Craft CLI

Release 1.1.0

Canonical Ltd.

Sep 01, 2022

GETTING STARTED

CHAPTER

ONE

TUTORIALS

1.1 Run a command based application with craft-cli

This tutorial will explain how to use Craft CLI to run an application that is based on commands.

Along the way you will define a simple command (named *unlink*, with the functionality of removing files), and call the appropriate library mechanisms for that command to be executed when running the application.

1.1.1 Prerequisites

Craft CLI is a standard Python library, so the best way to have it available is installed in a virtual environment.

The first step, then, is to create a virtual environment (you may skip this test if you already have one):

```
$ python3 -m venv env
```

Note that Python 3.8 or 3.9 are the supported versions.

Then enable the virtual environment and install Craft CLI:

```
$ source env/bin/activate
$ pip install craft-cli
```

1.1.2 Define the command and run it using the Dispatcher

First start with a class sub-classing *BaseCommand* with the appropriate attributes to name it and have automatic help texts, then provide a *fill_parser* method to declare what arguments are possible for this command, and finally a *run* method where the "real" functionality is implemented:

```
import pathlib
import textwrap
import sys
from craft_cli import (
    ArgumentParsingError,
    BaseCommand,
    CommandGroup,
    CraftError,
    Dispatcher,
    EmitterMode,
    ProvideHelpException,
```

(continues on next page)

(continued from previous page)

```
    emit,
)


class RemoveFileCommand(BaseCommand):
    """Remove the indicated file."""

    name = "unlink"
    help_msg = "Remove the indicated file."
    overview = textwrap.dedent("""
        Remove the indicated file.

        A file needs to be indicated. It is an argument error if the path does not exist
        or it's a directory.

        It will return succesfully if the file was properly removed.
    """)

    def fill_parser(self, parser):
        """Add own parameters to the general parser."""
        parser.add_argument("filepath", type=pathlib.Path, help="The file to be removed")

    def run(self, parsed_args):
        """Run the command."""
        if not parsed_args.filepath.exists() or parsed_args.filepath.is_dir():
            raise ArgumentParsingError("The indicated path is not a file or does not␣
↪exist.")
        try:
            parsed_args.filepath.unlink()
        except Exception as exc:
            raise CraftError(f"Problem removing the file: {exc}.")

        emit.message("File removed succesfully.")
```

Then initiate the `emit` object and call the `Dispatcher` functionality:

```
emit.init(EmitterMode.BRIEF, "example-app", "Starting example app v1.")
command_groups = [CommandGroup("Basic", [RemoveFileCommand])]
summary = "Example application for the craft-cli tutorial."

try:
    dispatcher = Dispatcher("example-app", command_groups, summary=summary)
    dispatcher.pre_parse_args(sys.argv[1:])
    dispatcher.load_command(None)
    dispatcher.run()
except (ArgumentParsingError, ProvideHelpException) as err:
    print(err, file=sys.stderr)  # to stderr, as argparse normally does
    emit.ended_ok()
except CraftError as err:
    emit.error(err)
except KeyboardInterrupt as exc:
    error = CraftError("Interrupted.")
```

(continues on next page)

(continued from previous page)

```
    error.__cause__ = exc
    emit.error(error)
except Exception as exc:
    error = CraftError(f"Application internal error: {exc!r}")
    error.__cause__ = exc
    emit.error(error)
else:
    emit.ended_ok()
```

Finally, put both chunks of code in a `example-app.py` file, and (having the virtual environment you prepared at the beginning still activated), run it. You should see the help message for the whole application (as a command is missing, which would be the same output if you pass the *help*, *-h* or *–help* parameters):

```
$ python example-app.py
Usage:
    example-app [help] <command>

Summary:    Example application for the craft-cli tutorial.

Global options:
       -h, --help:  Show this help message and exit
    -v, --verbose:  Show debug information and be more verbose
      -q, --quiet:  Only show warnings and errors, not progress
      --verbosity:  Set the verbosity level to 'quiet', 'brief',
                    'verbose', 'debug' or 'trace'",

Starter commands:

Commands can be classified as follows:
          Example:  unlink

For more information about a command, run 'example-app help <command>'.
For a summary of all commands, run 'example-app help --all'.
```

Ask help for specifically for the command:

```
$ python example-app.py help unlink
Usage:
    example-app unlink [options] <filepath>

Summary:
    Remove the indicated file.

    A file needs to be indicated. It is an argument error if the path does not exist
    or it's a directory.

    It will return succesfully if the file was properly removed.

Options:
       -h, --help:  Show this help message and exit
    -v, --verbose:  Show debug information and be more verbose
      -q, --quiet:  Only show warnings and errors, not progress
```

(continues on next page)

(continued from previous page)

```
      --verbosity:  Set the verbosity level to 'quiet', 'brief',
                    'verbose', 'debug' or 'trace'",

For a summary of all commands, run 'example-app help --all'.
```

Time to run the command on a file, you should see the successful message:

```
$ touch testfile
$ ls testfile
testfile
$ env/bin/python example-app.py unlink testfile
File removed succesfully.
$ ls testfile
ls: cannot access 'testfile': No such file or directory
```

Explore different error situations, first trying to remove a directory, then trying to remove a file but with "unexpected" problems:

```
$ mkdir testdir
$ python example-app.py unlink testdir
The indicated path is not a file or does not exist.

$ touch /tmp/testfile
$ sudo chown root /tmp/testfile
$ python example-app.py unlink /tmp/testfile
Problem removing the file: [Errno 1] Operation not permitted: '/tmp/testfile'.
Full execution log: '/home/user/.cache/example-app/log/example-app-20220114-120745.
↪861866.log'
```

Congratulations! You have built a complete application with good UX by using Craft CLI and implementing the functionality in one command.

CHAPTER

TWO

HOW TOS

2.1 Use a different logfile structure than the default

To override *the default management of application log files*, a file path can be specified when initiating the *emit* object, using the `log_filepath` parameter:

```
emit.init(mode, appname, greeting, log_filepath)
```

Note that if you use this option, is up to you to provide proper management of those files (e.g. to rotate them).

2.2 End the application with different return codes

To enable the application to return different return codes in different situations, you need wrap the Dispatcher in a specific way (having different return codes in the different situations), to take in consideration the returned value from the command's run, and of course make the application to actually return the specific value.

In the following code structure we see all these effects at once:

```
try:
    ...
    retcode = dispatcher.run()
    if retcode is None:
        retcode = 0
except ArgumentParsingError as err:
    print(err, file=sys.stderr)  # to stderr, as argparse normally does
    emit.ended_ok()
    retcode = 1
except ProvideHelpException as err:
    print(err, file=sys.stderr)  # to stderr, as argparse normally does
    emit.ended_ok()
    retcode = 0
except CraftError as err:
    emit.error(err)
    retcode = err.retcode
except KeyboardInterrupt as exc:
    error = CraftError("Interrupted.")
    error.__cause__ = exc
    emit.error(error)
    retcode = 1
except Exception as exc:
```

(continues on next page)

(continued from previous page)

```
    error = CraftError(f"Application internal error: {exc!r}")
    error.__cause__ = exc
    emit.error(error)
    retcode = 1
else:
    emit.ended_ok()
sys.exit(retcode)
```

In detail:

- the return code from the command's execution is bound when calling *dispatcher.run*, supporting the case of it not returning anything (defaults to *0*)
- have different return codes assigned for the different *except* situations, with two particular cases: for `ProvideHelpException` it's *0* as it's a normal exit situation when the user requested for help, and for `CraftError` where the return code is taken from the exception itself
- a *sys.exit* at the very end for the process to return the value

2.3 Raise more informational errors

To provide more information to the user in case of an error, you can use the `CraftError` exception provided by the `craft-cli` library.

So, in addition of just passing a message to the user...

```
raise CraftError("The indicated file does not exist.")
```

...you can provide more information:

- `details`: full error details received from a third party or extended information about the situation, useful for debugging but not to be normally shown to the user. E.g.:

```
raise CraftError(
    "Cannot access the indicated file.",
    details=f"File permissions: {oct(filepath.stat().st_mode)}")

raise CraftError(
    f"Server returned bad code {error_code}",
    details=f"Full server response: {response.content!r}")
```

- `resolution`: an extra line indicating to the user how the error may be fixed or avoided. E.g.:

```
raise CraftError(
    "Cannot remove the directory.",
    resolution="Confirm that the directory is empty and has proper permissions.")
```

- `docs_url`: an URL to point the user to documentation. E.g.:

```
raise CraftError(
    "Invalid configuration: bad version value.",
    docs_url="https://mystuff.com/docs/how-to-migrate-config")
```

- `reportable`: if an error report should be sent to some error-handling backend (like Sentry). E.g.:

```
raise CraftError(
    f"Unexpected subprocess return code: {proc.returncode}.",
    reportable=True)
```

- `retcode`: the code to return when the application finishes (see *how to use this when wrapping Dispatcher*)

You should use any combination of these, as looks appropriate.

For further information reported to the user and/or sent to the log file, you should create `CraftError` specifying the original exception (if any). E.g.:

```
try:
    ...
except IOError as exc:
    raise CraftError(f"Error when frunging the perculux: {exc}") from exc
```

Finally, if you want to build a hierarchy of errors in the application, you should start the tree inheriting `CraftError` to use this functionality.

2.4 Define and use other global arguments

To define more automatic global arguments than the ones provided automatically by `Dispatcher` (see *this explanation* for more information), use the `GlobalArgument` object to create all you need and pass them to the `Dispatcher` at instatiaton time.

Check *craft_cli.dispatcher.GlobalArgument* for more information about the parameters needed, but it's very straightforward to create these objects. E.g.:

```
ga_sec = GlobalArgument("secure_mode", "flag", "-s", "--secure", "Run the app in secure␣
↪mode")
```

To use it, just pass a list of the needed global arguments to the dispatcher using the `extra_global_args` option:

```
dispatcher = Dispatcher(..., extra_global_args=[ga_sec])
```

The `dispatcher.pre_parse_args` method returns the global arguments already parsed, as a dictionary. Use the name you gave to the global argument to check for its value and react properly. E.g.:

```
global_args = dispatcher.pre_parse_args(sys.argv[1:])
app_config.set_secure_mode(global_args["secure_mode"])
```

2.5 Set a default command in the application

To allow the application to run a command if none was given in the command line, you need to set a default command in the application when instantiating *craft_cli.dispatcher.Dispatcher*:

```
dispatcher = Dispatcher(..., default_command=MyImportantCommand)
```

This way `craft-cli` will run the specified command if none was given, e.g.:

```
$ my-super-app
```

And even run the specified default command if options are given for that command:

```
$ my-super-app --important-option
```

2.6 Allow temporarily other application to control the terminal

To be able to run another application (in other process) without interfering in the use of the terminal between the main application and the sub-executed one, you need to pause the emitter:

```
with emit.pause():
    subprocess.run(["someapp"])
```

When the emitter is paused the terminal is freed, and the emitter does not have control on what happens in the terminal there until it's resumed, not even for logging purposes.

The normal behaviour is resumed when the context manager exits (even if an exception was raised inside).

2.7 Create unit tests for code that uses Craft CLI Emitter

The library provides two fixtures that simplifies the testing of code using the Emitter when using `pytest`.

One of the fixtures (`init_emitter`) is even set with `autouse=True`, so it will automatically initialize the Emitter and tear it down after each test. This way there is nothing special you need to do in your code when testing it, just use it.

The other fixture (`emitter`) is very useful to test code interaction with Emitter. It provides an internal recording emitter that has several methods which help to test its usage.

The following example shows a simple usage, please refer to `craft_cli.pytest_plugin.RecordingEmitter` for more information about the provided functionality:

```
def test_super_function(emitter):
    """Check the super function."""
    result = super_function(42)
    assert result == "Secret of life, etc."
    emitter.assert_trace("Function properly called with magic number.")
```

2.8 Have a hidden option in a command

To have a command with an option that should not be shown in the help messages, effectively hidden from final users (e.g. because it's experimental), just use a special value in the option's *help*:

```
def fill_parser(self, parser):
    ...
    parser.add_argument("--experimental-behaviour", help=craft_cli.HIDDEN)
```

CHAPTER

THREE

EXPLANATIONS

3.1 About the appropriate mode to initiate *emit*

The first mandatory parameter of the `emit` object is `mode`, which controls the initial verboseness level of the system.

As the user can change the level later using global arguments when executing the application (this is the application default level), it's recommended to use `EmitterMode.BRIEF`, unless the application needs to honor any external configuration or indication (e.g. a `DEBUG` environment variable).

The values for `mode` are the following attributes of the `EmitterMode` enumerator:

- `EmitterMode.QUIET`: to present only error messages, if they happen
- `EmitterMode.BRIEF`: error and info messages, with nice progress indications
- `EmitterMode.VERBOSE`: for more verbose outputs, showing extra information to the user
- `EmitterMode.DEBUG`: aimed to provide useful information to the application developers; this includes timestamps on each line
- `EmitterMode.TRACE`: to also expose system-generated information (in general too overwhelming for debugging purposes but sometimes needed for particular analysis)

3.2 How Craft CLI manage the application logs

Unless overridden when `emit` is initiated (see *how to do that*), the application logs will be managed by the Craft CLI library, according to the following rules:

- one log file is always produced for each application run (only exposed to the user if the application ends in error or a verbose run was requested, for example by `--verbose`), naming the files with a timestamp so they are unique
- log files are located in a directory with the application name under the user's log directory
- only 5 files are kept, when reaching this limit the older file will be removed when creating the one for current run

3.3 Global and command specific arguments

One of the functionalities that the Dispatcher provides is global arguments handling: options that will be recognized and used no matter the position in the command line because they are not specific to any command, but global to all commands and the application itself.

For example, all these application executions are equivalent:

> <app> –verbose <command> <command-parameter> <app> <command> –verbose <command-parameter> <app> <command> <command-parameter> –verbose

The Dispatcher automatically provides the following global arguments, but more can be specified through the *extra_global_args* option (see *how to do that*):

- `-h` / `--help`: provides a help text for the application or command
- `-q` / `--quiet`: sets the `emit` output level to QUIET
- `-v` / `--verbose`: sets the `emit` output level to VERBOSE
- `--verbosity=LEVEL`: sets the `emit` output level to the specified level (allowed are `quiet`, `brief`, `verbose`, `debug` and `trace`).

Each command can also specify its own arguments parsing rules using the `fill_parser` method, which receives an ArgumentParser with all its features for parsing a command line argument. The parsing result will be passed to the command on execution, as the `parsed_args` parameter of the `run` method.

3.4 Group of commands

The Dispatcher's *command_groups* parameter is just a list *CommandGroup* objects, each of one grouping different commands for the different types of functionalities that may offer the application. See its reference here, but its use is quite straightforward. E.g.:

```
CommandGroup("Basic", [LoginCommand, LogoutCommand])
```

A list of these command groups is what is passed to the `Dispatcher` to run them as part of the application.

This grouping is uniquely for building the help exposed to the user, which improves the UX of the application.

When requesting the full application help, all commands will be grouped and presented in the order declared in each `CommandGroup` and in the list given to the `Dispatcher`, and when requesting help for one command, other commands from the same group are suggested to the user as related to the requested one.

3.5 What are hidden and common commands?

When preparing the automatic help messages Craft CLI will consider if a message is common or hidden.

Common commands are those that surely the users will use more frequently and to be learned first, and Craft CLI will list and describe shortly after the summary in the full help.

Hidden commands, on the other hand, will not appear at all in the help messages (but will just work if used), which is useful for deprecated commands (as they will disappear in a near future they should not be advertised) or aliases (multiple commands with different names but same functionality).

3.6 Presenting messages to the user

The main interface for the application to emit messages is the `emit` object. It handles everything that goes to screen and to the log file, even interfacing with the formal logging infrastructure to get messages from it.

It's a singleton, just import it wherever it needs to be used:

```
from craft_cli import emit
```

Before using it, though, it must be initiated. For example:

```
emit.init(EmitterMode.BRIEF, "example-app", "Starting example app v1.")
```

After bootstrapping the library as shown before, and importing `emit` wherever is needed, all its usage is just sending information to the user. The following sections describe the different ways of doing that.

3.6.1 Regular messages

The `message` method is for the final output of the running command.

```
def message(self, text: str) -> None:
```

E.g.:

```
emit.message("The meaning of life is 42.")
```

3.6.2 Progress messages

The `progress` method is to present all the messages that provide information on what the application is currently doing.

Messages shown this way are ephemeral in `QUIET` or `BRIEF` modes (overwritten by the next line) and will be truncated to the terminal's width in that case.

If a progress message is important enough that it should not be overwritten by the next ones, use `permanent=True`.

```
def progress(self, text: str, permanent: bool = False) -> None:
```

E.g.:

```
emit.progress("Assembling stuff...")
```

3.6.3 Progress bar

The `progress_bar` method is to be used in a potentially long-running single step of a command (e.g. a download or provisioning step).

It receives a *text* that should reflect the operation that is about to start, a `total` that will be the number to reach when the operation is completed, and optionally a *delta=False* to indicate that calls to `.advance` method should pass the total so far (by default is True, which implies that calls to `.advance` indicates the delta in the operation progress). Returns a context manager with the `.advance` method to call on each progress.

```
def progress_bar(self, text: str, total: Union[int, float], delta: bool = True) -> _
↪Progresser:
```

E.g.:

```
hasher = hashlib.sha256()
with emit.progress_bar("Hashing the file...", filepath.stat().st_size) as progress:
    with filepath.open("rb") as fh:
        while True:
            data = fh.read(65536)
            hasher.update(data)
            progress.advance(len(data))
            if not data:
                break
```

3.6.4 Verbose messages

Verbose messages are useful to provide more information to the user that shouldn't be exposed when in brief mode for clarity and simplicity.

```
def verbose(self, text: str) -> None:
```

E.g.:

```
emit.verbose("Deleted the temporary file.")
```

3.6.5 Debug messages

The `debug` method is to record everything that the user may not want to normally see but useful for the app developers to understand why things are failing or performing forensics on the produced logs.

```
def debug(self, text: str) -> None:
```

E.g.:

```
emit.debug(f"Hash calculated correctly: {hash_result}")
```

3.6.6 Trace messages

The `trace` method is a way to expose system-generated information, about the general process or particular information, which in general would be too overwhelming for debugging purposes but sometimes needed for particular analysis.

It only produces information to the screen and into the logs if the Emitters is set to TRACE mode.

```
def trace(self, text: str) -> None:
```

E.g.:

```
emit.trace(f"Headers of the server response: {response.headers}")
```

3.6.7 Get messages from subprocesses

The `open_stream` returns a context manager that can be used to get the standard output and/or error from the executed subprocess.

This way all the outputs of the subprocess will be captured by `craft-cli` and shown or not to the screen (according to verbosity setup) and always logged.

```
def open_stream(self, text: str) -> _StreamContextManager:
```

E.g.:

```
with emit.open_stream("Running ls") as stream:
    subprocess.run(["ls", "-l"], stdout=stream, stderr=stream)
```

3.6.8 Emitter modes and startup

The `emit` singleton object is first configured with an explicit call `init()`:

E.g.:

```
emit.init(
    EmitterMode.BRIEF,
    "craft",
    f"Starting craft version {__version__}",
    log_filepath=logpath,
)
```

It is only after this point that `emit` can be used for printing. Note that the mode is typically initialized to `EmitterMode.BRIEF`. The user can control the emitter mode through global arguments. The `Dispatcher`, as mentioned earlier, handles global arguments (including help). However, the `Dispatcher` only applies emitter mode changes during `pre_parse_args()` when parsing the global arguments (e.g. `--verbosity=trace`) later on in the code.

E.g.:

```
dispatcher.pre_parse_args(sys.argv[1:])
```

The implication of the two step process above is that between `init()` and `pre_parse_args()` tracing type messages will be dropped. If you wish to support configurable message verbosity levels during early initialisation, only do that after the dispatcher's `pre_parse_args()`.

Proposed emitter and dispatcher startup:

```
emit.init(...)
dispatcher = Dispatcher(...)
global_args = dispatcher.pre_parse_args(sys.argv[1:])
dispatcher.load_command(global_args)

<early initialisation with support for verbosity levels>

dispatcher.run()
```

3.6.9 How to easily try different message types

There is a collection of examples in the project, in the `examples.py` file. Some examples are very simple, exercising only one message type, but others use different combinations so it's easy to explore more complex behaviours.

To run them using the library, a virtual environment needs to be setup:

```
python3 -m venv env
env/bin/pip install -e .[dev]
source env/bin/activate
```

After that, is just a matter of running the file specifying which example to use:

```
./examples.py 18
```

We encourage you to adapt/improve/hack the examples in the file to play with different combinations of message types to learn and "feel" how the output would be in the different cases.

3.6.10 Understanding which/how messages are shown/logged

This is how texts are exposed to the screen for the different situations according to the selected verbosity level by the user running the application.

The last column of the table though is not about the screen: it indicates if the information will be present in the log created automatically by Craft CLI.

	QUIET	BRIEF	VERBOSE	DEBUG	TRACE	also to log-file
`.message(...)`	–	stdout permanent plain	stdout permanent plain	stdout permanent plain	stdout permanent plain	yes
`.progress(...)`	–	stderr transient (*) plain	stderr permanent plain	stderr permanent timestamp	stderr permanent timestamp	yes
`.progress(..., permanent=True)`	–	stderr permanent plain	stderr permanent plain	stderr permanent timestamp	stderr permanent timestamp	yes
`.progress_bar(...)`	–	stderr transient (*) plain	stderr permanent plain	stderr permanent timestamp	stderr permanent timestamp	first line only, without progress
`.open_stream(...)`	–	–	stderr permanent plain	stderr permanent timestamp	stderr permanent timestamp	yes
`.verbose(...)`	–	–	stderr permanent plain	stderr permanent timestamp	stderr permanent timestamp	yes
`.debug(...)`	–	–	–	stderr permanent timestamp	stderr permanent timestamp	yes
`.trace(...)`	–	–	–	–	stderr permanent timestamp	only when level=trace
captured logs (level > `logging.DEBUG`)	–	–	stderr permanent plain	stderr permanent timestamp	stderr permanent timestamp	yes
captured	–	–	–	stderr	stderr	yes

(*) when redirected to a file it doesn't make sense to have "transient" messages, so 'progress' messages will always end in a newline, and 'progress_bar' will just send its message line but without the progress indication.

When the application ends in error it should call the `emit.error()` method passing a `CraftError` instance. According to the verbosity level some information will be exposed or not. The following table details what happens in each case: which `CraftError` attribute is exposed and how the information is shown (similar to the other table above):

	QUIET	BRIEF	VERBOSE	DEBUG	TRACE	also to log-file
the error message	yes	yes	yes	yes	yes	yes
full tracebacks	no	no	no	yes	yes	yes
`.details`	no	no	no	yes	yes	yes
`.resolution`	yes	yes	yes	yes	yes	yes
`.docs_url`	yes	yes	yes	yes	yes	yes
how is it shown	stderr permanent plain	stderr permanent plain	stderr permanent plain	stderr permanent timestamp	stderr permanent timestamp	yes

CHAPTER

FOUR

CRAFT_CLI PACKAGE

4.1 Submodules

4.1.1 craft_cli.dispatcher module

Argument processing and command dispatching functionality.

class craft_cli.dispatcher.**BaseCommand**(*config*)

Bases: object

Base class to build application commands.

Subclass this to create a new command; the subclass must define the following attributes:

- name: the identifier in the command line
- help_msg: a one line help for user documentation
- overview: a longer multi-line text with the whole command description

Also it may override the following one to change its default:

- common: if it's a common/starter command, which are prioritized in the help (default to False)
- hidden: do not show in help texts, useful for aliases or deprecated commands (default to False)

It also must/can override some methods for the proper command behaviour (see each method's docstring).

The subclass must be declared in the corresponding section of command groups indicated to the Dispatcher.

Parameters **config** (Optional[Dict[str, Any]]) –

common = False

fill_parser(*parser*)

Specify command's specific parameters.

Each command parameters are independent of other commands, but note there are some global ones (see *main.Dispatcher._build_argument_parser*).

If this method is not overridden, the command will not have any parameters.

Parameters **parser** (_CustomArgumentParser) –

Return type None

help_msg: Optional[str] = None

hidden = False

name: Optional[str] = None

overview: Optional[str] = None

run(*parsed_args*)

Execute command's actual functionality.

It must be overridden by the command implementation.

This will receive parsed arguments that were defined in :meth:.fill_parser.

It should return None or the desired process' return code.

Parameters **parsed_args** (Namespace) –

Return type Optional[int]

class craft_cli.dispatcher.**CommandGroup**(*name*, *commands*)

Bases: tuple

Definition of a command group.

A list of these is what is passed to the Dispatcher to run commands as part of the application.

Parameters

- **name** – identifier of the command group (to be used in help texts).
- **commands** – a list of the commands in this group.

commands

name

class craft_cli.dispatcher.**Dispatcher**(*appname*, *commands_groups*, *, *summary=''*, *extra_global_args=None*, *default_command=None*)

Bases: object

Set up infrastructure and let the needed command run.

♪"Leeeeeet, the command ruuun"♪ https://www.youtube.com/watch?v=cv-0mmVnxPA

Parameters

- **appname** (str) – the name of the application
- **commands_groups** (List[*CommandGroup*]) – a list of command groups available to the user
- **summary** (str) – the summary of the application (for help texts)
- **extra_global_args** (Optional[List[*GlobalArgument*]]) – other automatic global arguments than the ones provided automatically
- **default_command** (Optional[Type[*BaseCommand*]]) – the command to run if none was specified in the command line

load_command(*app_config*)

Load a command.

Parameters **app_config** (Any) –

Return type *BaseCommand*

pre_parse_args(*sysargs*)

Pre-parse sys args.

Several steps:

- extract the global options and detects the possible command and its args
- validate global options and apply them

- validate that command is correct (NOT loading and parsing its arguments)

Parameters **sysargs** (List[str]) –

run()
Really run the command.

Return type Optional[int]

class craft_cli.dispatcher.**GlobalArgument**(*name*, *type*, *short_option*, *long_option*, *help_message*)
Bases: tuple

Definition of a global argument to be handled by the Dispatcher.

Parameters

- **name** – identifier of the argument (the reference in the dictionary returned by Dispatcher.pre_parse_args method)
- **type** – the argument type: flag for arguments that are set to True if specified (False by default), or option if a value is needed after it.
- **short_option** – the short form of the argument (a dash with a letter, e.g. -s); it can be None if the option does not have a short form.
- **long_option** – the long form of the argument (two dashes and a name, e.g. --secure).
- **help_message** – the one-line text that describes the argument, for building the help texts.

help_message

long_option

name

short_option

type

4.1.2 craft_cli.errors module

Error classes.

exception craft_cli.errors.**CraftError**(*message*, *, *details=None*, *resolution=None*, *docs_url=None*, *logpath_report=True*, *reportable=True*, *retcode=1*)
Bases: Exception

Signal a program error with a lot of information to report.

Variables

- **message** – the main message to the user, to be shown as first line (and probably only that, according to the different modes); note that in some cases the log location will be attached to this message.
- **details** – the full error details received from a third party which originated the error situation
- **resolution** – an extra line indicating to the user how the error may be fixed or avoided (to be shown together with 'message')
- **docs_url** – an URL to point the user to documentation (to be shown together with 'message')

- **logpath_report** – if the location of the log filepath should be presented in the screen as the final message
- **reportable** – if an error report should be sent to some error-handling backend (like Sentry)
- **retcode** – the code to return when the application finishes

Parameters

- **message** (str) –
- **details** (Optional[str]) –
- **resolution** (Optional[str]) –
- **docs_url** (Optional[str]) –
- **logpath_report** (bool) –
- **reportable** (bool) –
- **retcode** (int) –

4.1.3 craft_cli.helptexts module

Provide all help texts.

class craft_cli.helptexts.**HelpBuilder**(*appname*, *general_summary*, *command_groups*)

Bases: object

Produce the different help texts.

Parameters

- **appname** (str) –
- **general_summary** (str) –
- **command_groups** (List[ForwardRef]) –

get_command_help(*command*, *arguments*, *output_format*)

Produce the text for each command's help in any output format.

- command: the instantiated command for which help is prepared
- **arguments: all command options and parameters, with the (name, description) structure;** note that any argument with description being *HIDDEN* will be ignored
- output_format: the selected output format

The help text structure depends of the output format.

Parameters

- **command** (BaseCommand) –
- **arguments** (List[Tuple[str, str]]) –
- **output_format** (*OutputFormat*) –

Return type str

get_detailed_help(*global_options*)

Produce the text for the detailed help.

- global_options: options defined at application level (not in the commands), with the (options, description) structure

The help text has the following structure:

- usage
- summary
- global options
- all commands shown with description, grouped
- more help

Parameters **global_options** (List[Tuple[str, str]]) –

Return type str

get_full_help(*global_options*)

Produce the text for the default help.

- global_options: options defined at application level (not in the commands), with the (options, description) structure

The help text has the following structure:

- usage
- summary
- common commands listed and described shortly
- all commands grouped, just listed
- more help

Parameters **global_options** (List[Tuple[str, str]]) –

Return type str

get_usage_message(*error_message*, *command=''*)

Build a usage and error message.

The command is the extra string used after the application name to build the full command that will be shown in the usage message; for example, having an application name of "someapp": - if command is "" it will be shown "Try 'appname -h' for help". - if command is "version" it will be shown "Try 'appname version -h' for help"

The error message is the specific problem in the given parameters.

Parameters

- **error_message** (str) –
- **command** (str) –

Return type str

class craft_cli.helptexts.**OutputFormat**(*value*)

Bases: enum.Enum

An enumeration.

markdown = 2

plain = 1

craft_cli.helptexts.**process_overview_for_markdown**(*text*)
Process a regular overview to be rendered with markdown.

In detail:

- Join all lines for the same paragraph (as wrapping is responsibility of the renderer)
- Dedent and wrap with triple-backtick all indented blocks

Paragraphs are separated by empty lines

Parameters **text** (str) –

Return type str

4.1.4 craft_cli.messages module

Support for all messages, ok or after errors, to screen and log file.

class craft_cli.messages.**EmitterMode**(*value*)
Bases: enum.Enum

An enumeration.

BRIEF = 2

DEBUG = 4

QUIET = 1

TRACE = 5

VERBOSE = 3

4.1.5 craft_cli.pytest_plugin module

4.2 Module contents

Interact with Canonical services such as Charmhub and the Snap Store.

exception craft_cli.**ArgumentParsingError**
Bases: Exception

Exception used when an argument parsing error is found.

class craft_cli.**BaseCommand**(*config*)
Bases: object

Base class to build application commands.

Subclass this to create a new command; the subclass must define the following attributes:

- name: the identifier in the command line
- help_msg: a one line help for user documentation
- overview: a longer multi-line text with the whole command description

Also it may override the following one to change its default:

- common: if it's a common/starter command, which are prioritized in the help (default to False)
- hidden: do not show in help texts, useful for aliases or deprecated commands (default to False)

It also must/can override some methods for the proper command behaviour (see each method's docstring).

The subclass must be declared in the corresponding section of command groups indicated to the Dispatcher.

Parameters **config** (Optional[Dict[str, Any]]) –

common = False

fill_parser(*parser*)

Specify command's specific parameters.

Each command parameters are independent of other commands, but note there are some global ones (see *main.Dispatcher._build_argument_parser*).

If this method is not overridden, the command will not have any parameters.

Parameters **parser** (_CustomArgumentParser) –

Return type None

help_msg: Optional[str] = None

hidden = False

name: Optional[str] = None

overview: Optional[str] = None

run(*parsed_args*)

Execute command's actual functionality.

It must be overridden by the command implementation.

This will receive parsed arguments that were defined in :meth:.fill_parser.

It should return None or the desired process' return code.

Parameters **parsed_args** (Namespace) –

Return type Optional[int]

class craft_cli.**CommandGroup**(*name*, *commands*)

Bases: tuple

Definition of a command group.

A list of these is what is passed to the Dispatcher to run commands as part of the application.

Parameters

- **name** – identifier of the command group (to be used in help texts).
- **commands** – a list of the commands in this group.

commands

name

exception craft_cli.**CraftError**(*message*, ***, *details=None*, *resolution=None*, *docs_url=None*, *logpath_report=True*, *reportable=True*, *retcode=1*)

Bases: Exception

Signal a program error with a lot of information to report.

Variables

- **message** – the main message to the user, to be shown as first line (and probably only that, according to the different modes); note that in some cases the log location will be attached to this message.

- **details** – the full error details received from a third party which originated the error situation
- **resolution** – an extra line indicating to the user how the error may be fixed or avoided (to be shown together with 'message')
- **docs_url** – an URL to point the user to documentation (to be shown together with 'message')
- **logpath_report** – if the location of the log filepath should be presented in the screen as the final message
- **reportable** – if an error report should be sent to some error-handling backend (like Sentry)
- **retcode** – the code to return when the application finishes

Parameters

- **message** (str) –
- **details** (Optional[str]) –
- **resolution** (Optional[str]) –
- **docs_url** (Optional[str]) –
- **logpath_report** (bool) –
- **reportable** (bool) –
- **retcode** (int) –

class craft_cli.**Dispatcher**(*appname*, *commands_groups*, *, *summary=''*, *extra_global_args=None*, *default_command=None*)

Bases: object

Set up infrastructure and let the needed command run.

♪"Leeeeeet, the command ruuun"♪ https://www.youtube.com/watch?v=cv-0mmVnxPA

Parameters

- **appname** (str) – the name of the application
- **commands_groups** (List[*CommandGroup*]) – a list of command groups available to the user
- **summary** (str) – the summary of the application (for help texts)
- **extra_global_args** (Optional[List[*GlobalArgument*]]) – other automatic global arguments than the ones provided automatically
- **default_command** (Optional[Type[*BaseCommand*]]) – the command to run if none was specified in the command line

load_command(*app_config*)

Load a command.

Parameters **app_config** (Any) –

Return type *BaseCommand*

pre_parse_args(*sysargs*)

Pre-parse sys args.

Several steps:

- extract the global options and detects the possible command and its args

- validate global options and apply them
- validate that command is correct (NOT loading and parsing its arguments)

Parameters **sysargs** (List[str]) –

run()
Really run the command.

Return type Optional[int]

class craft_cli.**EmitterMode**(*value*)
Bases: enum.Enum

An enumeration.

BRIEF = 2

DEBUG = 4

QUIET = 1

TRACE = 5

VERBOSE = 3

class craft_cli.**GlobalArgument**(*name*, *type*, *short_option*, *long_option*, *help_message*)
Bases: tuple

Definition of a global argument to be handled by the Dispatcher.

Parameters

- **name** – identifier of the argument (the reference in the dictionary returned by Dispatcher.pre_parse_args method)
- **type** – the argument type: flag for arguments that are set to True if specified (False by default), or option if a value is needed after it.
- **short_option** – the short form of the argument (a dash with a letter, e.g. -s); it can be None if the option does not have a short form.
- **long_option** – the long form of the argument (two dashes and a name, e.g. --secure).
- **help_message** – the one-line text that describes the argument, for building the help texts.

help_message

long_option

name

short_option

type

exception craft_cli.**ProvideHelpException**
Bases: Exception

Exception used to provide help to the user.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

N

O

P

Q

R

S

T

V