Craft Parts

Release 1.18.0

Canonical Ltd.

Jan 20, 2023

CONTENTS

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

TUTORIALS

If you want to learn the basics from experience, then our tutorials will help you acquire the necessary competencies from real-life examples with fully reproducible steps.

1.1 Adding parts processing to an application

1.1.1 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
```

(continues on next page)

(continued from previous page)

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

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

HOW-TO GUIDES

2.1 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.

2.1.1 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
```

(continues on next page)

(continued from previous page)

```
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
...
```

2.1.2 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*.

2.2 Using craft-parts from the command line

2.2.1 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

THREE

REFERENCE

3.1 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.

class craft_parts.**LifecycleManager**(*all_parts*, ***, *application_name*, *cache_dir*, *work_dir='.'*, *arch=''*, *base=''*, *project_name=None*, *parallel_build_count=1*, *application_package_name=None*, *ignore_local_sources=None*, *extra_build_packages=None*, *extra_build_snaps=None*, *track_stage_packages=False*, *base_layer_dir=None*, *base_layer_hash=None*, *project_vars_part_name=None*, *project_vars=None*, ***custom_args*)

Coordinate the planning and execution of the parts lifecycle.

The lifecycle manager determines the list of actions that needs be executed in order to obtain a tree of installed files from the specification on how to process its parts, and provides a mechanism to execute each of these actions.

Parameters

- **all_parts** (Dict[str, Any]) – A dictionary containing the parts specification according to the parts schema. The format is compatible with the output generated by PyYAML's yaml.load.
- **application_name** (str) – A unique non-empty identifier for the application using Craft Parts. Valid application names contain upper and lower case letters, underscores or numbers, and must start with a letter.
- **project_name** (Optional[str]) – name of the project being built.
- **cache_dir** (Union[Path, str]) – The path to store cached packages and files. If not specified, a directory under the application name entry in the XDG base directory will be used.
- **work_dir** (Union[Path, str]) – The toplevel directory for work directories. The current directory will be used if none is specified.
- **arch** (str) – The architecture to build for. Defaults to the host system architecture.
- **base** (str) – The system base the project being processed will run on. Defaults to the system where Craft Parts is being executed.
- **parallel_build_count** (int) – The maximum number of concurrent jobs to be used to build each part of this project.
- **application_package_name** (Optional[str]) – The name of the application package, if required by the package manager used by the platform. Defaults to the application name.

- **ignore_local_sources** (Optional[List[str]]) – A list of local source patterns to ignore.
- **extra_build_packages** (Optional[List[str]]) – A list of additional build packages to install.
- **extra_build_snaps** (Optional[List[str]]) – A list of additional build snaps to install.
- **track_stage_packages** (bool) – Add primed stage packages to the prime state.
- **base_layer_dir** (Optional[Path]) – The path to the overlay base layer, if using overlays.
- **base_layer_hash** (Optional[bytes]) – The validation hash of the overlay base image, if using overlays. The validation hash should be constant for a given image, and should change if a different base image is used.
- **project_vars_part_name** (Optional[str]) – Project variables can only be set in the part matching this name.
- **project_vars** (Optional[Dict[str, str]]) – A dictionary containing project variables.
- **custom_args** (Any) – Any additional arguments that will be passed directly to callbacks.

action_executor()

Return a context manager for action execution.

Return type
ExecutionContext

clean(*step=Step.PULL*, ***, *part_names=None*)

Clean the specified step and parts.

Cleaning a step removes its state and all artifacts generated in that step and subsequent steps for the specified parts.

Parameters

- **step** (*Step*) – The step to clean. If not specified, all steps will be cleaned.
- **part_names** (Optional[List[str]]) – The list of part names to clean. If not specified, all parts will be cleaned and work directories will be removed.

Return type
None

get_primed_stage_packages(***, *part_name*)

Return the list of primed stage packages.

Parameters
part_name (str) – The name of the part to get primed stage packages from.

Return type
Optional[List[str]]

Returns
The sorted list of primed stage packages, or None if no state found.

get_pull_assets(***, *part_name*)

Return the part's pull state assets.

Parameters
part_name (str) – The name of the part to get assets from.

Return type
Optional[Dict[str, Any]]

Returns
The dictionary of the part's pull assets, or None if no state found.

plan(*target_step*, *part_names=None*)

Obtain the list of actions to be executed given the target step and parts.

Parameters

- **target_step** (*Step*) – The final step we want to reach.
- **part_names** (Optional[Sequence[str]]) – The list of parts to process. If not specified, all parts will be processed.
- **update** – Refresh the list of available packages.

Return type
List[*Action*]

Returns
The list of *Action* objects that should be executed in order to reach the target step for the specified parts.

property project_info: *ProjectInfo*

Obtain information about this project.

Return type
ProjectInfo

refresh_packages_list()

Update the available packages list.

The list of available packages should be updated before planning the sequence of actions to take. To ensure consistency between the scenarios, it shouldn't be updated between planning and execution.

Return type
None

reload_state()

Reload the ephemeral state from disk.

Return type
None

3.2 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).

3.2.1 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.

Permissions

Parts can declare read/write/execute permissions and ownership for the files they produce. This is achieved by adding a `permissions` subkey in the specific part:

```
# ...
parts:
  my-part:
    # ...
    permissions:
      - path: bin/my-binary
        owner: 1111
        group: 2222
        mode: "755"
```

The `permissions` subkey is a list of permissions definitions, each with the following keys:

- `path`: a string describing the file(s) and dir(s) that this definition applies to. The path should be relative, and supports wildcards. This field is *optional* and its absence is equivalent to `"*"`, meaning that the definition applies to all files produced by the part;
- `owner`: an integer describing the numerical id of the owner of the files. This field is *optional* in the general case but *mandatory* if `group` is specified;
- `group`: an integer describing the numerical id of the group for the files. The semantics are otherwise the same as `owner`, including being *optional* in the general case and *mandatory* if `owner` is specified;
- `mode`: string describing the desired permissions for the files as a number in base 8. This field is *optional*.

3.2.2 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`.

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.

3.3 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.

class craft_parts.**ActionType**(*value*)

The type of action to be executed.

Action execution can be modified according to its type:

`RUN`: execute the expected commands for step processing.

`RERUN`: clear the existing data and state before procceeding.

`UPDATE`: try to continue processing the step.

`SKIP`: don't execute this action.

`REAPPLY`: run the step commands without updating its state.

RUN = 0

RERUN = 1

SKIP = 2

UPDATE = 3

class craft_parts.**Action**(*part_name*, *step*, *action_type=ActionType.RUN*, *reason=None*, *project_vars=None*, *properties=ActionProperties(changed_files=None, changed_dirs=None)*)

The action to be executed for a given part.

Actions correspond to the operations required to run the lifecycle for each of the parts in the project specification.

Parameters

- **part_name** (`str`) – The name of the part this action will be performed on.
- **step** (*Step*) – The *Step* this action will execute.
- **action_type** (*ActionType*) – Action to run for this step.
- **reason** (`Optional[str]`) – A textual description of why this action should be executed.
- **project_vars** (`Optional[Dict[str, ProjectVar]]`) – The values of project variables from a previous execution, used if the action type is `ActionType.SKIP`.
- **properties** (*ActionProperties*) –

3.4 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*.

class craft_parts.**ProjectDirs**(*, *work_dir='.'*)

The project's main work directories.

Parameters

work_dir (Union[Path, str]) – The parent directory containing the parts, prime and stage subdirectories. If not specified, the current directory will be used.

Variables

- **work_dir** – The root of the work directories used for project processing.
- **parts_dir** – The directory containing work subdirectories for each part.
- **overlay_dir** – The directory containing work subdirectories for overlays.
- **overlay_mount_dir** – The mountpoint for the overlay filesystem.
- **overlay_packages_dir** – The cache directory for overlay packages.
- **overlay_work_dir** – The work directory for the overlay filesystem.
- **stage_dir** – The staging area containing installed files from all parts.
- **prime_dir** – The primed tree containing the final artifacts to deploy.

class craft_parts.**ProjectInfo**(*, *application_name*, *cache_dir*, *arch=''*, *base=''*, *parallel_build_count=1*, *project_dirs=None*, *project_name=None*, *project_vars_part_name=None*, *project_vars=None*, ***custom_args*)

Project-level information containing project-specific fields.

Parameters

- **application_name** (str) – A unique identifier for the application using Craft Parts.
- **project_name** (Optional[str]) – name of the project being built.
- **cache_dir** (Path) – The path to store cached packages and files. If not specified, a directory under the application name entry in the XDG base directory will be used.
- **arch** (str) – The architecture to build for. Defaults to the host system architecture.
- **parallel_build_count** (int) – The maximum number of concurrent jobs to be used to build each part of this project.
- **project_dirs** (Optional[*ProjectDirs*]) – The project work directories.
- **project_name** – The name of the project.
- **project_vars_part_name** (Optional[str]) – Project variables can be set only if the part name matches this name.
- **project_vars** (Optional[Dict[str, str]]) – A dictionary containing the project variables.
- **custom_args** (Any) – Any additional arguments defined by the application when creating a *LifecycleManager*.
- **base** (str) –

property application_name: str

Return the name of the application using craft-parts.

Return type
str

property arch_triplet: str

Return the machine-vendor-os platform triplet definition.

Return type
str

property base: str

Return the project build base.

Return type
str

property cache_dir: Path

Return the directory used to store cached files.

Return type
Path

property custom_args: List[str]

Return the list of custom argument names.

Return type
List[str]

property dirs: *ProjectDirs*

Return the project's work directories.

Return type
ProjectDirs

get_project_var(*name*, *, *raw_read=False*)

Get the value of a project variable.

Variables must be consumed by the application only after the lifecycle execution ends to prevent unexpected behavior if steps are skipped.

Parameters

- **name** (str) – The project variable name.
- **raw_read** (bool) – Whether the variable is read without access verifications.

Return type
str

Returns
The value of the variable.

Raises

- **ValueError** – If there is no project variable with the given name.
- **RuntimeError** – If the variable is consumed during the lifecycle execution.

property host_arch: str

Return the host architecture used for debs, snaps and charms.

Return type
str

property is_cross_compiling: bool

Whether the target and host architectures are different.

Return type
bool

property parallel_build_count: int

Return the maximum allowable number of concurrent build jobs.

Return type
int

property project_name: Optional[str]

Return the name of the project using craft-parts.

Return type
Optional[str]

property project_options: Dict[str, Any]

Obtain a project-wide options dictionary.

Return type
Dict[str, Any]

set_project_var(*name*, *value*, *raw_write=False*, ***, *part_name=None*)

Set the value of a project variable.

Variable values can be set once. Project variables are not intended for logic construction in user scripts, setting it multiple times is likely to be an error.

Parameters

- **name** (str) – The project variable name.
- **value** (str) – The new project variable value.
- **part_name** (Optional[str]) – If not None, variable setting is restricted to the named part.
- **raw_write** (bool) – Whether the variable is written without access verifications.

Raises

- **ValueError** – If there is no custom argument with the given name.
- **RuntimeError** – If a write-once variable is set a second time, or if a part name is specified and the variable is set from a different part.

Return type
None

property target_arch: str

Return the target architecture used for debs, snaps and charms.

Return type
str

class craft_parts.**PartInfo**(*project_info*, *part*)

Part-level information containing project and part fields.

Parameters

- **project_info** (*ProjectInfo*) – The project information.
- **part** (*Part*) – The part we want to obtain information from.

get_project_var(*name*, *, *raw_read=False*)

Get the value of a project variable.

Variables must be consumed by the application only after the lifecycle execution ends to prevent unexpected behavior if steps are skipped.

Parameters

- **name** (str) – The project variable name.
- **raw_read** (bool) – Whether the variable is read without access verifications.

Return type
str

Returns
The value of the variable.

Raises

- **ValueError** – If there is no project variable with the given name.
- **RuntimeError** – If the variable is consumed during the lifecycle execution.

property part_build_dir: Path

Return the subdirectory containing the part's build tree.

Return type
Path

property part_build_subdir: Path

Return the subdirectory in build containing the source subtree (if any).

Return type
Path

property part_install_dir: Path

Return the subdirectory to install the part's build artifacts.

Return type
Path

property part_name: str

Return the name of the part we're providing information about.

Return type
str

property part_src_dir: Path

Return the subdirectory containing the part's source code.

Return type
Path

property part_src_subdir: Path

Return the subdirectory in source containing the source subtree (if any).

Return type
Path

property part_state_dir: Path

Return the subdirectory containing this part's lifecycle state.

Return type
Path

property project_info: *ProjectInfo*

Return the project information.

Return type
ProjectInfo

set_project_var(*name*, *value*, *, *raw_write=False*)

Set the value of a project variable.

Variable values can be set once. Project variables are not intended for logic construction in user scripts, setting it multiple times is likely to be an error.

Parameters

- **name** (str) – The project variable name.
- **value** (str) – The new project variable value.
- **raw_write** (bool) – Whether the variable is written without access verifications.

Raises

- **ValueError** – If there is no custom argument with the given name.
- **RuntimeError** – If a write-once variable is set a second time, or if a part name is specified and the variable is set from a different part.

Return type
None

class craft_parts.**StepInfo**(*part_info*, *step*)

Step-level information containing project, part, and step fields.

Parameters

- **part_info** (*PartInfo*) – The part information.
- **step** (*Step*) – The step we want to obtain information from.

3.5 Exceptions

Craft parts errors.

exception craft_parts.errors.**CallbackRegistrationError**(*message*)

Bases: *PartsError*

Error in callback function registration.

Parameters
message (str) – the error message.

exception craft_parts.errors.**CopyFileNotFound**(*name*)

Bases: *PartsError*

An attempt was made to copy a file that doesn't exist.

Parameters
name (str) – The file name.

exception craft_parts.errors.**CopyTreeError**(*message*)

Bases: *PartsError*

Failed to copy or link a file tree.

Parameters
message (str) – The error message.

exception craft_parts.errors.**DebError**(*deb_path*, *command*, *exit_code*)

Bases: *PartsError*

A "deb"-related command failed.

Parameters

- **deb_path** (Path) –
- **command** (List[str]) –
- **exit_code** (int) –

exception craft_parts.errors.**FileOrganizeError**(*, *part_name*, *message*)

Bases: *PartsError*

Failed to organize a file layout.

Parameters

- **part_name** (str) – The name of the part being processed.
- **message** (str) – The error message.

exception craft_parts.errors.**FilesetConflict**(*conflicting_files*)

Bases: *PartsError*

Inconsistent stage to prime filtering.

Parameters
conflicting_files (Set[str]) – A set containing the conflicting file names.

exception craft_parts.errors.**FilesetError**(*, *name*, *message*)

Bases: *PartsError*

An invalid fileset operation was performed.

Parameters

- **name** (str) – The name of the fileset.
- **message** (str) – The error message.

exception craft_parts.errors.**InvalidAction**(*message*)

Bases: *PartsError*

An attempt was made to execute an action with invalid parameters.

Parameters
message (str) – The error message.

exception craft_parts.errors.**InvalidApplicationName**(*name*)

Bases: *PartsError*

The application name contains invalid characters.

Parameters
name (str) – The invalid application name.

exception craft_parts.errors.**InvalidArchitecture**(*arch_name*)

Bases: *PartsError*

The machine architecture is not supported.

Parameters
arch_name (str) – The unsupported architecture name.

exception craft_parts.errors.**InvalidControlAPICall**(*, *part_name*, *scriptlet_name*, *message*)

Bases: *PartsError*

A control API call was made with invalid parameters.

Parameters

- **part_name** (str) – The name of the part being processed.
- **scriptlet_name** (str) – The name of the scriptlet that originated the call.
- **message** (str) – The error message.

exception craft_parts.errors.**InvalidPartName**(*part_name*)

Bases: *PartsError*

An operation was requested on a part that's in the parts specification.

Parameters
part_name (str) – The invalid part name.

exception craft_parts.errors.**InvalidPlugin**(*plugin_name*, *, *part_name*)

Bases: *PartsError*

A request was made to use a plugin that's not registered.

Parameters

- **plugin_name** (str) – The invalid plugin name."
- **part_name** (str) – The name of the part defining the invalid plugin.

exception craft_parts.errors.**OsReleaseCodenameError**

Bases: *PartsError*

Failed to determine the host operating system version codename.

exception craft_parts.errors.**OsReleaseIdError**

Bases: *PartsError*

Failed to determine the host operating system identification string.

exception craft_parts.errors.**OsReleaseNameError**

Bases: *PartsError*

Failed to determine the host operating system name.

exception craft_parts.errors.**OsReleaseVersionIdError**

Bases: *PartsError*

Failed to determine the host operating system version.

exception craft_parts.errors.**OverlayPackageNotFound**(*, *part_name*, *package_name*)

Bases: *PartsError*

Failed to install an overlay package.

Parameters

- **part_name** (str) – The name of the part being processed.
- **message** – the error message.
- **package_name** (str) –

exception craft_parts.errors.**OverlayPermissionError**

Bases: *PartsError*

A project using overlays was processed by a non-privileged user.

exception craft_parts.errors.**OverlayPlatformError**

Bases: *PartsError*

A project using overlays was processed on a non-Linux platform.

exception craft_parts.errors.**PartDependencyCycle**

Bases: *PartsError*

A dependency cycle has been detected in the parts definition.

exception craft_parts.errors.**PartFilesConflict**(*, *part_name*, *other_part_name*, *conflicting_files*)

Bases: *PartsError*

Different parts list the same files with different contents.

Parameters

- **part_name** (str) – The name of the part being processed.
- **other_part_name** (str) – The name of the conflicting part.
- **conflicting_files** (List[str]) – The list of conflicting files.

exception craft_parts.errors.**PartSpecificationError**(*, *part_name*, *message*)

Bases: *PartsError*

A part was not correctly specified.

Parameters

- **part_name** (str) – The name of the part being processed.
- **message** (str) – The error message.

classmethod **from_validation_error**(*, *part_name*, *error_list*)

Create a PartSpecificationError from a pydantic error list.

Parameters

- **part_name** (str) – The name of the part being processed.
- **error_list** (List[ErrorDict]) – A list of dictionaries containing pydantic error definitions.

Return type
PartSpecificationError

exception craft_parts.errors.**PartsError**(*brief*, *details=None*, *resolution=None*)

Bases: Exception

Unexpected error.

Parameters

- **brief** (str) – Brief description of error.
- **details** (Optional[str]) – Detailed information.
- **resolution** (Optional[str]) – Recommendation, if any.

exception craft_parts.errors.**PluginBuildError**(*, *part_name*)

Bases: *PartsError*

Plugin build script failed at runtime.

Parameters
part_name (str) – The name of the part being processed.

exception craft_parts.errors.**PluginEnvironmentValidationError**(*, *part_name*, *reason*)

Bases: *PartsError*

Plugin environment validation failed at runtime.

Parameters

- **part_name** (str) – The name of the part being processed.
- **reason** (str) –

exception craft_parts.errors.**ScriptletRunError**(*, *part_name*, *scriptlet_name*, *exit_code*)

Bases: *PartsError*

A scriptlet execution failed.

Parameters

- **part_name** (str) – The name of the part being processed.
- **scriptlet_name** (str) – The name of the scriptlet that failed to execute.
- **exit_code** (int) – The execution error code.

exception craft_parts.errors.**StageFilesConflict**(*, *part_name*, *conflicting_files*)

Bases: *PartsError*

Files from a part conflict with files already being staged.

Parameters

- **part_name** (str) – The name of the part being processed.
- **conflicting_files** (List[str]) – The list of confictling files.

exception craft_parts.errors.**StagePackageNotFound**(*, *part_name*, *package_name*)

Bases: *PartsError*

Failed to install a stage package.

Parameters

- **part_name** (str) – The name of the part being processed.

- **package_name** (str) – The name of the package.

exception craft_parts.errors.**UndefinedPlugin**(*, *part_name*)

Bases: *PartsError*

The part didn't define a plugin and the part name is not a valid plugin name.

Parameters

part_name (str) – The name of the part with no plugin definition.

exception craft_parts.errors.**XAttributeError**(*key*, *path*, *is_write=False*)

Bases: *PartsError*

Failed to read or write an extended attribute.

Parameters

- **action** – The action being performed.
- **key** (str) – The extended attribute key.
- **path** (str) – The file path.
- **is_write** (bool) – Whether this is an attribute write operation.

exception craft_parts.errors.**XAttributeTooLong**(*key*, *value*, *path*)

Bases: *PartsError*

Failed to write an extended attribute because key and/or value is too long.

Parameters

- **key** (str) – The extended attribute key.
- **value** (str) – The extended attribute value.
- **path** (str) – The file path.

Source handler error definitions.

exception craft_parts.sources.errors.**ChecksumMismatch**(*, *expected*, *obtained*)

Bases: *SourceError*

A checksum doesn't match the expected value.

Parameters

- **expected** (str) – The expected checksum.
- **obtained** (str) – The actual checksum.

exception craft_parts.sources.errors.**IncompatibleSourceOptions**(*source_type*, *options*)

Bases: *SourceError*

Source specified options that cannot be used at the same time.

Parameters

- **source_type** (str) – The part's source type.
- **options** (List[str]) – The list of incompatible source options.

exception craft_parts.sources.errors.**InvalidSnapPackage**(*snap_file*)

Bases: *SourceError*

A snap package is invalid.

Parameters
snap_file (str) – The snap file name.

exception craft_parts.sources.errors.**InvalidSourceOption**(*, *source_type*, *option*)

Bases: *SourceError*

A source option is not allowed for the given source type.

Parameters

- **source_type** (str) – The part's source type.
- **option** (str) – The invalid source option.

exception craft_parts.sources.errors.**InvalidSourceType**(*source*)

Bases: *SourceError*

Failed to determine a source type.

Parameters
source (str) – The source defined for the part.

exception craft_parts.sources.errors.**NetworkRequestError**(*message*)

Bases: *SourceError*

A network request operation failed.

Parameters
message (str) – The error message.

exception craft_parts.sources.errors.**PullError**(*, *command*, *exit_code*)

Bases: *SourceError*

Failed pulling source.

Parameters

- **command** (Sequence) – The command used to pull the source.
- **exit_code** (int) – The command exit code.

exception craft_parts.sources.errors.**SourceError**(*brief*, *details=None*, *resolution=None*)

Bases: *PartsError*

Base class for source handler errors.

Parameters

- **brief** (str) –
- **details** (Optional[str]) –
- **resolution** (Optional[str]) –

exception craft_parts.sources.errors.**SourceNotFound**(*source*)

Bases: *SourceError*

Failed to retrieve a source.

Parameters
source (str) – The source defined for the part.

exception craft_parts.sources.errors.**SourceUpdateUnsupported**(*name*)

Bases: *SourceError*

The source handler doesn't support updating.

Parameters
name (str) – The source type.

exception craft_parts.sources.errors.**VCSError**(*message*)

Bases: *SourceError*

A version control system command failed.

Parameters
message (str) –

Exceptions raised by the packages handling subsystem.

exception craft_parts.packages.errors.**BuildPackageNotFound**(*package*)

Bases: *PackagesError*

A package listed in 'build-packages' was not found.

Parameters
package (str) – The name of the missing package.

exception craft_parts.packages.errors.**BuildPackagesNotInstalled**(*, *packages*)

Bases: *PackagesError*

Could not install all requested build packages.

Parameters
packages (Sequence[str]) – The packages to install.

exception craft_parts.packages.errors.**ChiselError**(*, *slices*, *output*)

Bases: *PackagesError*

A "chisel"-related command failed.

Parameters

- **slices** (List[str]) –
- **output** (str) –

exception craft_parts.packages.errors.**FileProviderNotFound**(*, *file_path*)

Bases: *PackagesError*

A file is not provided by any package.

Parameters
file_path (str) – The file path.

exception craft_parts.packages.errors.**PackageBackendNotSupported**(*backend*)

Bases: *PartsError*

Requested package resolved not supported on this host.

Parameters
backend (str) –

exception craft_parts.packages.errors.**PackageBroken**(*package_name*, *, *deps*)

Bases: *PackagesError*

Package has unmet dependencies.

Parameters

- **package_name** (str) – The name of the package with unmet dependencies.
- **deps** (Sequence[str]) – The list of unmet dependencies.

exception craft_parts.packages.errors.**PackageFetchError**(*message*)

Bases: *PackagesError*

Failed to fetch package from remote repository.

Parameters

message (str) – The error message.

exception craft_parts.packages.errors.**PackageListRefreshError**(*message*)

Bases: *PackagesError*

Failed to refresh the list of available packages.

Parameters

message (str) – The error message.

exception craft_parts.packages.errors.**PackageNotFound**(*package_name*)

Bases: *PackagesError*

Requested package doesn't exist in the remote repository.

Parameters

package_name (str) – The name of the missing package.

exception craft_parts.packages.errors.**PackagesDownloadError**(*, *packages*)

Bases: *PackagesError*

Failed to download packages from remote repository.

Parameters

packages (Sequence[str]) – The packages to download.

exception craft_parts.packages.errors.**PackagesError**(*brief*, *details=None*, *resolution=None*)

Bases: *PartsError*

Base class for package handler errors.

Parameters

- **brief** (str) –
- **details** (Optional[str]) –
- **resolution** (Optional[str]) –

exception craft_parts.packages.errors.**PackagesNotFound**(*packages*)

Bases: *PackagesError*

Requested package doesn't exist in the remote repository.

Parameters

- **package_name** – The names of the missing packages.
- **packages** (Sequence[str]) –

exception craft_parts.packages.errors.**SnapDownloadError**(*, *snap_name*, *snap_channel*)

Bases: *PackagesError*

Failed to download a snap.

Parameters

- **snap_name** (str) – The snap name.
- **snap_channel** (str) – The snap channel.

exception craft_parts.packages.errors.**SnapGetAssertionError**(*, *assertion_params*)

Bases: *PackagesError*

Failed to retrieve snap assertion.

Parameters

assertion_params (Sequence[str]) – The snap assertion parameters.

exception craft_parts.packages.errors.**SnapInstallError**(*, *snap_name*, *snap_channel*)

Bases: *PackagesError*

Failed to install a snap.

Parameters

- **snap_name** (str) – The snap name.
- **snap_channel** (str) – The snap channel.

exception craft_parts.packages.errors.**SnapRefreshError**(*, *snap_name*, *snap_channel*)

Bases: *PackagesError*

Failed to refresh a snap.

Parameters

- **snap_name** (str) – The snap name.
- **snap_channel** (str) – The snap channel.

exception craft_parts.packages.errors.**SnapUnavailable**(*, *snap_name*, *snap_channel*)

Bases: *PackagesError*

Failed to install or refresh a snap.

Parameters

- **snap_name** (str) – The snap name.
- **snap_channel** (str) – The snap channel.

exception craft_parts.packages.errors.**SnapdConnectionError**(*, *snap_name*, *url*)

Bases: *PackagesError*

Failed to connect to snapd.

Parameters

- **snap_name** (str) – The snap name.
- **url** (str) – The failed connection URL.

exception craft_parts.packages.errors.**UnpackError**(*package*)

Bases: *PackagesError*

Error unpacking stage package.

Parameters
package (str) – The package that failed to unpack.

3.6 Package reference

3.6.1 craft_parts package

Subpackages

craft_parts.executor package

Submodules

craft_parts.executor.collisions module

Helpers to detect conflicting staging files from multiple parts.

craft_parts.executor.collisions.**check_for_stage_collisions**(*part_list*)

Verify whether parts have conflicting files to stage.

Parameters
part_list (List[*Part*]) – The list of parts to be tested.

Raises
PartConflictError – If conflicts are found.

Return type
None

craft_parts.executor.collisions.**paths_collide**(*path1*, *path2*, *permissions_path1=None*, *permissions_path2=None*)

Check whether the provided paths conflict to each other.

If both paths have Permissions definitions, they are considered to be conflicting if the permissions are incompatible (as defined by permissions.permissions_are_compatible()).

Parameters

- **permissions_path1** (Optional[List[*Permissions*]]) – The list of Permissions that affect path1.
- **permissions_path2** (Optional[List[*Permissions*]]) – The list of Permissions that affect path2.
- **path1** (str) –
- **path2** (str) –

Return type
bool

craft_parts.executor.environment module

Helpers to handle part environment setting.

craft_parts.executor.environment.**expand_environment**(*data*, *, *info*, *skip=None*)

Replace global variables with their values.

Global variables are defined by craft-parts and are the subset of the CRAFT_* step execution environment variables that don't depend on the part or step being executed. The list of global variables include CRAFT_ARCH_TRIPLET, CRAFT_PROJECT_DIR, CRAFT_STAGE and CRAFT_PRIME. Additional global variables can be defined by the application using craft-parts.

Parameters

- **data** (Dict[str, Any]) – A dictionary whose values will have variable names expanded.
- **info** (*ProjectInfo*) – The project information.
- **skip** (Optional[List[str]]) – Keys to skip when performing expansion.

Return type
None

craft_parts.executor.environment.**generate_step_environment**(*, *part*, *plugin*, *step_info*)

Generate an environment to use during step execution.

Parameters

- **part** (*Part*) – The part being processed.
- **plugin** (*Plugin*) – The plugin used to build this part.
- **step_info** (*StepInfo*) – Information about the step to be executed.

Return type
str

Returns
The environment to use when executing the step.

craft_parts.executor.executor module

Definitions and helpers for the action executor.

class craft_parts.executor.executor.**ExecutionContext**(*, *executor*)

Bases: object

A context manager to handle lifecycle action executions.

Parameters
executor (*Executor*) –

execute(*actions*, *, *stdout=None*, *stderr=None*)

Execute the specified action or list of actions.

Parameters

- **actions** (Union[*Action*, List[*Action*]]) – An Action object or list of Action objects specifying steps to execute.
- **stdout** (Union[TextIO, int, None]) –
- **stderr** (Union[TextIO, int, None]) –

Raises
InvalidActionException – If the action parameters are invalid.

Return type
None

class craft_parts.executor.executor.**Executor**(*, *part_list*, *project_info*, *extra_build_packages=None*, *extra_build_snaps=None*, *track_stage_packages=False*, *ignore_patterns=None*, *base_layer_dir=None*, *base_layer_hash=None*)

Bases: object

Execute lifecycle actions.

The executor takes the part definition and a list of actions to run for a part and step. Action execution is stateless: no information is kept from the execution of previous parts. On-disk state information written after running each action is read by the sequencer before planning a new set of actions.

Parameters

- **part_list** (List[*Part*]) – The list of parts to process.
- **project_info** (*ProjectInfo*) – Information about this project.
- **track_stage_packages** (bool) – Add primed stage packages to the prime state.
- **extra_build_packages** (Optional[List[str]]) – Additional packages to install on the host system.
- **extra_build_snaps** (Optional[List[str]]) – Additional snaps to install on the host system.
- **ignore_patterns** (Optional[List[str]]) – File patterns to ignore when pulling local sources.
- **base_layer_dir** (Optional[Path]) –
- **base_layer_hash** (Optional[*LayerHash*]) –

clean(*initial_step*, *, *part_names=None*)

Clean the given parts, or all parts if none is specified.

Parameters

- **initial_step** (*Step*) – The step to clean. More steps may be cleaned as a consequence of cleaning the initial step.
- **part_names** (Optional[List[str]]) – A list with names of the parts to clean. If not specified, all parts will be cleaned and work directories will be removed.

Return type
None

epilogue()

Finish and clean the execution environment.

This method is called after executing lifecycle actions.

Return type
None

execute(*actions*, *, *stdout=None*, *stderr=None*)

Execute the specified action or list of actions.

Parameters

- **actions** (Union[*Action*, List[*Action*]]) – An `Action` object or list of `Action` objects specifying steps to execute.
- **stdout** (Union[TextIO, int, None]) –
- **stderr** (Union[TextIO, int, None]) –

Raises

InvalidActionException – If the action parameters are invalid.

Return type

None

prologue()

Prepare the execution environment.

This method is called before executing lifecycle actions.

Return type

None

craft_parts.executor.filesets module

Definitions and helpers to handle filesets.

class craft_parts.executor.filesets.**Fileset**(*entries*, *, *name=''*)

Bases: object

Helper class to process string lists.

Parameters

- **entries** (List[str]) –
- **name** (str) –

combine(*other*)

Combine the entries in this fileset with entries from another fileset.

Parameters

other (*Fileset*) – The fileset to combine with.

Return type

None

property entries: List[str]

Return the list of entries in this fileset.

Return type

List[str]

property excludes: List[str]

Return the list of files to be excluded.

Return type

List[str]

property includes: List[str]

Return the list of files to be included.

Return type
List[str]

property name: str

Return the fileset name.

Return type
str

remove(*item*)

Remove this entry from the list of files.

Parameters
item (str) – The item to remove.

Return type
None

craft_parts.executor.filesets.**migratable_filesets**(*fileset*, *srcdir*)

Return the files and directories that can be migrated.

Parameters

- **fileset** (*Fileset*) – The fileset to migrate.
- **srcdir** (str) –

Return type
Tuple[Set[str], Set[str]]

Returns
A tuple containing the set of files and the set of directories that can be migrated.

craft_parts.executor.migration module

Handle the execution of built-in or user specified step commands.

craft_parts.executor.migration.**clean_shared_area**(*, *part_name*, *shared_dir*, *part_states*, *overlay_migration_state*)

Clean files added by a part to a shared directory.

Parameters

- **part_name** (str) – The name of the part that added the files.
- **shared_dir** (Path) – The shared directory to remove files from.
- **part_states** (Dict[str, *StepState*]) – A dictionary mapping each part to the part's state for the step corresponding to the area being cleaned.
- **overlay_migration_state** (Optional[*MigrationState*]) – The state of the overlay migration to step.

Return type
None

craft_parts.executor.migration.**clean_shared_overlay**(*, *shared_dir*, *part_states*, *overlay_migration_state*)

Remove migrated overlay files from a shared directory.

Parameters

- **state_file** – The migration state file.
- **shared_dir** (Path) – The shared directory to remove files from.
- **part_states** (Dict[str, *StepState*]) – A dictionary mapping each part to the part's state for the step corresponding to the area being cleaned.
- **overlay_migration_state** (Optional[*MigrationState*]) –

Return type
None

craft_parts.executor.migration.**filter_dangling_whiteouts**(*files*, *dirs*, *, *base_dir*)

Remove dangling whiteout file and directory names.

Names corresponding to dangling files and directories (i.e. without a backing file in the base layer to be whited out) are to be removed from the provided sets of files and directory names.

Parameters

- **files** (Set[str]) – The set of files to be verified.
- **dirs** (Set[str]) – The set of directories to be verified.
- **base_dir** (Optional[Path]) –

Return type
Set[str]

Returns
The set of filtered out whiteout files.

craft_parts.executor.migration.**migrate_files**(*, *files*, *dirs*, *srcdir*, *destdir*, *missing_ok=False*, *follow_symlinks=False*, *oci_translation=False*, *fixup_func=<function <lambda>>*, *permissions=None*)

Copy or link files from a directory to another.

Files and directories are migrated from one step to the next during the lifecycle processing. Whenever possible, files are hard-linked instead of copied.

Parameters

- **files** (Set[str]) – The set of files to migrate.
- **dirs** (Set[str]) – The set of directories to migrate.
- **srcdir** (Path) – The directory containing entries to migrate.
- **destdir** (Path) – The directory to migrate entries to.
- **missing_ok** (bool) – Ignore entries that don't exist.
- **follow_symlinks** (bool) – Migrate symlink targets.
- **oci_translation** (bool) – Convert to OCI whiteout files and opaque dirs.
- **fixup_func** (Callable[..., None]) – A function to run on each migrated file.
- **permissions** (Optional[List[*Permissions*]]) – A list of permissions definitions to take into account when migrating the files (the original files are not modified).

Return type
Tuple[Set[str], Set[str]]

Returns
A tuple containing sets of migrated files and directories.

craft_parts.executor.organize module

Handle part files organization.

Installed part files can be reorganized according to a mapping specified under the *organize* entry in a part definition. In the key/value pair, the key represents the path of a file inside the part and the value represents how the file is going to be staged.

craft_parts.executor.organize.**organize_files**(*, *part_name*, *mapping*, *base_dir*, *overwrite*)

Rearrange files for part staging.

Parameters

- **fileset** – A fileset containing the *organize* file mapping.
- **base_dir** (Path) – Where the installed files are located.
- **overwrite** (bool) – Whether existing files should be overwritten. This is only used in build updates, when a part may organize over files it previously organized.
- **part_name** (str) –
- **mapping** (Dict[str, str]) –

Return type
None

craft_parts.executor.part_handler module

Definitions and helpers for part handlers.

class craft_parts.executor.part_handler.**PartHandler**(*part*, *, *part_info*, *part_list*, *track_stage_packages=False*, *overlay_manager*, *ignore_patterns=None*, *base_layer_hash=None*)

Bases: object

Handle lifecycle steps for a part.

Parameters

- **part** (*Part*) – The part being processed.
- **part_info** (*PartInfo*) – Information about the part being processed.
- **part_list** (List[*Part*]) – A list containing all parts.
- **track_stage_packages** (bool) –
- **overlay_manager** (*OverlayManager*) –
- **ignore_patterns** (Optional[List[str]]) –
- **base_layer_hash** (Optional[*LayerHash*]) –

clean_step(*step*)

Remove the work files and the state of the given step.

Parameters
step (*Step*) – The step to clean.

Return type
None

run_action(*action*, *, *stdout=None*, *stderr=None*)

Execute the given action for this part using a plugin.

Parameters

- **action** (*Action*) – The action to execute.
- **stdout** (Union[TextIO, int, None]) –
- **stderr** (Union[TextIO, int, None]) –

Return type
None

craft_parts.executor.step_handler module

Handle the execution of built-in or user specified step commands.

class craft_parts.executor.step_handler.**StepContents**(*files=<factory>*, *dirs=<factory>*)

Bases: object

Files and directories to be added to the step's state.

Parameters

- **files** (Set[str]) –
- **dirs** (Set[str]) –

dirs: Set[str]

files: Set[str]

class craft_parts.executor.step_handler.**StepHandler**(*part*, *, *step_info*, *plugin*, *source_handler*, *env*, *stdout=None*, *stderr=None*)

Bases: object

Executes built-in or user-specified step commands.

The step handler takes care of the execution of a step, using either a built-in set of actions to be taken, or executing a user-defined script defined in the part specification User-defined scripts may also call the built-in handler for a step by invoking a control utility. This class implements the built-in handlers and a FIFO-based mechanism and API to be used by the external control utility to communicate with the running instance.

Parameters

- **part** (*Part*) –
- **step_info** (*StepInfo*) –
- **plugin** (*Plugin*) –
- **source_handler** (Optional[*SourceHandler*]) –

- **env** (str) –
- **stdout** (Union[TextIO, int, None]) –
- **stderr** (Union[TextIO, int, None]) –

run_builtin()

Run the built-in commands for the current step.

Return type
StepContents

run_scriptlet(*scriptlet*, *, *scriptlet_name*, *step*, *work_dir*)

Execute a scriptlet.

Parameters

- **scriptlet** (str) – the scriptlet to run.
- **work_dir** (Path) – the directory where the script will be executed.
- **scriptlet_name** (str) –
- **step** (*Step*) –

Return type
None

Module contents

The action executor.

craft_parts.overlays package

Submodules

craft_parts.overlays.chroot module

Execute a callable in a chroot environment.

craft_parts.overlays.chroot.**chroot**(*path*, *target*, **args*, ***kwargs*)

Execute a callable in a chroot environment.

Parameters

- **path** (Path) – The new filesystem root.
- **target** (Callable) – The callable to run in the chroot environment.
- **args** (Any) – Arguments for target.
- **kwargs** (Any) – Keyword arguments for target.

Return type
Any

Returns
The target function return value.

craft_parts.overlays.errors module

Overlay error definitions.

exception craft_parts.overlays.errors.**OverlayChrootExecutionError**(*message*)

Bases: *OverlayError*

Failed to execute in a chroot environment.

Parameters
message (str) –

brief: str

exception craft_parts.overlays.errors.**OverlayError**(*brief*, *details=None*, *resolution=None*)

Bases: *PartsError*

Base class for overlay handler errors.

Parameters

- **brief** (str) –
- **details** (Optional[str]) –
- **resolution** (Optional[str]) –

brief: str

exception craft_parts.overlays.errors.**OverlayMountError**(*mountpoint*, *message*)

Bases: *OverlayError*

Failed to mount an overlay filesystem.

Parameters

- **mountpoint** (str) – The filesystem mount point.
- **message** (str) – The error message.

brief: str

exception craft_parts.overlays.errors.**OverlayUnmountError**(*mountpoint*, *message*)

Bases: *OverlayError*

Failed to unmount an overlay filesystem.

Parameters

- **mountpoint** (str) – The filesystem mount point.
- **message** (str) – The error message.

brief: str

craft_parts.overlays.layers module

Layer management and helpers.

class craft_parts.overlays.layers.**LayerHash**(*layer_hash*)

Bases: object

The layer validation hash for a part.

Parameters
layer_hash (bytes) –

classmethod for_part(*part*, *, *previous_layer_hash*)

Obtain the validation hash for a part.

Parameters

- **part** (*Part*) – The part being processed.
- **previous_layer_hash** (Optional[*LayerHash*]) – The validation hash of the previous layer in the overlay stack.

Return type
LayerHash

Returns
The validation hash computed for the layer corresponding to the given part.

hex()

Return the current hash value as a hexadecimal string.

Return type
str

classmethod load(*part*)

Read the part layer validation hash from persistent state.

Parameters
part (*Part*) – The part whose layer hash will be loaded.

Return type
Optional[*LayerHash*]

Returns
A layer hash object containing the loaded validaton hash, or None if the file doesn't exist.

save(*part*)

Save the part layer validation hash to persistent storage.

Parameters
part (*Part*) – The part whose layer hash will be saved.

Return type
None

class craft_parts.overlays.layers.**LayerStateManager**(*part_list*, *base_layer_hash*)

Bases: object

An in-memory layer state management helper for action planning.

Parameters

- **part_list** (List[*Part*]) – The list of parts in the project.

- **base_layer_hash** (Optional[*LayerHash*]) – The verification hash of the overlay base layer.

compute_layer_hash(*part*)

Calculate the layer validation hash for the given part.

Parameters
part (*Part*) – The part being processed.

Return type
LayerHash

Returns
The validation hash of the layer corresponding to the given part.

get_layer_hash(*part*)

Obtain the layer hash for the given part.

Parameters
part (*Part*) –

Return type
Optional[*LayerHash*]

get_overlay_hash()

Obtain the overlay validation hash.

Return type
bytes

set_layer_hash(*part*, *layer_hash*)

Store the value of the layer hash for the given part.

Parameters

- **part** (*Part*) –
- **layer_hash** (Optional[*LayerHash*]) –

Return type
None

craft_parts.overlays.overlay_fs module

Low level interface to OS overlayfs.

class craft_parts.overlays.overlay_fs.**OverlayFS**(*, *lower_dirs*, *upper_dir*, *work_dir*)

Bases: object

Linux overlayfs operations.

Parameters

- **lower_dirs** (List[Path]) –
- **upper_dir** (Path) –
- **work_dir** (Path) –

mount(*mountpoint*)

Mount an overlayfs.

Parameters
mountpoint (Path) – The filesystem mount point.

Raises
OverlayMountError – on mount error.

Return type
None

unmount()

Umount an overlayfs.

Raises
OverlayUnmountError – on unmount error.

Return type
None

craft_parts.overlays.overlay_fs.**is_opaque_dir**(*path*)

Verify if the given path corresponds to an opaque directory.

Overlayfs opaque directories are represented by directories with the extended attribute *trusted.overlay.opaque* set to *y*.

Parameters
path (Path) – The path of the file to verify.

Return type
bool

Returns
Whether the given path is an overlayfs opaque directory.

craft_parts.overlays.overlay_fs.**is_whiteout_file**(*path*)

Verify if the given path corresponds to a whiteout file.

Overlayfs whiteout files are represented as character devices with major and minor numbers set to 0.

Parameters
path (Path) – The path of the file to verify.

Return type
bool

Returns
Whether the given path is an overlayfs whiteout.

craft_parts.overlays.overlay_manager module

Overlay mount operations and package installation helpers.

class craft_parts.overlays.overlay_manager.**LayerMount**(*overlay_manager*, *top_part*, *pkg_cache=True*)

Bases: object

Mount the overlay layer stack for step processing.

Parameters

- **overlay_manager** (*OverlayManager*) – The overlay manager.

- **top_part** (*Part*) – The topmost part to mount.
- **pkg_cache** (bool) – Whether to mount the overlay package cache.

install_packages(*package_names*)

Install the specified packages on the local system.

Parameters
package_names (List[str]) – The list of packages to install.

Return type
None

class craft_parts.overlays.overlay_manager.**OverlayManager**(*, *project_info*, *part_list*, *base_layer_dir*)

Bases: object

Execution time overlay mounting and package installation.

Parameters

- **project_info** (*ProjectInfo*) – The project information.
- **part_list** (List[*Part*]) – A list of all parts in the project.
- **base_layer_dir** (Optional[Path]) – The directory containing the overlay base, or None if the project doesn't use overlay parameters.

property base_layer_dir: Optional[Path]

Return the path to the base layer, if any.

Return type
Optional[Path]

download_packages(*package_names*)

Download packages and populate the overlay package cache.

Parameters
package_names (List[str]) – The list of packages to download.

Return type
None

install_packages(*package_names*)

Install packages on the overlay area using chroot.

Parameters
package_names (List[str]) – The list of packages to install.

Return type
None

mkdirs()

Create overlay directories and mountpoints.

Return type
None

mount_layer(*part*, *, *pkg_cache=False*)

Mount the overlay step layer stack up to the given part.

Parameters

- **part** (*Part*) – The part corresponding to the topmost layer to mount.

- **cache** (*pkg*) – Whether the package cache layer is enabled.
- **pkg_cache** (bool) –

Return type
None

mount_pkg_cache()

Mount the overlay step package cache layer.

Return type
None

refresh_packages_list()

Update the list of available packages in the overlay system.

Return type
None

unmount()

Unmount the overlay step layer stack.

Return type
None

class craft_parts.overlays.overlay_manager.PackageCacheMount(*overlay_manager*)

Bases: object

Mount and umount the overlay package cache.

Parameters
overlay_manager (*OverlayManager*) – The overlay manager.

download_packages(*package_names*)

Download the specified packages to the local system.

Parameters
package_names (List[str]) – The list of packages to download.

Return type
None

refresh_packages_list()

Update the list of available packages in the overlay system.

Return type
None

craft_parts.overlays.overlays module

Overlay handling helpers.

Relevant OCI documentation available at: https://github.com/opencontainers/image-spec/blob/main/layer.md

craft_parts.overlays.overlays.**is_oci_opaque_dir**(*path*)

Verify if the given path corresponds to an opaque directory.

Parameters
path (Path) – The path of the file to verify.

Return type
bool

Returns
Whether the given path is an overlayfs opaque directory.

craft_parts.overlays.overlays.**is_oci_whiteout_file**(*path*)

Verify if the given path corresponds to an OCI whiteout file.

Parameters
path (Path) – The path of the file to verify.

Return type
bool

Returns
Whether the given path is an OCI whiteout file.

craft_parts.overlays.overlays.**oci_opaque_dir**(*path*)

Return the OCI opaque directory marker.

Parameters
path (Path) – The directory to mark as opaque.

Return type
Path

Returns
The corresponding OCI opaque directory marker path.

craft_parts.overlays.overlays.**oci_whited_out_file**(*whiteout_file*)

Find the whited out file corresponding to a whiteout file.

Parameters
whiteout_file (Path) – The whiteout file to process.

Return type
Path

Returns
The file that was whited out.

craft_parts.overlays.overlays.**oci_whiteout**(*path*)

Convert the given path to an OCI whiteout file name.

Parameters
path (Path) – The file path to white out.

Return type
Path

Returns
The corresponding OCI whiteout file name.

craft_parts.overlays.overlays.**visible_in_layer**(*lower_dir*, *upper_dir*)

Determine the files and directories that are visible in a layer.

Given a pair of directories containing lower and upper layer entries, list the files and subdirectories in the lower layer that would be directly visible when the layers are stacked (i.e. the visibility is not "blocked" by an entry with the same name that exists in the upper directory). The upper directory may contain OCI whiteout files and opaque dirs.

Parameters

- **lower_dir** (Path) – The lower directory.
- **upper_dir** (Path) – The upper directory.

Return type
Tuple[Set[str], Set[str]]

Returns
A tuple containing the sets of files and directories that are visible.

Module contents

Overlay filesystem management and helpers.

craft_parts.packages package

Submodules

craft_parts.packages.apt_cache module

Manages the state of packages obtained using apt.

class craft_parts.packages.apt_cache.**AptCache**(*, *stage_cache=None*, *stage_cache_arch=None*)

Bases: ContextDecorator

Transient cache for stage packages, or read-only for build packages.

Parameters

- **stage_cache** (Optional[Path]) –
- **stage_cache_arch** (Optional[str]) –

classmethod configure_apt(*application_package_name*)

Set up apt options and directories.

Parameters
application_package_name (str) –

Return type
None

fetch_archives(*download_path*)

Retrieve packages marked to be fetched.

Parameters
download_path (Path) – The directory to download files to.

Return type
List[Tuple[str, str, Path]]

Returns
A list of (<package-name>, <package-version>, <dl-path>) tuples.

get_installed_packages()

Obtain a list of all packages and versions installed on the system.

Return type
Dict[str, str]

Returns
A dictionary of files and installed versions.

`get_installed_version`(*package_name*, *, *resolve_virtual_packages=False*)

Obtain the version of the package currently installed on the system.

Parameters

- **package_name** (`str`) – The package installed on the system.
- **resolve_virtual_packages** (`bool`) – If the package is virtual, pick a non-virtual package that satisfies this virtual package name.

Return type
`Optional[str]`

Returns
The installed package version.

`get_packages_marked_for_installation`()

Obtain a list of packages and versions to be installed on the system.

Return type
`List[Tuple[str, str]]`

Returns
A list of (<package-name>, <package-version>) tuples.

`is_package_valid`(*package_name*)

Verify whether there is a valid package with the given name.

Parameters
package_name (`str`) – The name of the package to verify.

Return type
`bool`

Returns
Whether a package with the given name is valid.

`mark_packages`(*package_names*)

Mark the given package names to be fetched from the repository.

Parameters
package_names (`Set[str]`) – The set of package names to be marked.

Return type
`None`

`unmark_packages`(*unmark_names*)

Unmark packages and dependencies that are no longer required.

Parameters
unmark_names (`Set[str]`) – The names of the packages to unmark.

Return type
`None`

class craft_parts.packages.apt_cache.**LogProgress**

Bases: `AcquireProgress`

Internal Base class for text progress classes.

fail(*item*)

Handle failed item.

Parameters
item (AcquireItemDesc) –

Return type
None

fetch(*item*)

Handle item's data is fetch.

Parameters
item (AcquireItemDesc) –

Return type
None

craft_parts.packages.base module

Definition and helpers for the repository base class.

class craft_parts.packages.base.**BaseRepository**

Bases: ABC

Base implementation for a platform specific repository handler.

abstract classmethod configure(*application_package_name*)

Set up the repository.

Parameters
application_package_name (str) –

Return type
None

abstract classmethod download_packages(*package_names*)

Download the specified packages to the local package cache.

Parameters
package_names (List[str]) – A list with the names of the packages to download.

Return type
None

abstract classmethod fetch_stage_packages(*, *cache_dir*, *package_names*, *stage_packages_path*, *base*, *arch*, *list_only=False*)

Fetch stage packages to stage_packages_path.

Parameters

- **application_name** – A unique identifier for the application using Craft Parts.
- **package_names** (List[str]) – A list with the names of the packages to fetch.
- **base** (str) – The base this project will run on.
- **arch** (str) – The architecture of the packages to fetch.
- **list_only** (bool) – Whether to obtain a list of packages to be fetched instead of actually fetching the packages.

- **cache_dir** (Path) –
- **stage_packages_path** (Path) –

Stage_packages_path
The path stage packages will be fetched to.

Return type
List[str]

Returns
The list of all packages to be fetched, including dependencies.

abstract classmethod get_installed_packages()

Obtain a list of the installed packages and their versions.

Return type
List[str]

Returns
A list of installed packages in the form package=version.

abstract classmethod get_package_libraries(*package_name*)

Return a list of libraries in package_name.

Given the contents of package_name, return the subset of what are considered libraries from those contents, be it static or shared.

Parameters
package_name (str) – The package name to get library contents from.

Return type
Set[str]

Returns
A list of libraries that package_name provides, with paths.

abstract classmethod get_packages_for_source_type(*source_type*)

Return a list of packages required to to work with source_type.

Parameters
source_type (str) – A source type to handle.

Return type
Set[str]

Returns
A set of packages that need to be installed on the host.

abstract classmethod install_packages(*package_names*, *, *list_only=False*, *refresh_package_cache=True*)

Install packages on the host system.

This method needs to be implemented by using the appropriate mechanism to install packages on the system. If possible they should be marked as automatically installed to allow for easy removal. The method should return a list of the actually installed packages in the form "package=version".

If one of the packages cannot be found BuildPackageNotFound should be raised. If dependencies for a package cannot be resolved PackageBroken should be raised. If installing a package on the host failed BuildPackagesNotInstalled should be raised.

Parameters

- **package_names** (List[str]) – A list of package names to install.
- **list_only** (bool) – Only list the packages that would be installed.
- **refresh_package_cache** (bool) – Refresh the cache before installing.

Return type
List[str]

Returns
A list with the packages installed and their versions.

abstract classmethod is_package_installed(*package_name*)

Inform if a package is installed on the host system.

Parameters
package_name (str) – The package name to query.

Return type
bool

Returns
Whether the package is installed.

abstract classmethod refresh_packages_list()

Refresh the list of packages available in the repository.

If refreshing is not possible PackageListRefreshError should be raised.

Return type
None

abstract classmethod unpack_stage_packages(*, *stage_packages_path*, *install_path*, *stage_packages=None*, *track_stage_packages=False*)

Unpack stage packages.

Parameters

- **stage_packages_path** (Path) – The path to the directory containing the stage packages to unpack.
- **install_path** (Path) – The path stage packages will be unpacked to.
- **stage_packages** (Optional[List[str]]) – An optional list of the packages that were previously pulled.
- **track_stage_packages** (bool) –

Return type
None

class craft_parts.packages.base.**DummyRepository**

Bases: *BaseRepository*

A dummy repository.

classmethod configure(*application_package_name*)

Set up the repository.

Parameters
application_package_name (str) –

Return type
None

classmethod download_packages(*package_names*)

Download the specified packages to the local package cache.

Parameters
package_names (List[str]) –

Return type
None

classmethod fetch_stage_packages(***kwargs*)

Fetch stage packages to stage_packages_path.

Parameters
kwargs (Any) –

Return type
List[str]

classmethod get_installed_packages()

Obtain a list of the installed packages and their versions.

Return type
List[str]

classmethod get_package_libraries(*package_name*)

Return a list of libraries in package_name.

Parameters
package_name (str) –

Return type
Set[str]

classmethod get_packages_for_source_type(*source_type*)

Return a list of packages required to to work with source_type.

Parameters
source_type (str) –

Return type
Set[str]

classmethod install_packages(*package_names*, ***, *list_only=False*, *refresh_package_cache=True*)

Install packages on the host system.

Parameters

- **package_names** (List[str]) –
- **list_only** (bool) –
- **refresh_package_cache** (bool) –

Return type
List[str]

classmethod is_package_installed(*package_name*)

Inform if a packahe is installed on the host system.

Parameters
package_name (str) –

Return type
bool

classmethod refresh_packages_list()

Refresh the build packages cache.

Return type
None

classmethod unpack_stage_packages(*, *stage_packages_path*, *install_path*, *stage_packages=None*, *track_stage_packages=False*)

Unpack stage packages to install_path.

Parameters

- **stage_packages_path** (Path) –
- **install_path** (Path) –
- **stage_packages** (Optional[List[str]]) –
- **track_stage_packages** (bool) –

Return type
None

craft_parts.packages.base.**get_pkg_name_parts**(*pkg_name*)

Break package name into base parts.

Parameters
pkg_name (str) –

Return type
Tuple[str, Optional[str]]

craft_parts.packages.base.**mark_origin_stage_package**(*sources_dir*, *stage_package*)

Mark all files in sources_dir as coming from stage_package.

Parameters

- **sources_dir** (str) –
- **stage_package** (str) –

Return type
None

craft_parts.packages.base.**read_origin_stage_package**(*path*)

Read origin stage package.

Parameters
path (str) –

Return type
Optional[str]

craft_parts.packages.base.**write_origin_stage_package**(*path*, *value*)

Write origin stage package.

Parameters

- **path** (str) –
- **value** (str) –

Return type
None

craft_parts.packages.deb module

Support for deb files.

class craft_parts.packages.deb.**Ubuntu**

Bases: *BaseRepository*

Repository management for Ubuntu packages.

classmethod configure(*cls*, *application_package_name*)

Set up apt options and directories.

Parameters
application_package_name (str) –

Return type
None

classmethod download_packages(*package_names*)

Download the specified packages to the local package cache area.

Parameters
package_names (List[str]) –

Return type
None

classmethod fetch_stage_packages(*, *cache_dir*, *package_names*, *stage_packages_path*, *base*, *arch*, *list_only=False*)

Fetch stage packages to stage_packages_path.

Parameters

- **cache_dir** (Path) –
- **package_names** (List[str]) –
- **stage_packages_path** (Path) –
- **base** (str) –
- **arch** (str) –
- **list_only** (bool) –

Return type
List[str]

classmethod get_installed_packages(*cls*)

Obtain a list of the installed packages and their versions.

Return type
List[str]

classmethod get_package_libraries(*package_name*)

Return a list of libraries in package_name.

Parameters
package_name (str) –

Return type
Set[str]

classmethod get_packages_for_source_type(*cls*, *source_type*)

Return a list of packages required to to work with source_type.

Parameters
source_type (str) –

Return type
Set[str]

classmethod install_packages(*package_names*, *, *list_only=False*, *refresh_package_cache=True*)

Install packages on the host system.

Parameters

- **package_names** (List[str]) –
- **list_only** (bool) –
- **refresh_package_cache** (bool) –

Return type
List[str]

classmethod is_package_installed(*cls*, *package_name*)

Inform if a package is installed on the host system.

Parameters
package_name (str) –

Return type
bool

classmethod refresh_packages_list(*cls*)

Refresh the list of packages available in the repository.

Return type
None

classmethod unpack_stage_packages(*, *stage_packages_path*, *install_path*, *stage_packages=None*, *track_stage_packages=False*)

Unpack stage packages to install_path.

Parameters

- **stage_packages_path** (Path) –
- **install_path** (Path) –
- **stage_packages** (Optional[List[str]]) –
- **track_stage_packages** (bool) –

Return type
None

craft_parts.packages.deb.**get_cache_dirs**(*cache_dir*)

Return the paths to the stage and deb cache directories.

Parameters
cache_dir (Path) –

Return type
Tuple[Path, Path]

craft_parts.packages.deb.**get_packages_in_base**(*, *base*)

Get the list of packages for the given base.

Parameters
base (str) –

Return type
List[*DebPackage*]

craft_parts.packages.deb.**process_run**(*command*, ***kwargs*)

Run a command and log its output.

Parameters

- **command** (List[str]) –
- **kwargs** (Any) –

Return type
None

craft_parts.packages.deb_package module

Debian package representation.

class craft_parts.packages.deb_package.**DebPackage**(*name*, *arch=None*, *version=None*)

Bases: object

Debian package representation.

Parameters

- **name** (str) –
- **arch** (Optional[str]) –
- **version** (Optional[str]) –

arch: Optional[str] = None

classmethod from_unparsed(*package*)

Parse package supported in yaml.

Package Format: <package-name>[:<arch>][=<version>]

Examples: “foo”, “foo:i386”, “foo=1.5”, “foo:i386=1.5”

Parameters
package (str) – Package to parse.

Return type
DebPackage

Returns
DebPackage with populated arch & version, if any.

name: str

version: Optional[str] = None

craft_parts.packages.errors module

Exceptions raised by the packages handling subsystem.

exception craft_parts.packages.errors.**BuildPackageNotFound**(*package*)

Bases: *PackagesError*

A package listed in 'build-packages' was not found.

Parameters
package (str) – The name of the missing package.

brief: str

exception craft_parts.packages.errors.**BuildPackagesNotInstalled**(*, *packages*)

Bases: *PackagesError*

Could not install all requested build packages.

Parameters
packages (Sequence[str]) – The packages to install.

brief: str

exception craft_parts.packages.errors.**ChiselError**(*, *slices*, *output*)

Bases: *PackagesError*

A "chisel"-related command failed.

Parameters

- **slices** (List[str]) –
- **output** (str) –

brief: str

exception craft_parts.packages.errors.**FileProviderNotFound**(*, *file_path*)

Bases: *PackagesError*

A file is not provided by any package.

Parameters
file_path (str) – The file path.

brief: str

exception craft_parts.packages.errors.**PackageBackendNotSupported**(*backend*)

Bases: *PartsError*

Requested package resolved not supported on this host.

Parameters
backend (str) –

brief: str

exception craft_parts.packages.errors.**PackageBroken**(*package_name*, *, *deps*)

Bases: *PackagesError*

Package has unmet dependencies.

Parameters

- **package_name** (str) – The name of the package with unmet dependencies.
- **deps** (Sequence[str]) – The list of unmet dependencies.

brief: str

exception craft_parts.packages.errors.**PackageFetchError**(*message*)

Bases: *PackagesError*

Failed to fetch package from remote repository.

Parameters
message (str) – The error message.

brief: str

exception craft_parts.packages.errors.**PackageListRefreshError**(*message*)

Bases: *PackagesError*

Failed to refresh the list of available packages.

Parameters
message (str) – The error message.

brief: str

exception craft_parts.packages.errors.**PackageNotFound**(*package_name*)

Bases: *PackagesError*

Requested package doesn't exist in the remote repository.

Parameters
package_name (str) – The name of the missing package.

brief: str

exception craft_parts.packages.errors.**PackagesDownloadError**(*, *packages*)

Bases: *PackagesError*

Failed to download packages from remote repository.

Parameters
packages (Sequence[str]) – The packages to download.

brief: str

exception craft_parts.packages.errors.**PackagesError**(*brief*, *details=None*, *resolution=None*)

Bases: *PartsError*

Base class for package handler errors.

Parameters

- **brief** (str) –
- **details** (Optional[str]) –
- **resolution** (Optional[str]) –

brief: str

exception craft_parts.packages.errors.**PackagesNotFound**(*packages*)

Bases: *PackagesError*

Requested package doesn't exist in the remote repository.

Parameters

- **package_name** – The names of the missing packages.
- **packages** (Sequence[str]) –

brief: str

exception craft_parts.packages.errors.**SnapDownloadError**(*, *snap_name*, *snap_channel*)

Bases: *PackagesError*

Failed to download a snap.

Parameters

- **snap_name** (str) – The snap name.
- **snap_channel** (str) – The snap channel.

brief: str

exception craft_parts.packages.errors.**SnapGetAssertionError**(*, *assertion_params*)

Bases: *PackagesError*

Failed to retrieve snap assertion.

Parameters
assertion_params (Sequence[str]) – The snap assertion parameters.

brief: str

exception craft_parts.packages.errors.**SnapInstallError**(*, *snap_name*, *snap_channel*)

Bases: *PackagesError*

Failed to install a snap.

Parameters

- **snap_name** (str) – The snap name.
- **snap_channel** (str) – The snap channel.

brief: str

exception craft_parts.packages.errors.**SnapRefreshError**(*, *snap_name*, *snap_channel*)

Bases: *PackagesError*

Failed to refresh a snap.

Parameters

- **snap_name** (str) – The snap name.
- **snap_channel** (str) – The snap channel.

brief: str

exception craft_parts.packages.errors.**SnapUnavailable**(*, *snap_name*, *snap_channel*)

Bases: *PackagesError*

Failed to install or refresh a snap.

Parameters

- **snap_name** (str) – The snap name.
- **snap_channel** (str) – The snap channel.

brief: str

exception craft_parts.packages.errors.**SnapdConnectionError**(*, *snap_name*, *url*)

Bases: *PackagesError*

Failed to connect to snapd.

Parameters

- **snap_name** (str) – The snap name.
- **url** (str) – The failed connection URL.

brief: str

exception craft_parts.packages.errors.**UnpackError**(*package*)

Bases: *PackagesError*

Error unpacking stage package.

Parameters

package (str) – The package that failed to unpack.

brief: str

craft_parts.packages.normalize module

Definition and helpers for the repository base class.

craft_parts.packages.normalize.**fix_pkg_config**(*prefix_prepend*, *pkg_config_file*, *prefix_trim=None*)

Fix the prefix parameter in pkg-config files.

This function does 3 things: 1. Remove *prefix_trim* from the prefix. 2. Remove directories commonly added by staged snaps from the prefix. 3. Prepend *prefix_prepend* to the prefix.

The prepended stage directory depends on the source of the pkg-config file: - From snaps built via launchpad: */build/<snap-name>/stage* - From snaps built via a provider: */root/stage* - From snaps built locally: *<local-path-to-project>/stage* - Built during the build stage: the install directory

Parameters

- **pkg_config_file** (Path) – pkg-config (.pc) file to modify
- **prefix_prepend** (Path) – directory to prepend to the prefix
- **prefix_trim** (Optional[Path]) – directory to remove from prefix

Return type

None

craft_parts.packages.normalize.**normalize**(*unpack_dir*, *, *repository*)

Normalize unpacked artifacts.

Repository-specific packages are generally created to live in a specific distro. Normalize scans through the unpacked artifacts and slightly modifies them to work better in the Craft Parts build environment.

Parameters

- **unpack_dir** (Path) – Directory containing unpacked files to normalize.
- **repository** (Type[*BaseRepository*]) – The package format handler.

Return type
None

craft_parts.packages.platform module

Helpers to determine the repository for the platform.

craft_parts.packages.platform.**is_deb_based**(*distro=None*)

Verify the distribution packaging system.

Parameters
distro (Optional[str]) – The distribution name.

Return type
bool

Returns
Whether the distribution uses .deb packages.

craft_parts.packages.snaps module

Helpers to install snap packages.

class craft_parts.packages.snaps.**SnapPackage**(*snap*)

Bases: object

SnapPackage acts as a mediator to install or refresh a snap.

It uses information provided by snapd implicitly referring to the local and remote stores to obtain information about the snap, such as its confinement value and channel availability.

This information can also be used to determine if a snap should be installed or refreshed.

There are risks of the data falling out of date between the query and the requested action given that it is not possible to hold a global lock on snapd and the store data can change in between validation and execution.

Parameters
snap (str) –

download(*, *directory=None*)

Download a given snap.

Parameters
directory (Optional[str]) –

Return type
None

get_current_channel()

Obtain the current channel for this snap.

Return type
str

get_local_snap_info()

Return a local payload for the snap.

Validity of the results are determined by checking self.installed.

Return type
Optional[Dict[str, Any]]

get_store_snap_info()

Return a store payload for the snap.

Return type
Optional[Dict[str, Any]]

has_assertions()

Verify whether this snap has assertions.

Return type
bool

property in_store: bool

Whether this snap is available in the store.

Return type
bool

install()

Installs the snap onto the system.

Return type
None

property installed: bool

Whether this snap is currently installed on the system.

Return type
bool

is_classic()

Verify whether this snap is a classic snap.

Return type
bool

classmethod is_snap_installed(*snap*)

Verify whether the given snap is installed.

Parameters
snap (str) –

Return type
bool

is_valid()

Check if the snap is valid.

Return type
bool

classmethod is_valid_snap(*snap*)

Verify whether the given snap is valid.

Parameters
snap (str) –

Return type
bool

refresh()

Refresh a snap onto a channel on the system.

Return type
None

craft_parts.packages.snaps.**download_snaps**(*, *snaps_list*, *directory*)

Download snaps of the format <snap-name>/<channel> into directory.

The target directory is created if it does not exist.

Parameters

- **snaps_list** (Sequence[str]) –
- **directory** (str) –

Return type
None

craft_parts.packages.snaps.**get_assertion**(*assertion_params*)

Get assertion information.

Parameters
assertion_params (Sequence[str]) – a sequence of strings to pass to 'snap known'.

Returns
a stream of bytes from the assertion.

Return type
bytes

craft_parts.packages.snaps.**get_installed_snaps**()

Return all the snaps installed in the system.

Return type
List[str]

Returns
a list of "name=revision" for the snaps installed.

craft_parts.packages.snaps.**get_snapd_socket_path_template**()

Return the template for the snapd socket URI.

Return type
str

craft_parts.packages.snaps.**install_snaps**(*snaps_list*)

Install snaps of the format <snap-name>/<channel>.

Return type
List[str]

Returns
a list of "name=revision" for the snaps installed.

Parameters
snaps_list (Union[Sequence[str], Set[str]]) –

Module contents

Operations with platform-specific package repositories.

craft_parts.plugins package

Submodules

craft_parts.plugins.ant_plugin module

The Ant plugin.

class craft_parts.plugins.ant_plugin.**AntPlugin**(*, *properties*, *part_info*)

Bases: *JavaPlugin*

A plugin for Apache Ant projects.

The plugin requires the `ant` tool installed on the system. This can be achieved by adding the appropriate declarations to `build-packages` or `build-snaps`, or by having it installed or built in a different part. In this case, the name of the part supplying `ant` must be "ant-deps".

Additionally, Java projects need a dev kit (jdk) to build and a runtime environment (jre) to run. There are multiple choices here, but frequently adding `default-jdk-headless` to `build-packages` and `default-jre-headless` to `stage-packages` is enough.

Once built, the plugin will create the following structure in the part's install dir (which will later be staged/primed/packaged):

- A `bin/java` symlink pointing to the actual `java` binary provided by the jre;
- A `jar/` directory containing the .jar files generated by the build.

The ant plugin uses the common plugin keywords, plus the following ant- specific keywords:

- `ant-build-targets` (list of strings) The ant targets to build. These are directly passed to the `ant` command line.
- `ant-build-file` (str) The name of the main ant build file. Defaults to `build.xml`.
- `ant-properties` (dict of strings to strings) A series of key: value pairs that are passed to ant as properties (using the `-D{key}={value}` notation).

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
List[str]

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
Dict[str, str]

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
Set[str]

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
Set[str]

properties_class

alias of *AntPluginProperties*

validator_class

alias of *AntPluginEnvironmentValidator*

class craft_parts.plugins.ant_plugin.**AntPluginEnvironmentValidator**(*, *part_name*, *env*, *properties*)

Bases: *PluginEnvironmentValidator*

Check the execution environment for the Ant plugin.

Parameters

- **part_name** (str) – The part whose build environment is being validated.
- **env** (str) – A string containing the build step environment setup.
- **properties** (*PluginProperties*) –

validate_environment(*, *part_dependencies=None*)

Ensure the environment contains dependencies needed by the plugin.

Parameters
part_dependencies (Optional[List[str]]) – A list of the parts this part depends on.

Raises
PluginEnvironmentValidationError – If ant is invalid

and there are no parts named ant.

Return type
None

class craft_parts.plugins.ant_plugin.**AntPluginProperties**(**data*)

Bases: *PluginProperties*, *PluginModel*

The part properties used by the Ant plugin.

Parameters
data (Any) –

ant_build_file: Optional[str]

ant_build_targets: List[str]

ant_properties: Dict[str, str]

source: str

classmethod unmarshal(*data*)

Populate make properties from the part specification.

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

Return type
AntPluginProperties

Returns
The populated plugin properties data object.

Raises
pydantic.ValidationError – If validation fails.

craft_parts.plugins.autotools_plugin module

The autotools plugin implementation.

class craft_parts.plugins.autotools_plugin.**AutotoolsPlugin**(*, *properties*, *part_info*)

Bases: *Plugin*

The autotools plugin is used for autotools-based parts.

Autotools-based projects are the ones that have the usual *./configure && make && make install* instruction set.

This plugin will check for the existence of a 'configure' file, if one cannot be found, it will first try to run 'autogen.sh' or 'bootstrap' to generate one.

This plugin uses the common plugin keywords as well as those for "sources". For more information check the 'plugins' topic for the former and the 'sources' topic for the latter.

In addition, this plugin uses the following plugin-specific keywords:

- autotools-configure-parameters (list of strings) configure flags to pass to the build such as those shown by running './configure –help'

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
List[str]

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
Dict[str, str]

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
Set[str]

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
Set[str]

properties_class

alias of *AutotoolsPluginProperties*

class craft_parts.plugins.autotools_plugin.**AutotoolsPluginProperties**(**data*)

Bases: *PluginProperties*, *PluginModel*

The part properties used by the autotools plugin.

Parameters
data (Any) –

autotools_configure_parameters: List[str]

source: str

classmethod unmarshal(*data*)

Populate autotools properties from the part specification.

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

Return type
AutotoolsPluginProperties

Returns
The populated plugin properties data object.

Raises
pydantic.ValidationError – If validation fails.

craft_parts.plugins.base module

Plugin base class and definitions.

class craft_parts.plugins.base.**JavaPlugin**(*, *properties*, *part_info*)

Bases: *Plugin*

A base class for java-related plugins.

Provide common methods to deal with the java executable location and symlink creation.

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

properties_class: Type[*PluginProperties*]

class craft_parts.plugins.base.**Plugin**(*, *properties*, *part_info*)

Bases: ABC

The base class for plugins.

Variables

- **properties_class** – The plugin properties class.
- **validator_class** – The plugin environment validator class.

Parameters

- **part_info** (*PartInfo*) – The part information for the applicable part.
- **properties** (*PluginProperties*) – Part-defined properties.

abstract get_build_commands()

Return a list of commands to run during the build step.

Return type
List[str]

abstract get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
Dict[str, str]

abstract get_build_packages()

Return a set of required packages to install in the build environment.

Return type
Set[str]

abstract get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
Set[str]

classmethod get_out_of_source_build()

Return whether the plugin performs out-of-source-tree builds.

Return type
bool

properties_class: Type[*PluginProperties*]

set_action_properties(*action_properties*)

Store a copy of the given action properties.

Parameters
action_properties (*ActionProperties*) – The properties to store.

Return type
None

validator_class

alias of *PluginEnvironmentValidator*

class craft_parts.plugins.base.**PluginModel**(***data*)

Bases: BaseModel

Model for plugins using pydantic validation.

Plugins with configuration properties can use pydantic validation to unmarshal data from part specs. In this case, extract plugin-specific properties using the *extract_plugin_properties()* helper.

Parameters
data (Any) –

class Config

Bases: object

Pydantic model configuration.

alias_generator()

allow_mutation = False

extra = 'forbid'

validate_assignment = True

craft_parts.plugins.base.**extract_plugin_properties**(*data*, ***, *plugin_name*, *required=None*)

Obtain plugin-specifc entries from part properties.

Parameters

- **data** (Dict[str, Any]) – A dictionary containing all part properties.
- **plugin_name** (str) –
- **required** (Optional[List[str]]) –

Plugin_name
The name of the plugin.

Return type
Dict[str, Any]

Returns
A dictionary with plugin properties.

craft_parts.plugins.cmake_plugin module

The cmake plugin.

class craft_parts.plugins.cmake_plugin.**CMakePlugin**(***, *properties*, *part_info*)

Bases: *Plugin*

The cmake plugin is useful for building cmake based parts.

These are projects that have a CMakeLists.txt that drives the build. The plugin requires a CMakeLists.txt in the root of the source tree.

This plugin uses the common plugin keywords as well as those for "sources". For more information check the 'plugins' topic for the former and the 'sources' topic for the latter.

This implementation follows the syntax and behavior used in the Snapcraft cmake plugin for core20. Unlike the cmake plugin used for core18, `CMAKE_INSTALL_PREFIX` is not automatically set. To retain compatibility with the Snapcraft core18 plugin, define the cmake parameter `-DCMAKE_INSTALL_PREFIX=` in your project. This also allows libraries built using the cmake plugin and staged by a different part to be automatically recognized without defining additional parameters such as `CMAKE_INCLUDE_PATH` or `CMAKE_INSTALL_PATH`.

This plugin uses the following plugin-specific keywords:

- cmake-parameters (list of strings) parameters to pass to the build using the common cmake semantics.
- cmake-generator (string; default: "Unix Makefiles") Determine what native build system is to be used. Can be either *Ninja* or *Unix Makefiles* (default).

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
List[str]

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
Dict[str, str]

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
Set[str]

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
Set[str]

classmethod get_out_of_source_build()

Return whether the plugin performs out-of-source-tree builds.

Return type
bool

properties_class

alias of *CMakePluginProperties*

class craft_parts.plugins.cmake_plugin.**CMakePluginProperties**(***data*)

Bases: *PluginProperties*, *PluginModel*

The part properties used by the cmake plugin.

Parameters
data (Any) –

cmake_generator: str

cmake_parameters: List[str]

source: str

classmethod unmarshal(*data*)

Populate class attributes from the part specification.

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

Return type
CMakePluginProperties

Returns
The populated plugin properties data object.

Raises
pydantic.ValidationError – If validation fails.

craft_parts.plugins.dotnet_plugin module

The Dotnet plugin.

class craft_parts.plugins.dotnet_plugin.**DotPluginEnvironmentValidator**(*, *part_name*, *env*, *properties*)

Bases: *PluginEnvironmentValidator*

Check the execution environment for the Dotnet plugin.

Parameters

- **part_name** (str) – The part whose build environment is being validated.
- **env** (str) – A string containing the build step environment setup.
- **properties** (*PluginProperties*) –

validate_environment(*, *part_dependencies=None*)

Ensure the environment contains dependencies needed by the plugin.

Parameters
part_dependencies (Optional[List[str]]) – A list of the parts this part depends on.

Return type
None

class craft_parts.plugins.dotnet_plugin.**DotnetPlugin**(*, *properties*, *part_info*)

Bases: *Plugin*

A plugin for dotnet projects.

The dotnet plugin requires dotnet installed on your system. This can be achieved by adding the appropriate dotnet snap package to `build-snaps`, or to have it installed or built in a different part. In this case, the name of the part supplying the dotnet compiler must be "dotnet".

The dotnet plugin uses the common plugin keywords as well as those for "sources". Additionally, the following plugin-specific keywords can be used:

- `dotnet-build-configuration` (string) The dotnet build configuration to use. The default is "Release".
- `dotnet-self-contained-runtime-identifier` (string) Create a self contained dotnet application using the specified RuntimeIdentifier.

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
List[str]

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
Dict[str, str]

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
Set[str]

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
Set[str]

properties_class

alias of *DotnetPluginProperties*

validator_class

alias of *DotPluginEnvironmentValidator*

class craft_parts.plugins.dotnet_plugin.**DotnetPluginProperties**(**data*)

Bases: *PluginProperties*, *PluginModel*

The part properties used by the Dotnet plugin.

Parameters
data (Any) –

dotnet_build_configuration: str

dotnet_self_contained_runtime_identifier: Optional[str]

source: str

classmethod unmarshal(*data*)

Populate make properties from the part specification.

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

Return type
DotnetPluginProperties

Returns
The populated plugin properties data object.

Raises
pydantic.ValidationError – If validation fails.

craft_parts.plugins.dump_plugin module

The dump plugin.

This plugin just dumps the content from a specified part source.

class craft_parts.plugins.dump_plugin.**DumpPlugin**(*, *properties*, *part_info*)

Bases: *Plugin*

Copy the content from the part source.

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
List[str]

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
Dict[str, str]

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
Set[str]

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
Set[str]

properties_class

alias of *DumpPluginProperties*

class craft_parts.plugins.dump_plugin.**DumpPluginProperties**

Bases: *PluginProperties*

The part properties used by the dump plugin.

classmethod unmarshal(*data*)

Populate dump properties from the part specification.

'source' is a required part property.

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

Return type
DumpPluginProperties

Returns
The populated plugin properties data object.

Raises
ValueError – If a required property is not found.

craft_parts.plugins.go_plugin module

The Go plugin.

class craft_parts.plugins.go_plugin.**GoPlugin**(*, *properties*, *part_info*)

Bases: *Plugin*

A plugin for go projects using go.mod.

The go plugin requires a go compiler installed on your system. This can be achieved by adding the appropriate golang package to `build-packages`, or to have it installed or built in a different part. In this case, the name of the part supplying the go compiler must be "go".

The go plugin uses the common plugin keywords as well as those for "sources". Additionally, the following plugin-specific keywords can be used:

- `go-buildtags` (list of strings) Tags to use during the go build. Default is not to use any build tags.
- `go-generate` (list of strings) Parameters to pass to *go generate* before building. Each item on the list will be a separate *go generate* call. Default is not to call *go generate*.

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
`List[str]`

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
`Dict[str, str]`

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
`Set[str]`

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
`Set[str]`

properties_class

alias of *GoPluginProperties*

validator_class

alias of GoPluginEnvironmentValidator

class craft_parts.plugins.go_plugin.GoPluginEnvironmentValidator(*, part_name, env, properties)

Bases: PluginEnvironmentValidator

Check the execution environment for the Go plugin.

Parameters

- **part_name** (str) – The part whose build environment is being validated.
- **env** (str) – A string containing the build step environment setup.
- **properties** (PluginProperties) –

validate_environment(*, part_dependencies=None)

Ensure the environment contains dependencies needed by the plugin.

Parameters

part_dependencies (Optional[List[str]]) – A list of the parts this part depends on.

Raises

PluginEnvironmentValidationError – If go is invalid

and there are no parts named go.

Return type

None

class craft_parts.plugins.go_plugin.GoPluginProperties(**data)

Bases: PluginProperties, PluginModel

The part properties used by the Go plugin.

Parameters

data (Any) –

go_buildtags: List[str]

go_generate: List[str]

source: str

classmethod unmarshal(data)

Populate make properties from the part specification.

Parameters

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

Return type

GoPluginProperties

Returns

The populated plugin properties data object.

Raises

pydantic.ValidationError – If validation fails.

craft_parts.plugins.make_plugin module

The make plugin implementation.

class craft_parts.plugins.make_plugin.**MakePlugin**(*, *properties*, *part_info*)

Bases: *Plugin*

A plugin useful for building make-based parts.

Make-based projects are projects that have a Makefile that drives the build.

This plugin always runs 'make' followed by 'make install', except when the 'artifacts' keyword is used.

This plugin uses the common plugin keywords as well as those for "sources". For more information check the 'plugins' topic for the former and the 'sources' topic for the latter.

Additionally, this plugin uses the following plugin-specific keywords:

- make-parameters (list of strings) Pass the given parameters to the make command.

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
List[str]

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
Dict[str, str]

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
Set[str]

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
Set[str]

properties_class

alias of *MakePluginProperties*

class craft_parts.plugins.make_plugin.**MakePluginProperties**(**data*)

Bases: *PluginProperties*, *PluginModel*

The part properties used by the make plugin.

Parameters
data (Any) –

make_parameters: List[str]

source: str

classmethod unmarshal(*data*)

Populate make properties from the part specification.

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

Return type
MakePluginProperties

Returns
The populated plugin properties data object.

Raises
pydantic.ValidationError – If validation fails.

craft_parts.plugins.maven_plugin module

The maven plugin.

class craft_parts.plugins.maven_plugin.**MavenPlugin**(*, *properties*, *part_info*)

Bases: *JavaPlugin*

A plugin to build parts that use Maven.

The Maven build system is commonly used to build Java projects. This plugin requires a pom.xml in the root of the source tree.

This plugin uses the common plugin keywords as well as those for "sources". For more information check the 'plugins' topic for the former and the 'sources' topic for the latter.

Additionally, this plugin uses the following plugin-specific keywords:

- maven-parameters: (list of strings) Flags to pass to the build using the maven semantics for parameters.

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
List[str]

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
Dict[str, str]

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
Set[str]

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
Set[str]

properties_class

alias of *MavenPluginProperties*

validator_class

alias of *MavenPluginEnvironmentValidator*

class craft_parts.plugins.maven_plugin.**MavenPluginEnvironmentValidator**(*, *part_name*, *env*, *properties*)

Bases: *PluginEnvironmentValidator*

Check the execution environment for the maven plugin.

Parameters

- **part_name** (str) – The part whose build environment is being validated.
- **env** (str) – A string containing the build step environment setup.
- **properties** (*PluginProperties*) –

validate_environment(*, *part_dependencies=None*)

Ensure the environment contains dependencies needed by the plugin.

Parameters
part_dependencies (Optional[List[str]]) – A list of the parts this part depends on.

Raises
PluginEnvironmentValidationError – If go is invalid

and there are no parts named go.

Return type
None

class craft_parts.plugins.maven_plugin.**MavenPluginProperties**(**data*)

Bases: *PluginProperties*, *PluginModel*

The part properties used by the maven plugin.

Parameters
data (Any) –

maven_parameters: List[str]

source: str

classmethod unmarshal(*data*)

Populate class attributes from the part specification.

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

Return type
MavenPluginProperties

Returns
The populated plugin properties data object.

Raises
pydantic.ValidationError – If validation fails.

craft_parts.plugins.meson_plugin module

The Meson plugin.

class craft_parts.plugins.meson_plugin.**MesonPlugin**(*, *properties*, *part_info*)

Bases: *Plugin*

A plugin for meson projects.

The meson plugin requires meson installed on your system. This can be achieved by adding the appropriate meson package to `build-packages` or `build-snaps`, or to have it installed or built in a different part. In this case, the name of the part supplying meson must be "meson".

The meson plugin uses the common plugin keywords as well as those for "sources". Additionally, the following plugin-specific keywords can be used:

- `meson-parameters` (list of strings) List of parameters used to configure the meson based project.

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
`List[str]`

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
`Dict[str, str]`

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
`Set[str]`

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
`Set[str]`

classmethod get_out_of_source_build()

Return whether the plugin performs out-of-source-tree builds.

Return type
`bool`

properties_class

alias of *MesonPluginProperties*

validator_class

alias of *MesonPluginEnvironmentValidator*

class craft_parts.plugins.meson_plugin.**MesonPluginEnvironmentValidator**(*, *part_name*, *env*, *properties*)

Bases: *PluginEnvironmentValidator*

Check the execution environment for the Meson plugin.

Parameters

- **part_name** (str) – The part whose build environment is being validated.
- **env** (str) – A string containing the build step environment setup.
- **properties** (*PluginProperties*) –

validate_environment(*, *part_dependencies=None*)

Ensure the environment contains dependencies needed by the plugin.

Parameters
part_dependencies (Optional[List[str]]) – A list of the parts this part depends on.

Raises
PluginEnvironmentValidationError – If the environment is invalid.

Return type
None

class craft_parts.plugins.meson_plugin.**MesonPluginProperties**(**data*)

Bases: *PluginProperties*, *PluginModel*

The part properties used by the Go plugin.

Parameters
data (Any) –

meson_parameters: List[str]

source: str

classmethod unmarshal(*data*)

Populate make properties from the part specification.

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

Return type
MesonPluginProperties

Returns
The populated plugin properties data object.

Raises
pydantic.ValidationError – If validation fails.

craft_parts.plugins.nil_plugin module

The nil plugin.

Using this, parts can be defined purely by utilizing properties that are automatically included, e.g. stage-packages.

class craft_parts.plugins.nil_plugin.**NilPlugin**(*, *properties*, *part_info*)

Bases: *Plugin*

A plugin that defines no build commands.

The nil plugin is useful in two contexts:

First, it can be used for parts that identify no source, and can be defined purely by using built-in part properties such as `stage-packages`.

The second use is for parts that do define a source (which will be fetched), but for which the build step then needs to be explicitly defined using `override-build`; otherwise, even though the source is fetched, nothing will end up in that part's install directory. In short, for the case of a part that uses the nil plugin and defines a source, it is up to the developer to then define the `override-build` step that, in some way, populates the `$CRAFT_PART_INSTALL` directory.

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
`List[str]`

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
`Dict[str, str]`

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
`Set[str]`

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
`Set[str]`

properties_class

alias of *NilPluginProperties*

class craft_parts.plugins.nil_plugin.**NilPluginProperties**

Bases: *PluginProperties*

The part properties used by the nil plugin.

classmethod unmarshal(*data*)

Populate class attributes from the part specification.

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

Return type
NilPluginProperties

Returns
The populated plugin properties data object.

craft_parts.plugins.npm_plugin module

The npm plugin.

class craft_parts.plugins.npm_plugin.**NpmPlugin**(*, *properties*, *part_info*)

Bases: *Plugin*

A plugin for npm projects.

This plugin uses the common plugin keywords as well as those for "sources". For more information check the 'plugins' topic for the former and the 'sources' topic for the latter.

Additionally, this plugin uses the following plugin-specific keywords:

- npm-include-node (bool; default: False) If true, download and include the node binary and its dependencies. If npm-include-node is true, then npm-node-version must be defined.
- npm-node-version (str: default: None) Which version of node to download (e.g. "16.14.2")

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
List[str]

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
Dict[str, str]

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
Set[str]

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
Set[str]

properties_class

alias of *NpmPluginProperties*

validator_class

alias of *NpmPluginEnvironmentValidator*

class craft_parts.plugins.npm_plugin.**NpmPluginEnvironmentValidator**(*, *part_name*, *env*, *properties*)

Bases: *PluginEnvironmentValidator*

Check the execution environment for the npm plugin.

Parameters

- **part_name** (str) – The part whose build environment is being validated.
- **env** (str) – A string containing the build step environment setup.
- **properties** (*PluginProperties*) –

validate_environment(*, *part_dependencies=None*)

Ensure the environment has the dependencies to build npm applications.

Parameters
part_dependencies (Optional[List[str]]) – A list of the parts this part depends on.

Return type
None

class craft_parts.plugins.npm_plugin.**NpmPluginProperties**(**data*)

Bases: *PluginProperties*, *PluginModel*

The part properties used by the npm plugin.

Parameters
data (Any) –

classmethod node_version_defined(*values*)

If npm-include-node is true, then npm-node-version must be defined.

Parameters
values (Dict[str, Any]) –

Return type
Dict[str, Any]

npm_include_node: bool

npm_node_version: Optional[str]

source: str

classmethod unmarshal(*data*)

Populate class attributes from the part specification.

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

Return type
NpmPluginProperties

Returns
The populated plugin properties data object.

Raises
pydantic.ValidationError – If validation fails.

craft_parts.plugins.plugins module

Definitions and helpers to handle plugins.

craft_parts.plugins.plugins.**extract_part_properties**(*data*, *, *plugin_name*)

Get common part properties without plugin-specific entries.

Parameters

- **data** (Dict[str, Any]) – A dictionary containing all part properties.
- **plugin_name** (str) – The name of the plugin.

Return type
Dict[str, Any]

Returns
A dictionary containing only common part properties.

craft_parts.plugins.plugins.**get_plugin**(*, *part*, *part_info*, *properties*)

Obtain a plugin instance for the specified part.

Parameters

- **part** (*Part*) – The part requesting the plugin.
- **part_info** (*PartInfo*) – The part information data.
- **properties** (*PluginProperties*) – The plugin properties.

Return type
Plugin

Returns
The plugin instance.

craft_parts.plugins.plugins.**get_plugin_class**(*name*)

Obtain a plugin class given the name.

Parameters
name (str) – The plugin name.

Return type
Type[*Plugin*]

Returns
The plugin class.

Raises
ValueError – If the plugin name is invalid.

craft_parts.plugins.plugins.**get_registered_plugins**()

Return the list of currently registered plugins.

Return type
Dict[str, Type[*Plugin*]]

craft_parts.plugins.plugins.**register**(*plugins*)

Register part handler plugins.

Parameters
plugins (Dict[str, Type[*Plugin*]]) – a dictionary where the keys are plugin names and values are plugin classes. Valid plugins must subclass class:*Plugin*.

Return type
None

craft_parts.plugins.plugins.**unregister_all**()

Unregister all user-registered plugins.

Return type
None

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

The python plugin.

class craft_parts.plugins.python_plugin.**PythonPlugin**(*, *properties*, *part_info*)

Bases: *Plugin*

A plugin to build python parts.

It can be used for python projects where you would want to do:

- import python modules with a requirements.txt
- build a python project that has a setup.py
- install packages straight from pip

This plugin uses the common plugin keywords as well as those for "sources". For more information check the 'plugins' topic for the former and the 'sources' topic for the latter.

Additionally, this plugin uses the following plugin-specific keywords:

- `python-requirements` (list of strings) List of paths to requirements files.
- `python-constraints` (list of strings) List of paths to constraint files.
- `python-packages` (list) A list of dependencies to get from PyPI. If needed, pip, setuptools and wheel can be upgraded here.

This plugin also interprets these specific build-environment entries:

- `PARTS_PYTHON_INTERPRETER` (default: python3) The interpreter binary to search for in PATH.
- `PARTS_PYTHON_VENV_ARGS` Additional arguments for venv.

By default this plugin uses python from the base. If a different interpreter is desired, it must be bundled (including venv) and must be in PATH.

Use of python3-<python-package> in stage-packages will force the inclusion of the python interpreter.

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
`List[str]`

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
`Dict[str, str]`

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
`Set[str]`

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
`Set[str]`

properties_class

alias of *PythonPluginProperties*

class craft_parts.plugins.python_plugin.**PythonPluginProperties**(***data*)

Bases: *PluginProperties*, *PluginModel*

The part properties used by the python plugin.

Parameters
data (`Any`) –

python_constraints: List[str]

python_packages: List[str]

python_requirements: List[str]

source: str

classmethod unmarshal(*data*)

Populate make properties from the part specification.

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

Return type
PythonPluginProperties

Returns
The populated plugin properties data object.

Raises
pydantic.ValidationError – If validation fails.

craft_parts.plugins.rust_plugin module

The rust plugin.

class craft_parts.plugins.rust_plugin.**RustPlugin**(***, *properties*, *part_info*)

Bases: *Plugin*

A plugin for rust projects.

This plugin uses the common plugin keywords as well as those for "sources". For more information check the 'plugins' topic for the former and the 'sources' topic for the latter.

Additionally, this plugin uses the following plugin-specific keywords:

- rust-features (list of unique strings; default: []) List of rust features to install.
- rust-path (list of unique strings; default: ["."]) Path of rust project. Currently, only the first path in the list is used.

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
List[str]

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
Dict[str, str]

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
Set[str]

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
Set[str]

properties_class

alias of *RustPluginProperties*

validator_class

alias of *RustPluginEnvironmentValidator*

class craft_parts.plugins.rust_plugin.**RustPluginEnvironmentValidator**(*, *part_name*, *env*, *properties*)

Bases: *PluginEnvironmentValidator*

Check the execution environment for the Rust plugin.

Parameters

- **part_name** (str) – The part whose build environment is being validated.
- **env** (str) – A string containing the build step environment setup.
- **properties** (*PluginProperties*) –

validate_environment(*, *part_dependencies=None*)

Ensure the environment has the dependencies to build Rust applications.

Parameters
part_dependencies (Optional[List[str]]) – A list of the parts this part depends on.

Return type
None

class craft_parts.plugins.rust_plugin.**RustPluginProperties**(**data*)

Bases: *PluginProperties*, *PluginModel*

The part properties used by the rust plugin.

Parameters
data (Any) –

rust_features: ConstrainedListValue[str]

rust_path: ConstrainedListValue[str]

source: str

classmethod unmarshal(*data*)

Populate class attributes from the part specification.

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

Return type
RustPluginProperties

Returns
The populated plugin properties data object.

Raises
pydantic.ValidationError – If validation fails.

craft_parts.plugins.scons_plugin module

The SCons plugin.

class craft_parts.plugins.scons_plugin.**SConsPlugin**(*, *properties*, *part_info*)

Bases: *Plugin*

A plugin for SCons projects.

The plugin needs the `scons` tool which can be provisioned in one of the following ways:

- Add "scons" to the part's `build-packages`;
- Build a custom version of `scons` on a separate part called `scons-deps`

and have the part that uses this plugin depend on the `scons-deps` part.

Note that other dependencies (C/C++ compiler, Java compiler, etc) must be declared via `build-packages` or otherwise provisioned.

Since there is no "official" way of defining the target installation directory for SCons-built artifacts, the default build will set the DESTDIR environment variable which contains the root which the SConstruct file should use to configure its `Install()` builder target.

The plugin supports the following keywords:

- `scons-parameters` (list of strings) Additional values to pass to the `scons` and `scons install` command lines.

Parameters

- **properties** (*PluginProperties*) –
- **part_info** (*PartInfo*) –

get_build_commands()

Return a list of commands to run during the build step.

Return type
`List[str]`

get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
`Dict[str, str]`

get_build_packages()

Return a set of required packages to install in the build environment.

Return type
`Set[str]`

get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
`Set[str]`

properties_class

alias of *SConsPluginProperties*

validator_class

alias of *SConsPluginEnvironmentValidator*

class craft_parts.plugins.scons_plugin.**SConsPluginEnvironmentValidator**(*, *part_name*, *env*, *properties*)

Bases: *PluginEnvironmentValidator*

Check the execution environment for the SCons plugin.

Parameters

- **part_name** (str) – The part whose build environment is being validated.
- **env** (str) – A string containing the build step environment setup.
- **properties** (*PluginProperties*) –

validate_environment(*, *part_dependencies=None*)

Ensure the environment contains dependencies needed by the plugin.

Parameters
part_dependencies (Optional[List[str]]) – A list of the parts this part depends on.

Raises
PluginEnvironmentValidationError – If scons is invalid

and there are no parts named "scons-deps".

Return type
None

class craft_parts.plugins.scons_plugin.**SConsPluginProperties**(**data*)

Bases: *PluginProperties*, *PluginModel*

The part properties used by the SCons plugin.

Parameters
data (Any) –

scons_parameters: List[str]

source: str

classmethod unmarshal(*data*)

Populate make properties from the part specification.

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

Return type
SConsPluginProperties

Returns
The populated plugin properties data object.

Raises
pydantic.ValidationError – If validation fails.

craft_parts.plugins.validator module

Definitions and helpers for plugin environment validation.

craft_parts.plugins.validator.**COMMAND_NOT_FOUND = 127**

The shell error code for command not found.

class craft_parts.plugins.validator.**PluginEnvironmentValidator**(*, *part_name*, *env*, *properties*)

Bases: object

Base class for plugin environment validators.

Plugins may require certain environment elements to be present in order to build a part, regardless of how these elements were installed on the system. For example, a compiler may have been installed from a deb package, a snap, built from sources or even built by a different part. Plugin environment validators allow a plugin to ensure its execution environment is correct before building a part.

Parameters

- **part_name** (str) – The part whose build environment is being validated.
- **env** (str) – A string containing the build step environment setup.
- **properties** (*PluginProperties*) –

validate_dependency(*dependency*, *plugin_name*, *part_dependencies*, *argument='--version'*)

Validate that the environment has a required dependency.

<*dependency-name*> *–version* is executed to confirm the dependency is valid.

Parameters

- **dependency** (str) – name of the dependency to validate.
- **plugin_name** (str) – used to generate the part name that would satisfy the dependency.
- **part_dependencies** (Optional[List[str]]) – A list of the parts this part depends on.
- **argument** (str) – argument to call with the dependency. Default is *–version*.

Raises

PluginEnvironmentValidationError – If the environment is invalid.

Return type

str

Returns

output from executed dependency

validate_environment(*, *part_dependencies=None*)

Ensure the plugin execution environment is valid.

The environment is verified twice: during the execution prologue after build packages and snaps are installed (to provide an early error message if the environment is invalid), and before running the build step for the part. During the prologue validation the environment may be incomplete, so we pass a list of the part dependencies as a hint of which parts may be used to build the environment.

Parameters

part_dependencies (Optional[List[str]]) – A list of the parts this part depends on, so the validator can check if the required environment elements are supplied by another part when the method is called from the execution prologue. If the validation fails and list is empty, the error is final (the missing elements can't be supplied by a different part). The plugin may

require a specific part name as a hint that the part will attempt to supply missing environment elements.

Raises
PluginEnvironmentValidationError – If the environment is invalid.

Return type
None

Module contents

Craft Parts plugins subsystem.

class craft_parts.plugins.**Plugin**(*, *properties*, *part_info*)

Bases: ABC

The base class for plugins.

Variables

- **properties_class** – The plugin properties class.
- **validator_class** – The plugin environment validator class.

Parameters

- **part_info** (*PartInfo*) – The part information for the applicable part.
- **properties** (*PluginProperties*) – Part-defined properties.

abstract get_build_commands()

Return a list of commands to run during the build step.

Return type
List[str]

abstract get_build_environment()

Return a dictionary with the environment to use in the build step.

Return type
Dict[str, str]

abstract get_build_packages()

Return a set of required packages to install in the build environment.

Return type
Set[str]

abstract get_build_snaps()

Return a set of required snaps to install in the build environment.

Return type
Set[str]

classmethod get_out_of_source_build()

Return whether the plugin performs out-of-source-tree builds.

Return type
bool

properties_class: Type[*PluginProperties*]

set_action_properties(*action_properties*)

Store a copy of the given action properties.

Parameters
action_properties (*ActionProperties*) – The properties to store.

Return type
None

validator_class

alias of *PluginEnvironmentValidator*

class craft_parts.plugins.**PluginEnvironmentValidator**(*, *part_name*, *env*, *properties*)

Bases: object

Base class for plugin environment validators.

Plugins may require certain environment elements to be present in order to build a part, regardless of how these elements were installed on the system. For example, a compiler may have been installed from a deb package, a snap, built from sources or even built by a different part. Plugin environment validators allow a plugin to ensure its execution environment is correct before building a part.

Parameters

- **part_name** (str) – The part whose build environment is being validated.
- **env** (str) – A string containing the build step environment setup.
- **properties** (*PluginProperties*) –

validate_dependency(*dependency*, *plugin_name*, *part_dependencies*, *argument='--version'*)

Validate that the environment has a required dependency.

<dependency-name> –version is executed to confirm the dependency is valid.

Parameters

- **dependency** (str) – name of the dependency to validate.
- **plugin_name** (str) – used to generate the part name that would satisfy the dependency.
- **part_dependencies** (Optional[List[str]]) – A list of the parts this part depends on.
- **argument** (str) – argument to call with the dependency. Default is *–version*.

Raises
PluginEnvironmentValidationError – If the environment is invalid.

Return type
str

Returns
output from executed dependency

validate_environment(*, *part_dependencies=None*)

Ensure the plugin execution environment is valid.

The environment is verified twice: during the execution prologue after build packages and snaps are installed (to provide an early error message if the environment is invalid), and before running the build step for the part. During the prologue validation the environment may be incomplete, so we pass a list of the part dependencies as a hint of which parts may be used to build the environment.

Parameters
part_dependencies (Optional[List[str]]) – A list of the parts this part depends on, so the validator can check if the required environment elements are supplied by another part when the method is called from the execution prologue. If the validation fails and list is empty, the error is final (the missing elements can't be supplied by a different part). The plugin may require a specific part name as a hint that the part will attempt to supply missing environment elements.

Raises
PluginEnvironmentValidationError – If the environment is invalid.

Return type
None

class craft_parts.plugins.**PluginModel**(***data*)

Bases: BaseModel

Model for plugins using pydantic validation.

Plugins with configuration properties can use pydantic validation to unmarshal data from part specs. In this case, extract plugin-specific properties using the *extract_plugin_properties()* helper.

Parameters
data (Any) –

class Config

Bases: object

Pydantic model configuration.

alias_generator()

allow_mutation = False

extra = 'forbid'

validate_assignment = True

class craft_parts.plugins.**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.**extract_part_properties**(*data*, ***, *plugin_name*)

Get common part properties without plugin-specific entries.

Parameters

- **data** (Dict[str, Any]) – A dictionary containing all part properties.
- **plugin_name** (str) – The name of the plugin.

Return type
Dict[str, Any]

Returns
A dictionary containing only common part properties.

craft_parts.plugins.**extract_plugin_properties**(*data*, *, *plugin_name*, *required=None*)

Obtain plugin-specifc entries from part properties.

Parameters

- **data** (Dict[str, Any]) – A dictionary containing all part properties.
- **plugin_name** (str) –
- **required** (Optional[List[str]]) –

Plugin_name
The name of the plugin.

Return type
Dict[str, Any]

Returns
A dictionary with plugin properties.

craft_parts.plugins.**get_plugin**(*, *part*, *part_info*, *properties*)

Obtain a plugin instance for the specified part.

Parameters

- **part** (*Part*) – The part requesting the plugin.
- **part_info** (*PartInfo*) – The part information data.
- **properties** (*PluginProperties*) – The plugin properties.

Return type
Plugin

Returns
The plugin instance.

craft_parts.plugins.**get_plugin_class**(*name*)

Obtain a plugin class given the name.

Parameters
name (str) – The plugin name.

Return type
Type[*Plugin*]

Returns
The plugin class.

Raises
ValueError – If the plugin name is invalid.

craft_parts.plugins.**get_registered_plugins**()

Return the list of currently registered plugins.

Return type
Dict[str, Type[*Plugin*]]

craft_parts.plugins.**register**(*plugins*)

Register part handler plugins.

Parameters
plugins (Dict[str, Type[*Plugin*]]) – a dictionary where the keys are plugin names and values are plugin classes. Valid plugins must subclass class:*Plugin*.

Return type
None

craft_parts.plugins.**unregister_all**()

Unregister all user-registered plugins.

Return type
None

craft_parts.sources package

Submodules

craft_parts.sources.base module

Base classes for source type handling.

class craft_parts.sources.base.**FileSourceHandler**(*source*, *part_src_dir*, *, *cache_dir*, *source_tag=None*, *source_commit=None*, *source_branch=None*, *source_depth=None*, *source_checksum=None*, *source_submodules=None*, *command=None*, *project_dirs=None*, *ignore_patterns=None*)

Bases: *SourceHandler*

Base class for file source types.

Parameters
- **source** (str) –
- **part_src_dir** (Path) –
- **cache_dir** (Path) –
- **source_tag** (Optional[str]) –
- **source_commit** (Optional[str]) –
- **source_branch** (Optional[str]) –
- **source_depth** (Optional[int]) –
- **source_checksum** (Optional[str]) –
- **source_submodules** (Optional[List[str]]) –
- **command** (Optional[str]) –
- **project_dirs** (Optional[*ProjectDirs*]) –

- **ignore_patterns** (Optional[List[str]]) –

download(*filepath=None*)

Download the URL from a remote location.

Parameters
filepath (Optional[Path]) – the destination file to download to.

Return type
Path

abstract provision(*dst*, *keep=False*, *src=None*)

Process the source file to extract its payload.

Parameters

- **dst** (Path) –
- **keep** (bool) –
- **src** (Optional[Path]) –

Return type
None

pull()

Retrieve this source from its origin.

Return type
None

class craft_parts.sources.base.**SourceHandler**(*source*, *part_src_dir*, ***, *cache_dir*, *source_tag=None*, *source_commit=None*, *source_branch=None*, *source_depth=None*, *source_checksum=None*, *source_submodules=None*, *command=None*, *project_dirs=None*, *ignore_patterns=None*)

Bases: ABC

The base class for source type handlers.

Methods *check_if_outdated()* and update_source() can be overridden by subclasses to implement verification and update of source files.

Parameters

- **source** (str) –
- **part_src_dir** (Path) –
- **cache_dir** (Path) –
- **source_tag** (Optional[str]) –
- **source_commit** (Optional[str]) –
- **source_branch** (Optional[str]) –
- **source_depth** (Optional[int]) –
- **source_checksum** (Optional[str]) –
- **source_submodules** (Optional[List[str]]) –
- **command** (Optional[str]) –
- **project_dirs** (Optional[*ProjectDirs*]) –

- **ignore_patterns** (Optional[List[str]]) –

check_if_outdated(*target*, ***, *ignore_files=None*)

Check if pulled sources have changed since target was created.

Parameters

- **target** (str) – Path to target file.
- **ignore_files** (Optional[List[str]]) – Files excluded from verification.

Return type
bool

Returns
Whether the sources are outdated.

Raises
errors.SourceUpdateUnsupported – If the source handler can't check if files are outdated.

get_outdated_files()

Obtain lists of outdated files and directories.

Return type
Tuple[List[str], List[str]]

Returns
The lists of outdated files and directories.

Raises
errors.SourceUpdateUnsupported – If the source handler can't check if files are outdated.

abstract pull()

Retrieve the source file.

Return type
None

update()

Update pulled source.

Raises
errors.SourceUpdateUnsupported – If the source can't update its files.

Return type
None

craft_parts.sources.cache module

Cache base and file cache.

class craft_parts.sources.cache.**FileCache**(*cache_dir*, ***, *namespace='files'*)

Bases: object

Cache files based on the supplied key.

Parameters

- **cache_dir** (Path) –

- **namespace** (str) –

cache(*, *filename*, *key*)

Cache a file revision with hash in XDG cache, unless it already exists.

Parameters

- **filename** (str) – The path to the file to cache.
- **key** (str) – The key to cache the file under.

Return type
Optional[Path]

Returns
The path to the cached file, or None if the file was not cached.

clean()

Remove all files from the cache namespace.

Return type
None

get(*, *key*)

Get the filepath which matches the hash calculated with algorithm.

Parameters
key (str) – The key used to cache the file.

Return type
Optional[Path]

Returns
The path to cached file, or None if the file is not cached.

craft_parts.sources.checksum module

Helpers to compute and verify file checksums.

craft_parts.sources.checksum.**split_checksum**(*source_checksum*)

Split the given source checksum into algorithm and hash.

Parameters
source_checksum (str) – Source checksum in algorithm/hash format.

Return type
Tuple

Returns
a tuple consisting of the algorithm and the hash.

Raises
ValueError – If the checksum is not in the expected format.

craft_parts.sources.checksum.**verify_checksum**(*source_checksum*, *checkfile*)

Verify that checkfile corresponds to the given source checksum.

Parameters

- **source_checksum** (str) – Source checksum in algorithm/hash format.

- **checkfile** (Path) – The file to calculate the sum for with the algorithm defined in source_checksum.

Return type
Tuple

Returns
A tuple consisting of the algorithm and the hash.

Raises

- **ValueError** – If source_checksum is not of the form algorithm/hash.
- *ChecksumMismatch* – If checkfile does not match the expected hash calculated with the algorithm defined in source_checksum.

craft_parts.sources.deb_source module

The deb source handler.

class craft_parts.sources.deb_source.**DebSource**(*source*, *part_src_dir*, *, *cache_dir*, *source_tag=None*, *source_commit=None*, *source_branch=None*, *source_checksum=None*, *source_submodules=None*, *source_depth=None*, *project_dirs=None*, *ignore_patterns=None*)

Bases: *FileSourceHandler*

The "deb" file source handler.

Parameters

- **source** (str) –
- **part_src_dir** (Path) –
- **cache_dir** (Path) –
- **source_tag** (Optional[str]) –
- **source_commit** (Optional[str]) –
- **source_branch** (Optional[str]) –
- **source_checksum** (Optional[str]) –
- **source_submodules** (Optional[List[str]]) –
- **source_depth** (Optional[int]) –
- **project_dirs** (Optional[*ProjectDirs*]) –
- **ignore_patterns** (Optional[List[str]]) –

provision(*dst*, *keep=False*, *src=None*)

Extract deb file contents to the part source dir.

Parameters

- **dst** (Path) –
- **keep** (bool) –
- **src** (Optional[Path]) –

Return type
None

craft_parts.sources.errors module

Source handler error definitions.

exception craft_parts.sources.errors.**ChecksumMismatch**(*, *expected*, *obtained*)

Bases: *SourceError*

A checksum doesn't match the expected value.

Parameters

- **expected** (str) – The expected checksum.
- **obtained** (str) – The actual checksum.

brief: str

exception craft_parts.sources.errors.**IncompatibleSourceOptions**(*source_type*, *options*)

Bases: *SourceError*

Source specified options that cannot be used at the same time.

Parameters

- **source_type** (str) – The part's source type.
- **options** (List[str]) – The list of incompatible source options.

brief: str

exception craft_parts.sources.errors.**InvalidSnapPackage**(*snap_file*)

Bases: *SourceError*

A snap package is invalid.

Parameters
snap_file (str) – The snap file name.

brief: str

exception craft_parts.sources.errors.**InvalidSourceOption**(*, *source_type*, *option*)

Bases: *SourceError*

A source option is not allowed for the given source type.

Parameters

- **source_type** (str) – The part's source type.
- **option** (str) – The invalid source option.

brief: str

exception craft_parts.sources.errors.**InvalidSourceType**(*source*)

Bases: *SourceError*

Failed to determine a source type.

Parameters
source (str) – The source defined for the part.

brief: str

exception craft_parts.sources.errors.**NetworkRequestError**(*message*)

Bases: *SourceError*

A network request operation failed.

Parameters

message (str) – The error message.

brief: str

exception craft_parts.sources.errors.**PullError**(*, *command*, *exit_code*)

Bases: *SourceError*

Failed pulling source.

Parameters

- **command** (Sequence) – The command used to pull the source.
- **exit_code** (int) – The command exit code.

brief: str

exception craft_parts.sources.errors.**SourceError**(*brief*, *details=None*, *resolution=None*)

Bases: *PartsError*

Base class for source handler errors.

Parameters

- **brief** (str) –
- **details** (Optional[str]) –
- **resolution** (Optional[str]) –

brief: str

exception craft_parts.sources.errors.**SourceNotFound**(*source*)

Bases: *SourceError*

Failed to retrieve a source.

Parameters

source (str) – The source defined for the part.

brief: str

exception craft_parts.sources.errors.**SourceUpdateUnsupported**(*name*)

Bases: *SourceError*

The source handler doesn't support updating.

Parameters

name (str) – The source type.

brief: str

exception craft_parts.sources.errors.**VCSError**(*message*)

Bases: *SourceError*

A version control system command failed.

Parameters
message (str) –

brief: str

craft_parts.sources.file_source module

Implement the plain file source handler.

class craft_parts.sources.file_source.**FileSource**(*source*, *part_src_dir*, *, *cache_dir*, *source_tag=None*, *source_commit=None*, *source_branch=None*, *source_depth=None*, *source_submodules=None*, *source_checksum=None*, *project_dirs=None*, *ignore_patterns=None*)

Bases: *FileSourceHandler*

The plain file source handler.

Parameters

- **source** (str) –
- **part_src_dir** (Path) –
- **cache_dir** (Path) –
- **source_tag** (Optional[str]) –
- **source_commit** (Optional[str]) –
- **source_branch** (Optional[str]) –
- **source_depth** (Optional[int]) –
- **source_submodules** (Optional[List[str]]) –
- **source_checksum** (Optional[str]) –
- **project_dirs** (Optional[*ProjectDirs*]) –
- **ignore_patterns** (Optional[List[str]]) –

provision(*dst*, *keep=False*, *src=None*)

Process the source file to extract its payload.

Parameters

- **dst** (Path) –
- **keep** (bool) –
- **src** (Optional[Path]) –

Return type
None

craft_parts.sources.git_source module

Implement the git source handler.

class craft_parts.sources.git_source.**GitSource**(*source*, *part_src_dir*, *, *cache_dir*, *source_tag=None*, *source_commit=None*, *source_depth=None*, *source_branch=None*, *source_checksum=None*, *source_submodules=None*, *project_dirs=None*, *ignore_patterns=None*)

Bases: *SourceHandler*

The git source handler.

Retrieve part sources from a git repository. Branch, depth, commit and tag can be specified using part properties `source-branch`, `source-depth`, *source-commit* `, `source-tag`, and `source-submodules`.

Parameters

- **source** (str) –
- **part_src_dir** (Path) –
- **cache_dir** (Path) –
- **source_tag** (Optional[str]) –
- **source_commit** (Optional[str]) –
- **source_depth** (Optional[int]) –
- **source_branch** (Optional[str]) –
- **source_checksum** (Optional[str]) –
- **source_submodules** (Optional[List[str]]) –
- **project_dirs** (Optional[*ProjectDirs*]) –
- **ignore_patterns** (Optional[List[str]]) –

classmethod check_command_installed()

Check if git is installed.

Return type
bool

classmethod generate_version(*, *part_src_dir=None*)

Return the latest git tag from PWD or defined part_src_dir.

The output depends on the use of annotated tags and will return something like: '2.28+git.10.abcdef' where '2.28 is the tag, '+git' indicates there are commits ahead of the tag, in this case it is '10' and the latest commit hash begins with 'abcdef'. If there are no tags or the revision cannot be determined, this will return 0 as the tag and only the commit hash of the latest commit.

Parameters
part_src_dir (Optional[Path]) –

Return type
str

is_local()

Verify whether the git repository is on the local filesystem.

Return type
bool

pull()

Retrieve the local or remote source files.

Return type
None

classmethod version()

Get git version information.

Return type
str

craft_parts.sources.local_source module

The local source handler and helpers.

class craft_parts.sources.local_source.**LocalSource**(**args*, *copy_function=<function link_or_copy>*, ***kwargs*)

Bases: *SourceHandler*

The local source handler.

Parameters

- **args** (Any) –
- **copy_function** (Callable[..., None]) –
- **kwargs** (Any) –

check_if_outdated(*target*, ***, *ignore_files=None*)

Check if pulled sources have changed since target was created.

Parameters

- **target** (str) – Path to target file.
- **ignore_files** (Optional[List[str]]) – Files excluded from verification.

Return type
bool

Returns
Whether the sources are outdated.

get_outdated_files()

Obtain lists of outdated files and directories.

Return type
Tuple[List[str], List[str]]

Returns
The lists of outdated files and directories.

Raises
errors.SourceUpdateUnsupported – If the source handler can't check if files are outdated.

pull()

Retrieve the local source files.

Return type
None

update()

Update pulled source.

Call method *check_if_outdated()* before updating to populate the lists of files and directories to copy.

Return type
None

craft_parts.sources.snap_source module

The snap source handler.

class craft_parts.sources.snap_source.**SnapSource**(*source*, *part_src_dir*, *, *cache_dir*, *source_tag=None*, *source_commit=None*, *source_branch=None*, *source_depth=None*, *source_checksum=None*, *project_dirs=None*)

Bases: *FileSourceHandler*

Handles downloading and extractions for a snap source.

On provision, the meta directory is renamed to meta.<snap-name> and, if present, the same applies for the snap directory which shall be renamed to snap.<snap-name>.

Parameters

- **source** (str) –
- **part_src_dir** (Path) –
- **cache_dir** (Path) –
- **source_tag** (Optional[str]) –
- **source_commit** (Optional[str]) –
- **source_branch** (Optional[str]) –
- **source_depth** (Optional[int]) –
- **source_checksum** (Optional[str]) –
- **project_dirs** (Optional[*ProjectDirs*]) –

provision(*dst*, *keep=False*, *src=None*)

Provision the snap source.

Parameters

- **dst** (Path) – The destination directory to provision to.
- **keep** (bool) – Whether to keep the snap after provisioning is complete.
- **src** (Optional[Path]) – Force a new source to use for extraction.

raises errors.InvalidSnap: If trying to provision an invalid snap.

Return type
None

craft_parts.sources.sources module

Source handle utilities.

Unless the part plugin overrides this behaviour, a part can use these 'source' keys in its definition. They tell Craft Parts where to pull source code for that part, and how to unpack it if necessary.

- source: url-or-path

  A URL or path to some source tree to build. It can be local ('./src/foo') or remote ('https://foo.org/...'), and can refer to a directory tree or a tarball or a revision control repository ('git:...').

- source-type: git, bzr, hg, svn, tar, deb, rpm, or zip

  In some cases the source string is not enough to identify the version control system or compression algorithm. The source-type key can tell Craft Parts exactly how to treat that content.

- source-checksum: <algorithm>/<digest>

  Craft Parts will use the digest specified to verify the integrity of the source. The source-type needs to be a file (tar, zip, deb or rpm) and the algorithm either md5, sha1, sha224, sha256, sha384, sha512, sha3_256, sha3_384 or sha3_512.

- source-depth: <integer>

  By default clones or branches with full history, specifying a depth will truncate the history to the specified number of commits.

- source-branch: <branch-name>

  Craft Parts will checkout a specific branch from the source tree. This only works on multi-branch repositories from git and hg (mercurial).

- source-commit: <commit>

  Craft Parts will checkout the specific commit from the source tree revision control system.

- source-tag: <tag>

  Craft Parts will checkout the specific tag from the source tree revision control system.

- source-subdir: path

  When building, Snapcraft will set the working directory to be this subdirectory within the source.

- source-submodules: <list-of-submodules>

  Configure which submodules to fetch from the source tree. If source-submodules in defined and empty, no submodules are fetched. If source-submodules is not defined, all submodules are fetched (default behavior).

Note that plugins might well define their own semantics for the 'source' keywords, because they handle specific build systems, and many languages have their own built-in packaging systems (think CPAN, PyPI, NPM). In those cases you want to refer to the documentation for the specific plugin.

`craft_parts.sources.sources.`**`get_source_handler`**(*cache_dir*, *part*, *project_dirs*, *ignore_patterns=None*)

Return the appropriate handler for the given source.

Parameters

- **application_name** – The name of the application using Craft Parts.
- **part** (*Part*) – The part to get a source handler for.
- **project_dirs** (*ProjectDirs*) – The project's work directories.
- **cache_dir** (Path) –

- **ignore_patterns** (Optional[List[str]]) –

Return type
Optional[*SourceHandler*]

craft_parts.sources.sources.**get_source_type_from_uri**(*source*, *ignore_errors=False*)

Return the source type based on the given source URI.

Parameters

- **source** (str) – The source specification.
- **ignore_errors** (bool) – Don't raise InvalidSourceType if the source type could not be determined.

Raises
InvalidSourceType – If the source type is unknown.

Return type
str

craft_parts.sources.tar_source module

Implement the tar source handler.

class craft_parts.sources.tar_source.**TarSource**(*source*, *part_src_dir*, ***, *cache_dir*, *source_tag=None*, *source_commit=None*, *source_branch=None*, *source_depth=None*, *source_checksum=None*, *source_submodules=None*, *project_dirs=None*, *ignore_patterns=None*)

Bases: *FileSourceHandler*

The tar source handler.

Parameters

- **source** (str) –
- **part_src_dir** (Path) –
- **cache_dir** (Path) –
- **source_tag** (Optional[str]) –
- **source_commit** (Optional[str]) –
- **source_branch** (Optional[str]) –
- **source_depth** (Optional[int]) –
- **source_checksum** (Optional[str]) –
- **source_submodules** (Optional[List[str]]) –
- **project_dirs** (Optional[*ProjectDirs*]) –
- **ignore_patterns** (Optional[List[str]]) –

provision(*dst*, *keep=False*, *src=None*)

Extract tarball contents to the part source dir.

Parameters

- **dst** (Path) –

- **keep** (bool) –
- **src** (Optional[Path]) –

Return type
None

craft_parts.sources.zip_source module

Implement the zip file source handler.

class craft_parts.sources.zip_source.**ZipSource**(*source*, *part_src_dir*, *, *cache_dir*, *source_tag=None*, *source_branch=None*, *source_commit=None*, *source_depth=None*, *source_checksum=None*, *source_submodules=None*, *project_dirs=None*, *ignore_patterns=None*)

Bases: *FileSourceHandler*

The zip file source handler.

Parameters

- **source** (str) –
- **part_src_dir** (Path) –
- **cache_dir** (Path) –
- **source_tag** (Optional[str]) –
- **source_branch** (Optional[str]) –
- **source_commit** (Optional[str]) –
- **source_depth** (Optional[int]) –
- **source_checksum** (Optional[str]) –
- **source_submodules** (Optional[List[str]]) –
- **project_dirs** (Optional[*ProjectDirs*]) –
- **ignore_patterns** (Optional[List[str]]) –

provision(*dst*, *keep=False*, *src=None*)

Extract zip file contents to the part source dir.

Parameters

- **dst** (Path) –
- **keep** (bool) –
- **src** (Optional[Path]) –

Return type
None

Module contents

Source handler definitions and helpers.

craft_parts.state_manager package

Submodules

craft_parts.state_manager.build_state module

State definitions for the build step.

class craft_parts.state_manager.build_state.**BuildState**(***data*)

Bases: *StepState*

Context information for the build step.

Parameters
data (Any) –

assets: Dict[str, Any]

overlay_hash: Optional[str]

project_options_of_interest(*project_options*)

Return relevant project options concerning this step.

Parameters
project_options (Dict[str, Any]) – A dictionary containing all project options.

Return type
Dict[str, Any]

Returns
A dictionary containing project options of interest.

properties_of_interest(*part_properties*)

Return relevant properties concerning this step.

Parameters
part_properties (Dict[str, Any]) – A dictionary containing all part properties.

Return type
Dict[str, Any]

Returns
A dictionary containing properties of interest.

classmethod unmarshal(*data*)

Create and populate a new BuildState object from dictionary data.

The unmarshal method validates entries in the input dictionary, populating the corresponding fields in the state object.

Parameters
data (Dict[str, Any]) – The dictionary data to unmarshal.

Return type
BuildState

Returns
The newly created object.

Raises
TypeError – If data is not a dictionary.

craft_parts.state_manager.overlay_state module

State definitions for the overlay step.

class craft_parts.state_manager.overlay_state.**OverlayState**(**data*)

Bases: *StepState*

Context information for the overlay step.

Parameters
data (Any) –

part_properties: Dict[str, Any]

project_options: Dict[str, Any]

project_options_of_interest(*project_options*)

Return relevant project options concerning this step.

Parameters
project_options (Dict[str, Any]) – A dictionary containing all project options.

Return type
Dict[str, Any]

Returns
A dictionary containing project options of interest.

properties_of_interest(*part_properties*)

Return relevant properties concerning this step.

Parameters
part_properties (Dict[str, Any]) – A dictionary containing all part properties.

Return type
Dict[str, Any]

Returns
A dictionary containing properties of interest.

classmethod unmarshal(*data*)

Create and populate a new OverlayState object from dictionary data.

The unmarshal method validates entries in the input dictionary, populating the corresponding fields in the state object.

Parameters
data (Dict[str, Any]) – The dictionary data to unmarshal.

Return type
OverlayState

Returns
The newly created object.

Raises
TypeError – If data is not a dictionary.

craft_parts.state_manager.prime_state module

State definitions for the prime state.

class craft_parts.state_manager.prime_state.**PrimeState**(**data*)

Bases: *StepState*

Context information for the prime step.

Parameters
data (Any) –

dependency_paths: Set[str]

primed_stage_packages: Set[str]

project_options_of_interest(*project_options*)

Return relevant project options concerning this step.

Parameters
project_options (Dict[str, Any]) – A dictionary containing all project options.

Return type
Dict[str, Any]

Returns
A dictionary containing project options of interest.

properties_of_interest(*part_properties*)

Return relevant properties concerning this step.

Parameters
part_properties (Dict[str, Any]) – A dictionary containing all part properties.

Return type
Dict[str, Any]

Returns
A dictionary containing properties of interest.

classmethod unmarshal(*data*)

Create and populate a new PrimeState object from dictionary data.

The unmarshal method validates entries in the input dictionary, populating the corresponding fields in the state object.

Parameters
data (Dict[str, Any]) – The dictionary data to unmarshal.

Return type
PrimeState

Returns
The newly created object.

Raises
TypeError – If data is not a dictionary.

craft_parts.state_manager.pull_state module

State definitions for the pull step.

class craft_parts.state_manager.pull_state.**PullState**(****data**)

Bases: *StepState*

Context information for the pull step.

Parameters
data (Any) –

assets: Dict[str, Any]

outdated_dirs: Optional[List[str]]

outdated_files: Optional[List[str]]

project_options_of_interest(*project_options*)

Return relevant project options concerning this step.

Parameters
project_options (Dict[str, Any]) – A dictionary containing all project options.

Return type
Dict[str, Any]

Returns
A dictionary containing project options of interest.

properties_of_interest(*part_properties*)

Return relevant properties concerning this step.

Parameters
part_properties (Dict[str, Any]) – A dictionary containing all part properties.

Return type
Dict[str, Any]

Returns
A dictionary containing properties of interest.

classmethod unmarshal(*data*)

Create and populate a new PullState object from dictionary data.

The unmarshal method validates entries in the input dictionary, populating the corresponding fields in the state object.

Parameters
data (Dict[str, Any]) – The dictionary data to unmarshal.

Return type
PullState

Returns
The newly created object.

Raises
TypeError – If data is not a dictionary.

craft_parts.state_manager.reports module

Provide a report on why a step is outdated.

class craft_parts.state_manager.reports.**Dependency**(*part_name*, *step*)

Bases: object

The part and step that are a prerequisite to another step.

Parameters

- **part_name** (str) –
- **step** (*Step*) –

part_name: str

step: *Step*

class craft_parts.state_manager.reports.**DirtyReport**(*, *dirty_properties=None*, *dirty_project_options=None*, *changed_dependencies=None*)

Bases: object

The DirtyReport class explains why a given step is dirty.

A dirty step is defined to be a step that has run, but since doing so one of the following things have happened:

- One or more properties used by the step have changed.
- One of more project options have changed.
- One of more of its dependencies have been re-staged.

Parameters

- **dirty_properties** (Optional[List[str]]) –
- **dirty_project_options** (Optional[List[str]]) –
- **changed_dependencies** (Optional[List[*Dependency*]]) –

reason()

Get summarized report.

Return type
str

Returns
Short summary of why the part is dirty.

class craft_parts.state_manager.reports.**OutdatedReport**(*, *previous_step_modified=None*, *source_modified=False*, *outdated_files=None*, *outdated_dirs=None*)

Bases: object

The OutdatedReport class explains why a given step is outdated.

An outdated step is defined to be a step that has run, but since doing so one of the following things have happened:

- A step earlier in the lifecycle has run again.
- The source on disk has been updated.

Parameters

- **previous_step_modified** (Optional[*Step*]) –
- **source_modified** (bool) –
- **outdated_files** (Optional[List[str]]) –
- **outdated_dirs** (Optional[List[str]]) –

reason()

Get summarized report.

Return type
str

Returns
Short summary of why the step is outdated.

craft_parts.state_manager.stage_state module

State definitions for the stage step.

class craft_parts.state_manager.stage_state.**StageState**(***data*)

Bases: *StepState*

Context information for the stage step.

Parameters
data (Any) –

overlay_hash: Optional[str]

project_options_of_interest(*project_options*)

Return relevant project options concerning this step.

Parameters
project_options (Dict[str, Any]) – A dictionary containing all project options.

Return type
Dict[str, Any]

Returns
A dictionary containing project options of interest.

properties_of_interest(*part_properties*)

Return relevant properties concerning this step.

Parameters
part_properties (Dict[str, Any]) – A dictionary containing all part properties.

Return type
Dict[str, Any]

Returns
A dictionary containing properties of interest.

classmethod unmarshal(*data*)

Create and populate a new StageState object from dictionary data.

The unmarshal method validates entries in the input dictionary, populating the corresponding fields in the state object.

Parameters
data (Dict[str, Any]) – The dictionary data to unmarshal.

Return type
StageState

Returns
The newly created object.

Raises
TypeError – If data is not a dictionary.

craft_parts.state_manager.state_manager module

Part crafter step state management.

class craft_parts.state_manager.state_manager.**StateManager**(*, *project_info*, *part_list*, *ignore_outdated=None*)

Bases: object

Keep track of lifecycle execution state.

The State Manager tells whether a step should run based on current state information. The state database is initialized from state on disk, and after that it's maintained only in memory.

Parameters

- **project_info** (*ProjectInfo*) – The project information.
- **part_list** (List[*Part*]) – A list of this project's parts.
- **ignore_outdated** (Optional[List[str]]) – A list of file patterns to ignore when testing for outdated files.

check_if_dirty(*part*, *step*)

Verify whether a step is dirty.

A step is considered to be dirty if relevant properties or project options have changed since the step was executed. This means the step needs to be cleaned and run again. This is in contrast to an "outdated" step, which typically doesn't need to be cleaned, just updated with files from an earlier step in the lifecycle.

Parameters

- **part** (*Part*) – The part the step to be checked belongs to.
- **step** (*Step*) – The step to be checked.

Return type
Optional[*DirtyReport*]

Returns
A class:*DirtyReport* if the step is outdated, None otherwise.

check_if_outdated(*part*, *step*)

Verify whether a step is outdated.

A step is considered to be outdated if an earlier step in the lifecycle has been run more recently, or if the source code changed on disk. This means the step needs to be updated by taking modified files from the previous step. This is in contrast to a "dirty" step, which must be cleaned and run again.

Parameters

- **part** (*Part*) – The part the step to be checked belongs to.

- **step** (*Step*) – The step to be checked.

Return type
Optional[*OutdatedReport*]

Returns
An class:*OutdatedReport* if the step is outdated, None otherwise.

clean_part(*part*, *step*)

Remove the state for this and later steps in the given part.

Parameters

- **part** (*Part*) – The part corresponding to the state to remove.
- **step** (*Step*) – The step corresponding to the state to remove.

Return type
None

get_outdated_dirs(*part*)

Get the list of outdated directories for this part.

Parameters
part (*Part*) – The part being processed.

Return type
Optional[List[str]]

Returns
The list of outdated directories from the part's pull step.

get_outdated_files(*part*)

Get the list of outdated files for this part.

Parameters
part (*Part*) – The part being processed.

Return type
Optional[List[str]]

Returns
The list of outdated files from the part's pull step.

get_step_state_overlay_hash(*part*, *step*)

Get the overlay hash from the step state.

Parameters

- **part** (*Part*) – The part being processed.
- **part** – The step being processed.
- **step** (*Step*) –

Return type
bytes

Returns
The hash value for the layer corresponding to the part being processed.

has_step_run(*part*, *step*)

Determine if a given step of a given part has already run.

Parameters

- **part** (*Part*) – The part the step to be verified belongs to.
- **step** (*Step*) – The step to verify.

Return type
bool

Returns
Whether the step has already run.

mark_step_updated(*part*, *step*)

Mark the given part and step as updated.

Parameters

- **part** (*Part*) – The part being processed.
- **part** – The step being processed.
- **step** (*Step*) –

Return type
None

project_vars(*part*, *step*)

Obtain the project variables for a given part and step.

Parameters

- **part** (*Part*) – The part corresponding to the state to retrieve.
- **part** – The step corresponding to the state to retrieve.
- **step** (*Step*) –

Return type
Optional[Dict[str, *ProjectVar*]]

Returns
The project variables from the last execution.

set_state(*part*, *step*, *, *state*)

Set the state of the given part and step.

Parameters

- **part** (*Part*) – The part corresponding to the state to be set.
- **step** (*Step*) – The step corresponding to the state to be set.
- **state** (*StepState*) –

Return type
None

should_step_run(*part*, *step*)

Determine if a given step of a given part should run.

A given step should run if:

1. it hasn't already run
2. it's dirty
3. it's outdated
4. either (1), (2), or (3) apply to any earlier steps in the part's lifecycle.

Parameters

- **part** (*Part*) – The part the step to be verified belongs to.
- **step** (*Step*) – The step to verify.

Return type
bool

Returns
Whether the step should run.

update_state_timestamp(*part*, *step*)

Mark the step as recently modified.

Parameters

- **part** (*Part*) – The part corresponding to the state to update.
- **step** (*Step*) – The step corresponding to the state to update.

Return type
None

craft_parts.state_manager.states module

Helpers and definitions for lifecycle states.

craft_parts.state_manager.states.**get_overlay_migration_state_path**(*state_dir*, *step*)

Return the path to the overlay migration state file for the given step.

Parameters

- **state_dir** (Path) –
- **step** (*Step*) –

Return type
Path

craft_parts.state_manager.states.**get_step_state_path**(*part*, *step*)

Return the path to the state file for the given part and step.

Parameters

- **part** (*Part*) –
- **step** (*Step*) –

Return type
Path

craft_parts.state_manager.states.**load_overlay_migration_state**(*state_dir*, *step*)

Retrieve the overlay migration state for the given step.

Parameters

- **state_dir** (Path) – The path to the directory containing migration state files.
- **step** (*Step*) – The step corresponding to the migration state to load.

Return type
Optional[*MigrationState*]

craft_parts.state_manager.states.**load_step_state**(*part*, *step*)

Retrieve the persistent state for the given part and step.

Parameters

- **part** (*Part*) – The part corresponding to the state to load.
- **step** (*Step*) – The step corresponding to the state to load.

Return type
Optional[*StepState*]

Returns
The step state.

Raises
RuntimeError – If step is invalid.

craft_parts.state_manager.states.**remove**(*part*, *step*)

Remove the persistent state file for the given part and step.

Parameters

- **part** (*Part*) – The part whose state is to be removed.
- **step** (*Step*) – The step whose state is to be removed.

Return type
None

craft_parts.state_manager.step_state module

The step state preserves step execution context information.

class craft_parts.state_manager.step_state.**MigrationState**(***data*)

Bases: YamlModel

State information collected when migrating steps.

The migration state contains the paths to the files and directories that have been migrated. This information is used to remove migrated files from shared areas on step cleanup.

Parameters
data (Any) –

directories: Set[str]

files: Set[str]

marshal()

Create a dictionary containing the part state data.

Return type
Dict[str, Any]

Returns
The newly created dictionary.

classmethod unmarshal(*data*)

Create and populate a new state object from dictionary data.

Parameters
data (Dict[str, Any]) – A dictionary containing the data to unmarshal.

Return type
MigrationState

Returns
The state object containing the migration data.

write(*filepath*)

Write state data to disk.

Parameters
filepath (Path) – The path to the file to write.

Return type
None

class craft_parts.state_manager.step_state.**StepState**(***data*)

Bases: *MigrationState*, ABC

Contextual information collected when a step is executed.

The step state contains environmental and project-specific configuration data collected at step run time. Those properties are used to decide whether the step should run again on a new lifecycle execution.

Parameters
data (Any) –

class Config

Bases: object

Pydantic model configuration.

alias_generator()

allow_mutation = False

allow_population_by_field_name = True

extra = 'ignore'

validate_assignment = True

diff_project_options_of_interest(*other_project_options*)

Return project options that differ.

Take a dictionary of project_options and compare to our own, returning the set of project option names that are different. Both dictionaries are filtered prior to comparison, only relevant options are compared.

Parameters
other_project_options (Dict[str, Any]) – The project options to compare to the project options stored in this state.

Return type
Set[str]

diff_properties_of_interest(*other_properties*)

Return properties of interest that differ.

Take a dictionary of properties and compare to our own, returning the set of property names that are different. Both dictionaries are filtered prior to comparison, only relevant properties are compared.

Parameters
other_properties (Dict[str, Any]) – The properties to compare to the project options stored in this state.

Return type
Set[str]

part_properties: Dict[str, Any]

project_options: Dict[str, Any]

abstract project_options_of_interest(*project_options*)

Return relevant project options concerning this step.

Parameters
project_options (Dict[str, Any]) –

Return type
Dict[str, Any]

abstract properties_of_interest(*part_properties*)

Return relevant properties concerning this step.

Parameters
part_properties (Dict[str, Any]) –

Return type
Dict[str, Any]

classmethod unmarshal(*data*)

Create and populate a new state object from dictionary data.

Parameters
data (Dict[str, Any]) –

Return type
Any

craft_parts.state_manager.step_state.**validate_hex_string**(*value*)

Ensure that a pydantic model field is hexadecimal string.

Parameters
value (str) –

Return type
str

Module contents

Part state management.

craft_parts.utils package

Submodules

craft_parts.utils.deb_utils module

deb-related utilities used by both *packages* and *sources*.

craft_parts.utils.deb_utils.**extract_deb**(*deb_path*, *extract_dir*, *log_func*)

Extract file *deb_path* into `extract_dir.

Parameters

- **deb_path** (Path) –
- **extract_dir** (Path) –
- **log_func** (Callable[[str], None]) –

Return type

None

craft_parts.utils.file_utils module

File-related utilities.

class craft_parts.utils.file_utils.**NonBlockingRWFifo**(*path*)

Bases: object

A non-blocking FIFO for reading and writing.

Parameters

path (str) –

close()

Close the FIFO.

Return type

None

property path: str

Return the path to the FIFO file.

Return type

str

read()

Read from the FIFO.

Return type

str

write(*data*)

Write to the FIFO.

Parameters

data (str) – The data to write.

Return type

int

craft_parts.utils.file_utils.**calculate_hash**(*filename*, *, *algorithm*)

Calculate the hash of the given file.

Parameters

- **filename** (Path) – The path to the file to digest.
- **algorithm** (str) – The algorithm to use, as defined by hashlib.

Return type
str

Returns
The file hash.

Raises
ValueError – If the algorithm is unsupported.

craft_parts.utils.file_utils.**copy**(*source*, *destination*, *, *follow_symlinks=False*, *permissions=None*)

Copy source and destination files.

This function overwrites the destination if it already exists, and also tries to copy ownership information.

Parameters

- **source** (str) – The source to be copied to destination.
- **destination** (str) – Where to put the copy.
- **follow_symlinks** (bool) – Whether or not symlinks should be followed.
- **permissions** (Optional[List[*Permissions*]]) – The permissions definitions that should be applied to the new file.

Raises
CopyFileNotFound – If source doesn't exist.

Return type
None

craft_parts.utils.file_utils.**create_similar_directory**(*source*, *destination*, *permissions=None*)

Create a directory with the same permission bits and owner information.

Parameters

- **source** (str) – Directory from which to copy name, permission bits, and owner information.
- **destination** (str) – Directory to create and to which the source information will be copied.
- **permissions** (Optional[List[*Permissions*]]) – The permission definitions to apply to the new directory. If omitted, the new directory will have the same permissions and ownership of source.

Return type
None

craft_parts.utils.file_utils.**link**(*source*, *destination*, *, *follow_symlinks=False*)

Hard-link source and destination files.

Parameters

- **source** (str) – The source to which destination will be linked.
- **destination** (str) – The destination to be linked to source.

- **follow_symlinks** (bool) – Whether or not symlinks should be followed.

Raises
CopyFileNotFound – If source doesn't exist.

Return type
None

craft_parts.utils.file_utils.**link_or_copy**(*source*, *destination*, ***, *follow_symlinks=False*, *permissions=None*)

Hard-link source and destination files. Copy if it fails to link.

Hard-linking may fail (e.g. a cross-device link, or permission denied), so as a backup plan we just copy it. Note that we always copy the file if its `permissions` will change.

Parameters

- **source** (str) – The source to which destination will be linked.
- **destination** (str) – The destination to be linked to source.
- **follow_symlinks** (bool) – Whether or not symlinks should be followed.
- **permissions** (Optional[List[*Permissions*]]) – The permissions definitions that should be applied to the new file.

Return type
None

craft_parts.utils.file_utils.**link_or_copy_tree**(*source_tree*, *destination_tree*, *ignore=None*, *copy_function=<function link_or_copy>*)

Copy a source tree into a destination, hard-linking if possible.

Parameters

- **source_tree** (str) – Source directory to be copied.
- **destination_tree** (str) – Destination directory. If this directory already exists, the files in *source_tree* will take precedence.
- **ignore** (Optional[Callable[[str, List[str]], List[str]]]) – If given, called with two params, source dir and dir contents, for every dir copied. Should return list of contents to NOT copy.
- **copy_function** (Callable[..., None]) – Callable that actually copies.

Return type
None

craft_parts.utils.formatting_utils module

Text formatting utilities.

craft_parts.utils.formatting_utils.**humanize_list**(*items*, *conjunction*, *item_format='{!r}'*)

Format a list into a human-readable string.

Parameters

- **items** (Iterable[str]) – List to humanize.
- **conjunction** (str) – The conjunction used to join the final element to the rest of the list (e.g. 'and').

- **item_format** (str) – Format string to use per item.

Return type
str

craft_parts.utils.os_utils module

Utilities related to the operating system.

class craft_parts.utils.os_utils.**OsRelease**(*, *os_release_file='/etc/os-release'*)

Bases: object

A class to intelligently determine the OS on which we're running.

Parameters
os_release_file (str) –

id()

Return the OS ID.

Raises
OsReleaseIdError – If no ID can be determined.

Return type
str

name()

Return the OS name.

Raises
OsReleaseNameError – If no name can be determined.

Return type
str

version_codename()

Return the OS version codename.

This first tries to use the VERSION_CODENAME. If that's missing, it tries to use the VERSION_ID to figure out the codename on its own.

Raises
OsReleaseCodenameError – If no version codename can be determined.

Return type
str

version_id()

Return the OS version ID.

Raises
OsReleaseVersionIdError – If no version ID can be determined.

Return type
str

class craft_parts.utils.os_utils.**TimedWriter**

Bases: object

Enforce minimum times between writes.

Ensure subsequent writes happen at least at the specified minimum interval apart from each other, otherwise hosts with low tick resolution may generate files with identical timestamps.

classmethod write_text(*filepath*, *text*, *encoding=None*, *errors=None*)

Write text to the specified file.

Parameters

- **filepath** (Path) – The path to the file to write to.
- **text** (str) – The text to write.
- **encoding** (Optional[str]) – The name of the encoding used to encode and decode the file. See the corresponding parameter in os.open for details.
- **errors** (Optional[str]) – How encoding/decoding errors are handled, in the same format used in os.open.

Return type
None

craft_parts.utils.os_utils.**get_bin_paths**(*, *root*, *existing_only=True*)

List common system executable paths.

Parameters

- **root** (Path) – A path to prepend to each entry in the list.
- **existing_only** (bool) – Only list paths that are present in the system.

Return type
List[str]

Returns
The list of executable paths.

craft_parts.utils.os_utils.**get_include_paths**(*, *root*, *arch_triplet*)

List common include paths.

Parameters

- **root** (Path) – A path to prepend to each entry in the list.
- **arch_triplet** (str) –

Arch_triplet
The machine-vendor-os platform triplet definition.

Return type
List[str]

Returns
The list of include paths.

craft_parts.utils.os_utils.**get_library_paths**(*, *root*, *arch_triplet*, *existing_only=True*)

List common library paths.

Parameters

- **root** (Path) – A path to prepend to each entry in the list.
- **existing_only** (bool) – Only list paths that are present in the system.
- **arch_triplet** (str) –

Arch_triplet
The machine-vendor-os platform triplet definition.

Return type
`List[str]`

Returns
The list of library paths.

craft_parts.utils.os_utils.**get_pkg_config_paths**(*, *root*, *arch_triplet*)

List common pkg-config paths.

Parameters

- **root** (`Path`) – A path to prepend to each entry in the list.
- **arch_triplet** (`str`) –

Arch_triplet
The machine-vendor-os platform triplet definition.

Return type
`List[str]`

Returns
The list of pkg-config paths.

craft_parts.utils.os_utils.**get_system_info**()

Obtain running system information.

Return type
`str`

craft_parts.utils.os_utils.**is_dumb_terminal**()

Verify whether the caller is running on a dumb terminal.

Return type
`bool`

Returns
True if on a dumb terminal.

craft_parts.utils.os_utils.**is_inside_container**()

Determine if the application is in a container.

Return type
`bool`

Returns
Whether the process is running inside a container.

craft_parts.utils.os_utils.**is_snap**(*application_name*)

Verify whether we're running as a snap.

Application_name
The snap application name. If provided, check if it matches the snap name.

Parameters
application_name (`str`) –

Return type
`bool`

craft_parts.utils.os_utils.**mount**(*device*, *mountpoint*, **args*)

Mount a filesystem.

Parameters

- **device** (str) – The device to mount.
- **mountpoint** (str) – Where the device will be mounted.
- ***args** – Additional arguments to `mount(8)`.
- **args** (str) –

Raises

subprocess.CalledProcessError – on error.

Return type

None

craft_parts.utils.os_utils.**mount_overlayfs**(*mountpoint*, **args*)

Mount an overlay filesystem using fuse-overlayfs.

Parameters

- **mountpoint** (str) – Where the device will be mounted.
- ***args** – Additional arguments to `mount(8)`.
- **args** (str) –

Raises

subprocess.CalledProcessError – on error.

Return type

None

craft_parts.utils.os_utils.**process_run**(*command*, *log_func*, ***kwargs*)

Run a command and handle its output.

Parameters

- **command** (List[str]) –
- **log_func** (Callable[[str], None]) –
- **kwargs** (Any) –

Return type

None

craft_parts.utils.os_utils.**umount**(*mountpoint*, **args*)

Unmount a filesystem.

Parameters

- **mountpoint** (str) – The mount point or device to unmount.
- **args** (str) –

Raises

subprocess.CalledProcessError – on error.

Return type

None

craft_parts.utils.url_utils module

URL parsing and downloading helpers.

craft_parts.utils.url_utils.**download_request**(*request*, *destination*, *message=None*, *total_read=0*)

Download a request with nice progress bars.

Parameters

- **request** (Response) – The URL download request.
- **destination** (str) – The destination file name.
- **message** (Optional[str]) – The message shown in the progress bar.
- **total_read** (int) –

Return type
None

craft_parts.utils.url_utils.**get_url_scheme**(*url*)

Return the given URL's scheme.

Parameters
url (str) –

Return type
str

craft_parts.utils.url_utils.**is_url**(*url*)

Verify whether the given string is a valid URL.

Parameters
url (str) –

Return type
bool

Module contents

Utilities and helpers.

craft_parts.utils.**package_name**()

Return the topmost package name.

Return type
str

Submodules

craft_parts.actions module

Definitions of lifecycle actions and action types.

class craft_parts.actions.**Action**(*part_name*, *step*, *action_type=ActionType.RUN*, *reason=None*, *project_vars=None*, *properties=ActionProperties(changed_files=None, changed_dirs=None)*)

Bases: object

The action to be executed for a given part.

Actions correspond to the operations required to run the lifecycle for each of the parts in the project specification.

Parameters

- **part_name** (str) – The name of the part this action will be performed on.
- **step** (*Step*) – The Step this action will execute.
- **action_type** (*ActionType*) – Action to run for this step.
- **reason** (Optional[str]) – A textual description of why this action should be executed.
- **project_vars** (Optional[Dict[str, *ProjectVar*]]) – The values of project variables from a previous execution, used if the action type is ActionType.SKIP.
- **properties** (*ActionProperties*) –

action_type: *ActionType* = 0

part_name: str

project_vars: Optional[Dict[str, *ProjectVar*]] = None

properties: *ActionProperties* = ActionProperties(changed_files=None, changed_dirs=None)

reason: Optional[str] = None

step: *Step*

class craft_parts.actions.**ActionProperties**(*changed_files=None*, *changed_dirs=None*)

Bases: object

Properties defined for an action.

Parameters

- **changed_files** (Optional[List[str]]) –
- **changed_dirs** (Optional[List[str]]) –

changed_dirs: Optional[List[str]] = None

changed_files: Optional[List[str]] = None

class craft_parts.actions.**ActionType**(*value*)

Bases: IntEnum

The type of action to be executed.

Action execution can be modified according to its type:

RUN: execute the expected commands for step processing.

RERUN: clear the existing data and state before procceeding.

UPDATE: try to continue processing the step.

SKIP: don't execute this action.

REAPPLY: run the step commands without updating its state.

REAPPLY = 4

RERUN = 1

RUN = 0

SKIP = 2

UPDATE = 3

craft_parts.callbacks module

Register and execute callback functions.

class craft_parts.callbacks.**CallbackHook**(*function*, *step_list*)

Bases: tuple

function

Alias for field number 0

step_list

Alias for field number 1

craft_parts.callbacks.**register_epilogue**(*func*)

Register an execution epilogue callback function.

Parameters
func (Callable[[*ProjectInfo*], None]) – The callback function to run.

Return type
None

craft_parts.callbacks.**register_post_step**(*func*, ***, *step_list=None*)

Register a post-step callback function.

Parameters

- **func** (Callable[[*StepInfo*], bool]) – The callback function to run.
- **step_list** (Optional[List[*Step*]]) – The steps after which the callback function should run. If not specified, the callback function will be executed after all steps.

Return type
None

craft_parts.callbacks.**register_pre_step**(*func*, ***, *step_list=None*)

Register a pre-step callback function.

Parameters

- **func** (Callable[[*StepInfo*], bool]) – The callback function to run.
- **step_list** (Optional[List[*Step*]]) – The steps before which the callback function should run. If not specified, the callback function will be executed before all steps.

Return type
None

craft_parts.callbacks.**register_prologue**(*func*)

Register an execution prologue callback function.

Parameters
func (Callable[[*ProjectInfo*], None]) – The callback function to run.

Return type
None

craft_parts.callbacks.**run_epilogue**(*project_info*)

Run all registered execution epilogue callbacks.

Parameters
project_info (*ProjectInfo*) – The project information to be sent to callback functions.

Return type
None

craft_parts.callbacks.**run_post_step**(*step_info*)

Run all registered post-step callback functions.

Parameters
step_info (*StepInfo*) – the step information to be sent to the callback functions.

Return type
None

craft_parts.callbacks.**run_pre_step**(*step_info*)

Run all registered pre-step callback functions.

Parameters
step_info (*StepInfo*) – the step information to be sent to the callback functions.

Return type
None

craft_parts.callbacks.**run_prologue**(*project_info*)

Run all registered execution prologue callbacks.

Parameters
project_info (*ProjectInfo*) – The project information to be sent to callback functions.

Return type
None

craft_parts.callbacks.**unregister_all**()

Clear all existing registered callback functions.

Return type
None

craft_parts.ctl module

Helpers to invoke step execution handlers from the command line.

class craft_parts.ctl.**CraftCtl**

Bases: object

Client for the craft-parts ctl protocol.

Craftctl is used to execute built-in step handlers and to get and set variables in the running parts processor context.

classmethod run(*cmd*, *args*)

Handle craftctl commands.

Parameters

- **cmd** (str) – The command to handle.
- **args** (List[str]) – Command arguments.

Raises

RuntimeError – If the command is not handled.

Return type

Optional[str]

craft_parts.ctl.**main**()

Run the ctl client cli.

Return type

None

craft_parts.dirs module

Definitions for project directories.

class craft_parts.dirs.**ProjectDirs**(*, *work_dir='.'*)

Bases: object

The project's main work directories.

Parameters

work_dir (Union[Path, str]) – The parent directory containing the parts, prime and stage subdirectories. If not specified, the current directory will be used.

Variables

- **work_dir** – The root of the work directories used for project processing.
- **parts_dir** – The directory containing work subdirectories for each part.
- **overlay_dir** – The directory containing work subdirectories for overlays.
- **overlay_mount_dir** – The mountpoint for the overlay filesystem.
- **overlay_packages_dir** – The cache directory for overlay packages.
- **overlay_work_dir** – The work directory for the overlay filesystem.
- **stage_dir** – The staging area containing installed files from all parts.
- **prime_dir** – The primed tree containing the final artifacts to deploy.

craft_parts.errors module

Craft parts errors.

exception craft_parts.errors.**CallbackRegistrationError**(*message*)

Bases: *PartsError*

Error in callback function registration.

Parameters
message (str) – the error message.

brief: str

exception craft_parts.errors.**CopyFileNotFound**(*name*)

Bases: *PartsError*

An attempt was made to copy a file that doesn't exist.

Parameters
name (str) – The file name.

brief: str

exception craft_parts.errors.**CopyTreeError**(*message*)

Bases: *PartsError*

Failed to copy or link a file tree.

Parameters
message (str) – The error message.

brief: str

exception craft_parts.errors.**DebError**(*deb_path*, *command*, *exit_code*)

Bases: *PartsError*

A "deb"-related command failed.

Parameters

- **deb_path** (Path) –
- **command** (List[str]) –
- **exit_code** (int) –

brief: str

exception craft_parts.errors.**FileOrganizeError**(*, *part_name*, *message*)

Bases: *PartsError*

Failed to organize a file layout.

Parameters

- **part_name** (str) – The name of the part being processed.
- **message** (str) – The error message.

brief: str

exception craft_parts.errors.**FilesetConflict**(*conflicting_files*)

Bases: *PartsError*

Inconsistent stage to prime filtering.

Parameters

conflicting_files (Set[str]) – A set containing the conflicting file names.

brief: str

exception craft_parts.errors.**FilesetError**(*, *name*, *message*)

Bases: *PartsError*

An invalid fileset operation was performed.

Parameters

- **name** (str) – The name of the fileset.
- **message** (str) – The error message.

brief: str

exception craft_parts.errors.**InvalidAction**(*message*)

Bases: *PartsError*

An attempt was made to execute an action with invalid parameters.

Parameters

message (str) – The error message.

brief: str

exception craft_parts.errors.**InvalidApplicationName**(*name*)

Bases: *PartsError*

The application name contains invalid characters.

Parameters

name (str) – The invalid application name.

brief: str

exception craft_parts.errors.**InvalidArchitecture**(*arch_name*)

Bases: *PartsError*

The machine architecture is not supported.

Parameters

arch_name (str) – The unsupported architecture name.

brief: str

exception craft_parts.errors.**InvalidControlAPICall**(*, *part_name*, *scriptlet_name*, *message*)

Bases: *PartsError*

A control API call was made with invalid parameters.

Parameters

- **part_name** (str) – The name of the part being processed.
- **scriptlet_name** (str) – The name of the scriptlet that originated the call.
- **message** (str) – The error message.

brief: str

exception craft_parts.errors.**InvalidPartName**(*part_name*)

Bases: *PartsError*

An operation was requested on a part that's in the parts specification.

Parameters

part_name (str) – The invalid part name.

brief: str

exception craft_parts.errors.**InvalidPlugin**(*plugin_name*, *, *part_name*)

Bases: *PartsError*

A request was made to use a plugin that's not registered.

Parameters

- **plugin_name** (str) – The invalid plugin name."
- **part_name** (str) – The name of the part defining the invalid plugin.

brief: str

exception craft_parts.errors.**OsReleaseCodenameError**

Bases: *PartsError*

Failed to determine the host operating system version codename.

brief: str

exception craft_parts.errors.**OsReleaseIdError**

Bases: *PartsError*

Failed to determine the host operating system identification string.

brief: str

exception craft_parts.errors.**OsReleaseNameError**

Bases: *PartsError*

Failed to determine the host operating system name.

brief: str

exception craft_parts.errors.**OsReleaseVersionIdError**

Bases: *PartsError*

Failed to determine the host operating system version.

brief: str

exception craft_parts.errors.**OverlayPackageNotFound**(*, *part_name*, *package_name*)

Bases: *PartsError*

Failed to install an overlay package.

Parameters

- **part_name** (str) – The name of the part being processed.
- **message** – the error message.
- **package_name** (str) –

brief: str

exception craft_parts.errors.**OverlayPermissionError**

Bases: *PartsError*

A project using overlays was processed by a non-privileged user.

brief: str

exception craft_parts.errors.**OverlayPlatformError**

Bases: *PartsError*

A project using overlays was processed on a non-Linux platform.

brief: str

exception craft_parts.errors.**PartDependencyCycle**

Bases: *PartsError*

A dependency cycle has been detected in the parts definition.

brief: str

exception craft_parts.errors.**PartFilesConflict**(*, *part_name*, *other_part_name*, *conflicting_files*)

Bases: *PartsError*

Different parts list the same files with different contents.

Parameters

- **part_name** (str) – The name of the part being processed.
- **other_part_name** (str) – The name of the conflicting part.
- **conflicting_files** (List[str]) – The list of conflicting files.

brief: str

exception craft_parts.errors.**PartSpecificationError**(*, *part_name*, *message*)

Bases: *PartsError*

A part was not correctly specified.

Parameters

- **part_name** (str) – The name of the part being processed.
- **message** (str) – The error message.

brief: str

classmethod from_validation_error(*, *part_name*, *error_list*)

Create a PartSpecificationError from a pydantic error list.

Parameters

- **part_name** (str) – The name of the part being processed.
- **error_list** (List[ErrorDict]) – A list of dictionaries containing pydantic error definitions.

Return type

PartSpecificationError

exception craft_parts.errors.**PartsError**(*brief*, *details=None*, *resolution=None*)

Bases: Exception

Unexpected error.

Parameters

- **brief** (str) – Brief description of error.
- **details** (Optional[str]) – Detailed information.
- **resolution** (Optional[str]) – Recommendation, if any.

brief: str

details: Optional[str] = None

resolution: Optional[str] = None

exception craft_parts.errors.**PluginBuildError**(*, *part_name*)

Bases: *PartsError*

Plugin build script failed at runtime.

Parameters

part_name (str) – The name of the part being processed.

brief: str

exception craft_parts.errors.**PluginEnvironmentValidationError**(*, *part_name*, *reason*)

Bases: *PartsError*

Plugin environment validation failed at runtime.

Parameters

- **part_name** (str) – The name of the part being processed.
- **reason** (str) –

brief: str

exception craft_parts.errors.**ScriptletRunError**(*, *part_name*, *scriptlet_name*, *exit_code*)

Bases: *PartsError*

A scriptlet execution failed.

Parameters

- **part_name** (str) – The name of the part being processed.
- **scriptlet_name** (str) – The name of the scriptlet that failed to execute.
- **exit_code** (int) – The execution error code.

brief: str

exception craft_parts.errors.**StageFilesConflict**(*, *part_name*, *conflicting_files*)

Bases: *PartsError*

Files from a part conflict with files already being staged.

Parameters

- **part_name** (str) – The name of the part being processed.

- **conflicting_files** (List[str]) – The list of confictling files.

brief: str

exception craft_parts.errors.**StagePackageNotFound**(*, *part_name*, *package_name*)

Bases: *PartsError*

Failed to install a stage package.

Parameters

- **part_name** (str) – The name of the part being processed.
- **package_name** (str) – The name of the package.

brief: str

exception craft_parts.errors.**UndefinedPlugin**(*, *part_name*)

Bases: *PartsError*

The part didn't define a plugin and the part name is not a valid plugin name.

Parameters

part_name (str) – The name of the part with no plugin definition.

brief: str

exception craft_parts.errors.**XAttributeError**(*key*, *path*, *is_write=False*)

Bases: *PartsError*

Failed to read or write an extended attribute.

Parameters

- **action** – The action being performed.
- **key** (str) – The extended attribute key.
- **path** (str) – The file path.
- **is_write** (bool) – Whether this is an attribute write operation.

brief: str

exception craft_parts.errors.**XAttributeTooLong**(*key*, *value*, *path*)

Bases: *PartsError*

Failed to write an extended attribute because key and/or value is too long.

Parameters

- **key** (str) – The extended attribute key.
- **value** (str) – The extended attribute value.
- **path** (str) – The file path.

brief: str

craft_parts.infos module

Project, part and step information classes.

class craft_parts.infos.**PartInfo**(*project_info*, *part*)

Bases: object

Part-level information containing project and part fields.

Parameters

- **project_info** (*ProjectInfo*) – The project information.
- **part** (*Part*) – The part we want to obtain information from.

get_project_var(*name*, *, *raw_read=False*)

Get the value of a project variable.

Variables must be consumed by the application only after the lifecycle execution ends to prevent unexpected behavior if steps are skipped.

Parameters

- **name** (str) – The project variable name.
- **raw_read** (bool) – Whether the variable is read without access verifications.

Return type
str

Returns
The value of the variable.

Raises

- **ValueError** – If there is no project variable with the given name.
- **RuntimeError** – If the variable is consumed during the lifecycle execution.

property part_build_dir: Path

Return the subdirectory containing the part's build tree.

Return type
Path

property part_build_subdir: Path

Return the subdirectory in build containing the source subtree (if any).

Return type
Path

property part_install_dir: Path

Return the subdirectory to install the part's build artifacts.

Return type
Path

property part_name: str

Return the name of the part we're providing information about.

Return type
str

property part_src_dir: Path

Return the subdirectory containing the part's source code.

Return type
Path

property part_src_subdir: Path

Return the subdirectory in source containing the source subtree (if any).

Return type
Path

property part_state_dir: Path

Return the subdirectory containing this part's lifecycle state.

Return type
Path

property project_info: *ProjectInfo*

Return the project information.

Return type
ProjectInfo

set_project_var(*name*, *value*, *, *raw_write=False*)

Set the value of a project variable.

Variable values can be set once. Project variables are not intended for logic construction in user scripts, setting it multiple times is likely to be an error.

Parameters

- **name** (str) – The project variable name.
- **value** (str) – The new project variable value.
- **raw_write** (bool) – Whether the variable is written without access verifications.

Raises

- **ValueError** – If there is no custom argument with the given name.
- **RuntimeError** – If a write-once variable is set a second time, or if a part name is specified and the variable is set from a different part.

Return type
None

class craft_parts.infos.**ProjectInfo**(*, *application_name*, *cache_dir*, *arch=''*, *base=''*, *parallel_build_count=1*, *project_dirs=None*, *project_name=None*, *project_vars_part_name=None*, *project_vars=None*, ***custom_args*)

Bases: object

Project-level information containing project-specific fields.

Parameters

- **application_name** (str) – A unique identifier for the application using Craft Parts.
- **project_name** (Optional[str]) – name of the project being built.
- **cache_dir** (Path) – The path to store cached packages and files. If not specified, a directory under the application name entry in the XDG base directory will be used.

- **arch** (str) – The architecture to build for. Defaults to the host system architecture.
- **parallel_build_count** (int) – The maximum number of concurrent jobs to be used to build each part of this project.
- **project_dirs** (Optional[*ProjectDirs*]) – The project work directories.
- **project_name** – The name of the project.
- **project_vars_part_name** (Optional[str]) – Project variables can be set only if the part name matches this name.
- **project_vars** (Optional[Dict[str, str]]) – A dictionary containing the project variables.
- **custom_args** (Any) – Any additional arguments defined by the application when creating a LifecycleManager.
- **base** (str) –

property application_name: str

Return the name of the application using craft-parts.

Return type
str

property arch_triplet: str

Return the machine-vendor-os platform triplet definition.

Return type
str

property base: str

Return the project build base.

Return type
str

property cache_dir: Path

Return the directory used to store cached files.

Return type
Path

property custom_args: List[str]

Return the list of custom argument names.

Return type
List[str]

property dirs: *ProjectDirs*

Return the project's work directories.

Return type
ProjectDirs

get_project_var(*name*, *, *raw_read=False*)

Get the value of a project variable.

Variables must be consumed by the application only after the lifecycle execution ends to prevent unexpected behavior if steps are skipped.

Parameters

- **name** (str) – The project variable name.
- **raw_read** (bool) – Whether the variable is read without access verifications.

Return type
str

Returns
The value of the variable.

Raises

- **ValueError** – If there is no project variable with the given name.
- **RuntimeError** – If the variable is consumed during the lifecycle execution.

property host_arch: str

Return the host architecture used for debs, snaps and charms.

Return type
str

property is_cross_compiling: bool

Whether the target and host architectures are different.

Return type
bool

property parallel_build_count: int

Return the maximum allowable number of concurrent build jobs.

Return type
int

property project_name: Optional[str]

Return the name of the project using craft-parts.

Return type
Optional[str]

property project_options: Dict[str, Any]

Obtain a project-wide options dictionary.

Return type
Dict[str, Any]

set_project_var(*name*, *value*, *raw_write=False*, ***, *part_name=None*)

Set the value of a project variable.

Variable values can be set once. Project variables are not intended for logic construction in user scripts, setting it multiple times is likely to be an error.

Parameters

- **name** (str) – The project variable name.
- **value** (str) – The new project variable value.
- **part_name** (Optional[str]) – If not None, variable setting is restricted to the named part.
- **raw_write** (bool) – Whether the variable is written without access verifications.

Raises

- **ValueError** – If there is no custom argument with the given name.
- **RuntimeError** – If a write-once variable is set a second time, or if a part name is specified and the variable is set from a different part.

Return type
None

property target_arch: str

Return the target architecture used for debs, snaps and charms.

Return type
str

class craft_parts.infos.**ProjectVar**(***data*)

Bases: YamlModel

Project variables that can be updated using craftctl.

Parameters
data (Any) –

updated: bool

value: str

class craft_parts.infos.**StepInfo**(*part_info*, *step*)

Bases: object

Step-level information containing project, part, and step fields.

Parameters

- **part_info** (*PartInfo*) – The part information.
- **step** (*Step*) – The step we want to obtain information from.

craft_parts.lifecycle_manager module

The parts lifecycle manager.

class craft_parts.lifecycle_manager.**LifecycleManager**(*all_parts*, ***, *application_name*, *cache_dir*, *work_dir='.'*, *arch=''*, *base=''*, *project_name=None*, *parallel_build_count=1*, *application_package_name=None*, *ignore_local_sources=None*, *extra_build_packages=None*, *extra_build_snaps=None*, *track_stage_packages=False*, *base_layer_dir=None*, *base_layer_hash=None*, *project_vars_part_name=None*, *project_vars=None*, ***custom_args*)

Bases: object

Coordinate the planning and execution of the parts lifecycle.

The lifecycle manager determines the list of actions that needs be executed in order to obtain a tree of installed files from the specification on how to process its parts, and provides a mechanism to execute each of these actions.

Parameters

- **all_parts** (Dict[str, Any]) – A dictionary containing the parts specification according to the parts schema. The format is compatible with the output generated by PyYAML's yaml.load.
- **application_name** (str) – A unique non-empty identifier for the application using Craft Parts. Valid application names contain upper and lower case letters, underscores or numbers, and must start with a letter.
- **project_name** (Optional[str]) – name of the project being built.
- **cache_dir** (Union[Path, str]) – The path to store cached packages and files. If not specified, a directory under the application name entry in the XDG base directory will be used.
- **work_dir** (Union[Path, str]) – The toplevel directory for work directories. The current directory will be used if none is specified.
- **arch** (str) – The architecture to build for. Defaults to the host system architecture.
- **base** (str) – The system base the project being processed will run on. Defaults to the system where Craft Parts is being executed.
- **parallel_build_count** (int) – The maximum number of concurrent jobs to be used to build each part of this project.
- **application_package_name** (Optional[str]) – The name of the application package, if required by the package manager used by the platform. Defaults to the application name.
- **ignore_local_sources** (Optional[List[str]]) – A list of local source patterns to ignore.
- **extra_build_packages** (Optional[List[str]]) – A list of additional build packages to install.
- **extra_build_snaps** (Optional[List[str]]) – A list of additional build snaps to install.
- **track_stage_packages** (bool) – Add primed stage packages to the prime state.
- **base_layer_dir** (Optional[Path]) – The path to the overlay base layer, if using overlays.
- **base_layer_hash** (Optional[bytes]) – The validation hash of the overlay base image, if using overlays. The validation hash should be constant for a given image, and should change if a different base image is used.
- **project_vars_part_name** (Optional[str]) – Project variables can only be set in the part matching this name.
- **project_vars** (Optional[Dict[str, str]]) – A dictionary containing project variables.
- **custom_args** (Any) – Any additional arguments that will be passed directly to callbacks.

action_executor()

Return a context manager for action execution.

Return type
ExecutionContext

clean(*step=Step.PULL*, ***, *part_names=None*)

Clean the specified step and parts.

Cleaning a step removes its state and all artifacts generated in that step and subsequent steps for the specified parts.

Parameters

- **step** (*Step*) – The step to clean. If not specified, all steps will be cleaned.

- **part_names** (Optional[List[str]]) – The list of part names to clean. If not specified, all parts will be cleaned and work directories will be removed.

Return type
None

get_primed_stage_packages(*, *part_name*)

Return the list of primed stage packages.

Parameters
part_name (str) – The name of the part to get primed stage packages from.

Return type
Optional[List[str]]

Returns
The sorted list of primed stage packages, or None if no state found.

get_pull_assets(*, *part_name*)

Return the part's pull state assets.

Parameters
part_name (str) – The name of the part to get assets from.

Return type
Optional[Dict[str, Any]]

Returns
The dictionary of the part's pull assets, or None if no state found.

plan(*target_step*, *part_names=None*)

Obtain the list of actions to be executed given the target step and parts.

Parameters

- **target_step** (*Step*) – The final step we want to reach.
- **part_names** (Optional[Sequence[str]]) – The list of parts to process. If not specified, all parts will be processed.
- **update** – Refresh the list of available packages.

Return type
List[*Action*]

Returns
The list of Action objects that should be executed in order to reach the target step for the specified parts.

property project_info: *ProjectInfo*

Obtain information about this project.

Return type
ProjectInfo

refresh_packages_list()

Update the available packages list.

The list of available packages should be updated before planning the sequence of actions to take. To ensure consistency between the scenarios, it shouldn't be updated between planning and execution.

Return type
None

reload_state()

Reload the ephemeral state from disk.

Return type
None

craft_parts.main module

Part crafting command line tool.

This is the main entry point for the craft_parts package, invoked when running *python -mcraft_parts*. It provides basic functionality to process a parts specification and display the planned sequence of actions (using *–dry-run*) or execute them.

craft_parts.main.**main**()

Run the command-line interface.

Return type
None

craft_parts.parts module

Definitions and helpers to handle parts.

class craft_parts.parts.**Part**(*name*, *data*, *, *project_dirs=None*, *plugin_properties=None*)

Bases: object

Each of the components used in the project specification.

During the craft-parts lifecycle each part is processed through different steps in order to obtain its final artifacts. The Part class holds the part specification data and additional configuration information used during step processing.

Parameters

- **name** (str) – The part name.
- **data** (Dict[str, Any]) – A dictionary containing the part properties.
- **project_dirs** (Optional[*ProjectDirs*]) – The project work directories.
- **plugin_properties** (Optional[*PluginProperties*]) –

Raises
PartSpecificationError – If part validation fails.

property dependencies: List[str]

Return the list of parts this part depends on.

Return type
List[str]

property has_overlay: bool

Return whether this part declares overlay content.

Return type
bool

property overlay_dir: Path

Return the overlay directory.

Return type
Path

property part_build_dir: Path

Return the subdirectory containing the part build tree.

Return type
Path

property part_build_subdir: Path

Return the subdirectory in build containing the source subtree (if any).

Parts that have a source subdirectory and do not support out-of-source builds will have a build subdirectory.

Return type
Path

property part_install_dir: Path

Return the subdirectory to install the part build artifacts.

Return type
Path

property part_layer_dir: Path

Return the subdirectory containing the part overlay files.

Return type
Path

property part_packages_dir: Path

Return the subdirectory containing the part stage packages directory.

Return type
Path

property part_run_dir: Path

Return the subdirectory containing the part plugin scripts.

Return type
Path

property part_snaps_dir: Path

Return the subdirectory containing the part snap packages directory.

Return type
Path

property part_src_dir: Path

Return the subdirectory containing the part source code.

Return type
Path

property part_src_subdir: Path

Return the subdirectory in source containing the source subtree (if any).

Return type
Path

property part_state_dir: Path

Return the subdirectory containing the part lifecycle state.

Return type
Path

property parts_dir: Path

Return the directory containing work files for each part.

Return type
Path

property prime_dir: Path

Return the primed tree containing the artifacts to deploy.

Return type
Path

property stage_dir: Path

Return the staging area containing the installed files from all parts.

Return type
Path

class craft_parts.parts.PartSpec(**data)

Bases: BaseModel

The part specification data.

Parameters
data (Any) –

class Config

Bases: object

Pydantic model configuration.

alias_generator()

allow_mutation = False

extra = 'forbid'

validate_assignment = True

after: List[str]

build_attributes: List[str]

build_environment: List[Dict[str, str]]

build_packages: List[str]

build_snaps: List[str]

disable_parallel: bool

get_scriptlet(*step*)

Return the scriptlet contents, if any, for the given step.

Parameters
step (*Step*) – the step corresponding to the scriptlet to be retrieved.

Return type
Optional[str]

Returns
The scriptlet for the given step, if any.

marshal()

Create a dictionary containing the part specification data.

Return type
Dict[str, Any]

Returns
The newly created dictionary.

organize_files: Dict[str, str]

overlay_files: List[str]

overlay_packages: List[str]

overlay_script: Optional[str]

override_build: Optional[str]

override_prime: Optional[str]

override_pull: Optional[str]

override_stage: Optional[str]

permissions: List[*Permissions*]

plugin: Optional[str]

prime_files: List[str]

source: Optional[str]

source_branch: str

source_checksum: str

source_commit: str

source_depth: int

source_subdir: str

source_submodules: Optional[List[str]]

source_tag: str

source_type: str

stage_files: List[str]

stage_packages: List[str]

stage_snaps: List[str]

classmethod unmarshal(*data*)

Create and populate a new `PartSpec` object from dictionary data.

The unmarshal method validates entries in the input dictionary, populating the corresponding fields in the data object.

Parameters
data (`Dict[str, Any]`) – The dictionary data to unmarshal.

Return type
PartSpec

Returns
The newly created object.

Raises
TypeError – If data is not a dictionary.

classmethod validate_relative_path_list(*item*)

Check if the list does not contain empty of absolute paths.

Parameters
item (`str`) –

Return type
`str`

classmethod validate_root(*values*)

Check if the part spec has a valid configuration of packages and slices.

Parameters
values (`Dict[str, Any]`) –

Return type
`Dict[str, Any]`

craft_parts.parts.**get_parts_with_overlay**(*, *part_list*)

Obtain a list of parts that declare overlay parameters.

Parameters
part_list (`List[Part]`) – A list of all parts in the project.

Return type
`List[Part]`

Returns
A list of parts with overlay parameters.

craft_parts.parts.**has_overlay_visibility**(*part*, *, *part_list*, *viewers=None*)

Check if a part can see the overlay filesystem.

A part that declares overlay parameters and all parts depending on it are granted permission to see overlay filesystem.

Parameters

- **part** (*Part*) – The part whose overlay visibility will be checked.
- **viewers** (`Optional[Set[Part]]`) – Parts that are known to have overlay visibility.
- **part_list** (`List[Part]`) – A list of all parts in the project.

Return type
`bool`

Returns
Whether the part has overlay visibility.

craft_parts.parts.**part_by_name**(*name*, *part_list*)

Obtain the part with the given name from the part list.

Parameters

- **name** (str) – The name of the part to return.
- **part_list** (List[*Part*]) – The list of all known parts.

Return type
Part

Returns
The part with the given name.

craft_parts.parts.**part_dependencies**(*part*, *, *part_list*, *recursive=False*)

Return a set of all the parts upon which the named part depends.

Parameters

- **part** (*Part*) – The dependent part.
- **part_list** (List[*Part*]) –
- **recursive** (bool) –

Return type
Set[*Part*]

Returns
The set of parts the given part depends on.

craft_parts.parts.**part_list_by_name**(*names*, *part_list*)

Return a list of parts from part_list that are named in names.

Parameters

- **names** (Optional[Sequence[str]]) – The list of part names. If the list is empty or not defined, return all parts from part_list.
- **part_list** (List[*Part*]) – The list of all known parts.

Return type
List[*Part*]

Returns
The list of parts corresponding to the given names.

Raises
InvalidPartName – if a part name is not defined.

craft_parts.parts.**sort_parts**(*part_list*)

Perform an inneficient but easy to follow sorting of parts.

Parameters
part_list (List[*Part*]) – The list of parts to sort.

Return type
List[*Part*]

Returns
The sorted list of parts.

Raises
PartDependencyCycle – if there are circular dependencies.

craft_parts.parts.**validate_part**(*data*)

Validate the given part data against common and plugin models.

Parameters
data (Dict[str, Any]) – The part data to validate.

Return type
None

craft_parts.permissions module

Specify and apply permissions and ownership to part-owned files.

class craft_parts.permissions.**Permissions**(***data*)

Bases: BaseModel

Description of the ownership and permission settings for a set of files.

A Permissions object specifies that a given pattern-like path should be owned by owner with a given group, and have the read/write/execute bits defined by mode.

Notes

- path is optional and defaults to "everything";
- owner and group are optional if both are omitted - that is, if one of the pair is specified then both must be;
- mode is a string containing an integer in base 8. For example, "755",

"0755" and "0o755" are all accepted and are the equivalent of calling chmod 755 ....

Parameters
data (Any) –

applies_to(*path*)

Whether this Permissions' path pattern applies to path.

Parameters
path (Union[Path, str]) –

Return type
bool

apply_permissions(*target*)

Apply the permissions configuration to target.

Note that this method doesn't check if this Permissions's path pattern matches target; be sure to call applies_to() beforehand.

Parameters
target (Union[Path, str]) –

Return type
None

group: Optional[int]

mode: Optional[str]

property mode_octal: int

Get the mode as a base-8 integer.

Return type
int

owner: Optional[int]

path: str

classmethod validate_root(*values*)

Validate that "owner" and "group" are correctly specified.

Parameters
values (Dict[Any, Any]) –

Return type
Dict[Any, Any]

craft_parts.permissions.**apply_permissions**(*target*, *permissions*)

Apply all permissions configurations in permissions to target.

Parameters

- **target** (Union[Path, str]) –
- **permissions** (List[*Permissions*]) –

Return type
None

craft_parts.permissions.**filter_permissions**(*target*, *permissions*)

Get the subset of permissions whose path patterns apply to target.

Parameters

- **target** (Union[Path, str]) –
- **permissions** (List[*Permissions*]) –

Return type
List[*Permissions*]

craft_parts.permissions.**permissions_are_compatible**(*left*, *right*)

Whether two sets of permissions definitions are not in conflict with each other.

The function determines whether applying the two lists of Permissions to a given path would result in the same owner, group and mode.

Remarks:

- **If either of the parameters is None or empty, they are considered compatible**
  because they are understood to not be "in conflict".
- **Otherwise, the permissions are incompatible if one would they would set one**
  of the attributes (owner, group and mode) to different values, *even if* one of them would not modify the attribute at all.
- **The path attribute of the Permissions are completely ignored, as they**
  are understood to apply to the same file of interest through a previous call of filter_permissions().

type left
Optional[List[*Permissions*]]

param left
the first set of permissions.

type right
Optional[List[*Permissions*]]

param right
the second set of permissions.

rtype
bool

craft_parts.sequencer module

Determine the sequence of lifecycle actions to be executed.

class craft_parts.sequencer.**Sequencer**(*, *part_list*, *project_info*, *ignore_outdated=None*, *base_layer_hash=None*)

Bases: object

Obtain a list of actions from the parts specification.

The sequencer takes the parts definition and the current state of a project to plan all the actions needed to reach a given target step. State is read from persistent storage and updated entirely in memory. Sequencer operations never change disk contents.

Parameters

- **part_list** (List[*Part*]) – The list of parts to process.
- **project_info** (*ProjectInfo*) – Information about this project.
- **ignore_outdated** (Optional[List[str]]) – A list of file patterns to ignore when testing for outdated files.
- **base_layer_hash** (Optional[*LayerHash*]) –

plan(*target_step*, *part_names=None*)

Determine the list of steps to execute for each part.

Parameters

- **target_step** (*Step*) – The final step to execute for the given part names.
- **part_names** (Optional[Sequence[str]]) – The names of the parts to process.

Return type
List[*Action*]

Returns
The list of actions that should be executed.

reload_state()

Reload state from persistent storage.

Return type
None

craft_parts.steps module

Definitions and helpers to handle lifecycle steps.

class craft_parts.steps.**Step**(*value*)

Bases: IntEnum

All the steps needed to fully process a part.

Steps correspond to the tasks that must be fulfilled in order to process each of the parts in the project specification. In the PULL step sources for a part are retrieved and unpacked. The OVERLAY step is used to change the underlying filesystem. In the BUILD step artifacts are built and installed. In the STAGE step installed build artifacts from all parts and overlay contents are added to a staging area. These files are further processed in the PRIME step to obtain the final tree with the final payload for deployment.

BUILD = 3

OVERLAY = 2

PRIME = 5

PULL = 1

STAGE = 4

next_steps()

List the steps that should happen after the current step.

Return type
List[*Step*]

Returns
The list of next steps.

previous_steps()

List the steps that should happen before the current step.

Return type
List[*Step*]

Returns
The list of previous steps.

craft_parts.steps.**dependency_prerequisite_step**(*step*)

Obtain the step a given step may depend on.

Return type
Optional[*Step*]

Returns
The prerequisite step.

Parameters
step (*Step*) –

craft_parts.xattrs module

Helpers to read and write filesystem extended attributes.

craft_parts.xattrs.**read_xattr**(*path*, *key*)

Get extended attribute metadata from a file.

Parameters

- **path** (str) – The file to get metadata from.
- **key** (str) – The attribute key.

Return type
Optional[str]

Returns
The attribute value.

craft_parts.xattrs.**write_xattr**(*path*, *key*, *value*)

Add extended attribute metadata to a file.

Parameters

- **path** (str) – The file to add metadata to.
- **key** (str) – The attribute key.
- **value** (str) – The attribute value.

Return type
None

Module contents

Craft a project from several parts.

class craft_parts.**Action**(*part_name*, *step*, *action_type=ActionType.RUN*, *reason=None*, *project_vars=None*, *properties=ActionProperties(changed_files=None, changed_dirs=None)*)

Bases: object

The action to be executed for a given part.

Actions correspond to the operations required to run the lifecycle for each of the parts in the project specification.

Parameters

- **part_name** (str) – The name of the part this action will be performed on.
- **step** (*Step*) – The *Step* this action will execute.
- **action_type** (*ActionType*) – Action to run for this step.
- **reason** (Optional[str]) – A textual description of why this action should be executed.
- **project_vars** (Optional[Dict[str, *ProjectVar*]]) – The values of project variables from a previous execution, used if the action type is ActionType.SKIP.
- **properties** (*ActionProperties*) –

action_type: *ActionType* = 0

part_name: str

project_vars: Optional[Dict[str, *ProjectVar*]] = None

properties: *ActionProperties* = ActionProperties(changed_files=None, changed_dirs=None)

reason: Optional[str] = None

step: *Step*

class craft_parts.**ActionProperties**(*changed_files=None*, *changed_dirs=None*)

Bases: object

Properties defined for an action.

Parameters

- **changed_files** (Optional[List[str]]) –
- **changed_dirs** (Optional[List[str]]) –

changed_dirs: Optional[List[str]] = None

changed_files: Optional[List[str]] = None

class craft_parts.**ActionType**(*value*)

Bases: IntEnum

The type of action to be executed.

Action execution can be modified according to its type:

RUN: execute the expected commands for step processing.

RERUN: clear the existing data and state before procceeding.

UPDATE: try to continue processing the step.

SKIP: don't execute this action.

REAPPLY: run the step commands without updating its state.

REAPPLY = 4

RERUN = 1

RUN = 0

SKIP = 2

UPDATE = 3

class craft_parts.**LifecycleManager**(*all_parts*, ***, *application_name*, *cache_dir*, *work_dir='.'*, *arch=''*, *base=''*, *project_name=None*, *parallel_build_count=1*, *application_package_name=None*, *ignore_local_sources=None*, *extra_build_packages=None*, *extra_build_snaps=None*, *track_stage_packages=False*, *base_layer_dir=None*, *base_layer_hash=None*, *project_vars_part_name=None*, *project_vars=None*, ***custom_args*)

Bases: object

Coordinate the planning and execution of the parts lifecycle.

The lifecycle manager determines the list of actions that needs be executed in order to obtain a tree of installed files from the specification on how to process its parts, and provides a mechanism to execute each of these actions.

Parameters

- **all_parts** (Dict[str, Any]) – A dictionary containing the parts specification according to the parts schema. The format is compatible with the output generated by PyYAML's yaml.load.
- **application_name** (str) – A unique non-empty identifier for the application using Craft Parts. Valid application names contain upper and lower case letters, underscores or numbers, and must start with a letter.
- **project_name** (Optional[str]) – name of the project being built.
- **cache_dir** (Union[Path, str]) – The path to store cached packages and files. If not specified, a directory under the application name entry in the XDG base directory will be used.
- **work_dir** (Union[Path, str]) – The toplevel directory for work directories. The current directory will be used if none is specified.
- **arch** (str) – The architecture to build for. Defaults to the host system architecture.
- **base** (str) – The system base the project being processed will run on. Defaults to the system where Craft Parts is being executed.
- **parallel_build_count** (int) – The maximum number of concurrent jobs to be used to build each part of this project.
- **application_package_name** (Optional[str]) – The name of the application package, if required by the package manager used by the platform. Defaults to the application name.
- **ignore_local_sources** (Optional[List[str]]) – A list of local source patterns to ignore.
- **extra_build_packages** (Optional[List[str]]) – A list of additional build packages to install.
- **extra_build_snaps** (Optional[List[str]]) – A list of additional build snaps to install.
- **track_stage_packages** (bool) – Add primed stage packages to the prime state.
- **base_layer_dir** (Optional[Path]) – The path to the overlay base layer, if using overlays.
- **base_layer_hash** (Optional[bytes]) – The validation hash of the overlay base image, if using overlays. The validation hash should be constant for a given image, and should change if a different base image is used.
- **project_vars_part_name** (Optional[str]) – Project variables can only be set in the part matching this name.
- **project_vars** (Optional[Dict[str, str]]) – A dictionary containing project variables.
- **custom_args** (Any) – Any additional arguments that will be passed directly to callbacks.

action_executor()

Return a context manager for action execution.

Return type
ExecutionContext

clean(*step=Step.PULL*, ***, *part_names=None*)

Clean the specified step and parts.

Cleaning a step removes its state and all artifacts generated in that step and subsequent steps for the specified parts.

Parameters

- **step** (*Step*) – The step to clean. If not specified, all steps will be cleaned.
- **part_names** (Optional[List[str]]) – The list of part names to clean. If not specified, all parts will be cleaned and work directories will be removed.

Return type
None

get_primed_stage_packages(*, *part_name*)

Return the list of primed stage packages.

Parameters
part_name (str) – The name of the part to get primed stage packages from.

Return type
Optional[List[str]]

Returns
The sorted list of primed stage packages, or None if no state found.

get_pull_assets(*, *part_name*)

Return the part's pull state assets.

Parameters
part_name (str) – The name of the part to get assets from.

Return type
Optional[Dict[str, Any]]

Returns
The dictionary of the part's pull assets, or None if no state found.

plan(*target_step*, *part_names=None*)

Obtain the list of actions to be executed given the target step and parts.

Parameters

- **target_step** (*Step*) – The final step we want to reach.
- **part_names** (Optional[Sequence[str]]) – The list of parts to process. If not specified, all parts will be processed.
- **update** – Refresh the list of available packages.

Return type
List[*Action*]

Returns
The list of *Action* objects that should be executed in order to reach the target step for the specified parts.

property project_info: *ProjectInfo*

Obtain information about this project.

Return type
ProjectInfo

refresh_packages_list()

Update the available packages list.

The list of available packages should be updated before planning the sequence of actions to take. To ensure consistency between the scenarios, it shouldn't be updated between planning and execution.

Return type
None

`reload_state`()

Reload the ephemeral state from disk.

Return type
None

class craft_parts.**Part**(*name*, *data*, *, *project_dirs=None*, *plugin_properties=None*)

Bases: object

Each of the components used in the project specification.

During the craft-parts lifecycle each part is processed through different steps in order to obtain its final artifacts. The Part class holds the part specification data and additional configuration information used during step processing.

Parameters

- **name** (str) – The part name.
- **data** (Dict[str, Any]) – A dictionary containing the part properties.
- **project_dirs** (Optional[*ProjectDirs*]) – The project work directories.
- **plugin_properties** (Optional[*PluginProperties*]) –

Raises
PartSpecificationError – If part validation fails.

property dependencies: List[str]

Return the list of parts this part depends on.

Return type
List[str]

property has_overlay: bool

Return whether this part declares overlay content.

Return type
bool

property overlay_dir: Path

Return the overlay directory.

Return type
Path

property part_build_dir: Path

Return the subdirectory containing the part build tree.

Return type
Path

property part_build_subdir: Path

Return the subdirectory in build containing the source subtree (if any).

Parts that have a source subdirectory and do not support out-of-source builds will have a build subdirectory.

Return type
Path

property part_install_dir: Path

Return the subdirectory to install the part build artifacts.

Return type
Path

property part_layer_dir: Path

Return the subdirectory containing the part overlay files.

Return type
Path

property part_packages_dir: Path

Return the subdirectory containing the part stage packages directory.

Return type
Path

property part_run_dir: Path

Return the subdirectory containing the part plugin scripts.

Return type
Path

property part_snaps_dir: Path

Return the subdirectory containing the part snap packages directory.

Return type
Path

property part_src_dir: Path

Return the subdirectory containing the part source code.

Return type
Path

property part_src_subdir: Path

Return the subdirectory in source containing the source subtree (if any).

Return type
Path

property part_state_dir: Path

Return the subdirectory containing the part lifecycle state.

Return type
Path

property parts_dir: Path

Return the directory containing work files for each part.

Return type
Path

property prime_dir: Path

Return the primed tree containing the artifacts to deploy.

Return type
Path

property stage_dir: Path

Return the staging area containing the installed files from all parts.

Return type
Path

class craft_parts.PartInfo(*project_info*, *part*)

Bases: object

Part-level information containing project and part fields.

Parameters

- **project_info** (*ProjectInfo*) – The project information.
- **part** (*Part*) – The part we want to obtain information from.

get_project_var(*name*, *, *raw_read=False*)

Get the value of a project variable.

Variables must be consumed by the application only after the lifecycle execution ends to prevent unexpected behavior if steps are skipped.

Parameters

- **name** (str) – The project variable name.
- **raw_read** (bool) – Whether the variable is read without access verifications.

Return type
str

Returns
The value of the variable.

Raises

- **ValueError** – If there is no project variable with the given name.
- **RuntimeError** – If the variable is consumed during the lifecycle execution.

property part_build_dir: Path

Return the subdirectory containing the part's build tree.

Return type
Path

property part_build_subdir: Path

Return the subdirectory in build containing the source subtree (if any).

Return type
Path

property part_install_dir: Path

Return the subdirectory to install the part's build artifacts.

Return type
Path

property part_name: str

Return the name of the part we're providing information about.

Return type
str

property part_src_dir: Path

Return the subdirectory containing the part's source code.

Return type
Path

property part_src_subdir: Path

Return the subdirectory in source containing the source subtree (if any).

Return type
Path

property part_state_dir: Path

Return the subdirectory containing this part's lifecycle state.

Return type
Path

property project_info: *ProjectInfo*

Return the project information.

Return type
ProjectInfo

set_project_var(*name*, *value*, *, *raw_write=False*)

Set the value of a project variable.

Variable values can be set once. Project variables are not intended for logic construction in user scripts, setting it multiple times is likely to be an error.

Parameters

- **name** (str) – The project variable name.
- **value** (str) – The new project variable value.
- **raw_write** (bool) – Whether the variable is written without access verifications.

Raises

- **ValueError** – If there is no custom argument with the given name.
- **RuntimeError** – If a write-once variable is set a second time, or if a part name is specified and the variable is set from a different part.

Return type
None

exception craft_parts.**PartsError**(*brief*, *details=None*, *resolution=None*)

Bases: Exception

Unexpected error.

Parameters

- **brief** (str) – Brief description of error.
- **details** (Optional[str]) – Detailed information.
- **resolution** (Optional[str]) – Recommendation, if any.

brief: str

details: Optional[str] = None

resolution: Optional[str] = None

class craft_parts.**ProjectDirs**(*, *work_dir='.'*)

Bases: object

The project's main work directories.

Parameters

work_dir (Union[Path, str]) – The parent directory containing the parts, prime and stage subdirectories. If not specified, the current directory will be used.

Variables

- **work_dir** – The root of the work directories used for project processing.
- **parts_dir** – The directory containing work subdirectories for each part.
- **overlay_dir** – The directory containing work subdirectories for overlays.
- **overlay_mount_dir** – The mountpoint for the overlay filesystem.
- **overlay_packages_dir** – The cache directory for overlay packages.
- **overlay_work_dir** – The work directory for the overlay filesystem.
- **stage_dir** – The staging area containing installed files from all parts.
- **prime_dir** – The primed tree containing the final artifacts to deploy.

class craft_parts.**ProjectInfo**(*, *application_name*, *cache_dir*, *arch=''*, *base=''*, *parallel_build_count=1*, *project_dirs=None*, *project_name=None*, *project_vars_part_name=None*, *project_vars=None*, ***custom_args*)

Bases: object

Project-level information containing project-specific fields.

Parameters

- **application_name** (str) – A unique identifier for the application using Craft Parts.
- **project_name** (Optional[str]) – name of the project being built.
- **cache_dir** (Path) – The path to store cached packages and files. If not specified, a directory under the application name entry in the XDG base directory will be used.
- **arch** (str) – The architecture to build for. Defaults to the host system architecture.
- **parallel_build_count** (int) – The maximum number of concurrent jobs to be used to build each part of this project.
- **project_dirs** (Optional[*ProjectDirs*]) – The project work directories.
- **project_name** – The name of the project.
- **project_vars_part_name** (Optional[str]) – Project variables can be set only if the part name matches this name.
- **project_vars** (Optional[Dict[str, str]]) – A dictionary containing the project variables.
- **custom_args** (Any) – Any additional arguments defined by the application when creating a *LifecycleManager*.
- **base** (str) –

property application_name: str

Return the name of the application using craft-parts.

Return type
str

property arch_triplet: str

Return the machine-vendor-os platform triplet definition.

Return type
str

property base: str

Return the project build base.

Return type
str

property cache_dir: Path

Return the directory used to store cached files.

Return type
Path

property custom_args: List[str]

Return the list of custom argument names.

Return type
List[str]

property dirs: *ProjectDirs*

Return the project's work directories.

Return type
ProjectDirs

get_project_var(*name*, *, *raw_read=False*)

Get the value of a project variable.

Variables must be consumed by the application only after the lifecycle execution ends to prevent unexpected behavior if steps are skipped.

Parameters

- **name** (str) – The project variable name.
- **raw_read** (bool) – Whether the variable is read without access verifications.

Return type
str

Returns
The value of the variable.

Raises

- **ValueError** – If there is no project variable with the given name.
- **RuntimeError** – If the variable is consumed during the lifecycle execution.

property host_arch: str

Return the host architecture used for debs, snaps and charms.

Return type
str

property is_cross_compiling: bool

Whether the target and host architectures are different.

Return type
bool

property parallel_build_count: int

Return the maximum allowable number of concurrent build jobs.

Return type
int

property project_name: Optional[str]

Return the name of the project using craft-parts.

Return type
Optional[str]

property project_options: Dict[str, Any]

Obtain a project-wide options dictionary.

Return type
Dict[str, Any]

set_project_var(*name*, *value*, *raw_write=False*, ***, *part_name=None*)

Set the value of a project variable.

Variable values can be set once. Project variables are not intended for logic construction in user scripts, setting it multiple times is likely to be an error.

Parameters

- **name** (str) – The project variable name.
- **value** (str) – The new project variable value.
- **part_name** (Optional[str]) – If not None, variable setting is restricted to the named part.
- **raw_write** (bool) – Whether the variable is written without access verifications.

Raises

- **ValueError** – If there is no custom argument with the given name.
- **RuntimeError** – If a write-once variable is set a second time, or if a part name is specified and the variable is set from a different part.

Return type
None

property target_arch: str

Return the target architecture used for debs, snaps and charms.

Return type
str

class craft_parts.**Step**(*value*)

Bases: IntEnum

All the steps needed to fully process a part.

Steps correspond to the tasks that must be fulfilled in order to process each of the parts in the project specification. In the PULL step sources for a part are retrieved and unpacked. The OVERLAY step is used to change the underlying filesystem. In the BUILD step artifacts are built and installed. In the STAGE step installed build artifacts from all parts and overlay contents are added to a staging area. These files are further processed in the PRIME step to obtain the final tree with the final payload for deployment.

BUILD = 3

OVERLAY = 2

PRIME = 5

PULL = 1

STAGE = 4

next_steps()

List the steps that should happen after the current step.

Return type
List[*Step*]

Returns
The list of next steps.

previous_steps()

List the steps that should happen before the current step.

Return type
List[*Step*]

Returns
The list of previous steps.

class craft_parts.**StepInfo**(*part_info*, *step*)

Bases: object

Step-level information containing project, part, and step fields.

Parameters

- **part_info** (*PartInfo*) – The part information.
- **step** (*Step*) – The step we want to obtain information from.

craft_parts.**expand_environment**(*data*, ***, *info*, *skip=None*)

Replace global variables with their values.

Global variables are defined by craft-parts and are the subset of the CRAFT_* step execution environment variables that don't depend on the part or step being executed. The list of global variables include CRAFT_ARCH_TRIPLET, CRAFT_PROJECT_DIR, CRAFT_STAGE and CRAFT_PRIME. Additional global variables can be defined by the application using craft-parts.

Parameters

- **data** (Dict[str, Any]) – A dictionary whose values will have variable names expanded.
- **info** (*ProjectInfo*) – The project information.

- **skip** (Optional[List[str]]) – Keys to skip when performing expansion.

Return type
None

craft_parts.**validate_part**(*data*)

Validate the given part data against common and plugin models.

Parameters
data (Dict[str, Any]) – The part data to validate.

Return type
None

3.7 Changelog

3.7.1 1.18.0 (2023-01-19)

- Add SCons plugin
- Add Ant plugin
- Add Maven plugin
- Fix lifecycle work directory cleaning
- Make stage package tracking optional
- Improve chisel error handling
- Improve missing local source error message
- Documentation fixes and updates

3.7.2 1.17.1 (2022-11-23)

- Allow plus symbol in git url scheme

3.7.3 1.17.0 (2022-11-14)

- Fix go plugin mod download in jammy
- Remove hardcoded ubuntu version in chisel call
- Add plain file source handler
- Pass build attributes and state to post-step callback

3.7.4 1.16.0 (2022-10-20)

- Add file permission setting
- Take permissions into account when checking file collisions
- Only refresh overlay packages if necessary
- Generate separate environment setup file
- Make changed file list available to plugins

3.7.5 1.15.1 (2022-10-14)

- Fix device nodes in overlay base image

3.7.6 1.15.0 (2022-10-11)

- Add support to chisel slices
- Add `go-generate` property to the go plugin

3.7.7 1.14.2 (2022-09-22)

- Fix pypi release package

3.7.8 1.14.1 (2022-09-21)

- Fix stage/prime filter combination

3.7.9 1.14.0 (2022-09-09)

- Add API call to validate parts

3.7.10 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

3.7.11 1.12.1 (2022-08-19)

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

3.7.12 1.12.0 (2022-08-12)

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

3.7.13 1.11.0 (2022-08-12)

- Add API call to list registered plugins

3.7.14 1.10.2 (2022-08-03)

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

3.7.15 1.10.1 (2022-07-29)

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

3.7.16 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

3.7.17 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

3.7.18 1.8.1 (2022-07-05)

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

3.7.19 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

3.7.20 1.7.2 (2022-06-14)

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

3.7.21 1.7.1 (2022-05-21)

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

3.7.22 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

3.7.23 1.6.1 (2022-05-02)

- Fix stage package symlink normalization

3.7.24 1.6.0 (2022-04-29)

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

3.7.25 1.5.1 (2022-04-25)

- Fix extra build snaps installation

3.7.26 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

3.7.27 1.4.2 (2022-04-01)

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

3.7.28 1.4.1 (2022-03-30)

- Fix project variable adoption scope

3.7.29 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

3.7.30 1.3.0 (2022-03-05)

- Add meson plugin
- Adjustments in git source tests

3.7.31 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

3.7.32 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

3.7.33 1.1.1 (2022-01-05)

- Pin pydantic and pydantic-yaml dependency versions

3.7.34 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

3.7.35 1.0.4 (2021-11-10)

- Declare additional public API names
- Add git source handler

3.7.36 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

3.7.37 1.0.2 (2021-09-16)

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

3.7.38 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

3.7.39 1.0.0 (2021-08-05)

- Initial release

3.8 Indices and tables

- genindex
- modindex

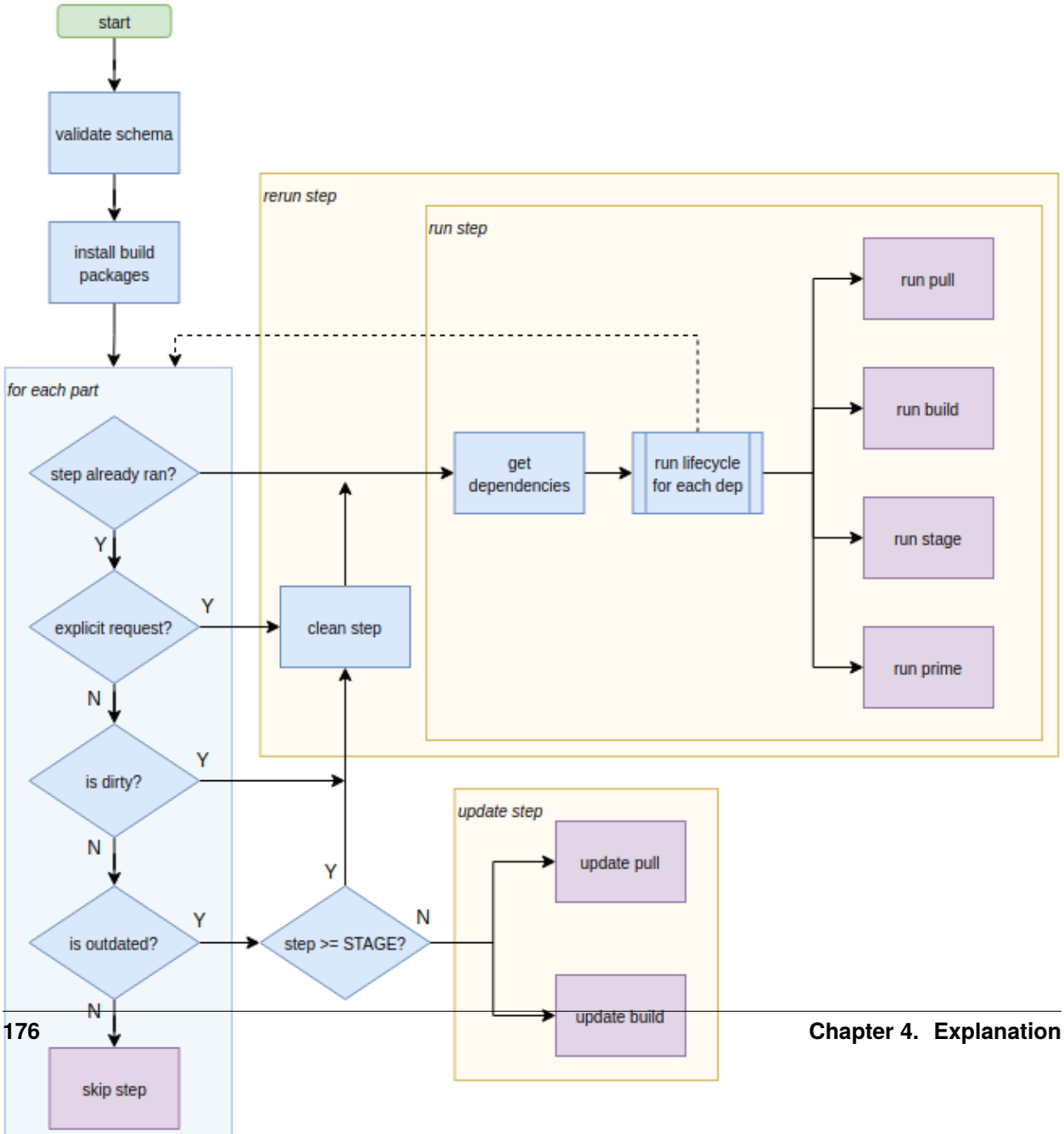
CHAPTER

FOUR

EXPLANATION

4.1 Lifecycle details

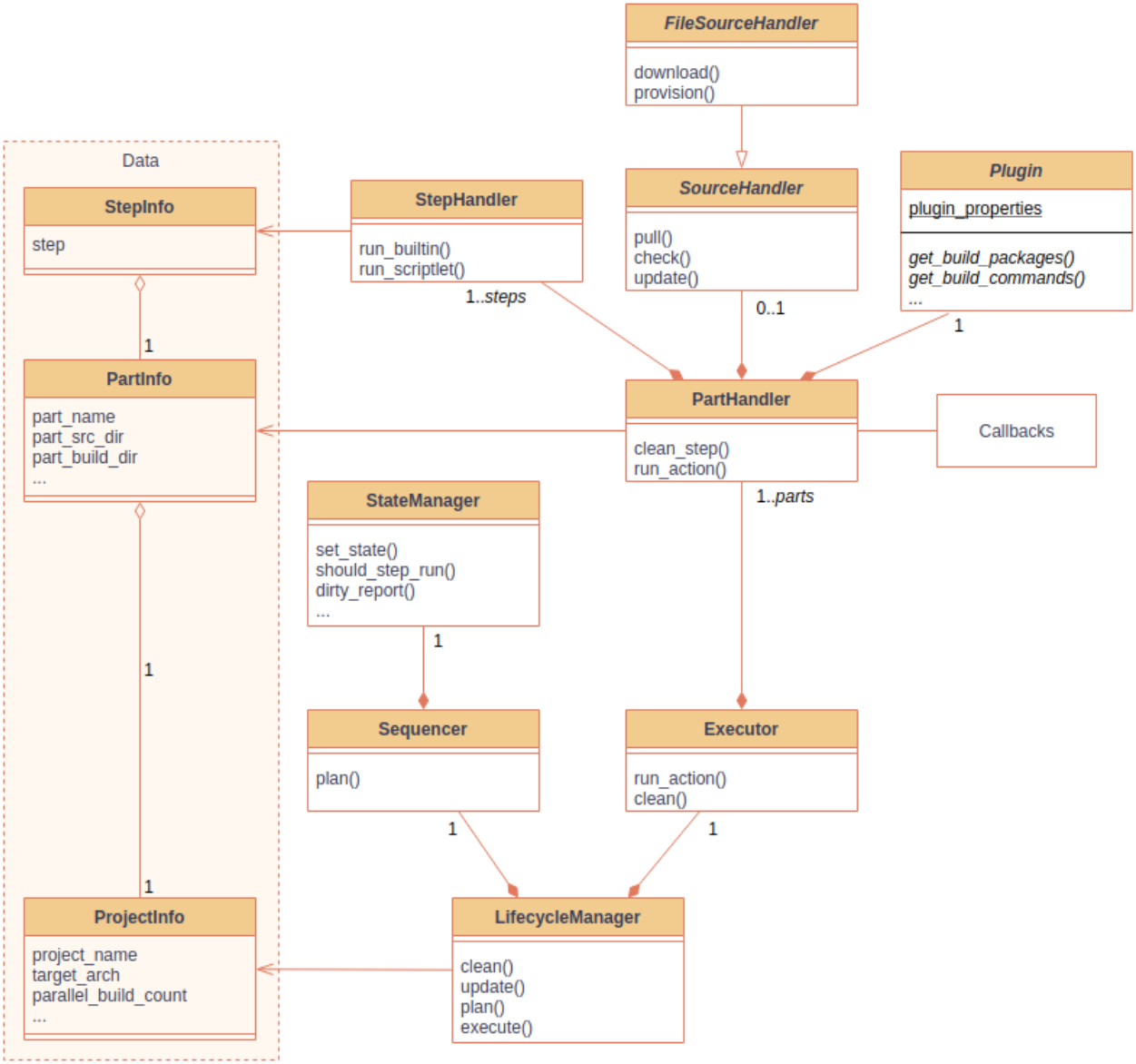
4.1.1 Lifecycle processing diagram



4.2 Implementation notes

4.2.1 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.



Tutorial **Get started** with a hands-on introduction to Craft Parts

How-to guides **Step-by-step guides** covering key operations and common tasks

Reference **Technical information** about Craft Parts' components and modules

Explanation **Discussion and clarification** of key topics

PYTHON MODULE INDEX

C

INDEX

C

D

E

F

G

H

I

N

O

P

R

S

W

X

Z