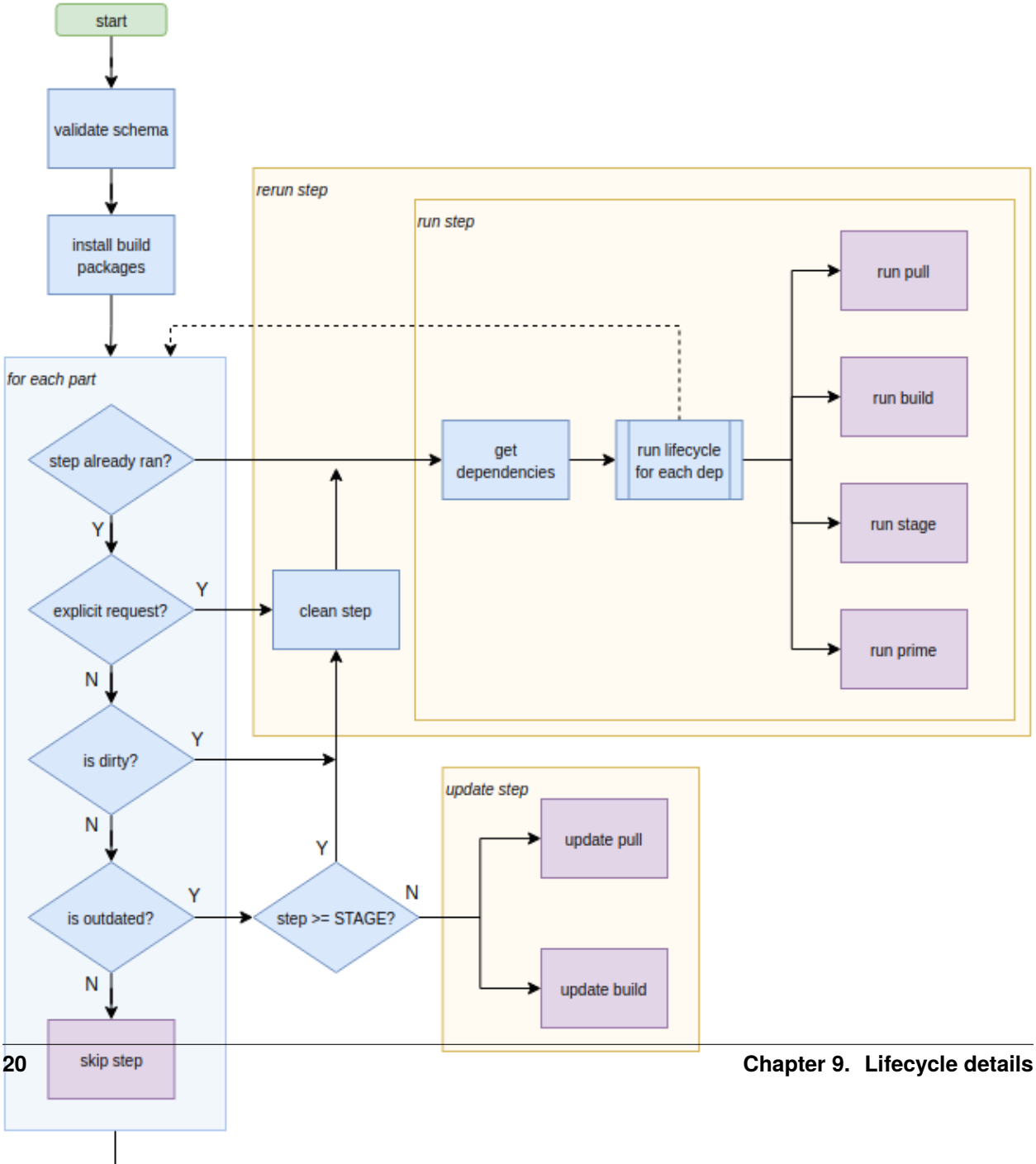
EXCEPTIONS

CHAPTER

NINE

LIFECYCLE DETAILS

9.1 Lifecycle processing diagram



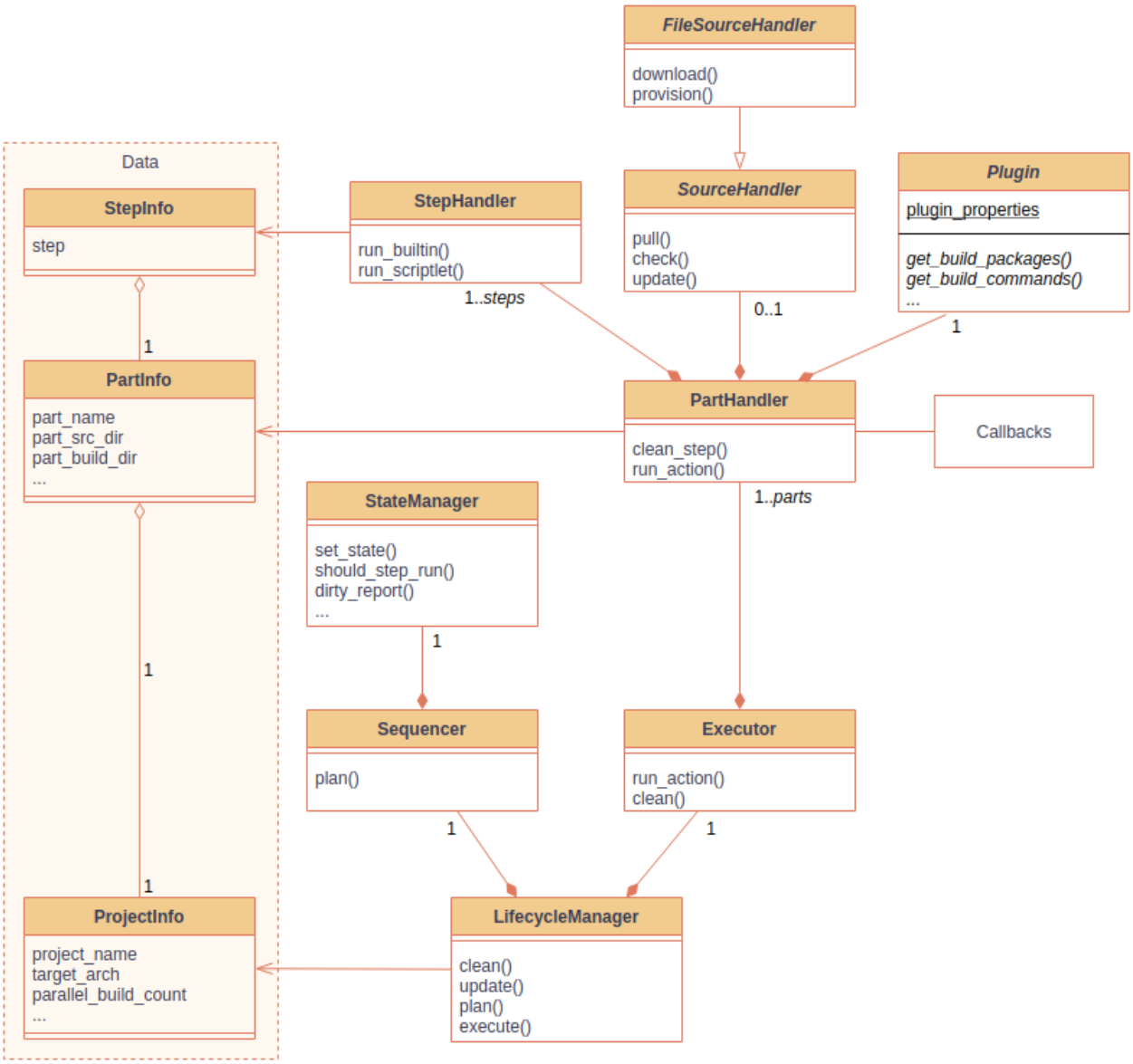
CHAPTER

TEN

IMPLEMENTATION NOTES

10.1 Class layout

The implementation reflects the two main lifecycle processing operations. All planning is done by the `Sequencer` class based on the parts definition and existing state loaded from disk. Execution of planned actions is handled by the `Executor` class.


FileSourceHandler
download()
provision()
Data
StepInfo
step
StepHandler
run_builtin()
run_scriptlet()
1..steps
SourceHandler
pull()
check()
update()
0..1
Plugin
plugin_properties
get_build_packages()
get_build_commands()
...
1
PartInfo
part_name
part_src_dir
part_build_dir
...
PartHandler
clean_step()
run_action()
Callbacks
1..parts
StateManager
set_state()
should_step_run()
dirty_report()
...
1
1
Sequencer
plan()
Executor
run_action()
clean()
1
1
1
ProjectInfo
project_name
target_arch
parallel_build_count
...
LifecycleManager
clean()
update()
plan()
execute()

CHAPTER

ELEVEN

PACKAGE REFERENCE

11.1 craft_parts package

11.1.1 Subpackages

craft_parts.executor package

Submodules

craft_parts.executor.collisions module

craft_parts.executor.environment module

craft_parts.executor.executor module

craft_parts.executor.filesets module

craft_parts.executor.migration module

craft_parts.executor.organize module

craft_parts.executor.part_handler module

craft_parts.executor.step_handler module

Module contents

craft_parts.overlays package

Submodules

craft_parts.overlays.chroot module

craft_parts.overlays.errors module

craft_parts.overlays.layers module

craft_parts.overlays.overlay_fs module

craft_parts.overlays.overlay_manager module

craft_parts.overlays.overlays module

Module contents

craft_parts.packages package

Submodules

craft_parts.packages.apt_cache module

craft_parts.packages.base module

craft_parts.packages.deb module

craft_parts.packages.deb_package module

craft_parts.packages.errors module

craft_parts.packages.normalize module

craft_parts.packages.platform module

craft_parts.packages.snaps module

Module contents

craft_parts.plugins package

Submodules

craft_parts.plugins.autotools_plugin module

craft_parts.plugins.base module

craft_parts.plugins.cmake_plugin module

craft_parts.plugins.dotnet_plugin module

craft_parts.plugins.dump_plugin module

craft_parts.plugins.go_plugin module

craft_parts.plugins.make_plugin module

craft_parts.plugins.meson_plugin module

craft_parts.plugins.nil_plugin module

craft_parts.plugins.npm_plugin module

craft_parts.plugins.plugins module

craft_parts.plugins.properties module

Definitions and helpers for plugin options.

class craft_parts.plugins.properties.**PluginProperties**

Bases: `object`

Options specific to a plugin.

PluginProperties should be subclassed into plugin-specific property classes and populated from a dictionary containing part properties.

classmethod unmarshal(*data*)

Populate class attributes from the part specification.

Parameters **data** (`Dict[str, Any]`) – A dictionary containing part properties.

Return type *PluginProperties*

Returns The populated plugin properties data object.

craft_parts.plugins.python_plugin module

craft_parts.plugins.rust_plugin module

craft_parts.plugins.validator module

Module contents

craft_parts.sources package

Submodules

craft_parts.sources.base module

craft_parts.sources.cache module

craft_parts.sources.checksum module

craft_parts.sources.deb_source module

craft_parts.sources.errors module

craft_parts.sources.git_source module

craft_parts.sources.local_source module

craft_parts.sources.snap_source module

craft_parts.sources.sources module

craft_parts.sources.tar_source module

craft_parts.sources.zip_source module

Module contents

craft_parts.state_manager package

Submodules

craft_parts.state_manager.build_state module

craft_parts.state_manager.overlay_state module

craft_parts.state_manager.prime_state module

craft_parts.state_manager.pull_state module

craft_parts.state_manager.reports module

craft_parts.state_manager.stage_state module

craft_parts.state_manager.state_manager module

craft_parts.state_manager.states module

craft_parts.state_manager.step_state module

Module contents

craft_parts.utils package

Submodules

craft_parts.utils.deb_utils module

craft_parts.utils.file_utils module

craft_parts.utils.formatting_utils module

craft_parts.utils.os_utils module

craft_parts.utils.url_utils module

Module contents

11.1.2 Submodules

craft_parts.actions module

craft_parts.callbacks module

craft_parts.ctl module

craft_parts.dirs module

craft_parts.errors module

craft_parts.infos module

craft_parts.lifecycle_manager module

craft_parts.main module

craft_parts.parts module

craft_parts.sequencer module

craft_parts.steps module

craft_parts.xattrs module

11.1.3 Module contents

CHAPTER

TWELVE

CHANGELOG

12.1 1.13.0 (2022-09-05)

- Add go generate support to go plugin
- Add support for deb sources
- Add source download request timeout
- Remove unnecessary overlay whiteout files

12.2 1.12.1 (2022-08-19)

- Revert changes to install prefix in cmake plugin to prevent stable base incompatibilities

12.3 1.12.0 (2022-08-12)

- Set install prefix in the cmake plugin
- Fix prefix path in the cmake plugin

12.4 1.11.0 (2022-08-12)

- Add API call to list registered plugins

12.5 1.10.2 (2022-08-03)

- Fix git source format error when cloning using depth
- Use host architecture when installing stage packages

12.6 1.10.1 (2022-07-29)

- Change staged snap pkgconfig prefix normalization to be predictable regardless of the path used for destructive mode packing

12.7 1.10.0 (2022-07-28)

- Add plugin class method to check for out of source builds
- Normalize file copy functions signatures
- Fix pkgconfig prefix in staged snaps

12.8 1.9.0 (2022-07-14)

- Prevent wildcard symbol conflict in stage and prime filters
- Apt installer changed to collect installed package versions after the installation

12.9 1.8.1 (2022-07-05)

- Fix execution of empty scriptlets
- List primed stage packages only if deb stage packages are defined

12.10 1.8.0 (2022-06-30)

- Add list of primed stage packages to prime state
- Add lifecycle manager methods to obtain pull state assets and the list of primed stage packages

12.11 1.7.2 (2022-06-14)

- Fix git repository updates
- Fix stage packages removal on build update

12.12 1.7.1 (2022-05-21)

- Fix stdout leak during snap package installation
- Fix plugin validation dependencies

12.13 1.7.0 (2022-05-20)

- Add support for application-defined environment variables
- Add package filter for core22
- Refresh packages list before installing packages
- Expand global variables in parts definition
- Adjust prologue/epilogue callback parameters
- Make plugin options available in plugin environment validator
- Fix readthedocs documentation generation

12.14 1.6.1 (2022-05-02)

- Fix stage package symlink normalization

12.15 1.6.0 (2022-04-29)

- Add zip source handler
- Clean up source provisioning
- Fix project variable setting for skipped parts

12.16 1.5.1 (2022-04-25)

- Fix extra build snaps installation

12.17 1.5.0 (2022-04-25)

- Add rust plugin
- Add npm plugin
- Add project name argument to LifecycleManager and set `CRAFT_PROJECT_NAME`
- Export symbols needed by application-defined plugins
- Refactor plugin environment validation

12.18 1.4.2 (2022-04-01)

- Fix craftctl error handling
- Fix long recursions in dirty step verification

12.19 1.4.1 (2022-03-30)

- Fix project variable adoption scope

12.20 1.4.0 (2022-03-24)

- Add cmake plugin
- Mount overlays using fuse-overlayfs
- Send execution output to user-specified streams
- Update craftctl commands
- Update step execution environment variables

12.21 1.3.0 (2022-03-05)

- Add meson plugin
- Adjustments in git source tests

12.22 1.2.0 (2022-03-01)

- Make git submodules fetching configurable
- Fix source type specification
- Fix testing in Python 3.10
- Address issues found by linters

12.23 1.1.2 (2022-02-07)

- Do not refresh already installed snaps
- Fix URL in setup.py
- Fix pydantic validation error handling
- Unpin pydantic and pydantic-yaml dependency versions
- Unpin pylint dependency version
- Remove unused requirements files

12.24 1.1.1 (2022-01-05)

- Pin pydantic and pydantic-yaml dependency versions

12.25 1.1.0 (2021-12-08)

- Add support to overlay step
- Use bash as step scriptlet interpreter
- Add plugin environment validation
- Add go plugin
- Add dotnet plugin

12.26 1.0.4 (2021-11-10)

- Declare additional public API names
- Add git source handler

12.27 1.0.3 (2021-10-19)

- Properly declare public API names
- Allow non-snap applications running on non-apt systems to invoke parts processing on build providers
- Use Bash as script interpreter instead of /bin/sh to stay compatible with Snapcraft V2 plugins

12.28 1.0.2 (2021-09-16)

- Fix local source updates causing removal of build artifacts and new files created in `override-pull`

12.29 1.0.1 (2021-09-13)

- Fix plugin properties test
- Use local copy of mutable source handler ignore patterns
- Use host state for apt cache and remove stage package refresh
- Add information to parts error in CLI tool
- Change CLI tool `--debug` option to `--trace` to be consistent with craft tools

12.30 1.0.0 (2021-08-05)

- Initial release

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

INDEX