# Riotkit-Do: Universal automation (DevOps) tool for elastic, shareable tasks and pipelines

*Release 0.0.35*

**Riotkit**

**Oct 19, 2021**

# CONTENTS:

RKD is a stable, open-source, multi-purpose automation tool which balance flexibility with simplicity. The primary language is Python and YAML syntax.

RiotKit-Do can be compared to **Gradle** and to **GNU Make**, by allowing both Python and Makefile-like YAML syntax.

**What can be achieved with RKD?**

- Simplify the scripts
- Put your Python and Bash scripts inside a YAML file (like in GNU Makefile)
- Do not reinvent the wheel (argument parsing, logs, error handling for example)
- Share the code across projects and organizations, use native Python Packaging to share tasks (like in Gradle)
- Natively integrate scripts with .env files
- Automatically generate documentation for your scripts
- Maintain your scripts in a good standard

**RKD can be used on PRODUCTION, for development, for testing, to replace some of Bash scripts inside docker containers, and for many more, where Makefile was used.**

YAML

Simplified Python

Classic Python

```
version: org.riotkit.rkd/yaml/v1
tasks:
    :hello1:
        extends: rkd.core.standardlib.syntax.PythonSyntaxTask
        arguments:
            "--name":
                required: True
                help: "Allows to specify a name"
        description: Prints your name
        execute: |
            self.io().info_msg(f'Hello {ctx.get_arg("--name")}, I\'m talking from YAML,␣
↪and you?')
            return True
```

```
from argparse import ArgumentParser
from rkd.core.api.contract import ExecutionContext
from rkd.core.api.decorators import extends
from rkd.core.api.syntax import ExtendedTaskDeclaration
from rkd.core.standardlib.syntax import PythonSyntaxTask


@extends(PythonSyntaxTask)
def hello_task():
    """
    Prints your name
    """

    def configure_argparse(task: PythonSyntaxTask, parser: ArgumentParser):
        parser.add_argument('--name', required=True, help='Allows to specify a name')
```

(continues on next page)

(continued from previous page)

```
    def execute(task: PythonSyntaxTask, ctx: ExecutionContext):
        task.io().info_msg(f'Hello {ctx.get_arg("--name")}, I\'m talking in Python, and␣
↪you?')
        return True

    return [configure_argparse, execute]


IMPORTS = [
    ExtendedTaskDeclaration(hello_task, name=':hello2')
]
```

```
from argparse import ArgumentParser
from rkd.core.api.contract import TaskInterface, ExecutionContext
from rkd.core.api.syntax import TaskDeclaration


class HelloFromPythonTask(TaskInterface):
    """
    Prints your name
    """

    def get_name(self) -> str:
        return ':hello3'

    def get_group_name(self) -> str:
        return ''

    def configure_argparse(self, parser: ArgumentParser):
        parser.add_argument('--name', required=True, help='Allows to specify a name')

    def execute(self, ctx: ExecutionContext) -> bool:
        self.io().info_msg(f'Hello {ctx.get_arg("--name")}, I\'m talking classic Python␣
↪syntax, and you?')
        return True


IMPORTS = [
    TaskDeclaration(HelloFromPythonTask()),
]
```

CHAPTER

ONE

# EXAMPLE USE CASES

- Docker based production environment with multiple configuration files, procedures (see: Harbor project)
- Database administrator workspace (importing dumps, creating new user accounts, plugging/unplugging databases)
- Development environment (executing migrations, importing test database, splitting tests and running parallel)
- On CI (prepare project to run on eg. Jenkins or Gitlab CI) - RKD is reproducible on local computer which makes inspection easier
- Application cluster management, deploying applications, adding users, setting permissons
- Automate things like certificate regeneration on production server, RKD can generate any application configs using JINJA2
- Installers (RKD has built-in commands for replacing lines in files, modifying .env files, asking user questions and validating answers)

# CHAPTER TWO

# INSTALL RKD

RiotKit-Do is delivered as a Python package that can be installed system-wide or in a virtual environment. The virtual environment installation is similar in concept to the Gradle wrapper (gradlew)

```
# download a wrapper that will automatically setup virtual environment and install RKD
# do not forget to commit wrapper to the GIT repository
wget https://github.com/riotkit-org/riotkit-do/blob/master/src/core/rkd/core/misc/
↪initial-structure/rkdw.py -O rkdw
chmod +x rkdw
./rkdw
```

# CHAPTER THREE

# GETTING STARTED IN FRESHLY CREATED STRUCTURE

The “Quick start” section ends up with a **.rkd** directory, a requirements.txt and ./rkdw

1. Call RKD using a wrapper in project directory **./rkdw**
2. Each time you install anything from **pip** in your project - add it to requirements.txt (or use `pipenv install`), additional RKD tasks can be installed from PIP
3. In **.rkd/makefile.yaml** add your tasks, pipelines and imports

CHAPTER

FOUR

# CREATE YOUR FIRST TASK WITH GETTING STARTED

CHAPTER

FIVE

# CHECK HOW TO USE COMMANDLINE TO RUN TASKS IN RKD WITH COMMANDLINE USAGE

CHAPTER

SIX

# SEE HOW TO IMPORT EXISTING TASKS TO YOUR MAKEFILE WITH IMPORTING TASKS PAGE

CHAPTER

SEVEN

# KEEP LEARNING

- YAML syntax is described also in *Tasks development - more examples* section
- Writing Python code in makefile.yaml requires to lookup *Tasks API*
- Learn how to import installed tasks via pip - *Importing tasks*
- You can also write tasks code in pure Python and redistribute those tasks via Python's PIP - see *Tasks development - more examples*
- With RKD you can create interactive installers - check the *Creating installer wizards with RKD* section

## 7.1 Getting started

RKD is project-focused, which means it is designed to provide an automation in scope of a project. That's why we call it a DevOps tool. Project is a GIT repository with a set of tasks to manage particular thing. The thing could be a development project, web application, a production database server, TLS certifications issuing, cluster configuration management, almost everything.

Scope of a project does not mean only GIT repository, first of all it means a runtime environment placed absolutely inside your project directory. Python's Virtual Environments are extensively used to provide a separated per-project environment with specific versions compatible with the currently used project.

To simplify usage of a project there is a *./rkdw* wrapper that transparently creates a virtual environment and installs RKD on-the-fly. It was inspired by the Gradle Project's *./gradlew* wrapper.

### 7.1.1 Where to place files

`.rkd` directory must always exists in your project. Inside `.rkd` directory you should place your makefile.yaml that will contain all of the required tasks.

Just like in UNIX/Linux, and just like in Python - there is an environment variable `RKD_PATH` that allows to define multiple paths to `.rkd` directories placed in other places - for example outside of your project. This gives a flexibility and possibility to build system-wide tools installable via Python's PIP.

### 7.1.2 Tutorial

Install RKD inside a project workspace

```
wget https://github.com/riotkit-org/riotkit-do/blob/master/src/core/rkd/core/misc/
↪initial-structure/rkdw.py -O rkdw && chmod +x rkdw
./rkdw
```

As you learned already - automation files should be placed inside `.rkd` directory in your project, let's create that directory and create an example Makefile.

```
mkdir -p .rkd
nano .rkd/makefile.yaml
```

Now put example content into your makefile.yaml

```yaml
version: org.riotkit.rkd/yaml/v1
environment:
    PYTHONPATH: "/project"
tasks:
    :hello:
        description: Prints variables
        environment:
            SOME_VAR: "HELLO"
        steps: |
            echo "SOME_VAR is ${SOME_VAR}, PYTHONPATH is ${PYTHONPATH}"
```

Save the file and run.

```
./rkdw :tasks  # see if your task is there
./rkdw :hello  # execute your task

# or combined :-)
./rkdw :tasks :hello

# or do it multiple times!
./rkdw :hello :hello :hello
```

That's it, now you are ready to proceed with the documentation to start writing your dreamed automation.

### 7.1.3 Environment variables

RKD natively reads .env (called also "dot-env files") at startup. You can define default environment values in .env, or in other .env-some-name files that can be included in `env_files` section of the YAML.

**Scope of environment variables**

`env_files` and `environment` blocks can be defined globally, which will end in including that fact in each task, second possibility is to define those blocks per task. Having both global and per-task block merges those values together and makes per-task more important.

**Example**

```yaml
version: org.riotkit.rkd/yaml/v1
environment:
```

(continues on next page)

(continued from previous page)

```
    PYTHONPATH: "/project"
tasks:
    :hello:
        description: Prints variables
        environment:
            SOME_VAR: "HELLO"
        steps: |
            echo "SOME_VAR is ${SOME_VAR}, PYTHONPATH is ${PYTHONPATH}"
```

### 7.1.4 Arguments parsing

Arguments parsing is a strong side of RKD. Each task has it's own argument parsing, it's own generated –help command. Python's argparse library is used, so Python programmers should feel like in home.

**Example**

```
version: org.riotkit.rkd/yaml/v1
environment:
    PYTHONPATH: "/project"
tasks:
    :hello:
        description: Prints your name
        arguments:
            "--name":
                required: true
                #option: store_true # for booleans/flags
                #default: "Unknown" # for default values
        steps: |
            echo "Hello ${ARG_NAME}"
```

```
rkd :hello --name Peter
```

### 7.1.5 Defining tasks in Python code

Defining tasks in Python gives wider possibilities - to access Python's libraries, better handle errors, write less tricky code. RKD has a similar concept to hashbangs in UNIX/Linux.

There are two supported hashbangs + no hashbang:

- #!python
- #!bash
- (just none there)

What can I do in such Python code? Everything! Import, print messages, execute shell commands, everything.

**Example**

```
version: org.riotkit.rkd/yaml/v1
environment:
    PYTHONPATH: "/project"
tasks:
```

(continues on next page)

(continued from previous page)

```
    :hello:
        description: Prints your name
        arguments:
            "--name":
                required: true
                #option: store_true # for booleans/flags
                #default: "Unknown" # for default values
        steps: |
            #!python
            print('Hello %s' % ctx.get_arg('--name'))
```

**Special variables**

- *this* - instance of current TaskInterface implementation
- *ctx* - instance of ExecutionContext

Please check *Tasks API* for those classes reference.

### 7.1.6 YAML syntax reference

Let's at the beginning start from analyzing an example.

```
version: org.riotkit.rkd/yaml/v1

# optional: Import tasks from Python packages
# This gives a possibility to publish tasks and share across projects, teams,␣
↪organizations
imports:
    - rkt_utils.db.WaitForDatabaseTask

# optional environment section would append those variables to all tasks
# of course the tasks can overwrite those values in per-task syntax
environment:
    PYTHONPATH: "/project/src"

# optional env files loaded there would append loaded variables to all tasks
# of course the tasks can overwrite those values in per-task syntax
#env_files:
#    - .some-dotenv-file

tasks:
    :check-is-using-linux:
        extends: rkd.core.standardlib.syntax.MultiStepLanguageAgnosticTask   # this a␣
↪default value
        description: Are you using Linux?
        # use sudo to become a other user, optional
        become: root
        steps:
            # steps can be defined as single step, or multiple steps
            # each step can be in a different language
            # each step can be a multiline string
            - "[[ $(uname -s) == \"Linux\" ]] && echo \"You are using Linux, cool\""
```

(continues on next page)

(continued from previous page)

```
            - echo "step 2"
            - |
                #!python
                print('Step 3')

    :hello:
        description: Say hello
        arguments:
            "--name":
                help: "Your name"
                required: true
                #default: "Peter"
                #option: "store_true" # for booleans
        steps: |
            echo "Hello ${ARG_NAME}"

            if [[ $(uname -s) == "Linux" ]]; then
                echo "You are a Linux user"
            fi
```

**extends** - Base Task that is going to be extended. Default value is `rkd.core.standardlib.syntax.MultiStepLanguageAgnosticTask` which allows to execute multiple steps in different languages

**imports** - Imports external tasks installed via Python' PIP. That's the way to easily share code across projects

**environment** - Can define default values for environment variables. Environment section can be defined for all tasks, or per task

**env_files** - Includes .env files, can be used also per task

**tasks** - List of available tasks, each task has a name, descripton, list of steps (or a single step), arguments

**Running the example:**

1. Create a .rkd directory
2. Create .rkd/makefile.yaml file
3. Paste/rewrite the example into the .rkd/makefile.yaml
4. Run `rkd :tasks` from the directory where the .rkd directory is placed
5. Run defined tasks `rkd :hello :check-is-using-linux`

**Example projects using Makefile YAML syntax:**

- Taiga docker image
- Taiga Events docker image
- K8S Workspace

Check *ADVANCED usage* page for description of all environment variables, mechanisms, good practices and more

## 7.2 Commandline usage

RKD command-line usage is highly inspired by GNU Make and Gradle, but it has its own extended possibilities to make your scripts smaller and more readable.

- Tasks are prefixed always with ":".
- Each task can handle it's own arguments (unique in RKD)
- "@" allows to propagate arguments to next tasks (unique in RKD)

### 7.2.1 Tasks arguments usage in shell and in scripts

**Executing multiple tasks in one command:**

```
./rkdw :task1 :task2
```

**Multiple tasks with different switches:**

```
./rkdw :task1 --hello  :task2 --world --become=root
```

Second task will run as root user, additionally with `--world` parameter.

**Tasks sharing the same switches**

Both tasks will receive switch "–hello"

```
# expands to:
#  :task1 --hello
#  :task2 --hello
./rkdw @ --hello :task1 :task2

# handy, huh?
```

**Advanced usage of shared switches**

Operator "@" can set switches anytime, it can also clear or replace switches in **NEXT TASKS**.

```
# expands to:
#   :task1 --hello
#   :task2 --hello
#   :task3
#   :task4 --world
#   :task5 --world
./rkdw @ --hello :task1 :task2 @ :task3 @ --world :task4 :task5
```

**Written as a pipeline (regular bash syntax)**

It's exactly the same example as above, but written multiline. It's recommended to write multiline commands if they are longer.

```
./rkdw @ --hello \
    :task1 \
    :task2 \
    @
    :task3 \
```

(continues on next page)

(continued from previous page)

```
    @ --world \
    :task4 \
    :task5
```

### Arguments

**Each task has it's own arguments parsing and –help method**

```
# see a list of commandline switches
./rkdw :task1 --help

# increase log level
./rkdw :task1 --log-level debug

# log output to file
./rkdw :task1 --log-to-file=/tmp/task1.log

# change user for task execution time
./rkdw :task1 --become=root
```

**Global commandline switches**

To apply default, global error level use a switch before all tasks.

```
./rkdw --log-level=debug :task1 :task2

# alternatively (changes log level on earlier stage than argument parsing)
RKD_SYS_LOG_LEVEL=debug ./rkdw :task1 :task2

# or like shown in 'Tasks arguments usage in shell and in scripts' - any commandline␣
↪switches
# can be propagated, including RKD internal switches
./rkdw @ --log-level=debug --task-workdir=/tmp :task1 :task2
```

## 7.2.2 Advanced: Blocks for error handling

Blocks allow to retry single failed task, or a group of tasks, execute a failure or rescue task.

**Tip:** Blocks cannot be nested.

**Retry a task - @retry**

Retry task until it will return success, up to defined retries. If there are multiple tasks, then a single task is repeated, not a whole block.

```
./rkdw '{@retry 3}' :unstable-task '{/@}'
```

**Retry a block (set of tasks) - @retry-block**

Works very similar to @retry, but in case, when at least one task fails - all tasks in the block are repeated.

```
./rkdw '{@retry-block 3}' :unstable-task :task2 '{/@}'
```

**Rescue - @rescue**

When a failure happens in any of tasks, then those tasks are interrupted and a rollback task is executed. Whole block status depends on the rollback task status. After a successful rollback execution next tasks from outside of the blocks are normally executed.

```
./rkdw :db:shutdown :db:backup '{@rescue :db:restore}' :db:upgrade '{/@}' :db:start
```

**Error - @error**

When at least one task fails, then a error task is notified and the execution is stopped.

```
./rkdw '{@error :notify "Task failed!"}' :some-task :some-other-task '{/@}'
```

## 7.3 Syntax

RKD is elastic. Different syntax allows to choose between ease of usage and extended possibilities.

### 7.3.1 YAML

- Best choice to start with RKD, perfect for simpler usage
- Gives clear view on what is defined, has obvious structure
- Created tasks are not possible to be shared as part of Python package

```
version: org.riotkit.rkd/yaml/v1
tasks:
    :hello1:
        extends: rkd.core.standardlib.syntax.PythonSyntaxTask
        arguments:
            "--name":
                required: True
                help: "Allows to specify a name"
        description: Prints your name
        execute: |
            self.io().info_msg(f'Hello {ctx.get_arg("--name")}, I\'m talking from YAML,␣
↪and you?')
            return True
```

### 7.3.2 Simplified Python

- Practical replacement for YAML syntax, good choice for more advanced tasks
- Has more flexibility on the structure, tasks and other code can be placed in different files and packages
- Created tasks are not possible to be shared as part of Python package, or at least difficult and should not be

```
from argparse import ArgumentParser
from rkd.core.api.contract import ExecutionContext
from rkd.core.api.decorators import extends
from rkd.core.api.syntax import ExtendedTaskDeclaration
from rkd.core.standardlib.syntax import PythonSyntaxTask


@extends(PythonSyntaxTask)
def hello_task():
    """
    Prints your name
    """

    def configure_argparse(task: PythonSyntaxTask, parser: ArgumentParser):
        parser.add_argument('--name', required=True, help='Allows to specify a name')

    def execute(task: PythonSyntaxTask, ctx: ExecutionContext):
        task.io().info_msg(f'Hello {ctx.get_arg("--name")}, I\'m talking in Python, and␣
↪you?')
        return True

    return [configure_argparse, execute]


IMPORTS = [
    ExtendedTaskDeclaration(hello_task, name=':hello2')
]
```

### 7.3.3 Classic Python

- Provides a full control without any limits on tasks extending
- Has more flexibility on the structure, tasks and other code can be placed in different files and packages
- Best fits for creating shareable tasks using local and remote Python packages

```
from argparse import ArgumentParser
from rkd.core.api.contract import TaskInterface, ExecutionContext
from rkd.core.api.syntax import TaskDeclaration


class HelloFromPythonTask(TaskInterface):
    """
    Prints your name
    """

    def get_name(self) -> str:
        return ':hello3'

    def get_group_name(self) -> str:
        return ''
```

(continues on next page)

(continued from previous page)

```
    def configure_argparse(self, parser: ArgumentParser):
        parser.add_argument('--name', required=True, help='Allows to specify a name')

    def execute(self, ctx: ExecutionContext) -> bool:
        self.io().info_msg(f'Hello {ctx.get_arg("--name")}, I\'m talking classic Python syntax, and you?')
        return True


IMPORTS = [
    TaskDeclaration(HelloFromPythonTask()),
]
```

### 7.3.4 Syntax reference

| Simplified Python | Python Class | YAML | Description |
|---|---|---|---|
| get_steps(task: MultiStepLanguageAgnosticTask) -> List[str]: | get_steps | steps: [“”] | List of steps in any language (only if extending MultiStep LanguageAgnosticTask) |
| stdin() | N/A | input: “” | Standard input text |
| @extends(ClassName) decorator on a main method | ClassName(BaseClass) | extends: package.name.ClassName | Which Base Task should be extended |
| execute(task: BaseClassNameTask, ctx: ExecutionContext): | execute(self, ctx: ExecutionContext) | execute: “” | Python code to execute |
| inner_execute(task: BaseClassNameTask, ctx: ExecutionContext): | inner_execute(self, ctx: ExecutionContext) | inner_execute: “” | Python code to execute inside inner_execute (if implemented by Base Task) |
| compile(task: BaseClassNameTask, event: CompilationLifecycleEvent): | compile(self, event: CompilationLifecycleEvent): | N/A | Python code to execute during Context compilation process |
| configure(task: BaseClassNameTask, event: ConfigurationLifecycleEvent): | configure(self, event: ConfigurationLifecycleEvent): | configure: “” | Python code to execute during Task configuration process |
| get_description() | get_description(self) | description: “” | Task description |
| get_group_name() | get_group_name() | N/A | Group name |
| internal=True in TaskDeclaration | internal=True in TaskDeclaration | internal: False | Is task considered internal? (hidden on :tasks list) |
| become in TaskDeclaration (or commandline switch) | become in TaskDeclaration (or commandline switch) | become: root | Change user for task execution time |
| workdir in TaskDeclaration | workdir in TaskDeclaration | workdir: /some/path | Change working directory for task execution time |
| configure_argparse(task: BaseClassNameTask, parser: ArgumentParser) | configure_argparse(self, parser: ArgumentParser) | arguments: {} | Configure argparse.ArgumentParser object |

## 7.4 Importing tasks

Tasks can be defined as installable Python's packages that you can import in your Makefile

**Please note:**

- To import a group of packages, the package you try to import need to have a defined **imports()** method inside of the package.
- The imported group does not need to import automatically dependent tasks (but it can, it is recommended), you need to read into the docs of specific package if it does so

### 7.4.1 1) Install a package

RKD defines dependencies using Python standards.

Example: Given we want to import tasks from package "rkt_armutils".

```
# Using virtualenv
echo "rkt_armutils==3.0" >> requirements.txt
pip install -r requirements.txt

# Alternatively, using pipenv (recommended)
pipenv install rkt_armutils==3.0
```

**Good practices:**

- Use fixed versions eg. 3.0 or even 3.0.0 and upgrade only intentionally to reduce your work. Automatic updates, especially of major versions could be unpredictable and introduce breaking changes into your project

**How do I check latest version?:**

- Simply install a package eg. `pip install rkt_armutils`, then do a `pip show rkt_armutils` and write the version

to the requirements.txt, or lookup a package first at https://pypi.org/project/rkt_armutils/ (where rkt_armutils is an example package)

### 7.4.2 2) In YAML syntax

Example: Given we want to import task "InjectQEMUBinaryIntoContainerTask", or we want to import whole "rkt_armutils.docker" group

```
imports:
    # Import whole package, if the package defines a group import (method imports())
    - rkt_armutils.docker

    # Or import single task
    - rkt_armutils.docker.InjectQEMUBinaryIntoContainerTask
```

### 7.4.3 2) In Python syntax

Example: Given we want to import task "InjectQEMUBinaryIntoContainerTask", or we want to import whole "rkt_armutils.docker" group

```
from rkd.core.api.syntax import TaskDeclaration
from rkt_armutils.docker import InjectQEMUBinaryIntoContainerTask

# ... (use "+" operator to append, remove "+" if you didn't define any import yet)
IMPORTS += [TaskDeclaration(InjectQEMUBinaryIntoContainerTask)]
```

### 7.4.4 3) Inline syntax

Tasks could be imported also in shell, for quick check, handy scripts, or for embedding inside other applications.

```
# note: Those examples requires "rkt_utils" package from PyPI
RKD_IMPORTS="rkt_utils.docker" rkd :docker:tag
RKD_IMPORTS="rkt_utils.docker:rkt_ciutils.boatci:rkd_python" rkd :tasks

# via commandline switch "--imports"
rkd --imports "rkt_utils.docker:rkt_ciutils.boatci:rkd_python" :tasks
```

*Note: The significant difference between environment variable and commandline switch is that the environment variable will be inherited into subshells of RKD, commandline argument not.*

For more information about this environment variable - check it's documentation page: *RKD_IMPORTS*

### 7.4.5 Ready to go? Check Built-in tasks that you can import in your Makefile

## 7.5 Extending tasks

### 7.5.1 Introduction

RKD is designed to provide ready-to-configure automations. In practice you can install almost ready set of tasks using Python's PIP tool, then adjust those tasks to your needs.

Every Base Task implements some mechanism, in this chapter we will use Docker Container as an example.

```
Given you have a Base Task RunInContainerBaseTask, it lets you do something, while a␣
↪container is running.
You can execute commands in container, copy files between host and the container.

According to the RunInContainerBaseTask's documentation you need to extend the it.
Which means to make your own task that extends RunInContainerBaseTask as a base,
then override a method, put your code.

Voilà! Your own task basing on RunInContainerBaseTask is now ready to be executed!
```

### 7.5.2 Practical tips

In order to successfully extend a Base Task a few steps needs to be marked. Check Base Task documentation, especially for:

- The list of methods that are recommended to be extended
- Which methods could be used in `configure()` and which in `execute()`
- Does the Base Task implement `inner_execute()`
- Note which methods to override needs to keep parent call, and if the parent should be called before or after the child (method that overrides parent)

**Caution:** inner_execute() will not work if it was not implemented by parent task. The sense of existence of inner_execute() is that it should be executed inside execute() at best moment of Base Task.

**Hint:** To avoid compatibility issues when upgrading Base Task version use only documented methods

### 7.5.3 Decorators

There are three decorators that allows to decide if the parent method will be executed:

- @after_parent (from rkd.core.api.decorators): Execute our method after original method
- @before_parent (from rkd.core.api.decorators): Execute our method before original method
- @without_parent (from rkd.core.api.decorators): Do not execute parent method at all

**Important:** No decorator in most case means that the parent method will not be executed at all

**Caution:** Not all methods supports decorators. For example argument parsing always inherits argument parsing from parent. Decorators can be used for configure, compile, execute, inner_execute.

**Warning:** Using multiple decorators for single method is not allowed and leads to syntax validation error.

YAML

Simplified Python

Classic Python

```
execute@after_parent: |
    print('I will e executed after parent method will be')
```

```
@before_parent
def execute(task: PythonSyntaxTask, ctx: ExecutionContext):
    print('I will e executed before parent method will be')
```

```
def execute(self, ctx: ExecutionContext):
    print('BEFORE PARENT')
    super().execute(ctx)  # make a parent method call
    print('AFTER PARENT')
```

### 7.5.4 Example #1: Using inner_execute

Check RunInContainerBaseTask documentation first. It says that execute() should not be overridden, but inner_execute() should be used instead.

Allows to work inside of a temporary docker container.

Configuration:

- mount(): Mount directories/files as volumes
- add_file_to_copy(): Copy given files to container before container starts
- user: Container username, defaults to "root"
- shell: Shell binary path, defaults to "/bin/sh"
- docker_image: Full docker image name with registry (optional), group, image name and tag
- entrypoint: Entrypoint
- command: Command to execute on entrypoint

Runtime:

- copy_to_container(): Copy files/directory to container immediately
- in_container(): Execute inside container

Example:

```
version: org.riotkit.rkd/yaml/v1
imports:
    - rkd.core.standardlib.docker.RunInContainerBaseTask

tasks:
    :something-in-docker:
        extends: rkd.core.standardlib.docker.RunInContainerBaseTask
        configure: |
            self.docker_image = 'php:7.3'
            self.user = 'www-data'
            self.mount(local='./build', remote='/build')
            self.add_file_to_copy('build.php', '/build/build.php')
        inner_execute: |
            self.in_container('php build.php')
            return True
        # do not extend just "execute", because "execute" is used by␣
↪RunInContainerBaseTask
        # to spawn docker container, run inner_execute(), and after␣
↪just destroy the container
```

### 7.5.5 Example #2: Advanced - extending a task that extends other task

In Example #1 there is a *base task that runs something inside a docker container*, going further in Example #2 there is a task that runs any code in a PHP container.

**Architecture:**

- Our example creates a Task from PhpScriptTask (we extend it, and create a "runnable" Task from it)
- `rkd.php.script.PhpScriptTask` extends `rkd.core.standardlib.docker.RunInContainerBaseTask`

Again, to properly prepare your task basing on existing Base Task check the Base Task documentation for tips. In case of PhpScriptTask the documentation says the parent `inner_execute` method should be executed to still allow providing PHP code via stdin. To coexist parent and new method in place of `inner_execute` just use one of decorators to control the inheritance behavior.

**Complete example:**

Execute a PHP code (using a docker container) Can be extended - this is a base task.

Inherits settings from *RunInContainerBaseTask*.

**Configuration:**

- script: Path to script to load instead of stdin (could be a relative path)
- version: PHP version. Leave None to use default 8.0-alpine version

**Example of usage:**

```yaml
version: org.riotkit.rkd/yaml/v2
imports:
    - rkd.php.script.PhpScriptTask
tasks:
    :yaml:test:php:
        extends: rkd.php.script.PhpScriptTask
        configure@before_parent: |
            self.version = '7.2-alpine'
        inner_execute@after_parent: |
            self.in_container('php --version')
            print('IM AFTER PARENT. At first the PHP code from "input"␣
↪will be executed.')
            return True
        input: |
            var_dump(getcwd());
            var_dump(phpversion());
```

**Example of usage with MultiStepLanguageAgnosticTask:**

```yaml
version: org.riotkit.rkd/yaml/v1
tasks:
    :exec:
        environment:
            PHP: '7.4'
            IMAGE: 'php'
        steps: |
            #!rkd.php.script.PhpLanguage
            phpinfo();
```

### Syntax reference

| Simplified Python | Python Class | YAML | Description |
|---|---|---|---|
| get_steps(task: MultiStepLanguageAgnosticTask) -> List[str]: | get_steps | steps: [“”] | List of steps in any language (only if extending MultiStep LanguageAgnosticTask) |
| stdin() | N/A | input: “” | Standard input text |
| @extends(ClassName) decorator on a main method | ClassName(BaseClass) | extends: package.name.ClassName | Which Base Task should be extended |
| execute(task: BaseClassNameTask, ctx: ExecutionContext): | execute(self, ctx: ExecutionContext) | execute: “” | Python code to execute |
| inner_execute(task: BaseClassNameTask, ctx: ExecutionContext): | inner_execute(self, ctx: ExecutionContext) | inner_execute: “” | Python code to execute inside inner_execute (if implemented by Base Task) |
| compile(task: BaseClassNameTask, event: CompilationLifecycleEvent): | compile(self, event: CompilationLifecycleEvent): | N/A | Python code to execute during Context compilation process |
| configure(task: BaseClassNameTask, event: ConfigurationLifecycleEvent): | configure(self, event: ConfigurationLifecycleEvent): | configure: “” | Python code to execute during Task configuration process |
| get_description() | get_description(self) | description: “” | Task description |
| get_group_name() | get_group_name() | N/A | Group name |
| internal=True in TaskDeclaration | internal=True in TaskDeclaration | internal: False | Is task considered internal? (hidden on :tasks list) |
| become in TaskDeclaration (or commandline switch) | become in TaskDeclaration (or commandline switch) | become: root | Change user for task execution time |
| workdir in TaskDeclaration | workdir in TaskDeclaration | workdir: /some/path | Change working directory for task execution time |
| configure_argparse(task: BaseClassNameTask, parser: ArgumentParser) | configure_argparse(self, parser: ArgumentParser) | arguments: { } | Configure argparse.ArgumentParser object |

## 7.6 Pipelines

Pipeline is a set of Tasks executing in selected order, with optional addition of error handling. Modifiers are changing behavior of Task execution, by implementing fallbacks, retries and error notifications.

**Tip:** Modifiers can be used together e.g. **@retry** + **@rescue**, **@retry** + **@error**

### 7.6.1 Basic pipeline

Basically the pipeline is a set of Tasks, it does not need to define any error handling.

**Tip:** Treat Pipeline as a shell command invocation - in practice a Pipeline is an alias, it is similar to a command executed in command line but a little bit more advanced.

The comparison isn't abstract, that's how Pipelines works and why there are shell examples of Pipelines.

YAML

Python

Shell

```
version: org.riotkit.rkd/yaml/v2

# ...

pipelines:
    :perform:
        tasks:
            - task: :start
            - task: :do-something
            - task: :stop
```

```
from rkd.core.api.syntax import Pipeline, PipelineTask as Task, PipelineBlock as Block,␣
↪TaskDeclaration

# ...

PIPELINES = [
    Pipeline(
        name=':perform',
        description='Example',
        to_execute=[
            Task(':start'),
            Task(':do-something'),
            Task(':stop')
        ]
    )
]
```

```
# :perform
./rkdw :start :do-something :stop
```

### 7.6.2 @retry

Simplest modifier that retries each failed task in a block up to maximum of N times.

The example actually combines **@retry** + **@rescue**. But **@retry** can be used alone.

**Warning:** When retrying a Pipeline inside of a Pipeline, then all that child Pipeline Tasks will be repeated, it will work like a @retry-block for inherited Pipeline.

**Syntax:**

YAML

Python

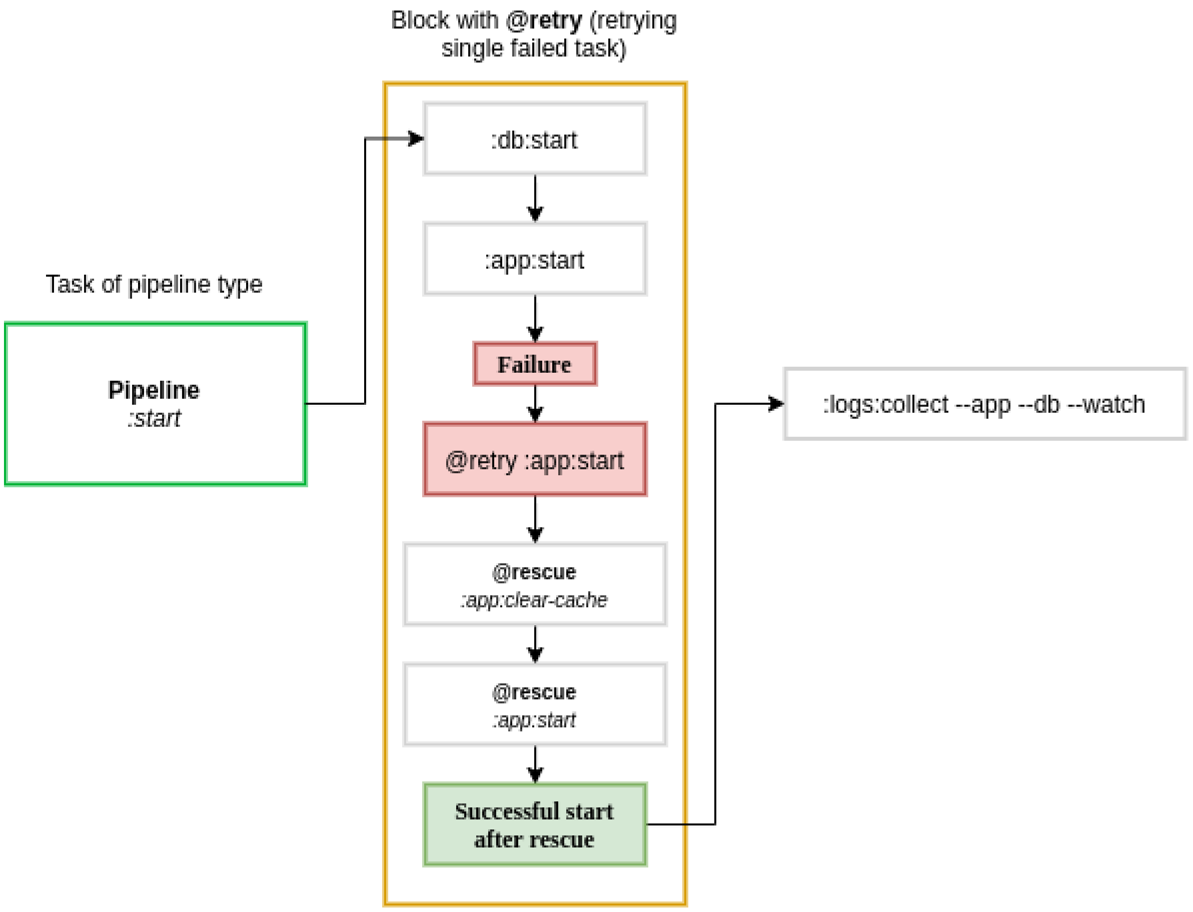
Shell

```
version: org.riotkit.rkd/yaml/v2

# ...

pipelines:
    :start:
        tasks:
            - block:
                  retry: 1  # retry max. 1 time
                  rescue: [':app:clear-cache', ':app:start']
                  tasks:
                      - task: [':db:start']
                      - task: [':app:start']
            - task: [':logs:collect', '--app', '--db', '--watch']
```

```
from rkd.core.api.syntax import Pipeline, PipelineTask as Task, PipelineBlock as Block,␣
↪TaskDeclaration

# ...

PIPELINES = [
    Pipeline(
        name=':start',
        description='Example',
        to_execute=[
            Block(rescue=':app:clear-cache :app:start', retry=1, tasks=[
                Task(':db:start'),
                Task(':app:start')
            ]),
            Task(':logs:collect', '--app', '--db', '--watch')
        ]
    )
]
```

```
# :start
./rkdw '{@rescue :app:clear-cache :app:start @retry 1}' :db:start :app:start '{/@}'␣
↪:logs:collect --app --db --watch
```

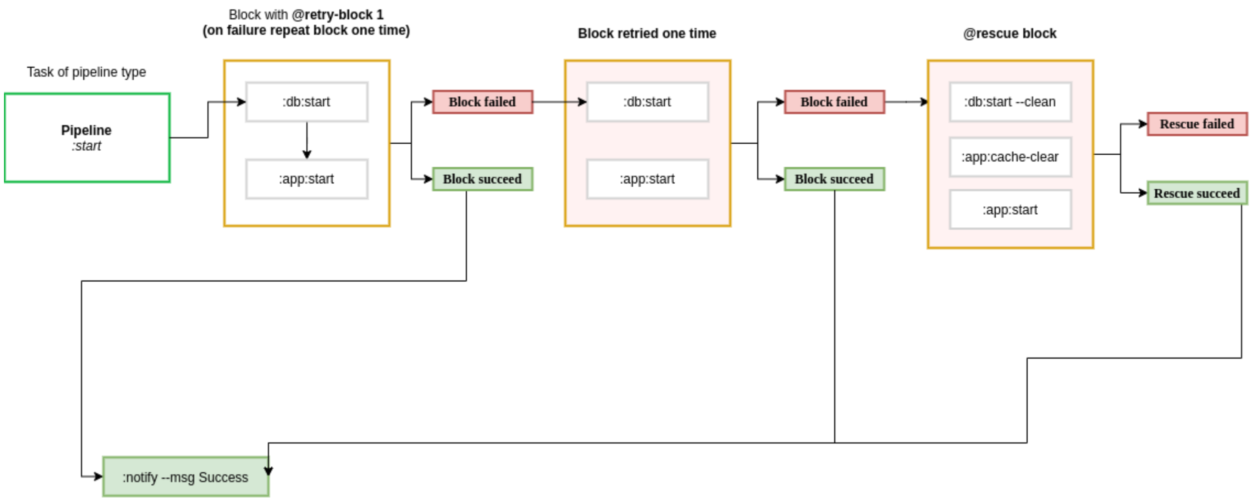
**Example workflow:**



### 7.6.3 @retry-block

Works in similar way as **@retry**, the difference is that if at least one task fails in a block, then all tasks from that blocks are repeated N times.

**Example workflow:**



### 7.6.4 @error

Executes a Task or set of Tasks when error happens. Does not affect the final result. After error task is finished the whole execution is stopped, no any more task will execute.

**Syntax:**

YAML

Python

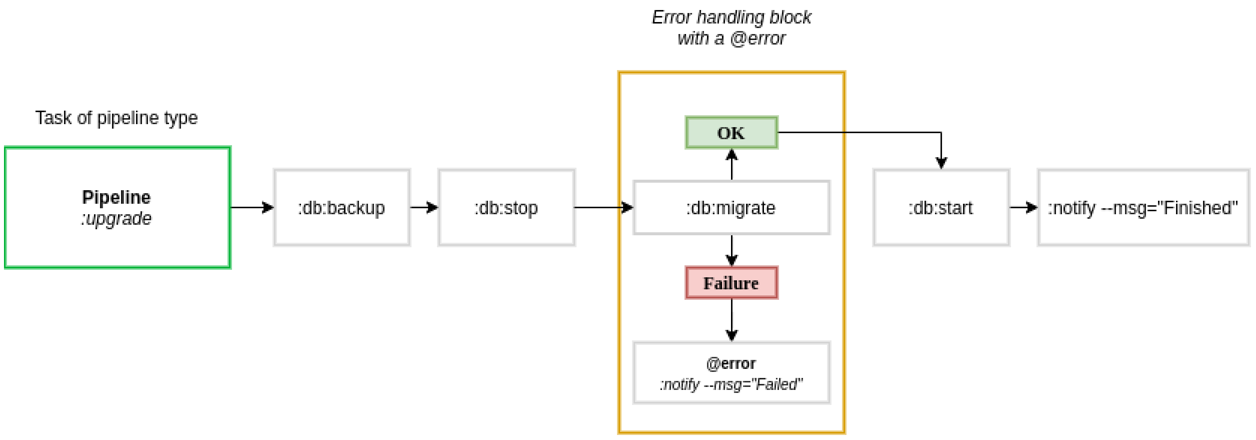
Shell

```
version: org.riotkit.rkd/yaml/v2

# ...

pipelines:
    :upgrade:
        tasks:
            - task: ":db:backup"
            - task: ":db:stop"
            - block:
                  error: [':notify', '--msg="Failed"']
                  tasks:
                      - task: [':db:migrate']
            - task: [":db:start"]
            - task: [":notify", '--msg', 'Finished']
```

```
from rkd.core.api.syntax import Pipeline, PipelineTask as Task, PipelineBlock as Block,␣
↪TaskDeclaration
from rkd.core.standardlib.core import DummyTask
from rkd.core.standardlib.shell import ShellCommandTask

# ...

PIPELINES = [
    Pipeline(
        name=':upgrade',
        description='Example',
        to_execute=[
            Task(':db:backup'),
            Task(':db:stop'),
            Block(error=':notify --msg="Failed"', tasks=[
                Task(':db:migrate')
            ]),
            Task(':db:start'),
            Task(':notify', '--msg', 'Finished')
        ]
    )
]
```

```
# :upgrade
./rkdw :db:backup :db:stop '{@error :notify --msg="Failed"}' :db:migrate '{/@}'␣
↪:db:start :notify --msg "Finished"
```

**Example workflow:**



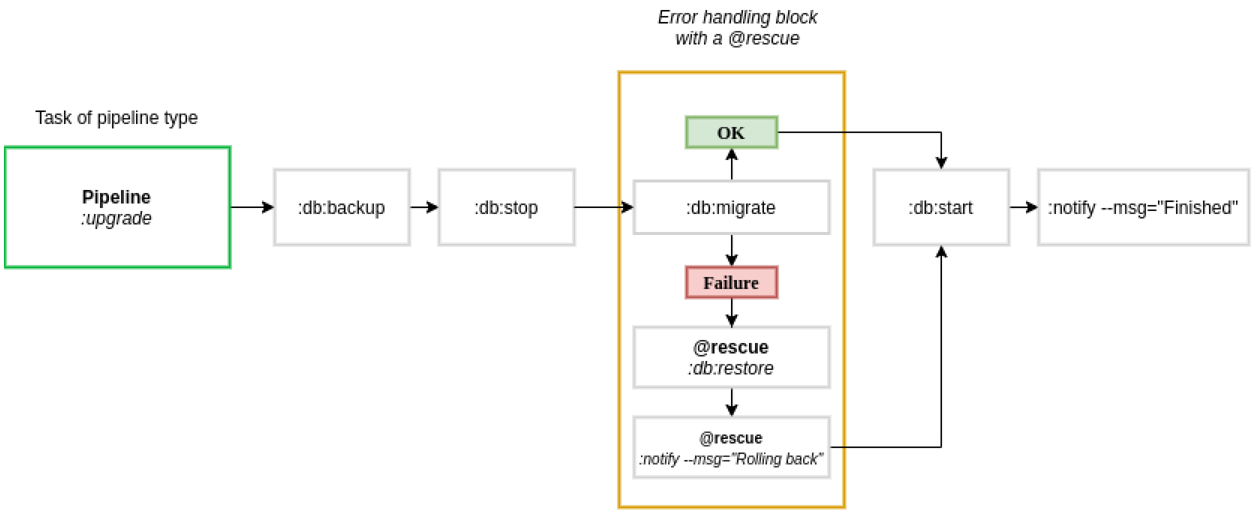
### 7.6.5 @rescue

Defines a Task that should be ran, when any of Task from given block will fail. Works similar as **@error**, but with difference that **@rescue** changes the result of pipeline execution.

**Tip:** When **@rescue** succeeds, then we assume that original Task that failed is now ok.

**Warning:** When rescuing a whole Pipeline inside other Pipeline, then failing Task will be rescued and the rest of Tasks from child Pipeline will be skipped.

**Example workflow:**



### 7.6.6 Order of modifiers execution

1. @retry: Each task is repeated until success or repeat limit
2. @retry-block: Whole block is repeated until success or repeat limit
3. @rescue: A rescue attempt of given Task or inherited Pipeline is attempted
4. @error: An error notification is sent, when all previous steps failed

### 7.6.7 Pipeline in Pipeline

A Pipeline inside a Pipeline is when we have defined a Pipeline, and one of it's called Tasks is other Pipeline.

```yaml
version: org.riotkit.rkd/yaml/v2

# ...

pipelines:
    :prepare_disk_space:
        tasks:
            - task: ":db:clear_buffers"
            - task: ":db:clear_temporary_directory"

    :upgrade:
        tasks:
            - task: ":db:backup"
            - task: ":db:stop"
            - task: ":prepare_disk_space"   # HERE IS OUR INHERITED PIPELINE
            - block:
                  error: [':notify', '--msg="Failed"']
                  tasks:
                      - task: [':db:migrate']
            - task: [":db:start"]
            - task: [":notify", '--msg', 'Finished']
```

### 7.6.8 Pipeline in Pipeline - how modifiers behave

Having Pipeline called inside other Pipeline, the inherited one is treated similar to a Task.

```
:pipeline_1 (@retry, @error, @rescue)
    :task_1
    :pipeline_2 (@retry, @rescue, ...)
        :subtask_1
        :subtask_2
    :task_3
```

When `:pipeline_2` fails then at first - *:pipeline_2* modifiers are called. In case, when *:pipeline_2* modifiers didn't rescue the Pipeline, then modifiers from parent level `:pipeline_1` are called.

**Warning:** When modifiers on main level of Pipeline fails, then parent Pipeline modifiers are inherited that behaves differently.

1. @retry from parent becomes a @retry-block of whole Pipeline (we retry a Pipeline now)
2. @rescue after rescuing a Task inside child Pipeline is skipping remaining Tasks in child Pipeline

### 7.6.9 Python syntax reference (API)

`class rkd.core.api.syntax.Pipeline`(*name: str*, *to_execute: List[Union[str, rkd.core.api.contract.PipelinePartInterface]]*, *env: Optional[Dict[str, str]] = None*, *description: str = ''*)

Task Caller

Has a name like a Task, but itself does not do anything than calling other tasks in selected order

`class rkd.core.api.syntax.PipelineTask`(*task: str*, **task_args*)

Represents a single task in a Pipeline

```
from rkd.core.api.syntax import Pipeline

PIPELINES = [
    Pipeline(
        name=':build',
        to_execute=[
            Task(':server:build --with-bluetooth'),
            Task(':client:build', '--with-bluetooth')
        ]
    )
]
```

`class rkd.core.api.syntax.PipelineBlock`(*tasks: List[*rkd.core.api.syntax.PipelineTask*]*, *retry: Optional[int] = None*, *retry_block: Optional[int] = None*, *error: Optional[str] = None*, *rescue: Optional[str] = None*)

Represents block of tasks

**Example of generated block:** {@retry 3} :some-task {/@}

```
from rkd.core.api.syntax import Pipeline, PipelineTask as Task, PipelineBlock as␣
↪Block, TaskDeclaration

Pipeline(
    name=':error-handling-example',
    description=':notify should be invoked after "doing exit" task, and execution␣
↪of a BLOCK should be interrupted',
    to_execute=[
        Task(':server:build'),
        Block(error=':notify -c "echo 'Build failed'"', retry=3, tasks=[
            Task(':docs:build', '--test'),
            Task(':sh', '-c', 'echo "doing exit"; exit 1'),
            Task(':client:build')
        ]),
        Task(':server:clear')
    ]
)
```

## 7.7 Project structure

Root level of the project should contain a hidden directory `.rkd`, there could be also defined subprojects as subdirectories of any depth.

**Example structure**

```
# project-level RKD files
.rkd/makefile.yaml
.rkd/makefile.py
rkdw

# some domain-specific files (e.g. web application)
src/Application/index.php
composer.json
composer.lock

# example subproject - documentation
docs/index.rst
docs/.rkd/makefile.yaml

# example second subproject - deployment to production
infrastructure/main.tf
infrastructure/variables.tf
infrastructure/outputs.tf
infrastructure/.rkd/makefile.py
```

**Example of usage of above project**

```
# build project, docs
./rkdw :docs:build :build
./rkdw :infrastructure:deploy
```

**Tip:** Divide Tasks in subprojects into smaller pieces to create an aggregated flow on project level, or on parent subproject level.

**Tip:** Design subprojects to be independent of Tasks in other subprojects to gain an easy way of testing smaller pieces of your automation.

### 7.7.1 Enabling subprojects

Subproject can be enabled only manually, there is no automatic discovery for performance and clarity reasons. A subproject can be included by it's parent Makefile. There can be an infinite depth of subprojects.

All tasks from subprojects are prefixed with the directory name (a subproject name).

makefile.yaml

makefile.py

```yaml
subprojects: ['docs', 'infrastructure']
```

```python
SUBPROJECTS = ['docs', 'infrastructure']
```

**Warning:** Subproject name should not contain "/" or any other special characters

**Warning:** Subprojects are loaded recursively step-by-step. Subproject cannot load sub-sub-subproject, it must go through step-by-step and include its closest children.

## 7.8 Built-in tasks

### 7.8.1 Shell

Provides tasks for shell commands execution - mostly used in YAML syntax and in Python modules.

#### :sh

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib.shell | rkd.core.standardlib.shell.ShellCommandTask | pip install rkd==SELECT VERSION | |

Executes a Bash script. Can be multi-line. Script can read from stdin instead of `-c` switch, if configured with `is_cmd_required = True` during configuration stage.

**Hint:** Phrase %RKD% is replaced with an rkd binary name

**Hint:** This is an extendable task. Read more in Extending tasks chapter.

**Example of plain usage:**

```bash
rkd :sh -c "ps aux"
```

**Example of task alias usage:**

```python

from rkd.core.api.syntax import TaskAliasDeclaration as Task

#
# Example of Makefile-like syntax
#

IMPORTS = []
```

(continues on next page)

(continued from previous page)

```
TASKS = [
    Task(':find-images', [
        ':sh', '-c', 'find ../../ -name \'*.png\''
    ]),

    Task(':build', [':sh', '-c', ''' set -x;
        cd ../../../

        chmod +x setup.py
        ./setup.py build

        ls -la
    ''']),

    # https://github.com/riotkit-org/riotkit-do/issues/43
    Task(':hello', [':sh', '-c', 'echo "Hello world"']),
    Task(':alias-in-alias-test', [':hello'])
]
```

### :exec

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib.shell | rkd.core.standardlib.shell.ExecProcessCommand | pip install rkd==SELECT VERSION | |

Works identically as **:sh**, but for spawns a single process. Does not allow a multi-line script syntax.

### BaseShellCommandWithArgumentParsingTask

**Warning:** To be used only in Python syntax

Creates a command that executes bash script and provides argument parsing using Python's argparse. Parsed arguments are registered as ARG_{{argument_name}} eg. –activity-type would be exported as ARG_ACTIVITY_TYPE.

```
IMPORTS += [
    BaseShellCommandWithArgumentParsingTask(
        name=":protest",
        group=":activism",
        description="Take action!",
        arguments_definition=lambda argparse: (
            argparse.add_argument('--activity-type', '-t', help='Select an activity type
↪')
        ),
        command='''
            echo "Let's act! Let's ${ARG_ACTIVITY_TYPE}!"
```

(continues on next page)

(continued from previous page)

```
        '''
    )
]
```

### MultiStepLanguageAgnosticTask

Used by default in YAML syntax (if "extends" is not specified). Allows to execute multiple steps in various languages.

It has very similar behavior to the GNU Makefile - each step is ran in a separate shell.

`class rkd.core.standardlib.syntax.MultiStepLanguageAgnosticTask`

Allows to define multiple shell/other language steps In YAML syntax it is a default task type (notice: there is no need to specify extends attribute)

**Bash example**

```
version: org.riotkit.rkd/yaml/v2
tasks:
    :example:
        steps: |
            echo "Hello from the Bash"
```

**Python example**

```
version: org.riotkit.rkd/yaml/v2
tasks:
    :example:
        steps: |
            #!python
            print('Hello')
```

**Multiple languages and steps example**

```
version: org.riotkit.rkd/yaml/v2
tasks:
    :example:
        steps:
            - |
                #!python
                print('Hello from Python')
            - ps aux
            - echo "Hello from Bash"
```

**Non-standard languages support**

```
version: org.riotkit.rkd/yaml/v2
imports:
    - rkd.php.script.PhpLanguage
environment:
    PHP: '8.0'
    IMAGE: 'php'
tasks:
    :example:
```

(continues on next page)

(continued from previous page)

```
        steps:
            - |
                #!rkd.php.script.PhpLanguage
                phpinfo();

            - |
                #!rkd.core.standardlib.jinja.Jinja2Language
                The used shell is {{ SHELL }}
```

## 7.8.2 Technical/Core

### :tasks

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib | rkd.core.standardlib.TasksListingTask | pip install rkd== SELECT VERSION | |

Lists all tasks that are loaded by all chained makefile.py configurations.

Environment variables:

- RKD_WHITELIST_GROUPS: (Optional) Comma separated list of groups to only show on the list
- RKD_ALIAS_GROUPS: (Optional) Comma separated list of groups aliases eg. ":international-workers-association->:iwa,:anarchist-federation->:fa"

### :version

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib | rkd.core.standardlib.VersionTask | pip install rkd== SELECT VERSION | |

Shows version of RKD and lists versions of all loaded tasks, even those that are provided not by RiotKit. The version strings are taken from Python modules as RKD strongly rely on Python Packaging.

### CallableTask

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib | rkd.core.standardlib.CallableTask | pip install rkd== SELECT VERSION | |

This is actually not a task to use directly, it is a template of a task to implement yourself. It's kind of a shortcut to create a task by defining a simple method as a callback.

```python
import os
from rkd.core.api.syntax import TaskDeclaration
from rkd.api.contract import ExecutionContext
from rkd.core.standardlib import CallableTask


def union_method(context: ExecutionContext) -> bool:
    os.system('xdg-open https://iwa-ait.org')
    return True


IMPORTS = [
    TaskDeclaration(CallableTask(':create-union', union_method))
]

TASKS = []
```

**class** rkd.core.standardlib.**CallableTask**(*name: str*, *callback: Callable[[rkd.core.api.contract.ExecutionContext, rkd.core.api.contract.TaskInterface], bool]*, *args_callback: Optional[Callable[[argparse.ArgumentParser], None]] = None*, *description: str = ''*, *group: str = ''*, *become: str = ''*, *argparse_options: Optional[List[rkd.core.api.contract.ArgparseArgument]] = None*)

Executes a custom callback - allows to quickly define a short, primitive task

**configure_argparse**(*parser: argparse.ArgumentParser*)

Allows a task to configure ArgumentParser (argparse)

```python
def configure_argparse(self, parser: ArgumentParser):
    parser.add_argument('--php', help='PHP version ("php" docker image tag)',␣
↪default='8.0-alpine')
    parser.add_argument('--image', help='Docker image name', default='php')
```

**execute**(*context: rkd.core.api.contract.ExecutionContext*) → bool

Executes a task. True/False should be returned as return

**get_become_as**() → str

User name in UNIX/Linux system, optional. When defined, then current task will be executed as this user (WARNING: a forked process would be started)

**get_declared_envs**() → Dict[str, str]

Dictionary of allowed envs to override: KEY -> DEFAULT VALUE

All environment variables fetched from the ExecutionContext needs to be defined there. Declared values there are automatically documented in –help

```python
@classmethod
def get_declared_envs(cls) -> Dict[str, Union[str, ArgumentEnv]]:
    return {
        'PHP': ArgumentEnv('PHP', '--php', '8.0-alpine'),
        'IMAGE': ArgumentEnv('IMAGE', '--image', 'php')
    }
```

**get_group_name**() → str
  Group name where the task belongs eg. ":publishing", can be empty.

**get_name**() → str
  Task name eg. ":sh"

### :rkd:create-structure

**Hint:** This is an extendable task. Read more in Extending tasks chapter.

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib | rkd.core.standardlib.CreateStructureTask | pip install rkd==SELECT VERSION | |

Creates a template structure used by RKD in current directory.

**API for developers:**

This task is extensible by class inheritance, you can override methods to implement your own task with changed behavior. It was designed to allow to create customized installers for tools based on RKD (custom RKD distributions), the example is RiotKit Harbor.

Look for "interface methods" in class code, those methods are guaranteed to not change from minor version to minor version.

class rkd.core.standardlib.**CreateStructureTask**
  Creates a RKD file structure in current directory

  This task is designed to be extended, see methods marked as "interface methods".

  **configure_argparse**(*parser: argparse.ArgumentParser*)
    Allows a task to configure ArgumentParser (argparse)

```
def configure_argparse(self, parser: ArgumentParser):
    parser.add_argument('--php', help='PHP version ("php" docker image tag)',␣
↪default='8.0-alpine')
    parser.add_argument('--image', help='Docker image name', default='php')
```

  **execute**(*ctx:* rkd.core.api.contract.ExecutionContext) → bool
    Executes a task. True/False should be returned as return

  **get_group_name**() → str
    Group name where the task belongs eg. ":publishing", can be empty.

  **get_name**() → str
    Task name eg. ":sh"

  **get_patterns_to_add_to_gitignore**(*ctx:* rkd.core.api.contract.ExecutionContext) → list
    List of patterns to write to .gitignore

    Interface method: to be overridden

  **on_creating_venv**(*ctx:* rkd.core.api.contract.ExecutionContext) → None
    When creating virtual environment

    Interface method: to be overridden

**on_files_copy**(*ctx: rkd.core.api.contract.ExecutionContext*) → None
When files are copied

Interface method: to be overridden

**on_git_add**(*ctx: rkd.core.api.contract.ExecutionContext*) → None
Action on, when adding files via *git add*

Interface method: to be overridden

**on_requirements_txt_write**(*ctx: rkd.core.api.contract.ExecutionContext*) → None
After requirements.txt file is written

Interface method: to be overridden

**on_startup**(*ctx: rkd.core.api.contract.ExecutionContext*) → None
When the command is triggered, and the git is not dirty

Interface method: to be overridden

**print_success_msg**(*use_pipenv: bool*, *ctx: rkd.core.api.contract.ExecutionContext*) → None
Emits a success message

Interface method: to be overridden

## :file:line-in-file

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib | rkd.core.standardlib.LineInFileTask | pip install rkd==SELECT VERSION | |

Similar to the Ansible's lineinfile, replaces/creates/deletes a line in file.

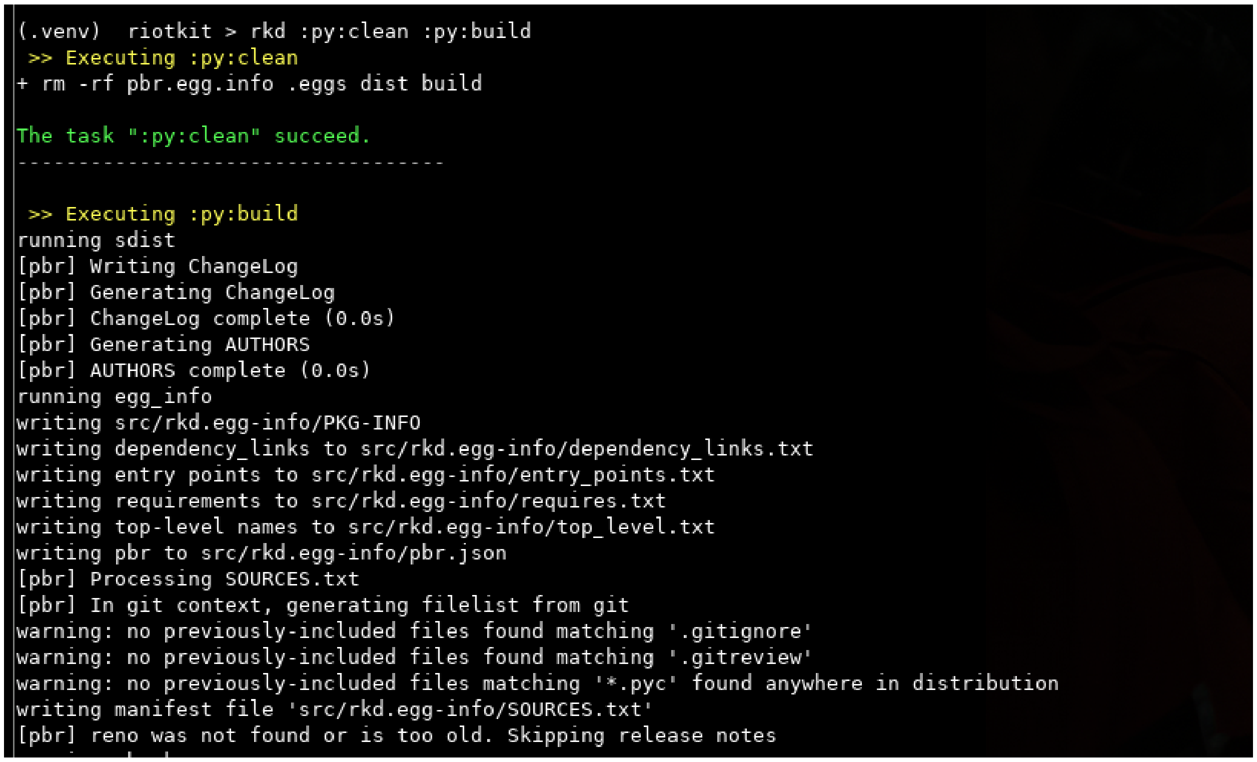
**Example usage:**

```
echo "Number: 10" > test.txt

rkd -rl debug :file:line-in-file test.txt --regexp="Number: ([0-9]+)?(.*)" --insert=
↪'Number: $match[0] / new: 10'
cat test.txt

rkd -rl debug :file:line-in-file test.txt --regexp="Number: ([0-9]+)?(.*)" --insert=
↪'Number: $match[0] / new: 6'
cat test.txt

rkd -rl debug :file:line-in-file test.txt --regexp="Number: ([0-9]+)?(.*)" --insert=
↪'Number: 50'
cat test.txt

rkd -rl debug :file:line-in-file test.txt --regexp="Number: ([0-9]+)?(.*)" --insert=
↪'Number: $match[0] / new: 90'
cat test.txt
```

## 7.8.3 Python

*This package was extracted from standardlib to rkd_python, but is maintained together with RKD as part of RKD core.*

Set of Python-related tasks for building, testing and publishing Python packages.



```
(.venv)  riotkit > rkd :py:clean :py:build
 >> Executing :py:clean
+ rm -rf pbr.egg.info .eggs dist build

The task ":py:clean" succeed.
-----------------------------------

 >> Executing :py:build
running sdist
[pbr] Writing ChangeLog
[pbr] Generating ChangeLog
[pbr] ChangeLog complete (0.0s)
[pbr] Generating AUTHORS
[pbr] AUTHORS complete (0.0s)
running egg_info
writing src/rkd.egg-info/PKG-INFO
writing dependency_links to src/rkd.egg-info/dependency_links.txt
writing entry points to src/rkd.egg-info/entry_points.txt
writing requirements to src/rkd.egg-info/requires.txt
writing top-level names to src/rkd.egg-info/top_level.txt
writing pbr to src/rkd.egg-info/pbr.json
[pbr] Processing SOURCES.txt
[pbr] In git context, generating filelist from git
warning: no previously-included files found matching '.gitignore'
warning: no previously-included files found matching '.gitreview'
warning: no previously-included files matching '*.pyc' found anywhere in distribution
writing manifest file 'src/rkd.egg-info/SOURCES.txt'
[pbr] reno was not found or is too old. Skipping release notes
```

### :py:publish

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd_python | rkd_python.PublishTask | pip install rkd_python== SELECT VERSION | |

Publish a package to the PyPI.

**Example of usage:**

```
rkd :py:publish --username=__token__ --password=.... --skip-existing --test
```

### :py:build

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd_python | rkd_python.BuildTask | pip install rkd_python== SELECT VERSION | |

Runs a build through setuptools.

### :py:install

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd_python | rkd_python.InstallTask | pip install rkd_python== SELECT VERSION | |

Installs the project as Python package using setuptools. Calls ./setup.py install.

### :py:clean

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd_python | rkd_python.CleanTask | pip install rkd_python== SELECT VERSION | |

Removes all files related to building the application.

### :py:unittest

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd_python | rkd_python.UnitTestTask | pip install rkd_python== SELECT VERSION | |

Runs Python's built'in unittest module to execute unit tests.

**Examples:**

```
rkd :py:unittest
rkd :py:unittest -p some_test
rkd :py:unittest --tests-dir=../test
```

### 7.8.4 PHP

#### PhpScriptTask

- configure: Should be overridden only with @before_parent decorator
- inner_execute: Should be overridden preserving original parent after or before
- input: A string of PHP code, optionally

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.php.script | rkd.php.script.PhpScriptTask | pip install rkd.php== SELECT VERSION | |

**Caution:** This is a Base Task. It is not a Task to run, but to create a **own, runnable Task** basing on it.

**Hint:** This is an extendable task. Read more in Extending tasks chapter.

`class rkd.php.script.PhpScriptTask`

# <sphinx:extending-tasks> Execute a PHP code (using a docker container) Can be extended - this is a base task.

Inherits settings from *RunInContainerBaseTask*.

**Configuration:**

- script: Path to script to load instead of stdin (could be a relative path)
- version: PHP version. Leave None to use default 8.0-alpine version

**Example of usage:**

```yaml
version: org.riotkit.rkd/yaml/v2
imports:
    - rkd.php.script.PhpScriptTask
tasks:
    :yaml:test:php:
        extends: rkd.php.script.PhpScriptTask
        configure@before_parent: |
            self.version = '7.2-alpine'
        inner_execute@after_parent: |
            self.in_container('php --version')
            print('IM AFTER PARENT. At first the PHP code from "input" will be executed.')
            return True
        input: |
            var_dump(getcwd());
            var_dump(phpversion());
```

**Example of usage with MultiStepLanguageAgnosticTask:**

```yaml
version: org.riotkit.rkd/yaml/v1
tasks:
```

(continues on next page)

(continued from previous page)

```
    :exec:
        environment:
            PHP: '7.4'
            IMAGE: 'php'
        steps: |
            #!rkd.php.script.PhpLanguage
            phpinfo();
```

# </sphinx:extending-tasks>

**configure**(*event: rkd.core.execution.lifecycle.ConfigurationLifecycleEvent*) → None
Executes before all tasks are executed. ORDER DOES NOT MATTER, can be executed in parallel.

**configure_argparse**(*parser: argparse.ArgumentParser*)
Allows a task to configure ArgumentParser (argparse)

```
def configure_argparse(self, parser: ArgumentParser):
    parser.add_argument('--php', help='PHP version ("php" docker image tag)',␣
↪default='8.0-alpine')
    parser.add_argument('--image', help='Docker image name', default='php')
```

**inner_execute**(*context:* rkd.core.api.contract.ExecutionContext) → bool
Execute a code when the container is up and running :param context: :return:

#### PhpLanguage

**class** rkd.php.script.**PhpLanguage**
Language extension for MultiStepLanguageAgnosticTask

```
version: org.riotkit.rkd/yaml/v1
tasks:
    :exec:
        environment:
            PHP: '7.4'
            IMAGE: 'php'
        steps: |
            #!rkd.php.script.PhpLanguage
            phpinfo();
```

### 7.8.5 ENV

Manipulates the environment variables stored in a .env file

RKD is always loading an .env file on startup, those tasks in this package allows to manage variables stored in .env file in the scope of a project.

#### :env:get

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib.env | rkd.core.standardlib.env.SetEnvTask | pip install rkd==SELECT VERSION | |

**Example of usage:**

```
rkd :env:get --name COMPOSE_PROJECT_NAME
```

#### :env:set

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib.env | rkd.core.standardlib.env.SetEnvTask | pip install rkd==SELECT VERSION | |

**Example of usage:**

```
rkd :env:set --name COMPOSE_PROJECT_NAME --value hello
rkd :env:set --name COMPOSE_PROJECT_NAME --ask
rkd :env:set --name COMPOSE_PROJECT_NAME --ask --ask-text="Please enter your name:"
```

### 7.8.6 JINJA

Renders JINJA2 files, and whole directories of files. Allows to render by pattern.

All includes and extends are by default looking in current working directory path.

#### :j2:render

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib.jinja | rkd.core.standardlib.jinja.FileRendererTask | pip install rkd==SELECT VERSION | |

Renders a single file from JINJA2.

**Example of usage:**

```
rkd :j2:render -s SOURCE-FILE.yaml.j2 -o OUTPUT-FILE.yaml
```

**Tip:** This Task is ready to be imported and executed, but can be also easily extended.

class rkd.core.standardlib.jinja.**FileRendererTask**

Renders a .j2 file using environment as input variables

**API**

To be used inside "execute":

- render(): Allows to render a JINJA template (from a string)
- render_to_file(): Renders a template to a file

**Example of API usage in YAML (if want to inherit the task):**

```
execute: |
    with open('some-file.j2', 'r') as f:
        task.render_to_file(f.read(), ctx, 'output.html')
```

**Usage**

```
./rkdw :j2:render --source=src.j2 --output=dst.html
```

## Jinja2Language

**Tip:** This class was designed especially with MultiStepLanguageAgnosticTask in mind, but can be easily used without it.

class rkd.core.standardlib.jinja.**Jinja2Language**

Jinja2 language extension for MultiStepLanguageAgnosticTask

**Usage using MultiStepLanguageAgnosticTask**

```
version: org.riotkit.rkd/yaml/v2
imports:
    - rkd.core.standardlib.jinja.Jinja2Language
tasks:
    :render:
        steps: |
            #!rkd.core.standardlib.jinja.Jinja2Language
            Test - RKD_PATH environment variable is {{ RKD_PATH }}.
            System PATH is {{ PATH }}, using shell {{ SHELL }}
```

**Usage standalone**

```
version: org.riotkit.rkd/yaml/v2
imports:
    - rkd.core.standardlib.jinja.Jinja2Language
tasks:
    :render:
        extends: rkd.core.standardlib.jinja.Jinja2Language
        input: |
            Test - RKD_PATH environment variable is {{ RKD_PATH }}.
            System PATH is {{ PATH }}, using shell {{ SHELL }}
```

```
./rkdw :render
./rkdw :render --output=/tmp/rendered
```

### :j2:directory-to-directory

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib.jinja | rkd.core.standardlib.jinja.FileRendererTask | pip install rkd==SELECT VERSION | |

Renders all files recursively in given directory to other directory.

Can remove source files after rendering them to the output files.

**Tip:** Use this Task in a docker entrypoint to create fully customizable configurations inside docker containers.

**Tip:** *Note: Pattern is a regexp pattern that matches whole path, not only file name*

**Tip:** *Note: Exclude pattern is matching on SOURCE files, not on target files*

**Example usage:**

```
rkd :j2:directory-to-directory \
    --source="/some/path/templates" \
    --target="/some/path/rendered" \
    --delete-source-files \
    --pattern="(.*).j2"
```

## 7.8.7 IO

### ArchivePackagingBaseTask

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib.io | rkd.core.standardlib.io.ArchivePackagingBaseTask | pip install rkd==SELECT VERSION | |

**Caution:** This is a Base Task. It is not a Task to run, but to create a **own, runnable Task** basing on it.

**Hint:** This is an extendable task. Read more in Extending tasks chapter.

`class rkd.core.standardlib.io.`**`ArchivePackagingBaseTask`**

**Supports:**

- dry-run mode (do not write anything to disk, just print messages)
- copies directories recursively
- .gitignore files support (manually added using API method)
- can work both as preconfigured and fully on runtime

Example (preconfigured):

```
@extends(ArchivePackagingBaseTask)
def PackIntoZipTask():
    def configure(task: ArchivePackagingBaseTask, event:␣
↪ConfigurationLifecycleEvent):
        task.archive_path = '/tmp/test-archive.zip'
        task.consider_gitignore('.gitignore')
        task.add('tests/samples/', './')

    return [configure]
```

Example (on runtime):

```
@extends(ArchivePackagingBaseTask)
def PackIntoZipTask():
    def configure(task: ArchivePackagingBaseTask, event:␣
↪ConfigurationLifecycleEvent):
        task.archive_path = '/tmp/test-archive.zip'

    def execute(task: ArchivePackagingBaseTask):
        task.consider_gitignore('.gitignore')
        task.add('tests/samples/', './')
        task.perform()

    return [configure, execute]
```

**add**(*src_path: str*, *target_path: Optional[str] = None*)
Enqueue file/directory to be added to the archive file

**Api** configure

**Parameters**

- **src_path** –
- **target_path** – Optional - name under which the file will be added to the archive

**Returns**

**configure_argparse**(*parser: argparse.ArgumentParser*)
Allows a task to configure ArgumentParser (argparse)

```
def configure_argparse(self, parser: ArgumentParser):
    parser.add_argument('--php', help='PHP version ("php" docker image tag)',␣
↪default='8.0-alpine')
    parser.add_argument('--image', help='Docker image name', default='php')
```

**execute**(*context:* rkd.core.api.contract.ExecutionContext) → bool
Executes a task. True/False should be returned as return

### 7.8.8 Docker

#### RunInContainerBaseTask

- inner_execute() should be used to execute a code while the container is running
- execute() should not be overridden

| Package to import | Single task to import | PIP package to install | Stable version |
|---|---|---|---|
| rkd.core.standardlib.docker | rkd.core.standardlib.docker.RunInContainerBaseTask | pip install rkd==SELECT VERSION | |

**Caution:** This is a Base Task. It is not a Task to run, but to create a **own, runnable Task** basing on it.

**Hint:** This is an extendable task. Read more in Extending tasks chapter.

`class rkd.core.standardlib.docker.RunInContainerBaseTask`

# <sphinx:extending-tasks>

Allows to work inside of a temporary docker container.

Configuration:

- mount(): Mount directories/files as volumes
- add_file_to_copy(): Copy given files to container before container starts
- user: Container username, defaults to "root"
- shell: Shell binary path, defaults to "/bin/sh"
- docker_image: Full docker image name with registry (optional), group, image name and tag
- entrypoint: Entrypoint
- command: Command to execute on entrypoint

Runtime:

- copy_to_container(): Copy files/directory to container immediately
- in_container(): Execute inside container

Example:

```yaml
version: org.riotkit.rkd/yaml/v1
imports:
    - rkd.core.standardlib.docker.RunInContainerBaseTask

tasks:
    :something-in-docker:
        extends: rkd.core.standardlib.docker.RunInContainerBaseTask
        configure: |
            self.docker_image = 'php:7.3'
            self.user = 'www-data'
```

(continues on next page)

(continued from previous page)

```
            self.mount(local='./build', remote='/build')
            self.add_file_to_copy('build.php', '/build/build.php')
        inner_execute: |
            self.in_container('php build.php')
            return True
        # do not extend just "execute", because "execute" is used by␣
↪RunInContainerBaseTask
        # to spawn docker container, run inner_execute(), and after just␣
↪destroy the container
```

# </sphinx:extending-tasks>

**add_file_to_copy**(*local: str*, *remote: str*) → None
Schedules a file to be copied during execution time

**Parameters**

- **local** –
- **remote** –

**Returns**

**copy_to_container**(*local: str*, *remote: str*) → None
Copies a file from host to container Can be used on execute stage

**Api**

**Parameters**

- **local** –
- **remote** –

**Returns**

**execute**(*context:* rkd.core.api.contract.ExecutionContext) → bool
Executes a task. True/False should be returned as return

**in_container**(*cmd: str*, *workdir: Optional[str] = None*, *user: Optional[str] = None*) → None
Execute a shell command inside of the container

**Parameters**

- **cmd** –
- **workdir** –
- **user** –

**Returns**

**mount**(*local: str*, *remote: str*, *mount_type: str = 'bind'*, *read_only: bool = False*) → None
Adds a mountpoint

**Parameters**

- **local** –
- **remote** –
- **mount_type** –
- **read_only** –

**Returns**

## 7.9 Working with environment variables

In a project-focused conception RKD is allowing to define environment variables in three places.

### 7.9.1 1) Dotenv

`.env` file is loaded on each RKD startup from directory, where `./rkdw` is launched.

### 7.9.2 2) Document scope

When operating on YAML there is a possibility to define a makefile-scoped environment variables, inline and loaded from dotenv file.

```
version: org.riotkit.rkd/yaml/v1
environment:
    STOP: "Police brutality"
env_files:
    - .env-prod
tasks: {}
```

### 7.9.3 3) Task scope

```
version: org.riotkit.rkd/yaml/v1
tasks:
    :task1:
        environment:
            STOP: "Police brutality"
        env_files:
            - .env-prod
        steps: |
            echo "STOP: ${STOP}"
```

### 7.9.4 4) Operating system scope

Traditional, expected way how to pass the environment variables.

```
STOP="Police brutality" ./rkdw :task1
```

### 7.9.5 Priority

Later has higher priority.

1. Dotenv loaded at startup
2. Document scope
3. Task scope
4. Operating system

## 7.10 Writing reusable tasks

There are different ways to achieve similar goal, to define the Task. In chapter about *Syntax* you can learn differences between those multiple ways.

Now we will focus on **Classic Python** syntax which allows to define Tasks as classes, those classes can be packaged into Python packages and reused across projects and event organizations.

### 7.10.1 Importing packages

Everytime a new project is created there is no need to duplicate same solutions over and over again. Even in simplest makefiles there are ready-to-use tasks from `rkd.core.standardlib` imported and used.

```
version: org.riotkit.rkd/yaml/v2
imports:
    - my_org.my_package1
```

### 7.10.2 Package index

A makefile can import a class or whole package. There is no any automatic class discovery, every package exports what was intended to export.

Below is explained how does it work that Makefile can import multiple tasks from `my_org.my_package1` without specifying classes one-by-one.

**Example package structure**

```
my_package1/
my_package1/__init__.py
my_package1/script.py
my_package1/composer.py
```

**Example __init__.py inside Python package e.g. my_org.my_package1**

```
from rkd.core.api.syntax import TaskDeclaration
from .composer import ComposerIntegrationTask                # (1)
from .script import PhpScriptTask, imports as script_imports  # (2)


# (3)
def imports():
```

(continues on next page)

(continued from previous page)

```
    return [
        TaskDeclaration(ComposerIntegrationTask()) # (5)
    ] + script_imports()  # (4)
```

- (1): **ComposerIntegrationTask** was imported from **composer.py** file
- (2): **imports as script_imports** other **def imports()** from **script.py** was loaded and used in **(4)**
- (3): **def imports()** defines which tasks will appear automatically in your build, when you import whole module, not a single class
- (5): **TaskDeclaration** can decide about custom task name, custom working directory, if the task is **internal** which means - if should be listed on :tasks

## 7.10.3 Task construction

### Basic example of how the Task looks

```
class GetEnvTask(TaskInterface):
    """Gets environment variable value"""

    def get_name(self) -> str:
        return ':get'

    def get_group_name(self) -> str:
        return ':env'

    def configure_argparse(self, parser: ArgumentParser):
        parser.add_argument('--name', '-e', help='Environment variable name',␣
↪required=True)

    def execute(self, context: ExecutionContext) -> bool:
        self.io().out(os.getenv(context.get_arg('--name'), ''))

        return True
```

### Basic configuration methods to implement

- **get_name():** Define a name e.g. `:my-task`
- **get_group_name():** Optionally a group name e.g. `:app1`
- **get_declared_envs():** List of allowed environment variables to be used inside of this Task
- **configure_argparse():** Commandline switches configuration, uses Python's native ArgParse
- **get_configuration_attributes()**: Optionally. If our Task is designed to be used as Base Task of other Task, then there we can limit which methods and class attributes can be called from **configure()** method

`class rkd.core.api.contract.`**`TaskInterface`**

`abstract configure_argparse`(*parser: argparse.ArgumentParser*)
Allows a task to configure ArgumentParser (argparse)

```python
def configure_argparse(self, parser: ArgumentParser):
    parser.add_argument('--php', help='PHP version ("php" docker image tag)',␣
↪default='8.0-alpine')
    parser.add_argument('--image', help='Docker image name', default='php')
```

**classmethod get_declared_envs()** → Dict[str, Union[str, rkd.core.api.contract.ArgumentEnv]]

Dictionary of allowed envs to override: KEY -> DEFAULT VALUE

All environment variables fetched from the ExecutionContext needs to be defined there. Declared values there are automatically documented in –help

```python
@classmethod
def get_declared_envs(cls) -> Dict[str, Union[str, ArgumentEnv]]:
    return {
        'PHP': ArgumentEnv('PHP', '--php', '8.0-alpine'),
        'IMAGE': ArgumentEnv('IMAGE', '--image', 'php')
    }
```

**abstract get_group_name()** → str

Group name where the task belongs eg. ":publishing", can be empty.

**abstract get_name()** → str

Task name eg. ":sh"

## Basic action methods

- **execute():** Contains the Task logic, there is access to environment variables, commandline switches and class attributes
- **inner_execute():** If you want to create a Base Task, then implement a call to this method inside **execute()**, so the Task that extends your Base Task can inject code inside **execute()** you defined
- **configure():** If our Task extends other Task, then there is a possibility to configure Base Task in this method
- **compile():** Code that will execute on compilation stage. There is an access to **CompilationLifecycleEvent** which allows several operations such as **task expansion** (converting current task into a Pipeline with dynamically created Tasks)

**class rkd.core.api.contract.ExtendableTaskInterface**

**compile**(*event: CompilationLifecycleEvent*) → None

Execute code after all tasks were collected into a single context

**configure**(*event: ConfigurationLifecycleEvent*) → None

Executes before all tasks are executed. ORDER DOES NOT MATTER, can be executed in parallel.

**abstract execute**(*context:* rkd.core.api.contract.ExecutionContext) → bool

Executes a task. True/False should be returned as return

**inner_execute**(*ctx:* rkd.core.api.contract.ExecutionContext) → bool

Method that can be executed inside execute() - if implemented.

**Use cases:**

- Allow child Task to inject code between e.g. database startup and database shutdown to execute some operations on the database

**Parameters ctx** –

**Returns**

### Additional methods that can be called inside execute() and inner_execute()

- **io():** Provides logging inside **execute()** and **configure()**
- **rkd() and sh():** Executes commands in subshells
- **py():** Executes Python code isolated in a subshell

`class rkd.core.api.contract.ExtendableTaskInterface`

`io()` → *rkd.core.api.inputoutput.IO*
Gives access to Input/Output object

`py`(*code: str = ''*, *become: Optional[str] = None*, *capture: bool = False*, *script_path: Optional[str] = None*, *arguments: str = ''*) → Optional[str]
Executes a Python code in a separate process

**NOTICE: Use instead of subprocess. Raw subprocess is less supported and output from raw subprocess**
may be not catch properly into the logs

`rkd`(*args: list*, *verbose: bool = False*, *capture: bool = False*) → str
Spawns an RKD subprocess

**NOTICE: Use instead of subprocess. Raw subprocess is less supported and output from raw subprocess**
may be not catch properly into the logs

`sh`(*cmd: str*, *capture: bool = False*, *verbose: bool = False*, *strict: bool = True*, *env: Optional[dict] = None*, *use_subprocess: bool = False*) → Optional[str]
Executes a shell script in bash. Throws exception on error. To capture output set capture=True

**NOTICE: Use instead of subprocess. Raw subprocess is less supported and output from raw subprocess**
may be not catch properly into the logs

## 7.11 ADVANCED usage

### 7.11.1 Troubleshooting

1. Output is corrupted or there is no output from a shell command executed inside of a task

The output capturing is under testing. The Python's subprocess module is skipping "sys.stdout" and "sys.stderr" by writing directly to /dev/stdout and /dev/stderr, which makes output capturing difficult.

Run rkd in compat mode to turn off output capturing from shell commands:

```
RKD_COMPAT_SUBPROCESS=true rkd :some-task-here
```

### 7.11.2 Loading priority

#### Environment variables loading order from .env and from .rkd

*Legend: Top is most important, the variables loaded on higher level are not overridden by lower level*

1. Operating system environment
2. Current working directory .env file
3. .env files from directories defined in RKD_PATH

#### Environment variables loading order in YAML syntax

*Legend: Top - is most important*

1. Operating system environment
2. .env file
3. Per-task "environment" section
4. Per-task "env_file" imports
5. Global "environment" section
6. Global "env_file" imports

#### Order of loading of makefile files in same .rkd directory

*Legend: Lower has higher priority (next is appending changes to previous)*

1. *.py
2. *.yaml
3. *.yml

#### Paths and inheritance

RKD by default search for .rkd directory in current execution directory - *./.rkd*.

**The search order is following (from lower to higher load priority):**

1. RKD's internals (we provide a standard tasks like *:tasks*, *:sh*, *:exec* and more)
2. */usr/lib/rkd*
3. User's home *~/.rkd*
4. Current directory *./.rkd*
5. *RKD_PATH*

**Custom path defined via environment variable**

RKD_PATH allows to define multiple paths that would be considered in priority.

```
export RKD_PATH="/some/path:/some/other/path:/home/user/riotkit/.rkd-second"
```

**How the makefiles are loaded?**

Each makefile is loaded in order, next makefile can override tasks of previous. That's why we at first load internals, then your tasks.

### Tasks execution

Tasks are executed one-by-one as they are specified in commandline or in TaskAlias declaration (commandline arguments).

```
rkd :task-1 :task-2 :task-3
```

1. task-1
2. task-2
3. task-3

A –keep-going can be specified after given task eg. :task-2 –keep-going, to ignore a single task failure and in consequence allow to go to the next task regardless of result.

## 7.11.3 Tasks development - more examples

RKD has multiple approaches to define a task. The first one is simpler - in makefile in YAML or in Python. The second one is a set of tasks as a Python package.

### Option 1) Simplest - in YAML syntax

Definitely the simplest way to define a task is to use YAML syntax, it is recommended for beginning users.

**Example 1:**

```
version: org.riotkit.rkd/yaml/v1
imports:
  - rkd.standardlib.jinja.RenderDirectoryTask

tasks:
  # see this task in "rkd :tasks"
  # run with "rkd :examples:bash-test"
  :examples:bash-test:
      description: Execute an example command in bash - show only python related tasks
      steps: |
             echo "RKD_DEPTH: ${RKD_DEPTH} # >= 2 means we are running rkd-in-rkd"
             echo "RKD_PATH: ${RKD_PATH}"
             rkd --silent :tasks | grep ":py"

  # try "rkd :examples:arguments-test --text=Hello --test-boolean"
  :examples:arguments-test:
      description: Show example usage of arguments in Bash
      arguments:
          "--text":
              help: "Adds text message"
              required: True
          "--test-boolean":
```

(continues on next page)

(continued from previous page)

```
                help: "Example of a boolean flag"
                action: store_true # or store_false
        steps:
          - |
            #!bash
            echo " ==> In Bash"
            echo " Text: ${ARG_TEXT}"
            echo " Boolean test: ${ARG_TEST_BOOLEAN}"
          - |
            #!python
            print(' ==> In Python')
            print(' Text: %s ' % ctx.args['text'])
            print(' Text: %s ' % str(ctx.args['test_boolean']))
            return True

    # run with "rkd :examples:list-standardlib-modules"
    :examples:list-standardlib-modules:
        description: List all modules in the standardlib
        steps:
          - |
            #!python
            ctx: ExecutionContext
            this: TaskInterface

            import os

            print('Hello world')
            print(os)
            print(ctx)
            print(this)

            return True

    :examples:with-other-workdir:
        description: "This task runs in /tmp"
        workdir: "/tmp"
        steps: |
            echo "I run in"
            pwd
```

**Example 2:**

```
version: org.riotkit.rkd/yaml/v1

environment:
    GLOBALLY_DEFINED: "16 May 1966, seamen across the UK walked out on a nationwide␣
↪strike for the first time in half a century. Holding solid for seven weeks, they won a␣
↪reduction in working hours from 56 to 48 per week "

env_files:
    - env/global.env
```

(continues on next page)

(continued from previous page)

```
tasks:
    :hello:
        description: |
            #1 line: 11 June 1888 Bartolomeo Vanzetti, Italian-American anarchist who␣
↪was framed & executed alongside Nicola Sacco, was born.
            #2 line: This is his short autobiography:
            #3 line: https://libcom.org/library/story-proletarian-life

        environment:
            INLINE_PER_TASK: "17 May 1972 10,000 schoolchildren in the UK walked out on␣
↪strike in protest against corporal punishment. Within two years, London state schools␣
↪banned corporal punishment. The rest of the country followed in 1987."
        env_files: ['env/per-task.env']
        steps: |
            echo " >> ENVIRONMENT VARIABLES DEMO"
            echo "Inline defined in this task: ${INLINE_PER_TASK}\n\n"
            echo "Inline defined globally: ${GLOBALLY_DEFINED}\n\n"
            echo "Included globally - global.env: ${TEXT_FROM_GLOBAL_ENV}\n\n"
            echo "Included in task - per-task.env: ${TEXT_PER_TASK_FROM_FILE}\n\n"
```

**Explanation of examples:**

1. "arguments" is an optional dict of arguments, key is the argument name, subkeys are passed directly to argparse
2. "steps" is a mandatory list or text with step definition in Bash or Python language
3. "description" is an optional text field that puts a description visible in ":tasks" task
4. "workdir" allows to optionally specify a working directory for a task
5. "environment" is a dict of environment variables that can be defined
6. "env_files" is a list of paths to .env files that should be included
7. "imports" imports a Python package that contains tasks to be used in the makefile and in shell usage

### Option 2) For Python developers - task as a class

This way allows to create tasks in a structure of a Python module. Such task can be packaged, then published to eg. PyPI (or other private repository) and used in multiple projects.

Each task should implement methods of **rkd.core.api.contract.TaskInterface** interface, that's the basic rule.

Following example task could be imported with path **rkd.standardlib.ShellCommandTask**, in your own task you would have a different package name instead of **rkd.standardlib**.

**Example task from RKD standardlib:**

```
class ShellCommandTask(ExtendableTaskInterface, MultiStepLanguageExtensionInterface):
    """
    Executes shell commands and scripts

    Extendable in two ways:
      - overwrite stdin()/input to execute a script
      - overwrite execute() to execute a Python code that could contain calls to self.
↪sh()
```

(continues on next page)

(continued from previous page)

```
    """

    # to be overridden in compile()
    is_cmd_required: bool  # Is --cmd switch required to be set?
    code: Optional[str]    # (Optional) Execute script from a variable value
    name: Optional[str]    # (Optional) Task name
    step_num: int

    def __init__(self):
        self.is_cmd_required = True
        self.code = None
        self.name = None
        self.step_num = 0

    def get_name(self) -> str:
        return ':sh' if not self.name else self.name

    def get_group_name(self) -> str:
        return ''

    def get_configuration_attributes(self) -> List[str]:
        return ['is_cmd_required']

    def configure_argparse(self, parser: ArgumentParser):
        parser.add_argument('--cmd', '-c', help='Shell command', required=self.is_cmd_
↪required)

    def with_predefined_details(self, code: str, name: str, step_num: int) ->
↪'ShellCommandTask':
        clone = copy(self)
        clone.code = code
        clone.name = name
        clone.step_num = step_num
        clone.is_cmd_required = False

        return clone

    def execute(self, context: ExecutionContext) -> bool:
        cmd = ''

        if context.get_input():
            cmd = context.get_input().read()

        if context.get_arg('cmd'):
            cmd = context.get_arg('cmd')

        if self.code:
            cmd = self.code

        try:
            # self.sh() and self.io() are part of the base class
            if cmd:
```

(continues on next page)

(continued from previous page)

```
                self.sh(cmd, capture=False)
            self.inner_execute(context)

        except CalledProcessError as e:
            self.io().error_msg(str(e))
            return False

        return True
```

**Explanation of example:**

1. The docstring in Python class is what will be shown in **:tasks as description**. You can also define your description by implementing **def get_description() -> str**
2. Name and group name defines a full name eg. :your-project:build
3. **def configure_argparse()** allows to inject arguments, and –help description for a task - it's a standard Python's argparse object to use
4. **def execute()** provides a context of execution, please read *Tasks API* chapter about it. In short words you can get commandline arguments, environment variables there.
5. **self.io()** is providing input-output interaction, please use it instead of print, please read *Tasks API* chapter about it.

### Option 3) Quick and elastic way in Python code of Makefile.py

Multiple Makefile files can be used at one time, you don't have to choose between YAML and Python. This opens a possibility to define more advanced tasks in pure Python, while you have most of the tasks in YAML. It's elastic - use YAML, or Python or both.

Let's define then a task in Python in a simplest method.

**Makefile.py**

```
from argparse import ArgumentParser
from rkd.core.api.contract import ExecutionContext
from rkd.core.api.decorators import extends
from rkd.core.api.syntax import ExtendedTaskDeclaration
from rkd.core.standardlib.syntax import PythonSyntaxTask


@extends(PythonSyntaxTask)
def hello_task():
    """
    Prints your name
    """

    def configure_argparse(task: PythonSyntaxTask, parser: ArgumentParser):
        parser.add_argument('--name', required=True, help='Allows to specify a name')

    def execute(task: PythonSyntaxTask, ctx: ExecutionContext):
        task.io().info_msg(f'Hello {ctx.get_arg("--name")}, I\'m talking in Python, and␣
↪you?')
        return True
```

(continues on next page)

(continued from previous page)

```
    return [configure_argparse, execute]


IMPORTS = [
    ExtendedTaskDeclaration(hello_task, name=':hello2')
]
```

#### Please check Tasks API for interfaces description

### 7.11.4 Global environment variables

Global switches designed to customize RKD per project. Put environment variables into your **.env** file, so you will no have to prepend them in the commandline every time.

Read also about *Environment variables loading order from .env and from .rkd*

#### RKD_WHITELIST_GROUPS

Allows to show only selected groups in the ":tasks" list. All tasks from hidden groups are still callable.

**Examples:**

```
RKD_WHITELIST_GROUPS=:rkd, rkd :tasks
RKD_WHITELIST_GROUPS=:rkd rkd :tasks
```

#### RKD_ALIAS_GROUPS

Alias group names, so it can be shorter, or even group names could be not typed at all.

*Notice: :tasks will rename a group with a first defined alias for this group*

**Examples:**

```
RKD_ALIAS_GROUPS=":rkd->:r" rkd :tasks :r:create-structure
RKD_ALIAS_GROUPS=":rkd->" rkd :tasks :create-structure
```

#### RKD_UI

Allows to toggle (true/false) the UI - messages like "Executing task X" or "Task finished", leaving only tasks stdout, stderr and logs.

### RKD_AUDIT_SESSION_LOG

Logs output of each executed task, when set to "true".

**Example structure of logs:**

```
# Note: This example requires "rkd-harbor" package to be installed from PyPI
RKD_AUDIT_SESSION_LOG=true harbor :service:list   # RiotKit Harbor is another project␣
↪based on RKD

# ls .rkd/logs/2020-06-11/11\:06\:02.068556/
task-1-init.log  task-2-harbor_service_list.log
```

### RKD_BIN

Defines a command that invokes RKD eg. `rkd`. When a custom distribution is present, then this value can different. For example project RiotKit Harbor has it's own command `harbor`, which is based on RKD, so the RKD_BIN=harbor would be defined in such project.

RKD_BIN is automatically generated, when executing task in a separate process, but it can be also set globally.

### RKD_DIST_NAME

Name of the Python package that wraps RKD (similar case as RKD_BIN use case)

### RKD_SYS_LOG_LEVEL

Use for debugging. The variable is read in very early stage of RKD initialization, before context preparation.

```
RKD_SYS_LOG_LEVEL=debug rkd :tasks
```

### RKD_IMPORTS

Allows to import a task, or group of tasks (module) inline, without need to create a Makefile. Useful in daily tasks to create handy shortcuts, also very useful for testing tasks and embedding them inside other applications.

"**:**" character is a separator for multiple imports.

```
# note: Those examples requires "rkt_utils" package from PyPI
RKD_IMPORTS="rkt_utils.docker" rkd :docker:tag
RKD_IMPORTS="rkt_utils.docker:rkt_ciutils.boatci:rkd_python" rkd :tasks
```

#### RKD_DEPTH

Internally used to detect if RKD is called from inside of RKD

### 7.11.5 Custom distribution

RiotKit Do can be used as a transparent framework for writing tasks for various usage, especially for specialized usage. To simplify usage for end-user RKD allows to create a custom distribution.

**Custom distribution allows to:**

- Define custom 'binary' name eg. "harbor" instead of "rkd"
- Hide unnecessary tasks in custom 'binary' (filter by groups - whitelist)
- Make shortcuts to tasks: Skip writing group name, make a group name to be appended by default

#### Example

```
import os
from rkd import main as rkd_main

def env_or_default(env_name: str, default: str):
    return os.environ[env_name] if env_name in os.environ else default

def main():
    os.environ['RKD_WHITELIST_GROUPS'] = env_or_default('RKD_WHITELIST_GROUPS', ':env,
↪:harbor,')
    os.environ['RKD_ALIAS_GROUPS'] = env_or_default('RKD_ALIAS_GROUPS', '->:harbor')
    os.environ['RKD_UI'] = env_or_default('RKD_UI', 'false')
    rkd_main()


if __name__ == '__main__':
    main()
```

```
$ harbor :tasks
[global]
:sh                                                # Executes shell scripts
:exec                                              # Spawns a shell process
:tasks                                             # Lists all enabled tasks
:version                                           # Shows version of RKD and of all␣
↪loaded tasks

[harbor]
:compose:ps                                        # List all containers
:start                                             # Create and start containers
:stop                                              # Stop running containers
:remove                                            # Forcibly stop running containers and␣
↪remove (keeps volumes)
:service:list                                      # Lists all defined containers in YAML␣
↪files (can be limited by --profile selector)
:service:up                                        # Starts a single service
```

(continues on next page)

(continued from previous page)

```
:service:down                                     # Brings down the service without␣
↪deleting the container
:service:rm                                       # Stops and removes a container and it
↪'s images
:pull                                             # Pull images specified in containers␣
↪definitions
:restart                                          # Restart running containers
:config:list                                      # Gets environment variable value
:config:enable                                    # Enable a configuration file - YAML
:config:disable                                   # Disable a configuration file - YAML
:prod:gateway:reload                              # Reload gateway, regenerate missing␣
↪SSL certificates
:prod:gateway:ssl:status                          # Show status of SSL certificates
:prod:gateway:ssl:regenerate                      # Regenerate all certificates with␣
↪force
:prod:maintenance:on                              # Turn on the maintenance mode
:prod:maintenance:off                             # Turn on the maintenance mode
:git:apps:update                                  # Fetch a git repository from the␣
↪remote
:git:apps:update-all                              # List GIT repositories
:git:apps:set-permissions                         # Make sure that the application would␣
↪be able to write to allowed directories (eg. upload directories)
:git:apps:list                                    # List GIT repositories

[env]
:env:get                                          # Gets environment variable value
:env:set                                          # Sets environment variable in the .
↪env file


Use --help to see task environment variables and switches, eg. rkd :sh --help, rkd --help
```

**Notices for above example:**

- No need to type eg. :harbor:config:list - just :config:list (RKD_ALIAS_GROUPS used)
- No "rkd" group is displayed (RKD_WHITELIST_GROUPS used)
- There is no information about task name (RKD_UI used)

### Read more in Global environment variables

### Customizing RKD resource files

Files like banner, internal Makefiles can be overridden in user's home directory, or in operating system-wide directory.

Here is the priority list, first matching result stops the search:

dist_name = env.distribution_name() # RKD_DIST_NAME env variable

**paths = [** # eg. ~/.local/share/rkd/banner.txt os.path.expanduser(('~/.local/share/%s/' + path) % dist_name),

# eg. /home/andrew/.local/lib/python3.8/site-packages/rkd/misc/banner.txt (get_user_site_packages() + '/%s/misc/' + path) % dist_name,

# eg. /usr/lib/python3.8/site-packages/rkd/misc/banner.txt (_get_global_site_packages() + '/%s/misc/' + path) % dist_name,

# eg. /usr/share/rkd/banner.txt ('/usr/share/%s/' + path) % dist_name

### 7.11.6 Tasks API

#### Each task must implement a TaskInterface (directly, or through a Base Task)

**Attention:** Methods marked as `abstract` must be implemented by your task that extends directly from TaskInterface.

**Tip:** During configuration and execution stage every task is having it's own ExecutionContext instance. ExecutionContext (called ctx) gives access to parameters, environment variables, user input (e.g. stdin) Do not try to manually read from stdin, or os.environment - read more about this topic in *Best practices* chapter.

`class rkd.core.api.contract.TaskInterface`

`abstract configure_argparse`(*parser: argparse.ArgumentParser*)
Allows a task to configure ArgumentParser (argparse)

```
def configure_argparse(self, parser: ArgumentParser):
    parser.add_argument('--php', help='PHP version ("php" docker image tag)',␣
↪default='8.0-alpine')
    parser.add_argument('--image', help='Docker image name', default='php')
```

`copy_internal_dependencies`(*task*)
Allows to execute a task-in-task, by copying dependent services from one task to other task

`exec`(*cmd: str*, *capture: bool = False*, *background: bool = False*) → Optional[str]
Starts a process in shell. Throws exception on error. To capture output set capture=True

**NOTICE: Use instead of subprocess. Raw subprocess is less supported and output from raw subprocess**
may be not catch properly into the logs

`abstract execute`(*context:* rkd.core.api.contract.ExecutionContext) → bool
Executes a task. True/False should be returned as return

`extends_task`()
Provides information if this Task has a Parent Task

**Returns**

`format_task_name`(*name: str*) → str
Allows to add a fancy formatting to the task name, when the task is displayed eg. on the :tasks list

`get_become_as`() → str
User name in UNIX/Linux system, optional. When defined, then current task will be executed as this user (WARNING: a forked process would be started)

`classmethod get_declared_envs`() → Dict[str, Union[str, rkd.core.api.contract.ArgumentEnv]]
Dictionary of allowed envs to override: KEY -> DEFAULT VALUE

All environment variables fetched from the ExecutionContext needs to be defined there. Declared values there are automatically documented in –help

```
@classmethod
def get_declared_envs(cls) -> Dict[str, Union[str, ArgumentEnv]]:
    return {
        'PHP': ArgumentEnv('PHP', '--php', '8.0-alpine'),
        'IMAGE': ArgumentEnv('IMAGE', '--image', 'php')
    }
```

**get_full_name**()
Returns task full name, including group name

abstract **get_group_name**() → str
Group name where the task belongs eg. ":publishing", can be empty.

abstract **get_name**() → str
Task name eg. ":sh"

**io**() → *rkd.core.api.inputoutput.IO*
Gives access to Input/Output object

**py**(*code: str = ''*, *become: Optional[str] = None*, *capture: bool = False*, *script_path: Optional[str] = None*, *arguments: str = ''*) → Optional[str]
Executes a Python code in a separate process

**NOTICE: Use instead of subprocess. Raw subprocess is less supported and output from raw subprocess**
may be not catch properly into the logs

**rkd**(*args: list*, *verbose: bool = False*, *capture: bool = False*) → str
Spawns an RKD subprocess

**NOTICE: Use instead of subprocess. Raw subprocess is less supported and output from raw subprocess**
may be not catch properly into the logs

**sh**(*cmd: str*, *capture: bool = False*, *verbose: bool = False*, *strict: bool = True*, *env: Optional[dict] = None*, *use_subprocess: bool = False*) → Optional[str]
Executes a shell script in bash. Throws exception on error. To capture output set capture=True

**NOTICE: Use instead of subprocess. Raw subprocess is less supported and output from raw subprocess**
may be not catch properly into the logs

**should_fork**() → bool
Decides if task should be ran in a separate Python process (be careful with it)

**silent_sh**(*cmd: str*, *verbose: bool = False*, *strict: bool = True*, *env: Optional[dict] = None*) → bool
sh() shortcut that catches errors and displays using IO().error_msg()

**NOTICE: Use instead of subprocess. Raw subprocess is less supported and output from raw subprocess**
may be not catch properly into the logs

### To include a task, wrap it in a declaration

**Note:** Task declaration declares a Task (TaskInterface implementation) to be a runnable task imported into a given Makefile.

**Tip:** With TaskDeclaration there is a possibility to customize things like task name, environment, working directory and other attributes.

`class rkd.core.api.syntax.TaskDeclaration`(*task:* rkd.core.api.contract.TaskInterface, *env: Optional[Dict[str, str]] = None*, *args: Optional[List[str]] = None*, *workdir: Optional[str] = None*, *internal: Optional[bool] = None*, *name: Optional[str] = None*)

Task Declaration is a DECLARED USAGE of a Task (instance of TaskInterface)

Examples of usage:

```
TaskDeclaration(MyNiceTask(), env={'SOME': 'thing'}, workdir='/tmp', name=
↪':custom:task:name')
```

### To create an alias for task or multiple tasks

**Note:** TaskAlias is a simplified pipeline form, it is a chain of tasks written in a string form.

`class rkd.core.api.syntax.TaskAliasDeclaration`(*name: str*, *to_execute: List[Union[str, rkd.core.api.contract.PipelinePartInterface]]*, *env: Optional[Dict[str, str]] = None*, *description: str = ''*)

Deprecated: Name will be removed in RKD 6.0

### Execution context provides parsed shell arguments and environment variables

`class rkd.core.api.contract.ExecutionContext`(*declaration: rkd.core.api.contract.TaskDeclarationInterface*, *parent: Optional[rkd.core.api.contract.GroupDeclarationInterface] = None*, *args: Dict[str, str] = {}*, *env: Dict[str, str] = {}*, *defined_args: Dict[str, dict] = {}*)

Defines which objects could be accessed by Task. It’s a scope of a single task execution.

`can_mutate_globals`() → bool

Is task having a special permissions to mutate globals such as OS environment :return:

`get_arg`(*name: str*) → Optional[str]

Get argument or option

**Usage:** ctx.get_arg(‘–name’) # for options ctx.get_arg(‘name’) # for arguments

**Raises:** KeyError when argument/option was not defined

**Returns:** Actual value or default value

**get_arg_or_env**(*name: str*) → Optional[str]

Provides value of user input

**Usage:** get_arg_or_env('–file-path') resolves into FILE_PATH env variable, and –file-path switch (file_path in argparse)

**Behavior:** When user provided explicitly switch eg. –history-id, then it's value will be taken in priority. If switch –history-id was not used, but user provided HISTORY_ID environment variable, then it will be considered.

If no switch provided and no environment variable provided, but a switch has default value - it would be returned.

If no switch provided and no environment variable provided, the switch does not have default, but environment variable has a default value defined, it would be returned.

When the –switch has default value (user does not use it, or user sets it explicitly to default value), and environment variable SWITCH is defined, then environment variable would be taken.

From RKD 2.1 the environment variable names can be mapped to any ArgParse switch.

Below example maps "COMMAND" environment variable to "–cmd" switch.

```
def get_declared_envs(self) -> Dict[str, Union[str, ArgumentEnv]]:
    return {
        'COMMAND': ArgumentEnv(name='COMMAND', switch='--cmd', default='')
    }
```

**Raises:** MissingInputException: When no switch and no environment variable was provided, then an exception is thrown.

**get_env**(*name: str, switch: str = '', error_on_not_used: bool = False*)

Get environment variable value

## Interaction with input and output

**Tip:** From inside a Task the IO can be accessed with `self.io()`

**Caution:** Every task has it's own instance of IO, with customized per-task log level.

class `rkd.core.api.inputoutput.IO`

Interacting with input and output - stdout/stderr/stdin, logging

**add_output_processor**(*callback: Callable[[Union[str, bytes], str], Union[str, bytes]]*)

Registers a output processing callback Each byte outputted by this IO instance will go through a set of registered processors

**Example use cases:**

- Hide sensitive information (secrets)
- Reformat output
- Strip long stdouts from commands
- Change colors
- Add/remove formatting

**Parameters** `callback` –

**Returns**

`capture_descriptors`(*target_files: List[str] = None*, *stream=None*, *enable_standard_out: bool = True*)
Capture stdout and stderr from a block of code - use with 'with'

`critical`(*text*)
Logger: critical

`debug`(*text*)
Logger: debug

`err`(*text*)
Standard error

`errln`(*text*)
Standard error + newline

`error`(*text*)
Logger: error

`error_msg`(*text*)
Error message

`static format_table`(*header: list*, *body: list*, *tablefmt: str = 'simple'*, *floatfmt: str = 'g'*, *numalign: str = 'decimal'*, *stralign: str = 'left'*, *missingval: str = ''*, *showindex: str = 'default'*, *disable_numparse: bool = False*, *colalign: Optional[str] = None*)
Renders a table

**Parameters:** header: body: tablefmt: floatfmt: numalign: stralign: missingval: showindex: disable_numparse: colalign:

**Returns:** Formatted table as string

`h1`(*text*)
Heading #1 (optional output)

`h2`(*text*)
Heading #2 (optional output)

`h3`(*text*)
Heading #3 (optional output)

`h4`(*text*)
Heading #3 (optional output)

`info`(*text*)
Logger: info

`info_msg`(*text*)
Informational message (optional output)

`internal`(*text*)
Logger: internal Should be used only by RKD core for more intensive logging

`internal_lifecycle`(*text*)
Should be used only by RKD core for more intensive logging :param text: :return:

`is_silent`() → bool
Is output silent? In silent mode OPTIONAL MESSAGES are not shown

`opt_errln`(*text*)
Optional errln()

**`opt_out`**(*text*)
: Optional output - fancy output skipped in –silent mode

**`opt_outln`**(*text*)
: Optional output - fancy output skipped in –silent mode + newline

**`out`**(*text*)
: Standard output

**`outln`**(*text*)
: Standard output + newline

**`print_group`**(*text*)
: Prints a colored text inside brackets [text] (optional output)

**`print_line`**()
: Prints a newline

**`print_opt_line`**()
: Prints a newline (optional output)

**`print_separator`**(*status: Optional[bool] = None*)
: Prints a text separator (optional output)

**`success_msg`**(*text*)
: Success message (optional output)

**`warn`**(*text*)
: Logger: warn

**`warn_msg`**(*text*) → None
: Warning message (optional output)

## Storing temporary files

**Tip:** From inside a Task the TempManager can be accessed with `self.temp`

`class rkd.core.api.temp.`**`TempManager`**(*chdir: str = './.rkd/'*)

Manages temporary files inside .rkd directory Using this class you make sure your code is more safe to use on Continuous Integration systems (CI)

**Usage:** path = self.temp.assign_temporary_file(mode=0o755)

**`assign_temporary_file`**(*mode: int = 493*) → str
: Assign a path for writing temporary files in RKD workspace

  Note: The RKD is executing the finally_clean_up() at the end of each task

  **Usage:**

  **try:** path = RKDTemp.assign_temporary_file_path() # (. . . ) some action there

  **finally:** RKDTemp.finally_clean_up()

**`finally_clean_up`**()
: Used to clean up all temporary files at the end of the code execution

  TaskExecutor is running this method after each finished task

### Parsing RKD syntax

**class** `rkd.core.api.parsing.`**`SyntaxParsing`**

**static** **`parse_import_as_type`**(*import_str: str*) → Type[*rkd.core.api.contract.TaskInterface*]

Import a Python class as a type

Example: rkd.core.standardlib.jinja.FileRendererTask

**Parameters** **`import_str`** –

**Returns**

**classmethod** **`parse_imports_by_list_of_classes`**(*classes_or_modules: List[str]*) → List[*rkd.core.api.syntax.TaskDeclaration*]

Parses a List[str] of imports, like in YAML syntax. Produces a List[TaskDeclaration] with imported list of tasks.

Could be used to import & validate RKD tasks.

**Examples:**

- rkd.core.standardlib
- rkd.core.standardlib.jinja.FileRendererTask

:raises ParsingException :return:

### Testing

**Tip:** BasicTestingCase is best for unit testing

**class** `rkd.core.api.testing.`**`BasicTestingCase`**(*methodName='runTest'*)

**Provides minimum of:**

- Doing backup of environment and cwd
- Methods for mocking task dependencies (RKD-specific like ExecutionContext)

**`environment`**(*environ: dict*)

Mocks environment

**Example usage:**

```
with self.environment({'RKD_PATH': SCRIPT_DIR_PATH + '/../docs/examples/env-
↪in-yaml/.rkd'}):
    # code there
```

**Parameters** **`environ`** –

**Returns**

**static** **`list_to_str`**(*in_list: list*) → List[str]

Execute __str__ on each list element, and replace element with the result `static mock_execution_context`(*task:* rkd.core.api.contract.TaskInterface, *args: Optional[Dict[str, Union[str, bool]]] = None*, *env: Optional[Dict[str, str]] = None*, *defined_args: Optional[Dict[str, dict]] = None*) → *rkd.core.api.contract.ExecutionContext*

Prepares a simplified rkd.core.api.contract.ExecutionContext instance

**Parameters**

- **task** –
- **args** –
- **env** –
- **defined_args** –

**Returns**

`static satisfy_task_dependencies`(*task:* rkd.core.api.contract.TaskInterface, *io: Optional[*rkd.core.api.inputoutput.IO*] = None*) → *rkd.core.api.contract.TaskInterface*

Inserts required dependencies to your task that implements rkd.core.api.contract.TaskInterface

**Parameters**

- **task** –
- **io** –

**Returns**

`setUp`() → None

Hook method for setting up the test fixture before exercising it.

`tearDown`() → None

Hook method for deconstructing the test fixture after testing it.

**Tip:** FunctionalTestingCase should be using for tests that are running single task and asserting output contents.

`class rkd.core.api.testing.FunctionalTestingCase`(*methodName='runTest'*)

Provides methods for running RKD task or multiple tasks with output and exit code capturing. Inherits OutputCapturingSafeTestCase.

`execute_mocked_task_and_get_output`(*task:* rkd.core.api.contract.TaskInterface, *args=None*, *env=None*) → str

Run a single task, capturing it's output in a simplified way. There is no whole RKD bootstrapped in this operation.

**Parameters**

- **task** (`TaskInterface`) –
- **args** (*dict*) –
- **env** (*dict*) –

**Returns**

`classmethod filter_out_task_events_from_log`(*out: str*)

Produces an array of events unformatted

```
[
    "Executing :sh -c echo 'Rocker' [part of :example]",
    "Executing :sh -c echo 'Kropotkin' [part of :example]",
    'Executing :sh -c echo "The Conquest of Bread"; exit 1 [part of :example]',
    'Retrying :sh -c echo "The Conquest of Bread"; exit 1 [part of :example]',
    'Executing :sh -c exit 0 ',
    'Executing :sh -c echo "Modern Science and Anarchism"; [part of :example]'
]
```

**Parameters** **out** –

**Returns**

**run_and_capture_output**(*argv: list*, *verbose: bool = False*) → Tuple[str, int]

Run task(s) and capture output + exit code. Whole RKD from scratch will be bootstrapped there.

**Example usage:** full_output, exit_code = self.run_and_capture_output([':tasks'])

**Parameters**

- **argv** (*list*) – List of tasks, arguments, commandline switches
- **verbose** (*bool*) – Print all output also to stdout

**Returns**

**with_temporary_workspace_containing**(*files: Dict[str, str]*)

Creates a temporary directory as a workspace and fills up with files specified in "file" parameter

**Parameters** **files** – Dict of [filename: contents to write to a file]

**Returns**

**class** rkd.core.api.testing.**OutputCapturingSafeTestCase**(*methodName='runTest'*)

Provides hooks for keeping stdout/stderr immutable between tests.

**setUp**() → None

Hook method for setting up the test fixture before exercising it.

**tearDown**() → None

Hook method for deconstructing the test fixture after testing it.

### 7.11.7 Working with YAML files

Makefile.yaml has checked syntax before it is parsed by RKD. A **jsonschema** library was used to validate YAML files against a JSON formatted schema file.

This gives the early validation of typing inside of YAML files, and a clear message to the user about place where the typo is.

### YAML parsing API

Schema validation is a part of YAML parsing, the preferred way of working with YAML files is to not only parse the schema but also validate. In result of this there is a class that wraps **yaml** library - **rkd.yaml_parser.YamlFileLoader**, use it instead of plain **yaml** library.

*Notice: The YAML and schema files are automatically searched in .rkd, .rkd/schema directories, including RKD_PATH*

**Example usage:**

```
from rkd.yaml_parser import YamlFileLoader

parsed = YamlFileLoader([]).load_from_file('deployment.yml', 'org.riotkit.harbor/
↪deployment/v1')
```

### FAQ

1. *FileNotFoundError: Schema "my-schema-name.json" cannot be found, looked in: ['.../riotkit-harbor', '/.../riotkit-harbor/schema', '/.../riotkit-harbor/.rkd/schema', '/home/.../.rkd/schema', '/usr/lib/rkd/schema', '/usr/lib/python3.8/site-packages/rkd/internal/schema']*

The schema file cannot be found, the name is invalid or file missing. The schema should be placed somewhere in the .rkd/schema directory - in global, in home directory or in project.

2. *rkd.exception.YAMLFileValidationError: YAML schema validation failed at path "tasks" with error: [] is not of type 'object'*

It means you created a list (starts with "-") instead of dictionary at "tasks" path.

**Example of what went wrong:**

```
tasks:
    - description: first
    - description: second
```

**Example of how it should be as an 'object' (dictionary):**

```
tasks:
    first:
        description: first

    second:
        description: second
```

### API

`class rkd.core.yaml_parser.`**`YamlFileLoader`**(*paths: List[str]*)

YAML loader extended by schema validation support

YAML schema is stored as JSON files in .rkd/schema directories. The Loader looks in all paths defined in RKD_PATH as well as in paths provided by ApplicationContext

**`find_path_by_name`**(*filename: str*, *subdir: str*) → str

Find schema in one of RKD directories or in current path

**`load`**(*stream*, *schema_name: str*)

Loads a YAML, validates and return parsed as dict/list

**load_from_file**(*filename: str*, *schema_name: str*)
Loads a YAML file from given path, a wrapper to load()

### 7.11.8 Creating installer wizards with RKD

**Wizard** is a component designed to create comfortable installers, where user has to answer a few questions to get the task done.

#### Concept

- User answers questions invoked by `ask()` method calls
- At the end the `finish()` is called, which acts as a commit, saves answers into `.rkd/tmp-wizard.json` by default and into the `.env` file (depends on if to_env=true was specified)
- Next RKD task executed can read `.rkd/tmp-wizard.json` looking for answers, the answers placed in .env are already loaded automatically as part of standard mechanism of environment variables support

#### Example Wizard

```python
from rkd.core.api.inputoutput import Wizard

# self is the TaskInterface instance, in Makefile.yaml it would be "this", in Python␣
↪code it is "self"
Wizard(self)\
    .ask('Service name', attribute='service_name', regexp='([A-Za-z0-9_]+)', default=
↪'redis')\
    .finish()
```

```
Service name [([A-Za-z0-9_]+)] [default: redis]:
    -> redis
```

**Example of application that is using Wizard to ask interactive questions**

#### Using Wizard results internally

Wizard is designed to keep the data on the disk, so you can access it in any other task executed, but this is not mandatory. You can skip committing changes to disk by not using `finish()` which **is flushing data to json and to .env files.**

Use `wizard.answers` to see all answers that would be put into json file, and `wizard.to_env` to browse all environment variables that would be set in .env if `finish()` would be used.

#### Example of loading stored values by other task

Wizard stores values into file and into .env file, so it can read it from file after it was stored there. This allows you to separate Wizard questions into one RKD task, and the rest of logic/steps into other RKD tasks.

```
from rkd.core.api.inputoutput import Wizard

# ... assuming that previously the Wizard was completed by user and the finish() method␣
↪was called ...

wizard = Wizard(self)
wizard.load_previously_stored_values()

print(wizard.answers, wizard.to_env)
```

#### API

**class** rkd.core.api.inputoutput.**Wizard**(*task: TaskInterface*, *filename: str = 'tmp-wizard.json'*)

**ask**(*title: str*, *attribute: str*, *regexp: str = ''*, *to_env: bool = False*, *default: Optional[str] = None*, *choices: list = []*, *secret: bool = False*) → *rkd.core.api.inputoutput.Wizard*

Asks user a question

**Usage:**

```
wizard = Wizard(self)
wizard.ask('In which year the Spanish social revolution has begun?',
           attribute='year',
           choices=['1936', '1910'])
wizard.finish()
```

**finish**() → *rkd.core.api.inputoutput.Wizard*

Commit all pending changes into json and .env files

**input**(*secret: bool = False*)

(Internal) Extracted for unit testing to make testing easier

**load_previously_stored_values**()

Load previously saved values

### 7.11.9 Best practices

#### Do not use os.getenv()

The ExecutionContext is providing processed environment variables. Variables could be overridden on some levels eg. in makefile.py - `rkd.core.api.syntax.TaskAliasDeclaration` can take a dict of environment variables to force override.

Use `context.get_env()` instead.

### Define your environment variables

*Note: Only in Python code*

By using `context.get_env()` you are enforced to implement a `TaskInterface.get_declared_envs()` returning a list of all environment variables used in your task code.

All defined environment variables will land in –help, which is considered as a task self-documentation.

### Use sh() and exec() to invoke commands

Using raw `subprocess` will make your commands output invisible in logs, as the subprocess is writting directly to stdout/stderr skipping sys.stdout and sys.stderr. The methods provided by RKD are buffering the output and making it possible to save to both file and to console.

### Do not print if you do not must, use io()

`rkd.core.api.inputoutput.IO` provides a standardized way of printing messages. The class itself distinct importance of messages, writing them to proper stdout/stderr and to log files.

`print` is also captured by IO, but should be used only eventually.

### Use tasks expansion or pipelines instead of dynamic tasks creation in Makefile

Makefiles are not designed to execute logic outside tasks execution. As long as it is possible use `compilation stage` to expand task into a group of tasks, see Task expansion pattern chapter of the documentation.

Standard way of tasks creation helps other people to understand your construction due to a common usage of a defined pattern in our documentation. Second argument is that things defined using Task expansion pattern are possible to package into a Python package.

### Invoke RKD subtasks as a pipeline or tasks expansion

Invoking tasks in the middle of one of tasks code signals an architecture fault. Your tasks probably are having too many responsibilities. To be SOLID split tasks into smaller pieces and create a pipeline or task expansion.

### Don't mix dependencies between subprojects - rethink project structure

RKD contexts are built at compilation stage, therefore it is possible that in `subproject` A you can use tasks from `subproject B`

**Solutions to avoid complex dependencies:**

- Extract common things into base tasks accessible in the PYTHONPATH, or best in a separate package
- Create aggregated pipelines on top level, before the subprojects, e.g. on project level. This requires to cut subprojects into smaller pieces to pilot the behavior from project level

Having complex dependencies in subprojects is again a signal of a invalid design.

#### Keep compilation and configuration stage fast

Compilation and configuration stages of every task are not intended to query databases, HTTP servers, to search recursively for files or directories. All other tasks in project will be affected if at least one task compilation would be slow.

#### Use configuration stage for validation

Configuration stage should be used for validation stage as it is executed for all tasks before first task is executed. Error messages at early stage, not in the middle of execution are very helpful in practice, increases quality of the automation.

### 7.11.10 Process isolation and permissions changing with sudo

Alternatively called "forking" is a feature of RKD similar to Gradle's JVM forking - the task can be run in a separate Python's process. This gives a possibility to run specific task as a specific user (eg. upgrade permissions to ROOT or downgrade to regular user)

#### Mechanism

RKD uses serialization to transfer data between processes - a standard `pickle` library is used. Pickle has limitations on what can be serialized - any inner-methods and lambdas cannot be returned by task.

To test if your task is compatible with running as a separate process simply add `--become=USER-NAME` to the commandline of your task. If it will fail due to serialization issue, then you will be notified with a nice stacktrace.

Technically the mechanism works on the task executor level, it means that process isolation is independent of the programming language as whole task's execute() is ran in a separate process, even if task is declared in YAML and has Bash steps.

#### Permissions changing with sudo

YAML syntax allows to define additional attribute `become`, that if defined then makes whole task to execute inside a separate Python process ran with sudo.

Additionally the RKD commandline supports a per-task parameter `--become`

#### Future usage

The mechanism is universal, it can be possibly used to sandbox, or even to execute tasks remotely. Currently we do not support such features but we do not say its impossible in the future.

### 7.11.11 Docker entrypoints under control

RKD has enough small footprint so that it can be used as an entrypoint in docker containers. There are a few features that are making RKD very attractive to use in this role.

### Environment variables

Defined commandline `--my-switch` can have optionally overridden value with environment variable. In docker it can help easily adjusting default values.

**Task needs to create an explicit declaration of environment variable:**

```
def get_declared_envs(self) -> Dict[str, ArgumentEnv]:
    return {
        'MY_SWITCH': ArgumentEnv(name='MY_SWITCH', switch='--switch-name', default=''),
    }
```

```
def execute(self, ctx: ExecutionContext) -> bool:
    # this one will look for a switch value, if switch has default value, then it will␣
↪look for an environment variable
    ctx.get_arg_or_env('--my-switch')
```

### Arguments propagation

When setting `ENTRYPOINT ["rkd", ":entrypoint"]` everything that will be passed as docker's CMD will be passed to rkd, so additional tasks and arguments can be appended.

### Tasks customization

It is a good practice to split your entrypoint into multiple tasks executed one-by-one. This gives you a possibility to create new Makefile in any place and modify `RKD_PATH` environment variable to add additional tasks or replace existing. The RKD_PATH has always higher priority than current `.rkd` directory.

**Possible options:**

- Create a bind-mount volume with additional `.rkd/makefile.yaml`, add `.rkd/makefile.yaml` into container and set RKD_PATH to point to `.rkd` directory
- Create new docker image having original in `FROM`, add `.rkd/makefile.yaml` into container and set RKD_PATH to point to `.rkd` directory

### Massive files rendering with JINJA2

`:j2:directory-to-directory` is a specially designed task to render JINJA2 templates recursively preserving a directory structure. You can create for example `templates/etc/nginx/nginx.conf.j2` and render `./templates/etc` into `/etc` with all files being copied on the fly.

**All jinja2 templates will have access to environment variables - with templating syntax you can define very advanced configuration files**

#### Privileges dropping

Often in entrypoint there are cache/uploads permissions corrected, so the `root` user is used. To migrate the application, to run the webserver the privileges could be dropped.

**Solutions:**

- In YAML syntax each task have a possible field to use: `become: user-name-here`
- In Python class TaskInterface has method `get_become_as()` that should return empty string or a username to use sudo with
- In commandline there is a switch `--become=user-name-here` that can be used with most of the tasks

### 7.11.12 Testing with PyTest

`rkd.core.api.testing` provides methods for running tasks with output capturing, a well as mocking RKD classes for unit testing of your task methods. To use our API just extend one of base classes.

#### Example: Running a task on a fully featured RKD executor

```
#!/usr/bin/env python3

import os
from rkd.core.api.testing import FunctionalTestingCase

SCRIPT_DIR_PATH = os.path.dirname(os.path.realpath(__file__))


class TestFunctional(FunctionalTestingCase):
    """
    Functional tests case of the whole application.
    Runs application like from the shell, captures output and performs assertions on the␣
↪results.
    """

    def test_tasks_listing(self):
        """ :tasks """

        full_output, exit_code = self.run_and_capture_output([':tasks'])

        self.assertIn(' >> Executing :tasks', full_output)
        self.assertIn('[global]', full_output)
        self.assertIn(':version', full_output)
        self.assertIn('succeed.', full_output)
        self.assertEqual(0, exit_code)
```

### Example: Mocking RKD-specific dependencies in TaskInterface

```
from rkd.core.api.inputoutput import BufferedSystemIO
from rkd.core.api.testing import FunctionalTestingCase

# ...

class SomeTestCase(FunctionalTestingCase):

    # ...

    def test_something_important(self):
        task = LineInFileTask()  # put your task class there
        io = BufferedSystemIO()

        BasicTestingCase.satisfy_task_dependencies(task, io=io)

        self.assertEqual('something', task.some_method())
```

### API

`class rkd.core.api.testing.`**`BasicTestingCase`**(*methodName='runTest'*)

**Provides minimum of:**

- Doing backup of environment and cwd
- Methods for mocking task dependencies (RKD-specific like ExecutionContext)

**`environment`**(*environ: dict*)

Mocks environment

**Example usage:**

```
with self.environment({'RKD_PATH': SCRIPT_DIR_PATH + '/../docs/examples/env-
↪in-yaml/.rkd'}):
    # code there
```

**Parameters** **`environ`** –

**Returns**

`static` **`list_to_str`**(*in_list: list*) → List[str]

Execute __str__ on each list element, and replace element with the result

`static` **`mock_execution_context`**(*task:* rkd.core.api.contract.TaskInterface, *args: Optional[Dict[str, Union[str, bool]]] = None*, *env: Optional[Dict[str, str]] = None*, *defined_args: Optional[Dict[str, dict]] = None*) → *rkd.core.api.contract.ExecutionContext*

Prepares a simplified rkd.core.api.contract.ExecutionContext instance

**Parameters**

- **`task`** –
- **`args`** –

- **env** –
- **defined_args** –

**Returns**

**static satisfy_task_dependencies**(*task:* rkd.core.api.contract.TaskInterface, *io: Optional[*rkd.core.api.inputoutput.IO*] = None*) → *rkd.core.api.contract.TaskInterface*

Inserts required dependencies to your task that implements rkd.core.api.contract.TaskInterface

**Parameters**

- **task** –
- **io** –

**Returns**

**setUp**() → None

Hook method for setting up the test fixture before exercising it.

**tearDown**() → None

Hook method for deconstructing the test fixture after testing it.

**class** rkd.core.api.testing.**FunctionalTestingCase**(*methodName='runTest'*)

Provides methods for running RKD task or multiple tasks with output and exit code capturing. Inherits OutputCapturingSafeTestCase.

**execute_mocked_task_and_get_output**(*task:* rkd.core.api.contract.TaskInterface, *args=None*, *env=None*) → str

Run a single task, capturing it's output in a simplified way. There is no whole RKD bootstrapped in this operation.

**Parameters**

- **task** (TaskInterface) –
- **args** (*dict*) –
- **env** (*dict*) –

**Returns**

**classmethod filter_out_task_events_from_log**(*out: str*)

Produces an array of events unformatted

```
[
    "Executing :sh -c echo 'Rocker' [part of :example]",
    "Executing :sh -c echo 'Kropotkin' [part of :example]",
    'Executing :sh -c echo "The Conquest of Bread"; exit 1 [part of :example]',
    'Retrying :sh -c echo "The Conquest of Bread"; exit 1 [part of :example]',
    'Executing :sh -c exit 0 ',
    'Executing :sh -c echo "Modern Science and Anarchism"; [part of :example]'
]
```

**Parameters** **out** –

**Returns**

**run_and_capture_output**(*argv: list*, *verbose: bool = False*) → Tuple[str, int]

Run task(s) and capture output + exit code. Whole RKD from scratch will be bootstrapped there.

**Example usage:** full_output, exit_code = self.run_and_capture_output([':tasks'])

**Parameters**

- **argv** (*list*) – List of tasks, arguments, commandline switches
- **verbose** (*bool*) – Print all output also to stdout

**Returns**

**with_temporary_workspace_containing**(*files: Dict[str, str]*)
Creates a temporary directory as a workspace and fills up with files specified in "file" parameter

**Parameters** **files** – Dict of [filename: contents to write to a file]

**Returns**

class rkd.core.api.testing.**OutputCapturingSafeTestCase**(*methodName='runTest'*)
Provides hooks for keeping stdout/stderr immutable between tests.

**setUp**() → None
Hook method for setting up the test fixture before exercising it.

**tearDown**() → None
Hook method for deconstructing the test fixture after testing it.

### 7.11.13 org.riotkit.rkd/yaml/v1 schema

**Tip:** Import this schema in your IDE for better static analysis of Makefiles written in YAML

YAML syntax marked with version `org.riotkit.rkd/yaml/v1` is validated using following schema:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "required": ["version"],
    "properties": {
        "version": {
            "type": "string",
            "minLength": 5
        },
        "imports": {
            "type": "array",
            "minItems": 0,
            "items": {
                "$ref": "#/definitions/import"
            }
        },
        "tasks": {
            "type": "object",
            "minItems": 0,
            "additionalProperties": {
                "$ref": "#/definitions/task"
            }
        },
```

(continues on next page)

(continued from previous page)

```
        "environment": {
            "type": "object",
            "minItems": 0
        },
        "env_files": {
            "type": "array"
        }
    },

    "definitions": {
        "task": {
            "type": "object",
            "properties": {
                "extends": {
                    "type": "string"
                },

                "description": {
                    "type": "string"
                },

                "arguments": {
                    "type": "object",
                    "additionalProperties": {
                        "$ref": "#/definitions/task-argument"
                    },
                    "minItems": 0
                },

                "steps": {
                    "type": ["array", "string"],
                    "minItems": 0
                },

                "environment": {
                    "type": "object",
                    "minItems": 0
                },

                "env_files": {
                    "type": "array"
                },

                "execute": {
                    "type": "string"
                },

                "execute@without_parent": {
                    "type": "string"
                },

                "execute@after_parent": {
```

(continues on next page)

(continued from previous page)

```
                    "type": "string"
                },

                "configure": {
                    "type": "string"
                },

                "configure@without_parent": {
                    "type": "string"
                },

                "configure@after_parent": {
                    "type": "string"
                },

                "inner_execute": {
                    "type": "string"
                },

                "inner_execute@without_parent": {
                    "type": "string"
                },

                "inner_execute@after_parent": {
                    "type": "string"
                }
            }
        },
        "task-argument": {
            "type": "object",
            "properties": {
                "help": {
                    "type": "string"
                },
                "required": {
                    "type": "boolean"
                },
                "action": {
                    "type": "string"
                },
                "metavar": {
                    "type": "string"
                },
                "type": {
                    "type": "string"
                },
                "nargs": {
                    "type": "string"
                },
                "default": {
                    "type": "string"
                },
```

(continues on next page)

(continued from previous page)

```
                "const": {
                    "type": "string"
                },
                "choices": {
                    "type": "array"
                },
                "dest": {
                    "type": "string"
                }
            }
        },
        "import": {
            "type": "string"
        }
    }
}
```

## 7.12 RKD Tech Specification

### 7.12.1 RTS-1: Extendable tasks

#### Abstract

Most of the existing build tools like GNU Make, SCons, Meson in opinion of RKD developers does not have enough good possibilities to share tasks across different projects, which means those tools are not scaling well.

#### Motivation

In order to provide a perfect DevOps tool that will allow sharing the code of universal mechanisms between projects, even organizations this RKD Tech Specification was designed.

Inspired how it looks in Gradle we decided to create a simplified DevOps tool that will be universal, will allow to install any task set as a Python package and then use it to manage databases, servers, build projects, generate configs and all other things that could be automated, parametrized.

#### Rationale

Common practice is to extract complex and universal mechanisms to separate packages, in our case it is a Python package. Packages can be shared across projects, even organizations, using already good and known mechanism - PyPI/PIP and Virtual Environment.

Inside project structure an already prepared mechanism can be imported from an installed package, then local project tasks can be created with already prepared configuration that is specific to the local project.

### Vocabulary

- `Task`: `Task` that actually runs, can be invoked and will produce result.
- `Base Task`: Task that acts like a template, other `Task` needs to be created from it and properly configured.
- `Decorator`: Marks an extended method that it should be executed instead of parent method, or before parent method, or after parent method. No decorator means replacing parent method. Internally in RKD Core called also `Markers`.

### Base tasks vs Customizations

Base Tasks are possible to be defined ONLY as Python classes inside Python modules (modules can be also local). The actual Tasks, the Customizations are defined in a simplified Python syntax or in YAML document syntax, those cannot be extended again. Regular Tasks are also possible to be written in pure Python as classes, there are no limits.

**Example of a Task that extends a Base Task, which means it is a Customization of a Base Task:**

```yaml
version: org.riotkit.rkd/yaml/v1
imports:
    - rkd.php.script.PhpScriptTask
tasks:
    :yaml:test:php:
        extends: rkd.php.script.PhpScriptTask
        # @before_parent is a Decorator, there could be only one decorator used
        configure@before_parent: |
            self.version = '7.2-alpine'
        inner_execute@after_parent: |
            print('IM AFTER PARENT')
            return True
        input: |
            # this is a PHP language
            var_dump(getcwd());
            var_dump(phpversion());
```

To create project-specific Base Task that extend other Base Task there is a requirement to define it as a Python class, and do it in a Pythonic way.

**Example of a Base Task that extends other Base Task:**

```python
import os
from typing import Optional

from rkd.core.execution.lifecycle import ConfigurationLifecycleEvent
from rkd.core.standardlib.docker import RunInContainerBaseTask


class PhpScriptTask(RunInContainerBaseTask):
    """
    Execute a PHP code (using a docker container)
    Can be extended - this is a base task.

    Inherits settings from `RunInContainerBaseTask`.

    Configuration:
```

(continues on next page)

(continued from previous page)

```
        script: Path to script to load instead of stdin (could be a relative path)
        version: PHP version. Leave None to use default 8.0-alpine version

    """

    script: Optional[str]
    version: Optional[str]

    def __init__(self):
        super().__init__()
        self.user = 'www-data'
        self.entrypoint = 'sleep'
        self.command = '9999999'
        self.script = None
        self.version = None

    def configure(self, event: ConfigurationLifecycleEvent) -> None:
        # please note: there is parent method called - RunInContainerBaseTask.
↪configure(event)
        super().configure(event)

        self.docker_image = '{image}:{version}'.format(
            image=event.ctx.get_arg_or_env('--image'),
            version=self.version if self.version else event.ctx.get_arg_or_env('--php')
        )

        self.mount(local=os.getcwd(), remote=os.getcwd())

    # ...
```

### Syntax

There exists actually three available syntax styles.

---

1. Python Class: Classic syntax

---

Classic syntax has no limits, it’s main purpose is to define Base Tasks, that could be extended later **due to its native construct could be packaged as PyPI/PIP package.**

```
import os
from argparse import ArgumentParser
from rkd.core.api.syntax import TaskDeclaration
from rkd.core.api.contract import TaskInterface, ExecutionContext

class GetEnvTask(TaskInterface):
    """Gets environment variable value"""

    def get_name(self) -> str:
        return ':get'
```

(continues on next page)

(continued from previous page)

```python
    def get_group_name(self) -> str:
        return ':env'

    def configure_argparse(self, parser: ArgumentParser):
        parser.add_argument('--name', '-e', help='Environment variable name',␣
↪required=True)

    def execute(self, context: ExecutionContext) -> bool:
        self.io().out(os.getenv(context.get_arg('--name'), ''))

        return True


IMPORTS = [
    TaskDeclaration(GetEnvTask())
]
```

2. Simplified Python syntax

Allows writing Tasks that extends Base Tasks in a very easy and short manner.

```python
from rkd.core.api.contract import ExecutionContext
from rkd.core.api.syntax import ExtendedTaskDeclaration
from rkd.core.api.decorators import before_parent, without_parent, after_parent, extends
from rkd.core.execution.lifecycle import ConfigurationLifecycleEvent
from rkd.php.script import PhpScriptTask

@extends(PhpScriptTask)
def MyTask():
    @without_parent
    def configure(task: PhpScriptTask, event: ConfigurationLifecycleEvent):
        task.version = '7.2-alpine'

    def inner_execute(task: PhpScriptTask, ctx: ExecutionContext):
        print('IM AFTER PARENT')
        return True

    def stdin():
        return """
            var_dump(getcwd());
            var_dump(phpversion());
        """

    return [configure, inner_execute, stdin]

IMPORTS = [
    ExtendedTaskDeclaration(name=':php', task=MyTask)
]
```

3. Document/YAML syntax

---

Has similar purpose as `Simplified Python syntax`, but should be simpler for non-programmers like System Administrators, or just for people that likes YAML declarations.

```
version: org.riotkit.rkd/yaml/v1
imports:
    - rkd.php.script.PhpScriptTask
tasks:
    :yaml:test:php:
        extends: rkd.php.script.PhpScriptTask
        configure@before_parent: |
            self.version = '7.2-alpine'
        inner_execute@after_parent: |
            print('IM AFTER PARENT')
            return True
        input: |
            var_dump(getcwd());
            var_dump(phpversion());

    # defining classic shell tasks is easiest with YAML syntax
    # "bash" and "python" can be also replaced with a full package name + class that␣
↪implements executing code in other language e.g. rkd.php.script.PhpScriptTask
    :yaml:test:multi:
        steps:
            - |
                #!bash
                echo "Hello world from Bash"
            - |
                #!python
                print("Hello from Python")
            - ps aux
            - ls -la
```

## Execute and Inner Execute concept

- `def execute(ctx: ExecutionContext) -> bool` is a main method that performs action of a task, as a result a boolean should be returned.
- `def inner_execute(ctx: ExecutionContext) -> bool` is a method that OPTIONALLY can be called by implementation of `execute()` method, to perform some e.g., transactional task

Base Tasks can implement a `execute()` and leave a possibility for a Customizations by calling `inner_execute(ctx)` from the inside of `execute()`, but not every Base Task may implement this. You need to carefully read docs for given Base Task.

**What are the cases for inner_execute?** - execute() launches a docker container, invokes `inner_execute()`, then removes the container. This allows to use the container from inside of `inner_execute(ctx)` method - execute() prepares required files, then invokes `inner_execute()` to perform some user-defined action, at the end cleans the workspace

### Table of method names

Despite three different syntax styles, there are slight differences the developer/ops needs to be aware of.

| Simplified Python | Python Class | YAML | Description |
|---|---|---|---|
| get_steps(task: MultiStepLanguageAgnosticTask) -> List[str]: | get_steps | steps: [""] | List of steps in any language (only if extending MultiStep LanguageAgnosticTask) |
| stdin() | • | input: "" | Standard input text |
| @extends(ClassName) decorator on a main method | ClassName(BaseClass) | extends: package.name.ClassName | Which Base Task should be extended |
| execute(task: BaseClassNameTask, ctx: ExecutionContext): | execute(self, ctx: ExecutionContext) | execute: "" | Python code to execute |
| inner_execute(task: BaseClassNameTask, ctx: ExecutionContext): | inner_execute(self, ctx: ExecutionContext) | inner_execute: "" | Python code to execute inside inner_execute (if implemented by Base Task) |
| compile(task: BaseClassNameTask, event: CompilationLifecycleEvent): | compile(self, event: CompilationLifecycleEvent): | None | Python code to execute during Context compilation process |
| configure(task: BaseClassNameTask, event: ConfigurationLifecycleEvent): | configure(self, event: ConfigurationLifecycleEvent): | configure: "" | Python code to execute during Task configuration process |
| get_description() | get_description(self) | description: "" | Task description |
| get_group_name() | get_group_name() | None | Group name |
| internal=True in TaskDeclaration | internal=True in TaskDeclaration | internal: False | Is task considered internal? (hidden on :tasks list) |
| become in TaskDeclaration (or commandline switch) | become in TaskDeclaration (or commandline switch) | become: root | Change user for task execution time |
| workdir in TaskDeclaration | workdir in TaskDeclaration | workdir: /some/path | Change working directory for task execution time |
| configure_argparse(task: BaseClassNameTask, parser: ArgumentParser) | configure_argparse(self, parser: ArgumentParser) | arguments: {} | Configure argparse.ArgumentParser object |

## 7.13 Development

RKD due to its technical and elastic nature is very abstract inside, therefore core concepts are explained in this chapter.

### 7.13.1 Input/Output

Every Task has its own instance of *IO*. Task compilation, context preparation stages are using *SystemIO*, so the *IO* configuration is having settings defined with commandline switches **before first task**.

**Reasons:**

- Each task can have different logging level
- Each task can log to different file (even cannot write to same file)
- Task has separate IO settings than RKD global UI
- –*no-ui* before first task disables RKD interface messages like *Successfully executed 2 tasks* but keeps interface produced by Task eg. >> *chown www-data:www-data /tmp/script.php*
- –*silent* before first task disables ALL interfaces both RKD and produced by Task. Only necessary messages are printed

**Global logging level - before first task examples**

```
./rkdw --no-ui :first-task :second-task

# defines log level on very early stage, before arguments parsing. Can be set to any␣
↪level including debug, info, warning, error
# "internal" is a level that contains internal RKD core debugging messages. Warning:␣
↪There could be a lot of messages
# use "debug" to debug your tasks
RKD_SYS_LOG_LEVEL=internal ./rkdw :first-task :second-task
```

**Logging levels:**

- internal: Includes RKD core internal messages
- debug: Includes task-related debugging messages
- info: User info messages
- warning: Warnings
- error: Errors
- fatal: Fatal errors

### 7.13.2 Lifecycle entities

Internally RKD has three types of objects that are used across the application - Task creation, Task usage declaration, Task execution scheduling.

### 1) Task Creation

`TaskInterface` implementations are considered to provide **importable Tasks** to be used in any automation project.

Example: I have `PostgreSQLRunTask` and I import it as `:pgsql:start`

```
# ...

class RenderDirectoryTask(TaskInterface):
    """Renders *.j2 files recursively in a directory to other directory"""

    def get_name(self) -> str:
        return ':directory-to-directory'

    def get_group_name(self) -> str:
        return ':j2'

    def execute(self, context: ExecutionContext) -> bool:
        # ...
```

Tasks should be defined mainly as part of installable libraries via PyPI, but could be also defined in local repository.

### 2) Task usage declaration - importing & preconfiguring Tasks in project code

`TaskDeclaration` declares that imported `TaskInterface` implementation would be used in our automation project under some name, with some environment variables, custom workspace and other little customizations that does not involve changing the code of imported Task.

`Pipeline`, `PipelineTask` and `PipelineBlock` defines complete Pipelines, with error handling, notifications, list of Tasks to execute.

**That's called static declaration of reproducible usage. Tasks are imported into a project defined in code, each Task is preconfigured and ready to be used in reproducible way.**

```
from rkd.core.api.syntax import Pipeline, PipelineTask as Task, PipelineBlock as Block,␣
↪TaskDeclaration
from rkd.core.standardlib.core import DummyTask
from rkd.core.standardlib.shell import ShellCommandTask

IMPORTS = [
    TaskDeclaration(ShellCommandTask(), internal=True)
]

PIPELINES = [
    Pipeline(
        name=':example',
        to_execute=[
            Block(rescue='...', tasks=[
                Task('...'),
            ]),
            Task('...'),
        ]
    )
]
```

#### 3) Runtime Task scheduling

Imported Tasks and declared for usage in a project are processed, when executed. This later stage is invisible to end-user and is performed internally on runtime, the entities are not known to the user.

Internally RKD must wrap any `TaskDeclaration` and process `Pipeline` into lower-level entities on first stage - resolving & compilation stage.

`TaskDeclaration` is wrapped by `DeclarationScheduledToRun` that **mixes** environment, arguments and other options **declared in code with everything that exists during execution that takes place now.**

`Pipeline` is translated into `GroupDeclaration` and associated `ArgumentBlock` objects which are no longer strings like `:db:start --listen=5432`, but are separate `TaskDeclaration` objects. There are all `@rescue` and `@error` modifiers resolved into objects, so everything is calculated on very early stage and therefore can be validated without disrupting later execution by any simple errors.

Each `TaskDeclaration` in Pipeline is wrapped into `DeclarationBelongingToPipeline` which acts very similar to `DeclarationScheduledToRun`, but it was named differently to distinct between something that was declared in the code (`DeclarationBelongingToPipeline`) from something that contains a set of information how the user invoked the command from shell (`DeclarationScheduledToRun`)

## 7.13.3 Task lifetime stages

#### 1) Construction

Tasks are created and imported into the `ApplicationContext`. Every `.rkd` directory context is parsed into `ApplicationContext`, then all contexts are merged into an unified `ApplicationContext`.

#### 2) Compilation

Unified `ApplicationContext` is compiled, compilation does two things:

1. Resolving all Pipelines into Groups of resolved Tasks
2. Executing `compile()` on all defined Tasks in `ApplicationContext`, regardless if they are called

#### 3) Configuration

`configure()` method is triggered on each Task that is scheduled to be executed.

#### 4) Execution

`execute()` method is triggered on each Task that is scheduled to be executed.

#### 5) Teardown

To be done. Not implemented yet.

# INDEX