simple_automation

oddlama

May 29, 2021

CONTENTS

Welcome to the official documentation of simple_automation! If you are visiting for the first time, we recommend that you read the *Introduction* page to get an overview of what simple_automation has to offer.

Otherwise, the sidebar and the table of contents below should let you easily access your topic of interest. You can also use the search function in the top left corner.

CHAPTER

ONE

INTRODUCTION

Simple automation is an ansible-inspired infrastructure and configuration management tool, but with a focus on minimalism and simplicity. Its intended uses are small-scale machine administation, system configuration tracking, or even just dotfiles management. Although it can certainly be used to administer larger infrastructures.

Hint: If you want a high level overview of the different library components and how they work together, please have a look at the *Architecture* page.

Here is an example of how to use simple automation to manage some global config files.

site.py

```
#!/usr/bin/env python3
from simple_automation import run_inventory, Inventory, Task
from simple_automation.transactions.basic import copy, directory, template

# -------- Define a task --------
class TaskZsh(Task):
    identifier = "zsh"
    description = "Installs global zsh configuration"

    def run(self, context):
        # Change permission defaults
        with context.defaults(umask=0o022, dir_mode=0o755, file_mode=0o644):
            # Template the zshrc, copy the zprofile
            directory(context, path="/etc/zsh")
            template(context, src="templates/zsh/zshrc.j2", dst="/etc/zsh/zshrc")
            copy(context, src="files/zsh/zprofile", dst="/etc/zsh/zprofile")

# -------- Setup your inventory --------
class MySite(Inventory):
    tasks = [TaskZsh]

    def register_inventory(self):
        self.manager.add_host("my_home_pc", ssh_host="root@localhost")

    def run(self, context):
        context.run_task(TaskZsh)
```

(continues on next page)

(continued from previous page)

```
# -------- Run the inventory --------
if __name__ == "__main__":
    run_inventory(MySite)
```

templates/zsh/zshrc.j2

```
# This is a zshrc for host {{ context.host.identifier }}
```

Possible output:

```
[>] Establishing ssh connection to root@localhost

[*] >>>> Task: zsh <<<<
[+] dir        /etc/zsh                      mode: 750 → 755
[+] template   /etc/zsh/zshrc                sha512sum: 3d48b060... → fac69135...
[.] copy       /etc/zsh/zprofile
```

1.1 Feature Overview

- Use python to write your configuration and don't be limited by a domain specific language.
- Supports encrypted variable storage.
- Executes all commands over a single ssh connection per host (→ fast execution)
- Concicse, readable output (optionally verbose but still quite compact)
- Tracking of arbitrary files and folders in a git repository. Allows you to easily keep track of your system's state over time.

1.1.1 Focus on minimalism and simplicity

This library has a very specific purpose: Conveniently make it possible to alter the state of remote hosts by executing commands over an SSH connection. We don't want to increase complexity by introducing niche features. We will provide everything that is necessary to get there, plus some convenience features that most users need (e.g. vaults). Because this library is small, you will be capable of building the rest yourself, if you require more.

The heart of this library is the remote execution framework and the transaction system, which consists of less than 1000 LoC. The rest is mostly predefined transactions which are provided for ease of use. Additionally, it has very few dependencies:

- jinja2 for templating
- pycryptodome for symmetrically encrypted vaults (scrypt + AES-256-GCM; library not required if feature not used).

Warning: Currently there is no supported way to become a privileged user on a host, when logging in as a unprivileged user. If you want to do things as root or an arbitrary user on a remote host, you will need to login as root.

1.2 Installation

Use can use pip to install simple_automation. If you want to help maintaining a package for your favourite distribution, feel free to reach out.

You will need `python>=3.9`.

Using pip

```
pip install simple_automation
```

Hint: Have a look at *Setting up a basic project* for an overview of how to setup a new inventory.

CHAPTER

TWO

EXAMPLES

In the following we will show some examples of how to use simple_automation.

2.1 Setting up a basic project

Typically, a project has a structure similar to the following:

```
my_project/
├── templates/          -- File templates
│   └── zsh/            -- (Best sorted by task)
│       └── zshrc.j2
├── tasks/              -- Task definitions
│   ├── __init__.py
│   └── zsh.py
├── myvault.asc         -- (Optional) Encrypted variable storage
└── site.py*            -- Your inventory and main executable
```

You will mainly have to deal with template files, task definitions and maintaining your inventory script. `site.py` is your main executable, and is the place where you will define your inventory, and where you will find the main execution routine.

site.py

```python
#!/usr/bin/env python3
from simple_automation import run_inventory, Inventory, SymmetricVault
from tasks.my_simple_task import MySimpleTask

# -------- Define your inventory --------
class MyInventory(Inventory):
    tasks = [MySimpleTask]

    # (Optional) Vaults store encrypted variables
    def register_vaults(self):
        # -------- Load vault --------
        self.vault = self.manager.add_vault(SymmetricVault, file="myvault.asc")

    # (Optional) Global variables
    def register_globals(self):
        # -------- Set global variables --------
        self.manager.set("tasks.my_simple_task.enabled", False)
```

```
    def register_inventory(self):
        # -------- Define Groups --------
        desktops = self.manager.add_group("desktops")
        desktops.set("system.is_desktop", True)
        desktops.copy("system.root_pw", self.vault)

        # -------- Define Hosts --------
        my_home_pc = self.manager.add_host("my_home_pc", ssh_host="root@localhost")
        my_laptop.add_group(desktops)

    def run(self, context):
        context.run_task(MySimpleTask)

# -------- Run the inventory --------
if __name__ == "__main__":
    run_inventory(MyInventory)
```

2.2 Defining a task

To create a new task, start by inheriting from *simple_automation.task.Task*. You need to specify an identifier for your task, which will for example be used for related variables. The description will be printed in verbose mode.

tasks/my_simple_task.py

```
from simple_automation import Task
from simple_automation.transactions.basic import copy, directory

# -------- Define a task --------
class MySimpleTask(Task):
    identifier = "mytask"
    description = "Just copies some files"

    def run(self, context):
        # Change permission defaults
        with context.defaults(umask=0o022, dir_mode=0o755, file_mode=0o644):
            # Create a directory and copy some files
            directory(context, path="/etc/zsh")
            copy(context, src="files/zsh/zshrc", dst="/etc/zsh/zshrc")
            copy(context, src="files/zsh/zprofile", dst="/etc/zsh/zprofile")
```

Hint: Default for user, umask, file/dir modes, file owner/group are strict by default. If not changed explicitly as shown above, the task will use `user='root', umask=0o077, dir_mode=0o700, file_mode=0o600, owner='root', group='root'`.

2.3 Task specific variables

You can define variables for your tasks, which you can use to customize e.g. installation paths, or to conditionally enable certain functionality. In *set_defaults()* you can define what default values your variables should have, if they are not overwritten by any globals, group variables or host variables.

Each task has an identifier. If you always use this identifier as part of your variable name, you can avoid clashes with other task variables.

Hint: All tasks automatically expose a variable named `tasks.{identifier}.enabled`, which you can use to conditionally disable a whole task.

Example:

```
from simple_automation import Task
from simple_automation.transactions.basic import template

class MyTask(Task):
    identifier = "mytask"
    description = "A short description"

    def set_defaults(self):
        self.manager.set(f"tasks.{self.identifier}.config_folder", "/etc/mytask")

    def run(self, context):
        # Use variables in templated parameters:
        template(context, src="templates/mytask/template.j2", dst="{{ tasks.mytask.
↪config_folder }}/config")

        # Use variables as a conditional
        if context.vars.get("tasks.mytask.some_boolean"):
            # ...
```

templates/mytask/template.j2

```
# This file's path is {{ tasks.mytask.config_folder }}/config
# and is saved on host {{ context.host.identifier }}
```

2.4 Global variables

You can set global variables by calling `self.manager.set()`. This is mainly helpful if you want to create customization points for your own global inventory *run()* routine.

```
from simple_automation import Inventory

class MyInventory(Inventory):
    # ...
```

(continues on next page)

(continued from previous page)

```
    def register_globals(self):
        # -------- Set global variables --------
        self.manager.set("install_dotfiles", False)

    def run(self, context):
        if context.vars.get("install_dotfiles"):
            # ...
```

2.5 Groups

If you have multiple hosts with related configuration needs, you can add them to groups to manage this common functionality. You might for example want to add all desktop machines into one group to install common software that you need on all of those hosts.

```
from simple_automation import Inventory

class MyInventory(Inventory):
    # ...

    def register_inventory(self):
        # -------- Define Groups --------
        self.desktops = self.manager.add_group("desktops")
        self.desktops.set("system.is_desktop", True)

        # -------- Define Hosts --------
        my_home_pc = self.manager.add_host("my_home_pc", ssh_host="root@localhost")
        my_home_pc.add_group(self.desktops)

    def run(self, context):
        # ...

        # Check if the current host belongs to a group
        if context.host in self.desktops:
            pass

        # Or examine a variable you set for that group
        if context.vars.get("system.is_desktop"):
            pass
```

2.6 Using transaction results

Sometimes you will need results from past transactions to determine what to do next. For example you might need to run some transactions only if a directory was created.

All transaction return an object of type *CompletedTransaction* which you can use to examine the initial and final transaction state.

Conditional execution based on directory creation state

```
from simple_automation import Task
from simple_automation.transactions.basic import directory

class MyTask(Task):
    # ...
    def run(self, context):
        # ...
        res = directory("/some/directory")
        if not res.initial_state["exists"]:
            # Directory didn't exist before
            # Do some additional work
```

2.7 Conditional execution based on command output

You might find yourself in the situation where you need the output of an arbitrary command, or a file on the remote system to determine the next steps. This can be done by directly executing a command on the remote system via the given context.

Hint: The method *context.remote_exec()* works similar to `subprocess.run()`, but is executed on the remote host. Please view the method documentation to see which parameters are avaiable.

Executing a remote command

```
from simple_automation import Task

class MyTask(Task):
    # ...
    def run(self, context):
        # ...
        remote_content = context.remote_exec(["cat", "/path/to/some/file"],␣
↪checked=True)
        content = remote_content.stdout
        # Use the content in your logic.
```

2.8 Tracking files

You can have tasks automatically check some files or directories into a git repository, so you can keep track of your system's state over time. This is as simple as deriving from *TrackedTask* instead of *Task*, and defining some additional class variables. Be sure to have a look at the documentation of *TrackedTask* to see which options are available.

Warning: Your chosen tracking repository should already have at least one commit. This is necessary because only then there will be a tracked branch when checking it out initially.

Hint: It may be beneficial to create your own base class for all tracked tasks, to set a common tracking repository. You will then only have to add all files and directories you want to track to *tracking_paths* in the actual task.

Define a common base task

```
from simple_automation import TrackedTask

class MyTrackedTask(TrackedTask):
    # Save the url into a vault so it doesn't leak into your management repository
    tracking_repo_url = "{{ tracking.repo_url }}"
    # Choose some path where the actual tracking repository will be cloned on your␣
↪machines
    tracking_local_dst = "/var/lib/root/tracking"
```

Track some files

Simply extend any of your task by inheriting from your new base task, then set the files and/or directories you want to track.

```
class TaskZshConfig(MyTrackedTask):
    tracking_paths = ["/etc/zsh"]
    # ...
```

A tracking-only task

It is perfectly valid to create a new task that does nothing but track some files.

```
class TaskTrackSomething(MyTrackedTask):
    identifier = "track_something"
    description = "Tracks something"
    tracking_paths = ["/etc/location1", "/var/lib/something_else"]
```

Track arbitrary information

You can also track arbitrary information, by querying this information in your tasks *run()* function and save it into a temporary destination.

```
# Track installed packages from portage
class TaskTrackInstalledPackages(MyTrackedTask):
    identifier = "track_installed_packages"
    description = "Tracks all installed packages"
    tracking_paths = ["/var/lib/root/installed_packages"]

    def run(self, context):
        # Change the command to fit your package manager
        save_output(context, command=["qlist", "-CIv"],
                    dst="/var/lib/root/installed_packages",
                    desc="Query installed packages")
```

2.9 Vaults

Vaults let you store variables in an encrypted file. This is useful when you want to safely store secrets in your management repository. By default we offer symmetrically encrypted vaults (scrypt+AES-256-GCM), or gpg encrypted vaults (convenient in combination with a smartcard or YubiKey).

For specific information on each, have a look at the respective class documentations:

- *SymmetricVault*
- *GpgVault*

A vault is just a variable storage, and therefore works similar to other variable storages like groups or hosts.

Creating/Editing a vault

If you have defined a vault, you can use `./site.py --edit-vault <vault_file>` to edit it. This will open `$EDITOR` and show the vault content in JSON format.

Using a vault

```
#!/usr/bin/env python3
from simple_automation import Inventory, SymmetricVault
from tasks.my_simple_task import MySimpleTask

# -------- Define your inventory --------
class MyInventory(Inventory):
    # ...
    def register_vaults(self):
        # You can optionally pass the unlock key / keyfile if needed
        self.vault = self.manager.add_vault(SymmetricVault, file="myvault.asc")
        # You may define multiple vaults. Store them in your instance to access them␣
↪later.

    def register_inventory(self):
        # ...
        # Copy root password from vault
        my_laptop.copy("system.root_pw", self.vault)
```

Creating a GpgVault

```
# ...
def register_vaults(self):
    self.vault = self.manager.add_vault(GpgVault, file="myvault.gpg", recipient="your_
↪keyid")
```

Hint: Use *copy()* to easily copy a variable from a vault into your globals, group or host variables.

CHAPTER

THREE

TRIVIA

3.1 Variable precedence

Global variables will be evaluated once at the start of the program:

1. Tasks will write their defaults to the global variables
2. Your `register_globals()` will be called.

When a new context is created (to execute your `run()` function), variables will be merged in the following order (lower entries overriding upper):

1. Globals
2. Groups in the order they have been added to the host
3. Host variables

3.2 Implicit templating variables

Management reminder with `simple_automation_managed`

This implicit global variable expands to a message stating that a file is managed by simple automation. Just put a comment like `# {{ simple_automation_managed }}` in the first line of your templates to use it.

Of course you can simply overwrite this in your globals or add your own variable if you want to customize the message.

Accessing the current host object directly from templates

You can directly access the python manager / host object in templated environments.

- `context.manager` will be the manager object.
- `context.host` will be the host attached to the current context.

3.3 Tasks have an implicit enabled variable

Tasks have an implicit enabled variable. Often you need a variable to enable or disable different tasks. Therefore, all tasks expose a variable called `tasks.{identifier}.enabled`, which by default is set to `True`. The `{identifier}` is whatever has been choosen as the task identifier.

Example

```
class MyInventory(Inventory):
    tasks = [MySimpleTask]

    def register_globals(self):
        # ...
        # Don't run MySimpleTask by default
        self.manager.set("tasks.my_simple_task.enabled", False)

    def register_inventory(self):
        # Do run the task only for hosts in the desktops group
        desktops = self.manager.add_group("desktops")
        desktops.set("tasks.my_simple_task.enabled", True)
        # ...

    def run(self, context):
        context.run_task(MySimpleTask)
```

3.4 Context remote execution defaults

By default, when a task is run, the context's default values are set to the following (strict) values:

Variable	Default	Description
user	`"root"`	User to execute commands as
umask	`0o077`	File system umask value
dir_mode	`0o700`	Permissions for directories created by *`directory()`* and alike.
file_mode	`0o600`	Permissions for files created by *`copy()`*, *`template()`* and alike.
owner	`"root"`	File/directory owner
group	`"root"`	File/directory group

It is recommended to always specify these defaults at the beginning of your task, so you know exactly what to expect.

3.5 Predefined transactions

See *Transactions* for an overview of available transactions.

3.6 Executing only a part of the script

If you separate your inventory *`run()`* method into several smaller methods, you will be able to run them individually. This can be beneficial especially for large scripts.

You can select methods to execute by passing them as a comma separated list to the `--scripts` command line option. They will be executed in order. If not given, the parameter `--scripts run` is assumed.

3.7 A unified `package()` command for different distributions

If you manage hosts with different distributions, it might be beneficial to create a wrapper around the `package()` transaction, which will chose the correct one for your hosts. This is as simple as:

```
# When defining your inventory
def register_inventory(self):
    distro_debian = self.manager.add_group("debians")
    distro_debian.set("system.distribution", "debian")

    distro_arch = self.manager.add_group("arch")
    distro_arch.set("system.distribution", "arch")

    # For all your hosts add them to the correct group
    my_host.add_group(distro_arch)

# And define a global transaction wrapper
def package(context, **kwargs):
    distro = context.vars.get("system.distribution")
    if distro == "arch":
        arch.package(**kwargs)
```

(continues on next page)

(continued from previous page)

```
    elif distro == "debian":
        apt.package(**kwargs)

# Now simply use package() in your tasks.
```

This approach is very flexible and would also allow you to e.g. add certain system dependent paths to these group settings to make your tasks work on any distribution.

3.8 Check if a host belongs to a group

```
def run(context):
    if self.my_host in self.some_group:
        # ...

    # Alternatively:
    if self.my_group in self.my_host.groups:
        # ...
```

3.9 Project to track dotfiles

You can simply create a single task that tracks all the locations you want to backup. Occasionally run the script and all your paths will be checked into a git repository.

3.10 Where to store secrets

Beware where and how you use secrets. If you have secrets, you should only ever store them in a vault so they won't appear in your management repository! See *Vaults* for information on how to use vaults.

Warning: Be careful, remote commands and their output may be printed in verbose or debugging modes! If you want to be certain that no secrets will ever be printed, only send them via `input=...` to the remote host in *remote_exec()*, or use them in files templated via *template()*.

You can easily copy secrets from a vault into any variable storage by using *copy()*.

```
def register_inventory(self):
    # Copy into globals
    self.manager.copy("some_variable", self.vault)
    # Copy into group variables
    self.my_group.copy("some_variable", self.vault)
    # Copy into host variables
    self.my_host.copy("some_variable", self.vault)
```

3.11 Setting additional instance variables

There are two ways of associating additional information with a host:

```
# Accessed via {{ var }} in templated contexts
self.my_host.set("var", "value")
```

```
# Accessed via {{ context.host.var }} in templated contexts
self.my_host.var = "value"
```

Both are fine, while the first might be more flexible, as it will allow you to inherit from global or group variables. You can use both approaches to access to arbitrary python objects from templated contexts, by using any object as the value.

3.12 Relative local paths

Local files given by `src=` in *copy()* or *template()* are relative to the project path where your main executable resides. You can override that behavior by passing a `main_directory` to *run_inventory()*.

3.13 Asking only once for a vault key for multiple vaults

You can either use a keyfile, or ask yourself for the password before running your inventory:

```
from simple_automation import Inventory, SymmetricVault, run_inventory
import getpass

global_key = None

# -------- Define your inventory --------
class MyInventory(Inventory):
    # ...
    def register_vaults(self):
        self.vault1 = self.manager.add_vault(SymmetricVault, file="vault1.asc",␣
↪key=global_key)
        self.vault2 = self.manager.add_vault(SymmetricVault, file="vault2.asc",␣
↪key=global_key)

# -------- Run the inventory --------
if __name__ == "__main__":
    global_key = getpass("Shared vault key: ")
    run_inventory(MyInventory)
```

Warning: The downside of this approach is that you will have to unlock your vault every time, even when you would for example edit any unrelated other vault.

CHAPTER

FOUR

FREQUENTLY ASKED QUESTIONS

4.1 Only show what changes would be applied to the remote systems

Use the `--pretend` command line parameter.

4.2 Only run on some of the defined hosts

Use the `--hosts` command line parameter.

4.3 Errors when trying to track files

Have you made sure that your tracking repository already has a commit? This is necessary because only then there will be a tracked branch when checking it out initially. If you delete the repository on your remote host, it will be checked out again the next time.

4.4 Change ssh port / parameters for a host

You can customize the ssh connection options with *set_ssh_port()* and *set_ssh_opts()*.

```
def register_inventory(self):
    my_host = self.manager.add_host("my_host", ssh_host="root@localhost")
    my_host.set_ssh_port(2222)
    my_host.set_ssh_opts(["-J", "jumphost@example.com"])
```

CHAPTER

FIVE

OVERVIEW

The *Manager* and *Context* are two of the main classes you will interact with. The Manager is used to register your inventory and the context is used to execute commands on remote hosts.

simple_automation.manager.Manager(...[, ...])	A class that manages a set of global variables, hosts, groups, and tasks.
simple_automation.context.Context(manager, host)	A context is a wrapper object around a host and an ssh connection to that host.

The following classes are instanciated by registering things in your inventory and are also used regularly. The most important classes are the *Task* and *TrackedTask* from which you can derive your own tasks.

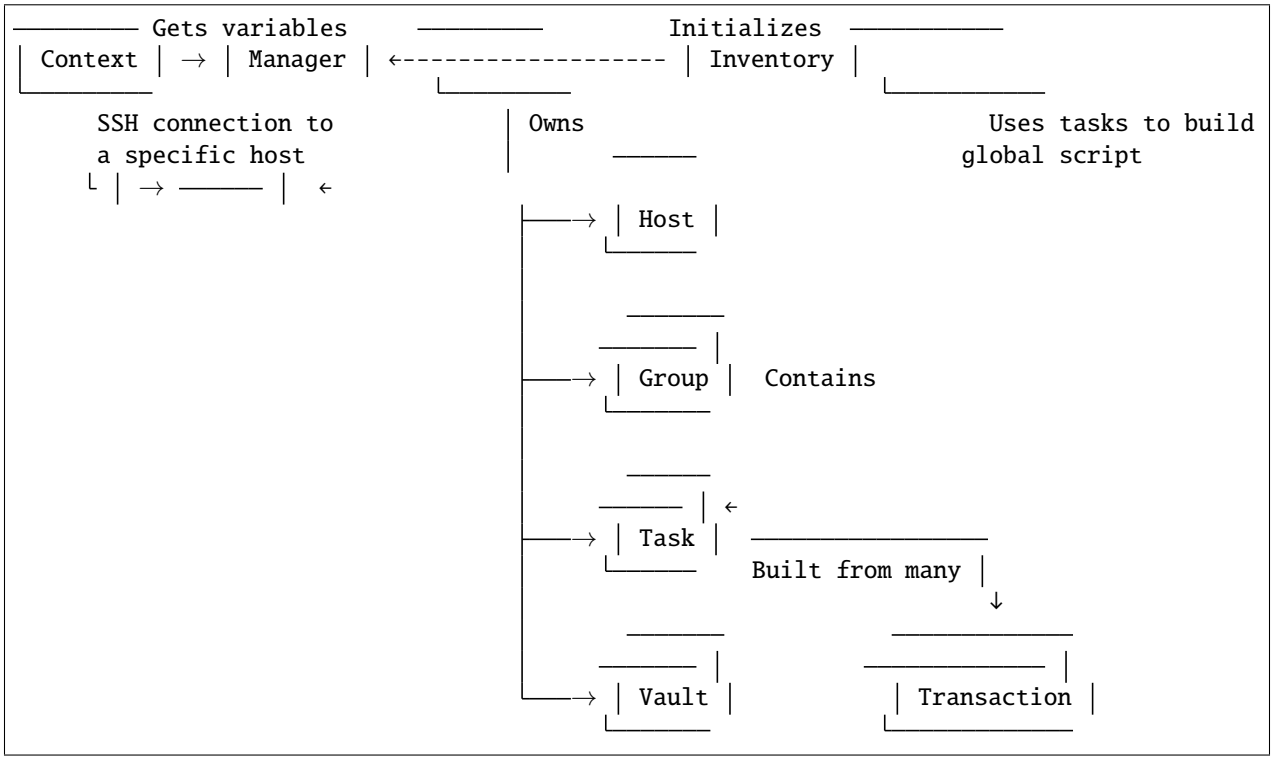
simple_automation.inventory.Inventory(manager)	An inventory is a collection of definitions for vaults, tasks, global variables, hosts and groups.
simple_automation.host.Host(manager, ...)	A Host represents a machine that can be reached via ssh, and which should be managed by simple_automation.
simple_automation.group.Group(manager, ...)	A group for hosts.
simple_automation.task.Task(manager)	A base class for tasks.
simple_automation.task.TrackedTask(manager)	A base class for tasks which want to track changes in a git repository.
simple_automation.vault.GpgVault(manager, ...)	A GpgVault is a Vault which saves its context encrypted with gpg.
simple_automation.vault.SymmetricVault(...)	A SymmetricVault is a Vault which saves its context symmetrically encrypted.

CHAPTER

SIX

ARCHITECTURE

In this document you will find a high-level overview of the library architecture.



6.1 Program start

1. A manager (and your inventory) is instanciated
2. Command line options are parsed
3. register_vaults is called
4. register_globals is called
5. register_inventory is called
6. The given entry function (run(context) by default) is called

6.2 Class functionality

6.2.1 Inventory

- "Builder" for the inventory + defines tasks to be run
- Registers hosts, tasks, groups and vaults
- Defines the actions to be done on the hosts

6.2.2 Host

- Represents a single host
- Stores host specific variables

6.2.3 Group

- Represents a group of hosts
- Stores variables that are shared between all belonging hosts

6.2.4 Vault

- Permanent encrypted storage for variables
- Useful to safely store and use secrets
- Can be checked into a git repository

6.2.5 Manager

- Initialized based on a passed inventory, which is called to build the inventory.
- Stores (i.e. owns) hosts, groups, tasks, vaults
- Provides CLI commands

6.2.6 Context

- Encapsulates an ssh connection to a specific host, and allows easy remote execution.
- Stores the actual processed set of variables defined for a host (merging globals and group variables)
- Stores intermediate data while the script is executed for a specific host

6.2.7 Task

- A collection of transactions and logic that achieves a bigger goal. (e.g. Install and configure zsh.)
- Can define and use variables, which allow easy per-host or per-group customization.

6.2.8 TrackedTask

- An extended task which will track a defined set of directories and files in a git repository, after it has finished execution.

6.2.9 Transaction function

- A single action that can be done on a remote host. (e.g. install package, template and upload file, create directory with given permissions, ...)
- A function that can be called inside a task to change remote state
- Examines current state
- Defines target state
- If changes are necessary, performs these changes on the remote
- Returns an object that stores information about what has been changed.

CHAPTER

SEVEN

TRANSACTIONS

Transactions are the main building blocks for tasks. You will use them to define what a task should do on the remote host. Every transaction will print a short summary of the changes it made to the remote. Below you will find a list of predefined transactions you can use.

7.1 Basic transactions

simple_automation.transactions.basic.directory(...)	Creates the given directory on the remote.
simple_automation.transactions.basic.directory_all(...)	Creates the given directories as if directory() was called for each of them.
simple_automation.transactions.basic.template(...)	Templates the given src file or given content and copies the output to the remote host at dst.
simple_automation.transactions.basic.template_all(...)	Templates each (src, dst) list entry, as if template() was called for each of them.
simple_automation.transactions.basic.copy(...)	Copies the given src file to the remote host at dst.
simple_automation.transactions.basic.copy_all(...)	Copies each (src, dst) list entry, as if copy() was called for each of them.
simple_automation.transactions.basic.save_output(...)	Saves the stdout of the given command on the remote host at remote dst.
simple_automation.transactions.basic.user(...)	Creates or modifies a unix user.
simple_automation.transactions.basic.group(...)	Creates or deletes a unix group.

7.2 Git transactions

simple_automation.transactions.git.checkout(...)	Checkout (and optionally update) the given git repository to dst.
simple_automation.transactions.git.clone(...)	Clone a git repository, without updating it, if it is already cloned.

7.3 Package transactions

By default, there is rudimentary support for installing packages via `apt`, `pacman` and `portage`.

simple_automation.transactions.package.apt.package(...)	Installs or uninstalls the given package name (depending on state == "present" or "absent").
simple_automation.transactions.package.pacman.package(...)	Installs or uninstalls the given package name (depending on state == "present" or "absent").
simple_automation.transactions.package.portage.package(...)	Installs or uninstalls the given package atom (depending on state == "present" or "absent").

simple_automation	Collects the main classes for easy importing in derived scripts.

CHAPTER

EIGHT

SIMPLE_AUTOMATION

Collects the main classes for easy importing in derived scripts.

Modules

simple_automation.checks	Provides some convenience checking functions that raise exceptions on errors.
simple_automation.context	Provides the Context class and related methods.
simple_automation.exceptions	Provides exception types for simple_automation.
simple_automation.group	Provides the group class.
simple_automation.host	Provides the host class.
simple_automation.inventory	Provides the Inventory class.
simple_automation.manager	Provides the manager class, which contains the toplevel logic of simple_automation and provides the CLI interface.
simple_automation.remote_dispatch	This module provides a standalone "server" that accepts commands on stdin and will be used to securely execute multiple commands over a single ssh connection.
simple_automation.task	Provides the Task class.
simple_automation.transaction	Provides the transaction class for easy state tracking and printing.
simple_automation.transactions	
simple_automation.utils	Provides utility functions.
simple_automation.vars	Provides the Vars class.
simple_automation.vault	Provides the Vault class for secure variable storage.
simple_automation.version	Provides version information

8.1 checks

Provides some convenience checking functions that raise exceptions on errors.

Functions

simple_automation.checks.**check_valid_key**(*key: str*, *msg: str = 'Invalid key'*)
Asserts that a given key is a valid identifier.

Parameters

- **key** (*str*) – The key to check.
- **msg** (*str, optional*) – The message to raise when the check fails.

simple_automation.checks.**check_valid_path**(*path*)
Asserts that a given path is non empty and absolute.

Parameters **path** (*str*) – The path to check.

simple_automation.checks.**check_valid_relative_path**(*path*)
Asserts that a given path is non empty and relative.

Parameters **path** (*str*) – The path to check.

8.2 context

Provides the Context class and related methods.

Classes

CompletedRemoteCommand()	A wrapper for the information returned by a remote command.
Context(manager, host)	A context is a wrapper object around a host and an ssh connection to that host.
RemoteDispatcher(context, command)	A wrapper class around a process that executes the remote dispatch script.

8.2.1 simple_automation.context.CompletedRemoteCommand

class simple_automation.context.**CompletedRemoteCommand**
Bases: object

A wrapper for the information returned by a remote command.

Methods

—

8.2.2 simple_automation.context.Context

class simple_automation.context.**Context**(*manager*, *host*)

Bases: object

A context is a wrapper object around a host and an ssh connection to that host. It is used to execute commands on the remote machine, and tracks state over the connection lifetime.

Methods

defaults	Creates a resource object, which sets the given defaults when it is entered, and resets them when it is exited.
exec_ssh_raw	Execute ssh to execute the given command on the remote host, directly via ssh.
init_ssh	Initialize environment on the remote host (temporary directory, remote exec script), so we can more easily execute commands on the remote.
remote_exec	Execute ssh to execute the given command on the remote host, via our built-in remote dispatch script.
run_task	Runs the registered instance (see Manager) for the given task class.
transaction	Begins a new transaction.

Attributes

debug	Forwards the corresponding variable from the associated manager.
pretend	Forwards the corresponding variable from the associated manager.
vars	Returns a Vars object containing the current active set of variables, which was merged from parent objects (global variables, group variables) on construction.
vars_dict	Returns a dictionary containing the current active set of variables, which was merged from parent objects (global variables, group variables) on construction.
verbose	Forwards the corresponding variable from the associated manager.

class ContextDefaults(*context*, *user: str*, *umask: int*, *dir_mode: int*, *file_mode: int*, *owner: str*, *group: str*)

Bases: object

Creates a resource object, which sets the given defaults when it is entered, and resets them when it is exited. Always use in a with statement. Statements may be nested.

property debug
Forwards the corresponding variable from the associated manager.

defaults(*user: Optional[str] = None*, *umask: Optional[int] = None*, *dir_mode: Optional[int] = None*, *file_mode: Optional[int] = None*, *owner: Optional[str] = None*, *group: Optional[str] = None*)
Creates a resource object, which sets the given defaults when it is entered, and resets them when it is exited. Always use in a with statement. Statements may be nested.

Example

```
def run(self, context):
    # Default context permissions apply here
    with context.defaults(owner="www", group="www"):
        # The directory owner and group will both be "www"
        directory(context, path="/var/www/test")
```

Parameters

- **user** (*str, optional*) – The user to execute commands as on the remote.
- **umask** (*int, optional*) – The umask to execute commands with on the remote.
- **dir_mode** (*int, optional*) – The directory mode for newly created directories on the remote.
- **file_mode** (*int, optional*) – The directory mode for newly created directories on the remote.
- **owner** (*str, optional*) – The owner of newly created files or directories on the remote.
- **group** (*str, optional*) – The group of newly created files or directories on the remote.

exec_ssh_raw(*command*)
Execute ssh to execute the given command on the remote host, directly via ssh.

Parameters **command** (*list[str]*) – The command to execute on the remote.

Returns The completed subprocess

Return type subprocess.CompletedProcess

init_ssh()
Initialize environment on the remote host (temporary directory, remote exec script), so we can more easily execute commands on the remote.

property pretend
Forwards the corresponding variable from the associated manager.

remote_exec(*command*, *checked=False*, *input=None*, *error_verbosity=None*, *user=None*, *umask=None*, *verbosity=None*)
Execute ssh to execute the given command on the remote host, via our built-in remote dispatch script. If checked is True, it will throw an exception if the remote command returns an unsuccessful exit status. checked=True also implies a default error_verbosity=0 and verbosity=2.

If both verbosity and error_verbosity trigger, the output will only be printed once.

Parameters

- **command** (*list[str]*) – The command to execute on the remote.

- **checked** (*bool, optional*) – If true, an exception will be raised if the command fails. Defaults to false.
- **input** (*bytes, optional*) – If not None, this will be passed to the command as stdin.
- **user** (*str, optional*) – A specific user to execute the command as. Defaults to the user set in the context.
- **umask** (*int, optional*) – A specific umask to execute the command with. Defaults to the umask set in the context.
- **verbosity** (*int, optional*) – If verbosity is not None and self.verbose >= verbosity, the command output will be printed. Read: verbosity is the number of -v flags needed so that the command's output will be shown. If verbosity is not given, the output will never be shown.
- **error_verbosity** (*int, optional*) – Same as verbosity, but only triggers when the command fails. E.g. calling with error_verbosity=1 causes both stdout and stderr to be printed, if the command fails and at least -v was given.

Returns The completed remote command

Return type *CompletedRemoteCommand*

run_task(*registered_task_class*)

Runs the registered instance (see Manager) for the given task class.

Parameters **registered_task_class** (*class(Task)*) – The task class that should be executed. The registered instance is found first, and then called.

transaction(*title: str*, *name: str*)

Begins a new transaction. Intended to be used in a 'with' statement. Each transaction will be shown to the user as a distinct unit.

A transaction must record an initial state and a final state, and may return success or failure (+reason).

A transaction should not actually alter the state of the remote, if context.pretend is set to True. In this case it should only examine and record what would be done.

A transaction may give additional variables to success() and failure(), which will be stored for later use.

Parameters

- **title** (*str*) – The title for the new transaction.
- **name** (*str*) – The name for the new transaction.

property vars

Returns a Vars object containing the current active set of variables, which was merged from parent objects (global variables, group variables) on construction.

property vars_dict

Returns a dictionary containing the current active set of variables, which was merged from parent objects (global variables, group variables) on construction.

property verbose

Forwards the corresponding variable from the associated manager.

8.2.3 simple_automation.context.RemoteDispatcher

class simple_automation.context.**RemoteDispatcher**(*context*, *command*)

Bases: object

A wrapper class around a process that executes the remote dispatch script. This will usually be an ssh command calling the script on a remote host, allowing us to send commands an receive output and return code information.

Methods

exec	Executes the given command on the remote machine as the user and with the umask given by the attached context.
expect	Waits until the given string is sent by the remote side.
read_len	Reads a length parameter from the remote process.
read_str	Reads a string from the remote process.
stop_and_wait	Stops the remote dispatcher, and waits until it exists.
write_data	Sends raw data to the remote process.
write_line	Sends a line to the remote process.
write_mode	Sends a mode to the remote dispatch process.
write_str	Sends the given string to the remote process.
write_str_list	Sends the given list of strings to the remote process.

exec(*command*, *input=None*, *user=None*, *umask=None*)

Executes the given command on the remote machine as the user and with the umask given by the attached context.

expect(*s*)

Waits until the given string is sent by the remote side.

read_len()

Reads a length parameter from the remote process.

read_str()

Reads a string from the remote process.

stop_and_wait()

Stops the remote dispatcher, and waits until it exists.

write_data(*data*)

Sends raw data to the remote process.

write_line(*s*)

Sends a line to the remote process.

write_mode(*mode*)

Sends a mode to the remote dispatch process.

write_str(*s*)

Sends the given string to the remote process.

write_str_list(*xs*)

Sends the given list of strings to the remote process.

8.3 exceptions

Provides exception types for simple_automation.

Exceptions

LogicError	Exception class for logic (i.e.
MessageError	An error type for which only the message will be printed, and the stacktrace supressed.
RemoteExecError(command, ret)	Exception class for remote execution errors.
SimpleAutomationError	Base exception class for simple_automation errors.
TransactionError(result)	Exception class for transaction errors.

8.3.1 simple_automation.exceptions.LogicError

exception simple_automation.exceptions.**LogicError**
Exception class for logic (i.e. "compile time") errors.

8.3.2 simple_automation.exceptions.MessageError

exception simple_automation.exceptions.**MessageError**
An error type for which only the message will be printed, and the stacktrace supressed.

8.3.3 simple_automation.exceptions.RemoteExecError

exception simple_automation.exceptions.**RemoteExecError**(*command*, *ret*)
Exception class for remote execution errors.

8.3.4 simple_automation.exceptions.SimpleAutomationError

exception simple_automation.exceptions.**SimpleAutomationError**
Base exception class for simple_automation errors.

8.3.5 simple_automation.exceptions.TransactionError

exception simple_automation.exceptions.**TransactionError**(*result*)
Exception class for transaction errors.

8.4 group

Provides the group class.

Classes

Group(manager, identifier)	A group for hosts.

8.4.1 simple_automation.group.Group

class simple_automation.group.**Group**(*manager*, *identifier*)

Bases: *simple_automation.vars.Vars*

A group for hosts. Can store variables which will have higher precedence than global variables, but lower than host variables.

Methods

copy	Copies a value from another vars object into this one.
get	Retrieves a variable by the given key.
set	Sets the given variable.

copy(*key*, *other_vars*)

Copies a value from another vars object into this one. Same as calling self.set(key, other_vars.get(key))

Parameters

- **key** (*str*) – The key that should be copied.
- **other_vars** (Vars) – The source variable storage where the key is copied from.

get(*key*, *default=None*)

Retrieves a variable by the given key. If no such key exists, it returns the given default value or throws a KeyError if no default is set.

Parameters

- **key** (*str*) – The key that should be read.
- **default** (*Any, optional*) – If not None, this will be returned in case the key is unset. By default None.

Returns The stored object.

Return type Any

set(*key*, *value*)

Sets the given variable.

Parameters

- **key** (*str*) – The key that should be read.
- **value** (*Any, optional*) – The value to be stored. Must be json (de-)serializable.

8.5 host

Provides the host class.

Classes

Host(manager, identifier, ssh_host)	A Host represents a machine that can be reached via ssh, and which should be managed by simple_automation.

8.5.1 simple_automation.host.Host

class simple_automation.host.**Host**(*manager*, *identifier*, *ssh_host*)

Bases: *simple_automation.vars.Vars*

A Host represents a machine that can be reached via ssh, and which should be managed by simple_automation. It can store variables which override gloabl and group variables, which may be used to customize the execution routine.

Methods

add_group	Adds this host to the given group, if it isn't already in that group.
copy	Copies a value from another vars object into this one.
get	Retrieves a variable by the given key.
set	Sets the given variable.
set_ssh_opts	Sets additional ssh parameters for this host's connection.
set_ssh_port	Sets the ssh port for the host's connection.

add_group(*group*)

Adds this host to the given group, if it isn't already in that group.

copy(*key*, *other_vars*)

Copies a value from another vars object into this one. Same as calling self.set(key, other_vars.get(key))

Parameters

- **key** (*str*) – The key that should be copied.
- **other_vars** (Vars) – The source variable storage where the key is copied from.

get(*key*, *default=None*)

Retrieves a variable by the given key. If no such key exists, it returns the given default value or throws a KeyError if no default is set.

Parameters

- **key** (*str*) – The key that should be read.
- **default** (*Any, optional*) – If not None, this will be returned in case the key is unset. By default None.

Returns The stored object.

Return type Any

set(*key*, *value*)
Sets the given variable.

Parameters

- **key** (*str*) – The key that should be read.
- **value** (*Any, optional*) – The value to be stored. Must be json (de-)serializable.

set_ssh_opts(*opts*)
Sets additional ssh parameters for this host's connection.

Parameters **opts** (*list[string]*) – Additional parameters to ssh

set_ssh_port(*port*)
Sets the ssh port for the host's connection.

Parameters **port** (*int*) – The port number to connect to

8.6 inventory

Provides the Inventory class.

Classes

Inventory(manager)	An inventory is a collection of definitions for vaults, tasks, global variables, hosts and groups.

8.6.1 simple_automation.inventory.Inventory

class simple_automation.inventory.**Inventory**(*manager*)
Bases: object

An inventory is a collection of definitions for vaults, tasks, global variables, hosts and groups. It also defines what tasks are run for each host.

Methods

register_globals	This function will be executed when your inventory should define its global variables.
register_inventory	This function will be executed when your inventory should define the hosts and groups.
register_tasks	This function will be executed when your inventory should define the tasks it want's to run.
register_vaults	This function will be executed when your inventory should define the vaults it want's to access.
run	This function is the default inventory script that will be executed for each host context, and is your main customization point.

Attributes

tasks	A list of task classes that should be registered to this inventory.

register_globals()
This function will be executed when your inventory should define its global variables. Use *self.manager.set()* to define them.

register_inventory()
This function will be executed when your inventory should define the hosts and groups. Use *self.manager.add_host()* and *self.manager.add_group()* to define them.

register_tasks()
This function will be executed when your inventory should define the tasks it want's to run. By default, this function will register all tasks given in the list self.tasks.

You can register tasks manually via a call to *self.manager.add_task()*. You may either save the return value of this function, and use ret.exec(context) to execute the task, or use the convenience method context.run_task(TaskClass).

register_vaults()
This function will be executed when your inventory should define the vaults it want's to access. Do this via a call to *self.manager.add_vault()*. Also remember to save the return value of this function, which will be a vault object you can later use to access stored values.

run(*context*)
This function is the default inventory script that will be executed for each host context, and is your main customization point. Here you can build your logic of what tasks to execute for a given host.

It is a good idea to split this up into several functions, because then you will be able to call those functions separately by selecting them via the `--scripts` command line option.

tasks = []
A list of task classes that should be registered to this inventory.

8.7 manager

Provides the manager class, which contains the toplevel logic of simple_automation and provides the CLI interface.

Classes

Manager(inventory_class[, main_directory])	A class that manages a set of global variables, hosts, groups, and tasks.
ThrowingArgumentParser([prog, usage, ...])	An argument parser that throws when invalid argument types are passed.

8.7.1 simple_automation.manager.Manager

class simple_automation.manager.**Manager**(*inventory_class*, *main_directory=None*)

Bases: *simple_automation.vars.Vars*

A class that manages a set of global variables, hosts, groups, and tasks. It provides the CLI interface and represents the main entry point for a simple automation script.

All relative paths (mainly files used in basic.template() or basic.copy()) will be interpreted relative from the location of the initially executed script. If you want to change this behavior, you can either set main_directory to a relative path, which will then be appended to that location, or to an absolute path.

Parameters

- **inventory_class** (*cls*) – The inventory class to instanciate.
- **main_directory** (*str, optional*) – The main directory of the script. Will be used to determine relative paths. If set to None, it will be set to the directory of the executed script file.

Methods

add_group	Registers a new group.
add_host	Registers a new host.
add_task	Registers a given task class.
add_vault	Registers a vault of the given class, with its storage at file.
copy	Copies a value from another vars object into this one.
get	Retrieves a variable by the given key.
main	The main program entry point.
set	Sets the given variable.

add_group(*identifier: str*)

Registers a new group.

Parameters **identifier** (*str*) – The identifier for the new group.

Returns The newly created group

Return type *Group*

add_host(*identifier: str*, *ssh_host: str*)

Registers a new host.

Parameters

- **identifier** (*str*) – The identifier for the new host.
- **ssh_host** (*str*) – The ssh host.

Returns The newly created host.

Return type *Host*

add_task(*task_class*)

Registers a given task class. This allows the task to register variable defaults. You can either save the returned instance yourself and call task.exec() when you want to run it, or you can use context.run_task(task_class) to run a registered task automatically.

Parameters **identifier** (*str*) – The identifier for the new task.

Returns The newly created task.

Return type *Task*

add_vault(*vault_class*, *file: str*, ***kwargs*)
Registers a vault of the given class, with its storage at file. Additional parameters are forwarded to the vault constructor.

Parameters

- **vault_class** (*class(Vault)*) – The vault class to instanciate.
- **file** (*str*) – A file relative to the project directory that will be passed onto the vault.
- ****kwargs** – Will be forwarded to the Vault constructor.

Returns The newly created vault

Return type *Vault*

copy(*key*, *other_vars*)
Copies a value from another vars object into this one. Same as calling self.set(key, other_vars.get(key))

Parameters

- **key** (*str*) – The key that should be copied.
- **other_vars** (Vars) – The source variable storage where the key is copied from.

get(*key*, *default=None*)
Retrieves a variable by the given key. If no such key exists, it returns the given default value or throws a KeyError if no default is set.

Parameters

- **key** (*str*) – The key that should be read.
- **default** (*Any, optional*) – If not None, this will be returned in case the key is unset. By default None.

Returns The stored object.

Return type Any

main()
The main program entry point. This will parse arguments and call the user-supplied function on the defined inventory, when the script should be executed.

set(*key*, *value*)
Sets the given variable.

Parameters

- **key** (*str*) – The key that should be read.
- **value** (*Any, optional*) – The value to be stored. Must be json (de-)serializable.

8.7.2 simple_automation.manager.ThrowingArgumentParser

class simple_automation.manager.**ThrowingArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=True*)

Bases: argparse.ArgumentParser

An argument parser that throws when invalid argument types are passed.

Methods

add_argument	
add_argument_group	
add_mutually_exclusive_group	
add_subparsers	
convert_arg_line_to_args	
error	Raises an exception on error.
exit	
format_help	
format_usage	
get_default	
parse_args	
parse_intermixed_args	
parse_known_args	
parse_known_intermixed_args	
print_help	
print_usage	
register	
set_defaults	

add_argument(*dest*, ..., *name=value*, ...)
add_argument(*option_string*, *option_string*, ..., *name=value*, ...) → None

error(*message*)
Raises an exception on error.

Exceptions

ArgumentParserError	Error class for argument parsing errors.

8.7.3 simple_automation.manager.ArgumentParserError

exception simple_automation.manager.**ArgumentParserError**
Error class for argument parsing errors.

Functions

simple_automation.manager.**run_inventory**(*inventory_class*, *main_directory=None*)
Instanciates a manager given an inventory class and runs the manager's CLI.

Parameters

- **inventory_class** (*str*) – The inventory class to instanciate.
- **main_directory** (*str, optional*) – The main directory of the script. Will be used to determine relative paths. If set to None, it will be set to the directory of the executed script file.

8.8 remote_dispatch

This module provides a standalone "server" that accepts commands on stdin and will be used to securely execute multiple commands over a single ssh connection. This script allows us to execute a command as-is without interpreting any arguments, and retrieve the exit status as well as capture and forwad stdout and stderr back to the client.

Classes

Dispatcher()	The main dispatcher.
ExecutionSettings()	Execution settings for the next command.

8.8.1 simple_automation.remote_dispatch.Dispatcher

class simple_automation.remote_dispatch.**Dispatcher**
Bases: object

The main dispatcher. Parses the protocol and executes commands.

Methods

handle_exec	Handles the exec mode packet.
handle_set_debug	Handles the debugging mode packet.
handle_set_input	Handles the input mode packet.
handle_set_umask	Handles the umask mode packet.
handle_set_user	Handles the user mode packet.
main	Begin by changing directory to /tmp.
run_command	Runs a given command using the saved execution settings.

handle_exec()
Handles the exec mode packet. Reads a command, and executes it. stdout and stderr will be captured and returned to the client.

handle_set_debug()
Handles the debugging mode packet. If debugging is enabled, we will print every executed command and the relevant settings.

handle_set_input()
Handles the input mode packet. The given input will be used as stdin for the next command execution.

handle_set_umask()
Handles the umask mode packet. The given umask will be used for the next command execution.

handle_set_user()
Handles the user mode packet. Validates the given uid / resolves a username, which will then be used for the next command. The gid will be set to the primary gid of that user.

main()
Begin by changing directory to /tmp. Then listen for packets on stdin and loop until stdin is closed.

run_command(*command*)
Runs a given command using the saved execution settings. The settings will be reset afterwards.

8.8.2 simple_automation.remote_dispatch.ExecutionSettings

class simple_automation.remote_dispatch.**ExecutionSettings**
Bases: object

Execution settings for the next command.

Methods

—

Functions

simple_automation.remote_dispatch.**read_data**()
Read arbitrary data

simple_automation.remote_dispatch.**read_len**()
Read a newline delimited length

simple_automation.remote_dispatch.**read_mode**()
Read and return a mode (newline terminated string)

simple_automation.remote_dispatch.**read_str**()
Read arbitrary string

simple_automation.remote_dispatch.**read_str_list**()
Read a string list

simple_automation.remote_dispatch.**resolve_script_path**()
Guard script name resolution because the remote variant will not be executed from a file.

simple_automation.remote_dispatch.**write_data**(*data*)
Write arbitrary binary data (sends a length, newline, data)

simple_automation.remote_dispatch.**write_mode**(*mode*)
Write a mode (newline terminated string)

simple_automation.remote_dispatch.**write_str**(*s*)
Write arbitrary string

8.9 task

Provides the Task class.

Classes

Task(manager)	A base class for tasks.
TrackedTask(manager)	A base class for tasks which want to track changes in a git repository.

8.9.1 simple_automation.task.Task

class simple_automation.task.**Task**(*manager*)

Bases: object

A base class for tasks. A task executes on a host context and runs a set of transactions on this context.

Examples

Every tasks needs an identifier and a description. The identifier can be used as a namespace for related variables.

```
from simple_automation import Task

class TaskZshConfig(Task):
    identifier = "zsh"
    description = "Installs a global zsh configuration"

    def run(self, context):
        with context.defaults(umask=0o022, dir_mode=0o755, file_mode=0o644):
            # Copy configuration
            directory(context, path="/etc/zsh")
            template(context, src="templates/zsh/zshrc.j2", dst="/etc/zsh/zshrc")
            template(context, src="templates/zsh/zprofile.j2", dst="/etc/zsh/
↪zprofile")
```

Methods

enabled	Returns true, if our enable variable is set to true in the given context.
exec	Executes the actual task, as well as the pre and post functions in respective order, if the task is enabled for the current context.
post_run	Called after self.run() is called.
pre_run	Called before self.run() is called.
run	To be overwritten by a subclass.
set_defaults	Optional callback for subclasses.

Attributes

description	A short description of what the task does.
identifier	The identifier of this task.

description = None

A short description of what the task does.

enabled(*context*)

Returns true, if our enable variable is set to true in the given context.

exec(*context*)

Executes the actual task, as well as the pre and post functions in respective order, if the task is enabled for the current context.

identifier = None
The identifier of this task.

post_run(*context*)
Called after self.run() is called. No-op by default.

pre_run(*context*)
Called before self.run() is called. Prints the task's title by default.

run(*context*)
To be overwritten by a subclass. Contain's the task's logic.

set_defaults()
Optional callback for subclasses. Called when the task may define its global defaults. Default variable keys should be named like "tasks.{self.identifier}.variable_name".

Examples

Define variables so your task can be customized easily for your different systems.

```
from simple_automation import Task

class TaskZshConfig(Task):
    identifier = "zsh"
    description = "Installs a global zsh configuration"

    def set_defaults(self):
        self.manager.set("tasks.zsh.config_folder", "/etc/zsh")

    def run(self, context):
        # ...
        template(context, src="templates/zsh/zshrc.j2", dst="{{ tasks.zsh.
↪config_folder }}/zshrc")
```

8.9.2 simple_automation.task.TrackedTask

class simple_automation.task.**TrackedTask**(*manager*)
Bases: *simple_automation.task.Task*

A base class for tasks which want to track changes in a git repository. All tracking will be done after *run()* has finished.

Examples

Generally, to track files or directories, you have to inherit from *TrackedTask* instead of *Task*. It may be beneficial to create your own base class for all tracked tasks, to set a common tracking repository. You will then only have to add all files and directories you want to track to *tracking_paths* in the actual task.

```
from simple_automation import TrackedTask

class MyTrackedTask(TrackedTask):
    tracking_repo_url = "{{ tracking.repo_url }}"
    tracking_local_dst = "/var/lib/root/tracking"
```

To track a config file, simply extend your task with the correct tracking path:

```
class TaskZshConfig(MyTrackedTask):
    tracking_paths = ["/etc/zsh"]
    # ...
```

To track dynamic information, first save it to a file:

```
# Track installed packages from portage
class TaskTrackInstalledPackages(MyTrackedTask):
    identifier = "track_installed_packages"
    description = "Tracks all installed packages"
    tracking_paths = ["/var/lib/root/installed_packages"]

    def run(self, context):
        save_output(context, command=["qlist", "-CIv"],
                    dst="/var/lib/root/installed_packages",
                    desc="Query installed packages")
```

Methods

enabled	Returns true, if our enable variable is set to true in the given context.
exec	Executes the actual task, as well as the pre and post functions in respective order, if the task is enabled for the current context.
post_run	Called after self.run() is called.
pre_run	Called before self.run() is called.
run	To be overwritten by a subclass.
set_defaults	Optional callback for subclasses.

Attributes

description	A short description of what the task does.
identifier	The identifier of this task.
tracking_git_commit_opts	Extra options to 'git commit'.
tracking_group	The group which will own tracking related files.
tracking_local_dst	The path where the local clone of the repository will be.
tracking_paths	A list of directories and/or files that should be tracked.
tracking_repo_configs	A dictionary of git configs to be set locally when the repository is first created.
tracking_repo_url	The remote url to the repository which will be used as the tracking repo.
tracking_subpath	The subpath in the repository where the tracked files will be held.
tracking_user	The user to execute all tracking commands as, and the user which will own all related files.

class TaskInitializeTracking(*tracked_task*)
Bases: *simple_automation.task.Task*

A sub-task used to initialize the tracking repository.

We remember the tracked parent task, so so we have access to the tracking specific variables later.

enabled(*context*)
Returns true, if our enable variable is set to true in the given context.

exec(*context*)
Executes the actual task, as well as the pre and post functions in respective order, if the task is enabled for the current context.

post_run(*context*)
Called after self.run() is called. No-op by default.

pre_run(*context*)
Called before self.run() is called. Prints the task's title by default.

run(*context*)
To be overwritten by a subclass. Contain's the task's logic.

set_defaults()
Optional callback for subclasses. Called when the task may define its global defaults. Default variable keys should be named like "tasks.{self.identifier}.variable_name".

Examples

Define variables so your task can be customized easily for your different systems.

```
from simple_automation import Task

class TaskZshConfig(Task):
    identifier = "zsh"
    description = "Installs a global zsh configuration"

    def set_defaults(self):
        self.manager.set("tasks.zsh.config_folder", "/etc/zsh")

    def run(self, context):
        # ...
        template(context, src="templates/zsh/zshrc.j2", dst="{{ tasks.zsh.
↪config_folder }}/zshrc")
```

description = None
A short description of what the task does.

enabled(*context*)
Returns true, if our enable variable is set to true in the given context.

exec(*context*)
Executes the actual task, as well as the pre and post functions in respective order, if the task is enabled for the current context.

identifier = None
The identifier of this task.

post_run(*context*)
Called after self.run() is called. No-op by default.

pre_run(*context*)
Called before self.run() is called. Prints the task's title by default.

run(*context*)
To be overwritten by a subclass. Contain's the task's logic.

set_defaults()
Optional callback for subclasses. Called when the task may define its global defaults. Default variable keys should be named like "tasks.{self.identifier}.variable_name".

Examples

Define variables so your task can be customized easily for your different systems.

```
from simple_automation import Task

class TaskZshConfig(Task):
    identifier = "zsh"
    description = "Installs a global zsh configuration"

    def set_defaults(self):
        self.manager.set("tasks.zsh.config_folder", "/etc/zsh")

    def run(self, context):
        # ...
        template(context, src="templates/zsh/zshrc.j2", dst="{{ tasks.zsh.
↪config_folder }}/zshrc")
```

tracking_git_commit_opts = []
Extra options to 'git commit'. Will be templated by the currently executed context.

tracking_group = 'root'
The group which will own tracking related files.

tracking_local_dst = None
The path where the local clone of the repository will be. Will be templated by the currently executed context.

tracking_paths = []
A list of directories and/or files that should be tracked. Will be templated by the currently executed context.

tracking_repo_configs = {'user.email': 'root@localhost', 'user.name': '{{ context.host.identifier }}'}
A dictionary of git configs to be set locally when the repository is first created. Values will be templated by the currently executed context.

By default, it sets user.name to the host identifier, and user.email to root@localhost. (for each entry, 'git config –local {key} {value}' is executed). Useful to set name and email, and maybe a gpg signing key.

tracking_repo_url = None
The remote url to the repository which will be used as the tracking repo. Will be templated by the currently executed context. For example: - (via ssh) "git@github.com:myuser/tracked-system-settings" - (via https) "https://{{ personal_access_token }}@github.com/myuser/tracked-system-settings" Remember to put secrets into a vault so they aren't checked into your repository in plain text. We recommend using ssh, as secrets in the url will be printed to the terminal when executed.

tracking_subpath = '{{ context.host.identifier }}'
The subpath in the repository where the tracked files will be held. Will be templated by the currently executed context. It defaults to the identifier of the machine.

tracking_user = 'root'
The user to execute all tracking commands as, and the user which will own all related files. The rsync will always be called as root so it may access arbitrary paths.

8.10 transaction

Provides the transaction class for easy state tracking and printing.

Classes

ActiveTransaction()	Represents a transaction on a remote host.
CompletedTransaction(transaction, success, store)	A CompletedTransaction is a manifest of the initial and final state of an transition.
Transaction(context, title, name)	A wrapper around a transaction context that enforces usage of the 'with' statement to modify the transaction.

8.10.1 simple_automation.transaction.ActiveTransaction

class simple_automation.transaction.**ActiveTransaction**
Bases: object

Represents a transaction on a remote host. Transactions are operational units, which alter the state of a remote from an initial state A to a known target state. When they begin, they must examine the initial state, and transition the remote into the target state. This possible state change will be presented to the user.

Methods

extra_info	Purely extraneous information that will be shown additionally to the user.
failure	Completes the transaction, marking it as failed with the given reason.
final_state	Records the (expected) final state of the remote.
finalize	Finalizes this transaction, which will verify that all states are set corectly, and print the transaction.
initial_state	Records the observed initial state of the remote.
success	Completes the transaction with successful status.
unchanged	Sets the final state to the initial state and returns success(**kwargs).

extra_info(***kwargs*)
Purely extraneous information that will be shown additionally to the user.

Parameters ****kwargs** – Extra information assocations

failure(*reason*, *set_final_state=False*, ***kwargs*)
Completes the transaction, marking it as failed with the given reason. If reason is a RemoteExecError, additional information will be printed.

Parameters

- **reason** (*str*) – The reason for the failure.
- **set_final_state** (*bool*) – If true, the final transaction state will be set. Defaults to false.
- ****kwargs** – Final state assocations

Returns The completed transaction.

Return type *CompletedTransaction*

final_state(***kwargs*)
Records the (expected) final state of the remote.

Parameters ****kwargs** – Final state assocations

finalize(*context*, *transaction*)
Finalizes this transaction, which will verify that all states are set corectly, and print the transaction.

Parameters

- **context** (Context) – The associated context.
- **transaction** (Transaction) – The transaction that should be finalized.

initial_state(***kwargs*)
Records the observed initial state of the remote.

Parameters ****kwargs** – Initial state assocations

success(***kwargs*)
Completes the transaction with successful status.

Parameters ****kwargs** – Final state assocations

Returns The completed transaction.

Return type *CompletedTransaction*

unchanged(***kwargs*)
Sets the final state to the initial state and returns `success(**kwargs)`.

Parameters ****kwargs** – Final state assocations

Returns The completed transaction.

Return type *CompletedTransaction*

8.10.2 simple_automation.transaction.CompletedTransaction

class simple_automation.transaction.**CompletedTransaction**(*transaction*, *success*, *store*, *failure_reason=None*)

Bases: object

A CompletedTransaction is a manifest of the initial and final state of an transition. Additionally, it records a success status, a changed flag to indicate that at least one action has actually been performed, as well as additional stored values defined by the specific transaction for later use.

Instance variables

success [bool] True if the transaction result was successful.

changed [bool] True if the transaction did change anything.

failure_reason [str, optional] The failure reason that will be attached if the transaction failed.

initial_state [dict] The observed initial state.

final_state [dict] The final state of the system. If success=False, this will be the initial state.

extra_info [dict, optional] Optional extra information given by the transaction

Parameters

- **transaction** (Transaction) – The transaction that is about to be completed.
- **success** (*bool*) – True if the transaction result was successful.
- **store** (*dict*) – Additional value store that can later be used by callers to retrieve additional information.
- **failure_reason** (*str, optional*) – The failure reason that will be attached if the transaction failed.

Methods

8.10.3 simple_automation.transaction.Transaction

class simple_automation.transaction.**Transaction**(*context*, *title*, *name*)

Bases: object

A wrapper around a transaction context that enforces usage of the 'with' statement to modify the transaction.

Parameters

- **context** (Context) – The context on which the transaction will be executed.
- **title** (*str*) – The title of the transaction for printed output.
- **name** (*str*) – The name of the transaction for printed output.

Methods

8.11 transactions

Modules

simple_automation.transactions.basic	Provides basic transactions.
simple_automation.transactions.git	Provides git related transactions.
simple_automation.transactions.package	
simple_automation.transactions.utils	Provides basic transaction utilities.

8.11.1 transactions.basic

Provides basic transactions.

Functions

simple_automation.transactions.basic.**copy**(*context:* simple_automation.context.Context, *src: str*, *dst: str*, *mode=None*, *owner=None*, *group=None*)

Copies the given src file to the remote host at dst.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **src** (*str*) – The local source file path relative to the project directory. Will be templated.
- **dst** (*str*) – The remote destination file path. Will be templated.
- **mode** (*int, optional*) – The new file mode. Defaults the current context file creation mode.
- **owner** (*str, optional*) – The new file owner. Defaults the current context owner.
- **group** (*str, optional*) – The new file group. Defaults the current context group.

Returns The completed transaction

Return type *CompletedTransaction*

simple_automation.transactions.basic.**copy_all**(*context:* simple_automation.context.Context, *src_dst_pairs: list*, *mode=None*, *owner=None*, *group=None*)

Copies each (src, dst) list entry, as if copy() was called for each of them.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **src_dst_pairs** (*list[(str, str)]*) – A list of (src, dst) pairs corresponding to the parameters from copy().
- **mode** (*int, optional*) – The new file mode. Defaults the current context file creation mode.
- **owner** (*str, optional*) – The new file owner. Defaults the current context owner.

- **group** (*str, optional*) – The new file group. Defaults the current context group.

Returns The completed transaction

Return type *CompletedTransaction*

simple_automation.transactions.basic.**directory**(*context: simple_automation.context.Context*, *path: str*, *mode=None*, *owner=None*, *group=None*)

Creates the given directory on the remote.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **path** (*str*) – The directory path to create (will be templated). Parent directory must exist.
- **mode** (*int, optional*) – The new directory mode. Defaults the current context directory creation mode.
- **owner** (*str, optional*) – The new directory owner. Defaults the current context owner.
- **group** (*str, optional*) – The new directory group. Defaults the current context group.

Returns The completed transaction

Return type *CompletedTransaction*

simple_automation.transactions.basic.**directory_all**(*context: simple_automation.context.Context*, *paths: list*, *mode=None*, *owner=None*, *group=None*)

Creates the given directories as if directory() was called for each of them.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **paths** (*str*) – The directory paths to create (each will be templated). Parent directory must exist for each directory. Executed in order.
- **mode** (*int, optional*) – The new directory mode. Defaults the current context directory creation mode.
- **owner** (*str, optional*) – The new directory owner. Defaults the current context owner.
- **group** (*str, optional*) – The new directory group. Defaults the current context group.

Returns The completed transaction

Return type *CompletedTransaction*

simple_automation.transactions.basic.**group**(*context: simple_automation.context.Context*, *name: str*, *state: str = 'present'*, *system: bool = False*)

Creates or deletes a unix group.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **name** (*str*) – The name of the user to create or modify. Will be templated.
- **state** (*str*) – If “present” the user will be added / modified, if “absent” the user will be deleted ignoring all other parameters.

- **system** (*bool*) – If `True` the user will be created as a system user. This has no effect on existing users.

Returns The completed transaction

Return type *CompletedTransaction*

simple_automation.transactions.basic.**save_output**(*context: simple_automation.context.Context*, *command: list*, *dst: str*, *desc=None*, *mode=None*, *owner=None*, *group=None*)

Saves the stdout of the given command on the remote host at remote dst. Using –pretend will still run the command, but won't save the output. Changed status reflects if the file contents changed. Optionally accepts file mode, owner and group, if not given, context defaults are used.

Parameters

- **context** (`Context`) – The context providing the execution context and templating dictionary.
- **command** (*list[str]*) – A list containing the command and its arguments. Each one will be templated.
- **dst** (*str*) – The remote destination file path. Will be templated.
- **desc** (*str*) – A description to be printed in the summary when executing.
- **mode** (*int, optional*) – The new file mode. Defaults the current context file creation mode.
- **owner** (*str, optional*) – The new file owner. Defaults the current context owner.
- **group** (*str, optional*) – The new file group. Defaults the current context group.

Returns The completed transaction

Return type *CompletedTransaction*

simple_automation.transactions.basic.**template**(*context: simple_automation.context.Context*, *dst: str*, *src: Optional[str] = None*, *content: Optional[str] = None*, *mode=None*, *owner=None*, *group=None*)

Templates the given src file or given content and copies the output to the remote host at dst. Either content or src must be specified.

Parameters

- **context** (`Context`) – The context providing the execution context and templating dictionary.
- **src** (*str, optional*) – The local source file path relative to the project directory. Will be templated. Mutually exclusive with content.
- **content** (*str, optional*) – The content for the file. Will be templated. Mutually exclusive with src.
- **dst** (*str*) – The remote destination file path. Will be templated.
- **mode** (*int, optional*) – The new file mode. Defaults the current context file creation mode.
- **owner** (*str, optional*) – The new file owner. Defaults the current context owner.
- **group** (*str, optional*) – The new file group. Defaults the current context group.

Returns The completed transaction

Return type *CompletedTransaction*

simple_automation.transactions.basic.**template_all**(*context: simple_automation.context.Context*, *src_dst_pairs: list*, *mode=None*, *owner=None*, *group=None*)

Templates each (src, dst) list entry, as if template() was called for each of them.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **src_dst_pairs** (*list[(str, str)]*) – A list of (src, dst) pairs corresponding to the parameters from template().
- **mode** (*int, optional*) – The new file mode. Defaults the current context file creation mode.
- **owner** (*str, optional*) – The new file owner. Defaults the current context owner.
- **group** (*str, optional*) – The new file group. Defaults the current context group.

Returns The completed transaction

Return type *CompletedTransaction*

simple_automation.transactions.basic.**user**(*context: simple_automation.context.Context*, *name: str*, *group: Optional[str] = None*, *groups: Optional[list] = None*, *append_groups: bool = False*, *state: str = 'present'*, *system: bool = False*, *shell: Optional[str] = None*, *password: Optional[str] = None*, *home: Optional[str] = None*, *create_home=True*)

Creates or modifies a unix user. Because we interally call `userdel`, removing a user will also remove it's associated primary group if no other user belongs to it.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **name** (*str*) – The name of the user to create or modify. Will be templated.
- **group** (*str, optional*) – The primary group of the user. If given, the group must already exists. Otherwise, a group will be created with the same name as the user, if a user is created by this action. Will be templated.
- **groups** (*list[str], optional*) – Supplementary groups for the user.
- **append_groups** (*bool*) – If `True`, the user will be added to all given supplementary groups. If `False`, the user will be added to exactly the given supplementary groups and removed from other groups.
- **state** (*str*) – If "present" the user will be added / modified, if "absent" the user will be deleted ignoring all other parameters.
- **system** (*bool*) – If `True` the user will be created as a system user. This has no effect on existing users.
- **shell** (*str*) – Specifies the shell for the user. Defaults to `/sbin/nologin` if not given but a user needs to be created. Will be templated.
- **password** (*str, optional*) – Will update the password hash to the given vaule of the user. Use `!` to lock the account. Defaults to '!' if not given but a user needs to be created. Will be templated. You can generate a password hash by using the following script:

```
>>> import crypt, getpass
>>> crypt.crypt(getpass.getpass(), crypt.mksalt(crypt.METHOD_SHA512))
'$6$rwn5z9MlYvcnE222$9wOP6Y6EcnF.cZ7BUjttWeSQNdOQWI...'
```

- **home** (*str, optional*) – The home directory for the user. Will be left empty if not given but a user needs to be created. Will be templated.
- **create_home** (*bool*) – If `True` and home was given, create the home directory of the user if it doesn't exist.

Returns The completed transaction

Return type *CompletedTransaction*

8.11.2 transactions.git

Provides git related transactions.

Functions

simple_automation.transactions.git.**checkout**(*context:* simple_automation.context.Context, *url: str*, *dst: str*, *update: bool = True*, *depth=None*)

Checkout (and optionally update) the given git repository to dst.

Parameters

- **context** (`Context`) – The context providing the execution context and templating dictionary.
- **url** (*str*) – The url of the git repository to checkout. Will be templated.
- **dst** (*str*) – The remote destination path for the repository. Will be templated.
- **update** (*bool, optional*) – Also tries to update the repository if it is already cloned. Defaults to true.
- **depth** (*str, optional*) – Restrict repository cloning depth. Beware that updates might not work correctly because of forced updates.

Returns The completed transaction

Return type *CompletedTransaction*

simple_automation.transactions.git.**clone**(*context:* simple_automation.context.Context, *url: str*, *dst: str*, *depth=None*)

Clone a git repository, without updating it, if it is already cloned. Same as calling *checkout()* with update=False.

Parameters

- **context** (`Context`) – The context providing the execution context and templating dictionary.
- **url** (*str*) – The url of the git repository to checkout. Will be templated.
- **dst** (*str*) – The remote destination path for the repository. Will be templated.
- **depth** (*str, optional*) – Restrict repository cloning depth. Beware that updates might not work correctly because of forced updates.

Returns The completed transaction

Return type *CompletedTransaction*

8.11.3 transactions.package

Modules

simple_automation.transactions.package.apt	Provides apt related transactions.
simple_automation.transactions.package.pacman	Provides pacman related transactions.
simple_automation.transactions.package.portage	Provides portage related transactions.
simple_automation.transactions.package.utils	Provides package related utils.

transactions.package.apt

Provides apt related transactions.

Functions

simple_automation.transactions.package.apt.**is_installed**(*context:* simple_automation.context.Context, *name: str*)

Queries whether or not the given package name is installed on the remote.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **name** (*str*) – The package name to query. Will be templated.

Returns True if the package is installed

Return type bool

simple_automation.transactions.package.apt.**package**(*context:* simple_automation.context.Context, *name: str*, *state='present'*, *opts: Optional[list] = None*)

Installs or uninstalls the given package name (depending on state == "present" or "absent"). Additional options to apt-get can be passed via opts, and will be appended before the package name. opts will be templated.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **name** (*str*) – The package name to be installed or uninstalled. Will be templated.
- **state** (*str, optional*) – The desired state, either "present" or "absent". Defaults to "present".
- **opts** (*list[str]*) – Additional options to pacman. Will be templated.

Returns The completed transaction

Return type *CompletedTransaction*

transactions.package.pacman

Provides pacman related transactions.

Functions

simple_automation.transactions.package.pacman.**is_installed**(*context:* simple_automation.context.Context, *name: str*)

Queries whether or not the given package name is installed on the remote.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **name** (*str*) – The package name to query. Will be templated.

Returns True if the package is installed

Return type bool

simple_automation.transactions.package.pacman.**package**(*context:* simple_automation.context.Context, *name: str*, *state='present'*, *opts: Optional[list] = None*)

Installs or uninstalls the given package name (depending on state == "present" or "absent"). Additional options to pacman can be passed via opts, and will be appended before the package name. opts will be templated.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **name** (*str*) – The package name to be installed or uninstalled. Will be templated.
- **state** (*str, optional*) – The desired state, either "present" or "absent". Defaults to "present".
- **opts** (*list[str]*) – Additional options to pacman. Will be templated.

Returns The completed transaction

Return type *CompletedTransaction*

transactions.package.portage

Provides portage related transactions.

Functions

simple_automation.transactions.package.portage.**is_installed**(*context:* simple_automation.context.Context, *atom: str*, *packages: Optional[list] = None*)

Queries whether or not the given package atom is installed on the remote.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **atom** (*str*) – The package name to query. Will be templated.
- **packages** (*list[str]*) – Additional options to portage. Will be templated.

Returns True if the package is installed

Return type bool

simple_automation.transactions.package.portage.**list_packages**(*context: simple_automation.context.Context*)

Returns a dictionary of all installed packages on the remote system. The dictionary maps from "{category}/{name}" → any INFO_ATOMS → str/None

Parameters **context** (Context) – The context providing the execution context and templating dictionary.

Returns All package atoms that are installed on the remote system.

Return type list[str]

simple_automation.transactions.package.portage.**package**(*context: simple_automation.context.Context*, *atom: str*, *state='present'*, *oneshot=False*, *opts: Optional[list] = None*)

Installs or uninstalls the given package atom (depending on state == "present" or "absent"). Additional options to emerge can be passed via opts, and will be appended before the package atom. opts will be templated.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **atom** (*str*) – The package name to be installed or uninstalled. Will be templated.
- **state** (*str, optional*) – The desired state, either "present" or "absent". Defaults to "present".
- **oneshot** (*bool, optional*) – Use portage option –oneshot. Defaults to false.
- **opts** (*list[str]*) – Additional options to portage. Will be templated.

Returns The completed transaction

Return type *CompletedTransaction*

transactions.package.utils

Provides package related utils.

Functions

simple_automation.transactions.package.utils.**generic_package**(*context: simple_automation.context.Context*, *atom: str*, *state: str*, *is_installed*, *install*, *uninstall*)

Installs or uninstalls (depending if state == "present" or "absent") the given package atom. Additional options to emerge can be passed via opts, and will be appended before the package atom. opts will be templated.

Parameters

- **context** (Context) – The context providing the execution context and templating dictionary.
- **atom** (*str*) – The package name to be installed or uninstalled. Will be templated.
- **state** (*str*) – The desired state, either "present" or "absent".
- **is_installed** (*Callable[[Context, str], bool]*) – Callback used to determine if a given package is installed.
- **install** (*Callable[[Context, str], None]*) – Callback used to install a package on the remote.
- **uninstall** (*Callable[[Context, str], None]*) – Callback used to uninstall a package on the remote.

Returns The completed transaction

Return type *CompletedTransaction*

8.11.4 transactions.utils

Provides basic transaction utilities.

Functions

simple_automation.transactions.utils.**remote_sha512sum**(*context: simple_automation.context.Context*, *path: str*)

Fetch the sha512sum of a remote file.

Parameters

- **context** (Context) – The host execution context
- **path** (*str*) – The path to the remote file.

Returns The hexlified sha512sum of the path on the remote host, or None if an error occurred.

Return type str

simple_automation.transactions.utils.**remote_stat**(*context*, *path*)

Runs stat on the given remote path.

Parameters

- **context** (Context) – The host execution context
- **path** (*str*) – The path to run stat on.

Returns A tuple of (file_type, str_octal_mode, owner, group), where file_type is one of ["file", "directory", "link", "other"]

Return type (str, str, str, str)

simple_automation.transactions.utils.**remote_upload**(*context: simple_automation.context.Context*, *get_content*, *title: str*, *name: str*, *dst: str*, *mode=None*, *owner=None*, *group=None*)

Calls get_content and saves the resulting string as a file on the remote host at dst. No arguments will be templated, this is task of the calling function. Optionally accepts file mode, owner and group, if not given, context defaults are used.

Parameters

- **context** (Context) – The host execution context
- **get_content** (*Callable[[],str]*) – A function that is called to get the content that should be uploaded.
- **title** (*str*) – The title for the generated transaction
- **name** (*str*) – The name for the generated transaction
- **dst** (*str*) – The remote destination for the uplaoded file
- **mode** (*int, optional*) – The new file mode. Defaults the current context file creation mode.
- **owner** (*str, optional*) – The new file owner. Defaults the current context owner.
- **group** (*str, optional*) – The new file group. Defaults the current context group.

Returns The completed transaction

Return type *CompletedTransaction*

simple_automation.transactions.utils.**resolve_mode_owner_group**(*context: simple_automation.context.Context*, *mode*, *owner*, *group*, *fallback_mode*)

Canonicalize mode, owner and group. If any of them is None, the respective variable will be replaced with the context default.

Parameters

- **context** (Context) – The host execution context
- **mode** (*int*) – The mode to canonicalize. May be None.
- **owner** (*str*) – User id or name for the owner. User will be verified as existing on the remote.
- **group** (*str*) – Group id or name for the group. Group will be verified as existing on the remote.
- **fallback_mode** (*int*) – The fallback_mode to canonicalize in case mode = None. Usually set to either context.file_mode or context.dir_mode.

Returns A tuple of (resolved_mode, resolved_owner, resolved_group)

Return type (str, str, str)

simple_automation.transactions.utils.**template_str**(*context: simple_automation.context.Context*, *content: str*) → str

Renders the given string template.

Parameters

- **context** (Context) – The context providing the templating dictionary
- **content** (*str*) – The string to template

Returns The templated string

Return type str

8.12 utils

Provides utility functions.

Functions

simple_automation.utils.**align_ellipsis**(*s*, *width*)
Shrinks the given string to width (including an ellipsis character), and additionally pads the string with spaces to match the given with.

simple_automation.utils.**choice_yes**(*msg: str*) → bool
Awaits user choice (Y/n).

simple_automation.utils.**ellipsis**(*s*, *width*)
Shrinks the given string to width (including an ellipsis character).

simple_automation.utils.**merge_dicts**(*source*, *destination*)
Recursively merges two dictionaries source and destination. The source dictionary will only be read, but the destination dictionary will be overwritten.

simple_automation.utils.**print_transaction**(*context*, *transaction*)
Prints the transaction summary

simple_automation.utils.**print_transaction_early**(*transaction*)
Prints the transaction summary early (i.e. without changes)

simple_automation.utils.**print_transaction_title**(*transaction*, *title_color*, *status_char*)
Prints the transaction title and name

8.13 vars

Provides the Vars class.

Classes

Vars()	The Vars class represents a nested dictionary, that allows setting variables in children dicts by using '.' in the key as a delimiter.

8.13.1 simple_automation.vars.Vars

class simple_automation.vars.**Vars**
Bases: object

The Vars class represents a nested dictionary, that allows setting variables in children dicts by using '.' in the key as a delimiter.

Methods

copy	Copies a value from another vars object into this one.
get	Retrieves a variable by the given key.
set	Sets the given variable.

copy(*key*, *other_vars*)
Copies a value from another vars object into this one. Same as calling self.set(key, other_vars.get(key))

Parameters

- **key** (*str*) – The key that should be copied.
- **other_vars** (Vars) – The source variable storage where the key is copied from.

get(*key*, *default=None*)
Retrieves a variable by the given key. If no such key exists, it returns the given default value or throws a KeyError if no default is set.

Parameters

- **key** (*str*) – The key that should be read.
- **default** (*Any, optional*) – If not None, this will be returned in case the key is unset. By default None.

Returns The stored object.

Return type Any

set(*key*, *value*)
Sets the given variable.

Parameters

- **key** (*str*) – The key that should be read.
- **value** (*Any, optional*) – The value to be stored. Must be json (de-)serializable.

8.14 vault

Provides the Vault class for secure variable storage.

Classes

GpgVault(manager, file, recipient)	A GpgVault is a Vault which saves its context encrypted with gpg.
SymmetricVault(manager, file[, keyfile, key])	A SymmetricVault is a Vault which saves its context symmetrically encrypted.
Vault(manager, file)	A base-class for vaults.

8.14.1 simple_automation.vault.GpgVault

class simple_automation.vault.**GpgVault**(*manager*, *file: str*, *recipient: str*)

Bases: *simple_automation.vault.Vault*

A GpgVault is a Vault which saves its context encrypted with gpg. This can be convenient if you e.g. use a YubiKey or similar hardware to store your encryption keys.

Initializes the gpg encrypted vault from the given file and recipient.

Parameters

- **manager** (*Manager*) – The manager to which this vault is registered.
- **file** (*str*) – The file which serves as the permanent storage.
- **recipient** (*str*) – Only needed for encryption (when editing). Reflects the gpg command line parameter '–recipient'. If you don't plan on using the editing feature, the recipient may be set to None.

Methods

copy	Copies a value from another vars object into this one.
decrypt	Decrypts the vault (using self.decrypt_content) and loads the content into our Vars.
decrypt_content	Decrypts the given ciphertext.
edit	Opens an $EDITOR containing the loaded content as a pretty printed json, and updates the internal representation as well as the original vault file, if the content changed after the editor exists.
encrypt	Encrypts the currently stored Vars (using self.encrypt_content) and overwrites the vault file.
encrypt_content	Encrypts the given plaintext.
get	Retrieves a variable by the given key.
set	Sets the given variable.

copy(*key*, *other_vars*)

Copies a value from another vars object into this one. Same as calling self.set(key, other_vars.get(key))

Parameters

- **key** (*str*) – The key that should be copied.
- **other_vars** (*Vars*) – The source variable storage where the key is copied from.

decrypt()

Decrypts the vault (using self.decrypt_content) and loads the content into our Vars.

decrypt_content(*ciphertext: bytes*) → bytes

Decrypts the given ciphertext.

Parameters **ciphertext** (*bytes*) – The bytes to decrypt.

Returns The plaintext

Return type bytes

edit()
Opens an $EDITOR containing the loaded content as a pretty printed json, and updates the internal representation as well as the original vault file, if the content changed after the editor exists.

encrypt() → bytes
Encrypts the currently stored Vars (using self.encrypt_content) and overwrites the vault file.

encrypt_content(*plaintext: bytes*) → bytes
Encrypts the given plaintext.

Parameters **plaintext** (*bytes*) – The bytes to encrypt.

Returns The ciphertext

Return type bytes

get(*key*, *default=None*)
Retrieves a variable by the given key. If no such key exists, it returns the given default value or throws a KeyError if no default is set.

Parameters

- **key** (*str*) – The key that should be read.
- **default** (*Any, optional*) – If not None, this will be returned in case the key is unset. By default None.

Returns The stored object.

Return type Any

set(*key*, *value*)
Sets the given variable.

Parameters

- **key** (*str*) – The key that should be read.
- **value** (*Any, optional*) – The value to be stored. Must be json (de-)serializable.

8.14.2 simple_automation.vault.SymmetricVault

class simple_automation.vault.**SymmetricVault**(*manager*, *file: str*, *keyfile=None*, *key=None*)
Bases: *simple_automation.vault.Vault*

A SymmetricVault is a Vault which saves its context symmetrically encrypted. Content is encrypted with a salted key (+scrypt) using AES-256-GCM.

Initializes the vault from the given file and key/keyfile. If neither key nor keyfile is provided, the key will be read via getpass(). The key may be given as str or bytes. If the key is given a a str, it will automatically be converted to bytes (without encoding) before usage.

Parameters

- **manager** (Manager) – The manager to which this vault is registered.
- **file** (*str*) – The file which serves as the permanent storage.
- **keyfile** (*str, optional*) – A file which contains the decryption key. Defaults to None.
- **key** (*str, optional*) – The decryption key. Defaults to None.

Methods

copy	Copies a value from another vars object into this one.
decrypt	Decrypts the vault (using self.decrypt_content) and loads the content into our Vars.
decrypt_content	Decrypts the given ciphertext.
edit	Opens an $EDITOR containing the loaded content as a pretty printed json, and updates the internal representation as well as the original vault file, if the content changed after the editor exists.
encrypt	Encrypts the currently stored Vars (using self.encrypt_content) and overwrites the vault file.
encrypt_content	Encrypts the given plaintext.
get	Retrieves a variable by the given key.
get_key	Loads the decryption key.
kdf	Derives the actual aeskey from a given salt and the saved key.
set	Sets the given variable.

copy(*key*, *other_vars*)
Copies a value from another vars object into this one. Same as calling self.set(key, other_vars.get(key))

Parameters

- **key** (*str*) – The key that should be copied.
- **other_vars** (Vars) – The source variable storage where the key is copied from.

decrypt()
Decrypts the vault (using self.decrypt_content) and loads the content into our Vars.

decrypt_content(*ciphertext: bytes*) → bytes
Decrypts the given ciphertext.

Parameters **ciphertext** (*bytes*) – The bytes to decrypt.

Returns The plaintext

Return type bytes

edit()
Opens an $EDITOR containing the loaded content as a pretty printed json, and updates the internal representation as well as the original vault file, if the content changed after the editor exists.

encrypt() → bytes
Encrypts the currently stored Vars (using self.encrypt_content) and overwrites the vault file.

encrypt_content(*plaintext: bytes*) → bytes
Encrypts the given plaintext.

Parameters **plaintext** (*bytes*) – The bytes to encrypt.

Returns The ciphertext

Return type bytes

get(*key*, *default=None*)
Retrieves a variable by the given key. If no such key exists, it returns the given default value or throws a KeyError if no default is set.

Parameters

- **key** (*str*) – The key that should be read.
- **default** (*Any, optional*) – If not None, this will be returned in case the key is unset. By default None.

Returns The stored object.

Return type Any

get_key()
Loads the decryption key.

kdf(*salt*)
Derives the actual aeskey from a given salt and the saved key.

set(*key*, *value*)
Sets the given variable.

Parameters

- **key** (*str*) – The key that should be read.
- **value** (*Any, optional*) – The value to be stored. Must be json (de-)serializable.

8.14.3 simple_automation.vault.Vault

class simple_automation.vault.**Vault**(*manager*, *file: str*)
Bases: *simple_automation.vars.Vars*

A base-class for vaults.

Parameters

- **manager** (Manager) – The manager to which this vault is registered.
- **file** (*str*) – The file which serves as the permanent storage.

Methods

copy	Copies a value from another vars object into this one.
decrypt	Decrypts the vault (using self.decrypt_content) and loads the content into our Vars.
decrypt_content	Decrypts the given ciphertext.
edit	Opens an $EDITOR containing the loaded content as a pretty printed json, and updates the internal representation as well as the original vault file, if the content changed after the editor exists.
encrypt	Encrypts the currently stored Vars (using self.encrypt_content) and overwrites the vault file.
encrypt_content	Encrypts the given plaintext.
get	Retrieves a variable by the given key.
set	Sets the given variable.

copy(*key*, *other_vars*)
Copies a value from another vars object into this one. Same as calling self.set(key, other_vars.get(key))

Parameters

- **key** (*str*) – The key that should be copied.
- **other_vars** (Vars) – The source variable storage where the key is copied from.

decrypt()
Decrypts the vault (using self.decrypt_content) and loads the content into our Vars.

decrypt_content(*ciphertext: bytes*) → bytes
Decrypts the given ciphertext. Should be implemented by subclasses.

Parameters **ciphertext** (*bytes*) – The bytes to decrypt.

Returns The plaintext

Return type bytes

edit()
Opens an $EDITOR containing the loaded content as a pretty printed json, and updates the internal representation as well as the original vault file, if the content changed after the editor exists.

encrypt() → bytes
Encrypts the currently stored Vars (using self.encrypt_content) and overwrites the vault file.

encrypt_content(*plaintext: bytes*) → bytes
Encrypts the given plaintext. Should be implemented by subclasses.

Parameters **plaintext** (*bytes*) – The bytes to encrypt.

Returns The ciphertext

Return type bytes

get(*key*, *default=None*)
Retrieves a variable by the given key. If no such key exists, it returns the given default value or throws a KeyError if no default is set.

Parameters

- **key** (*str*) – The key that should be read.
- **default** (*Any, optional*) – If not None, this will be returned in case the key is unset. By default None.

Returns The stored object.

Return type Any

set(*key*, *value*)
Sets the given variable.

Parameters

- **key** (*str*) – The key that should be read.
- **value** (*Any, optional*) – The value to be stored. Must be json (de-)serializable.

8.15 version

Provides version information

CHAPTER

NINE

LICENSE

MIT License

Copyright (c) 2021 oddlama

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

F

G

H

I

K

L

M

P

R

S

T

U

V

W