groundwork Documentation

Release 0.1.13

team useblocks

Mar 27, 2018

Contents

groundwork is a Python based microframework for highly reusable *applications* and their components.
Its functionality is based on exchangeable, well-documented and well-tested *plugins* and *patterns*.

It is designed to support any kind of a Python application: command line scripts, desktop programs or web applications.

groundwork enables applications to activate and deactivate plugins during runtime and to control dynamic plugin behaviors like plugin status, used signals, registered commands and much more.

The functionality of plugins can easily be extended by the usage of inheritable patterns.
Thus, groundwork supports developers with time-saving solutions for:

- *Command line interfaces*
- Loose inter-plugin communication via *signals and receivers*
- *Shared objects* to provide and request content to and from other plugins
- Static and dynamic *documents* for an overall documentation

Additional, ready-to-use solutions can be easily integrated into groundwork applications by the usage of third-party plugins and patterns from the groundwork community (like groundwork-database or groundwork-web). See *Additional Packages for groundwork* for more information.

CHAPTER 1

Example

The following code defines a plugin with command line support and creates a groundwork *application*, which activates the *plugin*:

```
from groundwork import App
from groundwork.patterns import GwCommandsPattern

class MyPlugin(GwCommandsPattern):
    def _init_(self, app, *args, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, *args, **kwargs)

    def activate(self):
        self.commands.register(command='hello',
                               description='prints "hello world"',
                               function=self.greetings)

    def greetings(self):
        print("Hello world")

if __name__ == "__main__":
    my_app = App(plugins=[MyPlugin])        # Creates app and registers MyPlugin
    my_app.plugins.activate(["My Plugin"])  # Initialise and activates 'My Plugin'
    my_app.commands.start_cli()             # Starts the command line interface
```

The following commands can be used on a command line now:

```
python my_app.py hello      # Prints 'Hello world'
python my_app.py            # Prints a list of available commands
python my_app.py hello -h   # Prints syntax help for the hello command
```

CHAPTER 2

Tutorial

For the case you wish a more use-case oriented introduction into groundwork, we have set up a tutorial with a huge amount of code examples.

This tutorial starts with the groundwork basics, lets you create your first groundwork command line application, gives introductions how to add database support and finally ends with your own groundwork based web application.

Beside groundwork itself, it also uses the community packages groundwork-database and groundwork-web.

The tutorial is available under useblocks.github.io/groundwork-tutorial.

CHAPTER 3

User's Guide

3.1 Foreword

3.1.1 Challenges

The initial version of groundwork was created inside an environment of cross-company development teams, with very different skills on Python and its ecosystem.

Main challenges were the needed understanding of already existing code, missing responsibilities for artefacts besides the code (like tests and documentation) and a tight project plan, which never contained time slots needed to teach and update team members about important changes on the code.

Besides these challenges on the project level, there were also a lot of challenges on code level when it came to understanding architecture, databases, algorithms, interactions and more of a running application with dynamic and extensible behavior.

3.1.2 Goals

groundwork was created to take most of these challenges and to provide easy, understandable and plugin-able solutions.

The groundwork team has defined goals, which shall be applicable for all groundwork based applications:

A plugin bundles everything

Besides the code itself, a *plugin* also provide tests, documentation and meta data for its functionality.

Like other application/plugin frameworks, groundwork is responsible to "glue" all plugin code together to a single application.

But it also cares about test cases of a plugin and makes tests of all used plugins available inside a single test suite. Furthermore, it also collects all plugin documentation and creates a single, overall documentation for developers and

users. The meta data of a plugin is collected as well and made available inside the documentation and - if desired - also in the application.

Injections

To lower the learning curve, commonly used libraries and their core functions can directly be injected into groundwork plugins by using groundwork patterns.

For instance: Instead of initialising and configuring Blinker for signals and Click for command line interfaces by yourself, groundworks provides `self.signal.send("Yehaa")` and `self.commands.register(...)` directly inside plugin classes.

By defining own *patterns*, it is very easy to provide team members additional injected functions. E.g. `self.web.route()` for registering a web route or `self.db.sql()` to execute a SQL statement.

However, the library and its objects can still be made available and directly accessible to support uncommon or not yet supported use cases.

Realtime documentation

Nowadays it's really hard to get the big picture of an existing application. Normally only some kind of documentation and the code itself are available as information source. However, the former is rarely well maintained and the later gives you a structured, but too deep view which takes hours or even days to understand.

groundwork tries to retrieve and provide a lot of information from the executed code directly during runtime. For instance, it is able to show registered and used signals or to create a list of available commands.

These information depends on the activated plugins, which may change during runtime and affect the documentation as well.

3.1.3 Technical background

groundwork was created to glue code-snippets from various developers together and make their nested functions easily available.

In Python this is commonly achieved by importing modules, initialising a class of them and storing the class instance in a local or global variable. However, these mechanism aren't really dynamic and the relationship between different classes and objects is hard-coded, without any chance to change it during runtime.

groundwork uses *plugins*, based on cooperative-multi-inheritance to load and manage needed attributes and functions from reusable *patterns*. It also enforces the usage of the groundwork *`GwBasePattern`* for all plugins to make common attributes and functions available.

For more information about cooperative-multi-inheritance see:

- Raymond Hettinger - Super considered super! - PyCon 2015
- Python docs for classes and inheritance

3.2 Installation

Warning:

groundwork does currently support Python3.4 or higher only.

Python2.x is not supported!

3.2.1 System-wide installation

You can use pip to install groundwork in your local python environment:

```
sudo pip install groundwork
```

On windows system **sudo** is not needed.

3.2.2 Virtual environment

A virtual environment allows you to install and test python packages without any affect on a system-wide installation.

If not done yet, use pip to install **virtualenv**:

```
sudo pip install virtualenv
```

Create a virtual environment in your preferred folder with:

```
virtualenv venv      # venv will be the name of the folder. You may change it.
```

To activate it, run:

```
. venv/bin/activate
```

Or on windows:

```
venv\scripts\activate
```

After that your virtual environment is installed and activated. Now you can install groundwork:

```
pip install groundwork
```

3.3 Quickstart

3.3.1 Applications

Create an app

Create a file named **my_app.py** and add the following code:

```
from groundwork import App

if __name__ == "__main__":
    my_app = App()
    my_app.plugins.activate(["GwPluginInfo"])
    my_app.commands.start_cli()
```

This code performs the following actions:

- It creates a groundwork application app via `my_app = App()`

- **It activates the plugin `GwPluginInfo`, which is part of groundwork itself.**
  - During activation, GWPluginInfo registers a command called **`plugin_list`** by its own.
- It starts the command line interface

Run an app

Open a command line interface, change to folder, which contains **my_app.py**, and execute:

```
python my_app.py
```

This will start groundwork and its command line interface.

Because no command was added as parameter, groundwork complains about it. To change this, simply add the needed command

```
python my_app.py plugin_list
```

This will print a list of all available plugins, including some helpful information about them.

3.3.2 Plugins

Activate registered plugins

Activating already registered plugins like `GwPluginInfo` is easy. All you need to know is the name of a plugin.:

```
my_app.activate(["GwPluginInfo", "GwCommandsInfo", "GwSignalInfo"])
```

groundwork knows these names, because it automatically scans the used python environment for packages, which are providing groundwork plugins. See *Plugin registration* for more details or *Packaging and Installation* for using this mechanism for own plugins.

Create own plugins

The easiest way of creating a groundwork plugin is by defining a class, which inherits from *`GwBasePattern`*. But before activation, it also needs to be registered, what can be done during application initialisation:

```
from groundwork import App
from groundwork.patterns import GwBasePattern

class MyPlugin(GwBasePattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self): pass

    def deactivate(self): pass


my_app = App(plugins=[MyPlugin])     # Register your plugin class
my_app.plugins.activate(["My Plugin"])       # And activate it
```

You can also use the plugin object itself to perform the activation:

```
# Instead of
# my_app = App(plugins=[MyPlugin])
# my_app.activate(["My Plugin"])
my_app = App()
my_plugin = MyPlugin(app=my_app)
my_plugin.activate()
```

Note: If a plugin inherits from any pattern, *GwBasePattern* is no longer needed as the pattern itself does already inherit from this class.

Warning: The `__init__` routine of a plugin class **must** always set a name and call the next `__init__` routine in the inheritance chain (in this order!).

Also make sure that your `__init__` can handle **app** as the first argument and additional, optional keyword arguments.

If this is missed, the patterns and their objects are not initialized and configured the right way.

So always use:

```
def __init__(self, app, **kwargs):
    self.name = "My Plugin"
    super().__init__(app, **kwargs)
```

3.3.3 Patterns

Using patterns

Patterns are used to inject new functionality to a plugin. There are patterns for registering commands, generating different types of documentation, activating web support and much more.

A plugin can inherit multiple patterns:

```
class MyPlugin(GwCommandPattern, GwDocumentPattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)
```

This code example gives MyPlugin functions to register new commands and new documents.

If your are using a coding environment with code completion, just type `self.` to see all available functions , including the inherited ones.

Writing patterns

A pattern is more or less a plugin without any **activation** or **deactivation** function. Like plugins, it must also inherit from *GwBasePattern*.

A pattern is allowed to multiply inherit from other patterns as well.

You can find an example with multiple inheritance in the *Pattern Example Code*.

3.4 Architecture

A groundwork application knows 3 levels of abstraction: *applications*, *plugins* and *patterns*.

These three levels were chosen to reflect the need of functional and technical separation. Additionally, it allows granular code reuse and recombination.

3.4.1 Definitions

Application

An *application* bundles several functionalities, which are provided by *plugins*.

Therefore, an *application* is a package of *plugins* and has one or several functional focuses. Like a web app for weather services or a console script for text file manipulation.

Plugin

A *plugin* has a strongly user oriented functional focus. Examples are user account handling, error monitoring or functions for viewing log files.

It may also provide some sort of user interface, like console commands or web pages.

If a plugin needs technical resources like a database connection, a web server or command registration, it needs to use *patterns*.

A plugin can be activated and deactivated during application runtime.

Pattern

A *pattern* provides technical resources to *plugins*. They are responsible for setting up database connections, providing APIs for command registration or web route handling.

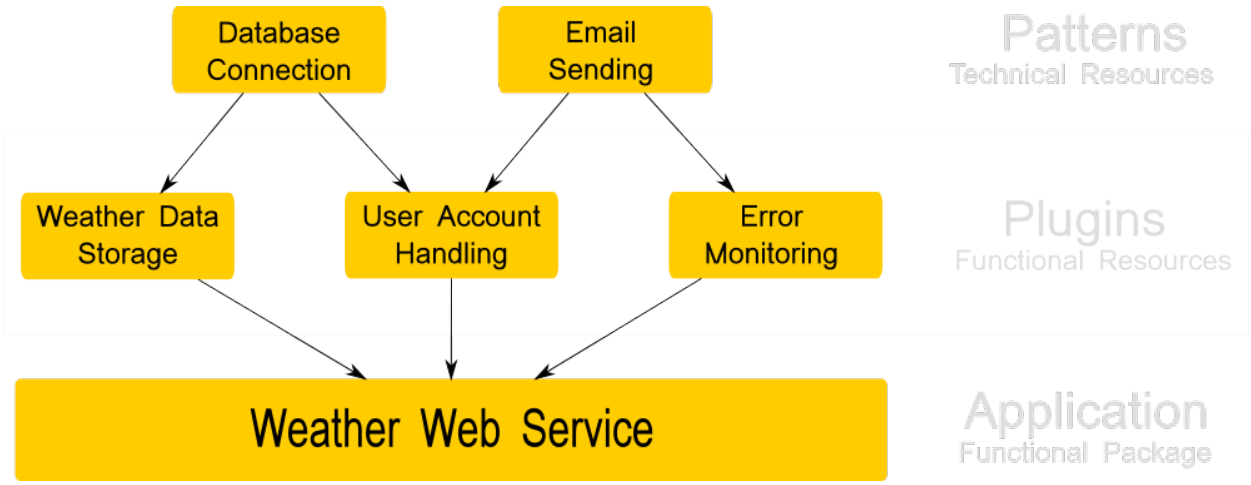
A plugin uses patterns by deriving from pattern classes. Several patterns can be invoked by using multi inheritance. Thus, patterns are strongly coupled with related plugins during application runtime.

Patterns become automatically activated as soon as an application activates the first plugin that inherits from the pattern. A pattern gets automatically deactivated, if all plugins are deactivated, which inherit from the pattern. With plugins deriving from patterns, the plugin activation order becomes unimportant because resources provided by patterns will already be registered on the app if the first plugin wants to use it. This is important for functionality like database connections or web frameworks which are commonly only instantiated once but used for multiple plugins.

3.4.2 Example

The following image shows an application example for a weather web service. The service has 3 main features:

- Provide weather information. Data source is a database.
- Allow user registration, in which users are stored to the database and then receive a welcome email.
- Error handling. Service administrators shall get emails if problems occur.



The 3 features are separated into 3 plugins having a use case related focus: weather storage, user handling and error monitoring.

All plugins need a total of 2 technical resources: A database and a way to send emails. These are realised by 2 different patterns: a database connection pattern and an email sending pattern.

The application itself is configured to load the 3 plugins during startup. The related patterns are getting loaded and configured automatically.

Code examples

The following code snippets give a first impression how such an architecture can be realised.

patterns.py

The following code defines the 2 patterns for database connections and email sending:

```
from groundwork.patterns import GwBasePattern


class DatabasePattern(GwBasePattern):
    def __init__(self):
        self.database = Database()  # Database has functions: store(), get()


class EmailPattern(GwBasePattern):
    def __init__(self):
        self.email = Email()  # Email has functions: send()
```

plugins.py

The 3 features are realised by the following 3 plugins:

```
from .patterns import DatabasePattern, EmailPattern


class WeatherStorePlugin(DatabasePattern):
    def __init__(self, app, **kwargs):
```

```
        self.name = "Weather Store"
        super().__init__(app, **kwargs)

    def activate(self):
        self.database.store(MyWeatherData)

    def get_weather(location):
        return self.database.get(location)

    def deactivate(self):
        pass


class UserHandling(DatabasePattern, EmailPattern):
    def __init__(self, app, **kwargs):
        self.name = "User Handling"
        super().__init__(app, **kwargs)

    def register_user(self, username, email):
        self.database.store(User(username, email))
        self.email.send(email, "Welcome %s" % username)

    def deactivate(self):
        pass


class ErrorMonitoring(EmailPattern):
    def __init__(self, app, **kwargs):
        self.name = "Error Monitoring"
        super().__init__(app, **kwargs)

    def activate(self):
        self.admin = "admin@my_company.com"

    def error_detected(traceback):
        self.email.send(self.admin, "Error found! %s" % traceback)

    def deactivate(self):
        pass
```

app.py

The application itself only needs to load the three plugins:

```
from groundwork import App
from .plugins import WeatherStorePlugin, UserHandling, ErrorMonitoring

# Load application and register plugins
my_app = App(plugins=[WeatherStorePlugin, UserHandling, ErrorMonitoring])

# Activate plugins
my_app.activate(["Weather Store", "User Handling", "Error Monitoring"])
```

3.5 Application

The groundwork *App* is a container mostly for configurations, plugins/patterns and their needed objects.

Most attributes are added during runtime by patterns, which need a single instance of an object per application. For instance: A list of registered commands, a database connection object, a web application.

To initialise a groundwork application, simply do:

```
from groundwork import App

my_app = App()
```

3.5.1 Configuration

groundwork can load multiple configuration files during application initialisation. These values are available via `my_app.config.get("MY_CONFIG_PARAMETER")`.

It is also possible to reload the configuration or to extend it by additional configuration files during runtime:

```
from groundwork import App

my_app = App(config_files=["config1.py", "config2.py"])      # Load 2 config files
my_app.config.load(["config3.py"])                           # Load a third config file
```

A configuration file must be python file, which defines variables on root level. Only variables with an uppercase name are used for the groundwork configuration:

```
# config1.py

import os

APP_NAME = "My awesome application"                 # Is used as config parameter

config_file_location = __file__                     # Is not used as config parameter
APP_PATH = os.path.dirname(config_file_location)    # Is used as config parameter

APP_PLUGINS = [ "My Plugin",                        # Is used as config parameter
                "GwPluginInfo",
                "GwCommandInfo]
```

After application initialisation, the configuration can be used. For instance to activate needed plugins:

```
from groundwork import APP

my_app = App(config_files=["config1.py"])
my_app.plugins.activate(my_app.config.get("APP_PLUGINS"))
```

3.5.2 Plugin registration

Before a plugin can be activated for a groundwork application, it must be registered.

groundwork does this registration automatically for all python packages in the current python environment. For not-packaged plugins, they must be registered by the application developer her/himself.

Packaged plugins

A packaged plugin is part of a python package, which provides a setup.py and was installed via `python setup.py install` or related pip/easy_install commands in the current python environment.

The package must use the entry_point **groundwork.plugin** and provide a class for each entry_point. Example from the groundwork package itself:

```
setup(
    name='groundwork',

    # A lot of other information....

    entry_points={
        'groundwork.plugin': [
            'gw_plugin_info = groundwork.plugins.gw_plugin_info:GwPluginInfo',
            'gw_signal_info = groundwork.plugins.gw_signal_info:GwSignalInfo',
            'gw_command_info = groundwork.plugins.gw_commands_info:GwCommandInfo'
        ]
    }
)
```

During application initialisation, groundwork registers all plugins, which are provided by this way automatically. They can be activated after app initialisation:

```
from groundwork import App

my_app = App()
my_app.plugins.activate(["GwPluginInfo", "GwSignalInfo"])
```

Registration of own plugins

If a groundwork plugin is not part of a package and not made available via entry_point, it must be registered by the application developer. This can be done during application initialisation or later:

```
from groundwork import App
from groundwork.patterns import GwBasePattern

class MyPlugin(GwBasePattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self): pass

    def deactivate(self): pass


# Registration during initialisation
my_app = App(plugins=[MyPlugin])

# Registration after initialisation
from my_module import AnotherPlugin
my_app.plugins.classes.register([AnotherPlugin])

# Activation
my_app.plugins.activate(["My Plugin", "AnotherPlugin"])
```

3.5.3 Plugin activation

Before a plugin registers its commands, signals or anything else, it must be activated.

groundwork supports two ways of activation:

- Activation by application
- Activation by plugin

Here is an example, which demonstrates both ways:

```
from groundwork import App
from groundwork.patterns import GwBasePattern

class MyPlugin(GwBasePattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self): pass

    def deactivate(self): pass

# Activation by application
my_app = App(plugins=[MyPlugin])                          # Registration
my_app.plugins.activate(["My Plugin"])                    # Activation

# Activation by plugin
my_plugin2 = MyPlugin(app=my_app, name="MyPlugin2")       # Registration
my_plugin2.activate()                                     # Activation
```

3.5.4 Plugin deactivation

Like for plugin activation, also the plugin deactivation supports two ways of deactivating a plugin:

```
# Follow up of the plugin activation example...

# Deactivation by application
my_app.deactivate(["MyPlugin"])

# Deactivation by plugin
my_plugin2.deactivate()
```

3.5.5 Handling errors

A plugin registration or activation can easily fail. Reasons may be bad code, missing dependencies, already registered classes and more.

By default groundwork logs only a warning if a registration or activation fails.

You can ask groundwork to throw also an exception, if problems occur. This behavior can be activated by setting the parameter `strict=True` during application initialisation:

```
from groundwork import App

class MyBadPlugin():
```

```
    pass

my_app = App(strict=True)
my_app.registers([MyBadPlugin])      # will throw an exception

my_app.strict = False
my_app.registers([MyBadPlugin])      # will log a warning only
```

3.5.6 Logging

A groundwork application provides its own logger object, which is available under `my_app.log`:

```
from groundwork import App

my_app = App()
my_app.log.info("Loading plugins")
my_app.log.debug("Activating Plugin X")
```

This logger is used by most application related objects. Plugins have their own logger, which is available under `self.log` inside an plugin class.

Configuration

All loggers (application and plugins) can be configured by a configuration parameter called **GROUNDWORK_LOGGING** inside a used configuration file.

The value of this parameter must be a dictionary. Its structure is described in the python documentation for logging.

Example of a configuration for groundwork logs:

```
GROUNDWORK_LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'standard': {
            'format': '%(asctime)s [%(levelname)s] %(name)s: %(message)s'
        },
        'extended': {
            'format': "%(levelname)-8s %(name)-40s - %(asctime)s - %(message)s"
        },
        'debug': {
            'format': "%(name)s - %(asctime)s - [%(levelname)s] - %(module)s:
↪%(funcName)s(%(lineno)s) - %(message)s"
        },
    },
    'handlers': {
        'default': {
            'formatter': 'standard',
            'class': 'logging.StreamHandler',
            'level': 'DEBUG'
        },
        'console_stdout': {
            'formatter': 'extended',
            'class': 'logging.StreamHandler',
            'stream': sys.stdout,
            'level': 'DEBUG'
```

```
        },
        'file': {
            "class": "logging.handlers.RotatingFileHandler",
            "formatter": "debug",
            "filename": "logs/app.log",
            "maxBytes": 1024000,
            "backupCount": 3,
            'level': 'DEBUG'
        },
        'file_my_plugin': {
            "class": "logging.handlers.RotatingFileHandler",
            "formatter": "debug",
            "filename": "logs/my_plugin.log",
            "maxBytes": 1024000,
            "backupCount": 3,
            'level': 'DEBUG'
        },
    },
    'loggers': {
        '': {
            'handlers': ['default'],
            'level': 'WARNING',
            'propagate': True
        },
        'groundwork': {
            'handlers': ['console_stdout', 'file'],
            'level': 'INFO',
            'propagate': False
        },
        'MyPlugin': {
            'handlers': ['console_stdout', 'file_my_plugin'],
            'level': 'DEBUG',
            'propagate': False
        },
    }
}
```

3.6 Plugins

Plugins are used by groundwork to load specific functions into a groundwork *application*.

In most cases a plugin should have a functional focus, like providing some documentation about signals or providing some commands to the user to handle specific tasks on data.

A plugin can be activated and deactivated during runtime. And it can be loaded from python packages or from own code.

The following rules apply for each groundwork plugin.

- A plugin contains code, documentation and tests.
- A plugin provides routines for activation and deactivation during runtime.
- A plugin inherits directly or indirectly from *GwBasePattern*.

During development of a plugin, *patterns* can be used to extend its functionality or grant access to specific objects, like a database engine to perform database actions.

3.6.1 Registration

The registration of a plugin must happen by using the application:

```
from groundwork import App

my_app = App()
my_app.plugins.activate(["GwPluginInfo"])
```

For more information please read *Plugin registration* of the chapter *Application*.

3.6.2 Activation and Deactivation

Plugins can be activated and deactivated during runtime. There are two ways of doing this:

- By the *activate()*/ *deactivate()* function, accessible by `my_app.plugins.activate()` or `my_app.plugins.deactivate()`.
- By the activation/deactivation function of the plugin itself, accessible by `my_plugin.activate()` or `my_plugin.deactivate()`.

For a code example, please take a look into *Plugin activation* and *Plugin deactivation* of the application documentation.

3.6.3 Development of own plugins

To start the development of own plugins, simply create a new class and inherit from *GwBasePattern*:

```
from groundwork.patterns import GwBasePattern

class MyPlugin(GwBasePattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self): pass

    def deactivate(self): pass
```

Warning: It is very important to call the `__init__` routine of parent classes. Otherwise they can't deliver functions and objects, which you may need. Also no signals are registered, which inform interested functions when your plugin gets activated or deactivated. So no automatic cleanup would happen, like erasing all registered commands of your plugin.

Also make sure that your `__init__` can handle **app** as the first argument and additional, optional keyword arguments.

Provided variables

The groundwork *GwBasePattern* creates the following variables for your plugin and makes them directly available:

- **self.path**: The absolute path of the python-file, which contains your plugin (directory + file name)
- **self.dir**: The absolute directory, which contains your plugin (directory only)
- **self.file**: The name of the file, which contains your plugin (file name only)

- **self.version**: An initial version (0.0.1), if this was not set by your plugin during initialisation
- **self.active**: True, if the plugin got activated.
- **self.needed_plugins**: Empty tuple, if it was not set by your plugin during initialisation

Using signals and receivers

You are free to add signals or connect receivers to them:

```
from groundwork.patterns import GwBasePattern

class MyPlugin(GwBasePattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self):
        self.signals.register(signal="My signal",
                              description="Informing about something")

        self.signals.connect(receiver="My signal receiver",
                             signal="My signal",
                             function=self.fancy_stuff,
                             description="Doing some fancy stuff")

    def fancy_stuff(plugin, **kwargs):
        print("FANCY STUFF!!! " * 50)
```

For more details about signals, please read *Signals and Receivers*.

Note: Each plugin sends automatically signals when it gets activated or deactivated. The used signals are: plugin_activate_pre, plugin_activate_post, plugin_deactivate_pre and plugin_deactivate_post.

Please see *Signals and Receivers* for more information.

3.6.4 Using patterns

Patterns can be used to extend your plugin with new functions and objects.

groundwork itself provides 6 patterns:

- *GwBasePattern*
- *GwCommandsPattern*
- *GwDocumentsPattern*
- *GwRecipesPattern*
- *GwSharedObjectsPattern*
- *GwThreadsPattern*

You can load multiple patterns into your plugin:

```
from groundwork.patterns import GwCommandsPattern, GwDocumentsPattern,␣
↪GwSharedObjectsPattern

# GwBasePattern is no longer needed, because the used patterns already inherit from␣
↪it.
class MyPlugin(GwCommandsPattern, GwDocumentsPattern, GwSharedObjectsPattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self):
        self.commands.register(...)
        self.documents.register(...)
        self.shared_objects.register(...)
```

For more information about these patterns, please read the related chapters: *Commands*, *Documents*, *Recipes*, *Shared Objects* and *Threads*.

3.6.5 Logging

Each plugin has its own logger, which name is the name of the plugin. It is accessible via `self.log` inside a plugin class:

```
from groundwork.patterns import GwBasePattern

class MyPlugin(GwBasePattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)
        self.log.info("Initialisation done for %s" % self.name)

    def activate(self):
        self.log.debug("Starting activation")
        self.log.info("Activation done")
```

For each logger, and therefore for each plugin, it is possible to register handlers to monitor specific plugins and log messages in detail.

For instance: Store all messages of “My Plugin” inside a file called “my_plugin.log”. All other messages go to “app.log”.

For details how to configure groundworks logging, please see *logging configuration*.

3.6.6 Plugin dependencies

A plugin can have dependencies to other plugins and it needs to be sure that these plugins are activated in the current app.

Therefore a plugin can specify the names of needed plugins and groundwork cares about their activation:

```
from groundwork.patterns import GwBasePattern

class MyPlugin(GwBasePattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
```

```
        self.needed_plugins = ("AnotherPlugin", "AndAnotherPlugin")
        super().__init__(app, **kwargs)
```

During plugin activation, groundwork does the following:

- Read in `self.needed_plugins`
- For each plugin name
  - Check, if a plugin with this name exists in app.plugins (objects/instantiated plugins)
    - If yes: activate it instantly (if not done yet)
    - If no: check for plugin classes with this name in app.plugin.classes (classes, not instantiated)
      - If yes: Instantiate and activate it
      - If not: Throw error

3.7 Patterns

Patterns are used to extend the functionality of a plugin. So most patterns provides use-case specific functions like register commands, store users and more.

3.7.1 Using patterns

The usage of a pattern is defined by a plugin during its development. The plugin itself decides to inherit from one or multiple patterns:

```
from groundwork.patterns import GwCommandsPattern, GwDocumentsPattern, GwSharedObjects

class MyPlugin(GwCommandsPattern, GwDocumentsPattern):          # Used Patterns
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        self.super().__init__(app, **kwargs)
```

This inheritance can not be changed during runtime or via configuration. It's hard coded inside a plugins code.

3.7.2 Developing own patterns

A pattern is more or less a plugin without any **activation** or **deactivation** function. Like plugins, it must also inherit from *GwBasePattern*. A pattern is allowed to multiply inherit from other patterns as well. Example:

```
from groundwork import App
from groundwork.patterns import GwCommandPattern, GwDocumentPattern

class MyPattern(GwCommandPattern, GwDocumentPattern):
    def __init__(self, app, **kwargs):
        super().__init__(app, **kwargs)

    def my_register(self, command_name, command_func):
    """ Registers and documents a new command"""
        self.commands.register(command_name, command_func, ...)
        self.documents.register(command_name, ...)
```

```
class MyPlugin(MyPattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self):
        # Your new function
        self.my_register(command_name = "print_me", command_func = self.print_me)

        # But you also have access to all functions from
        # GwCommandPattern and GwDocumentPattern
        self.commands.register(...)
        self.documents.register(...)

    def print_me(self):
        print("I'm %s." % self.name)

my_app = App([MyPlugin])
my_app.activate(["My Plugin"])
```

3.7.3 Logging

Patterns are using the same logger as the plugin, which has inherit from this pattern. Example:

```
from groundwork import App
from groundwork.patterns import GwBasePattern


class MyPattern(GwBasePatter):
    def __init__(self, app, **kwargs):
        super().__init__(app, **kwargs)

        self.log.debug("Initialising pattern 'MyPattern'")


class MyPlugin(MyPattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

        self.log.info("Initialising MyPlugin")


class AnotherPlugin(MyPattern):
    def __init__(self, app, **kwargs):
        self.name = "Another Plugin"
        super().__init__(app, **kwargs)

        self.log.info("Initialising AnotherPlugin")


my_app = App(plugins=[MyPlugin, AnotherPlugin])
my_app.plugins.activate(["My Plugin", "Another Plugin"])

my_app.log.info("Start application")
```

The output of this would be like:

```
MyPlugin         DEBUG    Initialising pattern 'MyPattern'
MyPlugin         INFO     Initialising MyPlugin
AnotherPlugin    DEBUG    Initialising pattern 'MyPattern'
AnotherPlugin    INFO     Initialising MyPlugin
groundwork       INFO     Start application
```

For more details about logging see *Plugin Logging* and *Application Logging*

3.8 Signals and Receivers

Signals and receivers are used to loosely connect plugins. Every plugin can register and send signals. And every plugin can register a receiver for a specific signal.

A signal is defined by its unique name and should have a meaningful description.

A receiver is connected to a specific signal and is defined additionally by an unique name, a function and a description.

groundwork also stores the plugin, which has registered a signal or a receiver.

Note: groundwork is internally using the library Blinker and made most of its functions available.

3.8.1 Use case: User creation

Let's imagine we have 3 active plugins:

- **GwUserManager** - For creating users in database
- **GwEMail** - For sending e-mails
- **GwChat** - For sending chat messages

If a user gets created, the GwUSerManager sends the signal "User created" and adds the created user object.

Both, GwEMail and GwChat, have registered receivers to the signal "User created". So GwEMail gets called, it fetches the e-mail address from the attached user object and sends a "Welcome" message to the user. GwChat gets also called and sends a chat message to the chat room of the development team and informs them that a new user has been created.

3.8.2 Working with signals

Signals and receivers can be used inside plugins, without the need of using any specific pattern. As groundwork itself uses signals for some internal processes, signals and receivers are already part of *GwBasePattern*.

Register a signal

To register a signal, simply use the *register()* function of self.signals:

```
from grundwork.patters import GwBasePattern

class MyPlugin(GwBasePattern):
    def __init__(self, app, **kwargs):
```

```
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self):
        self.signals.register("my_signal", "this is my first signal")
```

You are able to get all signals, which were registered by your plugin by using *get()*:

```
...
def activate(self):
    self.signals.register("my_signal", "this is my first signal")
    my_signals = self.signals.get()                              # Returns a dictionary
    my_single_signal = self.signals.get(signal="my_signal")      # Return Signal or␣
↪None
```

Send a signal

Sending a signal can be done by every plugin, even if it has not registered any signals or receivers.

However, a signal, which shall be send, must already be registered. Otherwise an exception is thrown.:

```
...
def activate(self):
    self.signals.register(signal ="my_signal",
                          description="this is my first signal")

    self.signals.send("my_signal")  # Will work
    self.signals.send("not_registered__signal")  # Will throw an exception
```

Note: Also the application can send signals by using *send()*, like `my_app.signals.send("my_signal", plugin=self)`.

Signals installed by groundwork

groundwork installs 4 signals during start up:

- plugin_activate_pre
- plugin_activate_post
- plugin_deactivate_pre
- plugin_activate_post

This signals are called automatically if a plugin gets activated or deactivated.

The difference between **pre** and **post** is that **pre** is called before any action is done by the plugin. And **post** is called after the plugin did some action for de/activation.

3.8.3 Working with receivers

Any plugin can register a receiver for any signal. Even if the signal itself will never be send or even registered.

Register a receiver

To register a receiver, a callback function is needed, which gets executed, if the receiver gets called.

Registration of receiver is done by the function *connect()*:

```
from grundwork.patters import GwBasePattern

class MyPlugin(GwBasePattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self):
        self.signals.connect(receiver="My signal receiver",
                             signal="My signal",
                             function=self.fancy_stuff,
                             description="Doing some fancy")

    def fancy_stuff(plugin, **kwargs):
        print("FANCY STUFF!!! " * 50)
```

The used function must accept as first parameter the sender/plugin, which send the signal. After this multiple, optional keyword arguments must be accepted as well.

The parameter **sender** can be used during registration to receive signals only from specific senders/plugins.

Best practice: Pattern clean up

Lets say, a pattern provides a function to register web-routes. During activation, the plugin registers some of them. But during deactivation is forgets to unregister them, so that they are still registered and available.

The pattern should register to **plugin_deactivate_post** and make sure that everything gets unregistered.

Example:

```
class GwWebPattern(GwBasePattern):
    def __init__(self, app, **kwargs):
        self.signals.connect(receiver="%s_web_route_deactivation" % self.name,
                             signal="plugin_deactivate_post",
                             function=self.__deactivate_commands,
                             description="Deactivate commands for %s" % self.name,
                             sender=self)    # We only need signals from this plugin

    def __deactivate_web_routes(self, plugin, *args, **kwargs):
        web_routes = self.web_routes.get()
        for web_route in web_routes.keys():
            self.web_routes.unregister(web_route)
```

Unregister a receiver

To disconnect a receiver from a signal, use the *disconnect()* function:

```
class MyPlugin(GwBasePattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)
```

```
    def activate(self):
        self.signals.connect(receiver="%s_my_deactivation" % self.name, ... )

    def deactivate(self):
        self.signals.disconnect("%s_my_deactivation" % self.name)
```

3.8.4 Signals and receivers on application level

All signals and receivers can be accessed on application level via *get()*:

```
from groundwork import App

my_app = App()
my_app.signals.register("app_signal", "signal from application", plugin=app)
signals = my_app.signals.get()
```

It is also possible to register new signals and receivers. But inside the application an additional parameter called **plugin** is necessary. This parameter gets set automatically inside plugins. However on application level this must be set by the developer.

3.9 Commands

Commands are used to provide access to different function via a command line interface (CLI).

groundwork cares automatically about CLI setup, help messages and command arguments.

However the command line interface must be started by the application itself.

3.9.1 Starting the CLI

To start the cli, be sure that at least one plugin gets activated, which is using the pattern *GwCommandsPattern*.

After application initialisation and plugin activations, *start_cli()* must be called:

```
from groundwork import App
from groundword.plugins import GwCommandInfo

my_app = App()
my_app.plugins.activate(["GwCommandInfo"])
my_app.commands.start_cli()
```

3.9.2 Registering commands

To register commands, a plugin must inherit from *GwCommandsPattern* and use the function *register()*.

```
from groundwork.patterns import GwCommandsPattern

class MyPlugin(GwCommandsPattern):
    def __init__(self, app, **kwargs)
        self.name = "My Plugin"
        super().__init__(app, **kwargs)
```

```
    def activate(self):
        self.commands.register(command="my_command",
                               description="executes something",
                               function=self.my_command,
                               params=[])

    def my_command(self, plugin, **kwargs):
        print("Yehaaa")
```

3.9.3 Using arguments and options

groundworks's command line support is based on click.

For arguments and options, groundwork is using the definition and native classes of click:

- Arguments are positional parameters to a command
- Options are usually optional value on a command.

To use them, you have to pass instances of them to the `params` parameter of the function *`register()`*.

```
from groundwork.patterns import GwCommandsPattern
from click import Argument, Option

class MyPlugin(GwCommandsPattern):
    def __init__(self, app, **kwargs)
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self):
        self.commands.register(command="my_command",
                               description="executes something",
                               function=self.my_command,
                               params=[Option(("--force", "-f"),
                                              required=False,
                                              help="Will force something...",
                                              default=False,
                                              is_flag=True)])

    def my_command(self, plugin, force, **kwargs):
        if force:
            print("FORCE Yehaaa")
        else:
            print("Maybe Yehaaa")
```

For detailed parameter description, please take a look into the documentation of click for arguments and options

3.9.4 Unregister a command

A command can also be unregistered during runtime.

Simply use *`unregister()`* and pass the name of the command:

```
...
```

```
def deactivate(self):
    self.commands.unregister("my_command")
```

3.10 Shared Objects

Shared objects are used to provide or access objects to or from other plugins.

As example, a plugin may be responsible for creating and updating users by setting up a database and make some tests before any change happens. It could provide a shared object, which functions allow other plugins to create users quite easily without the need to know all the details (database, tests, ...).

There are no restrictions for a shared object, it can be any python object.

Note: As a shared object can be anything, you should be sure that this object is really good documented for other plugin developers.

And if you access a shared object, you should also make some tests to guarantee that the shared object behaves like expected.

3.10.1 Registration

Like for commands or signals, there is also a *register()* function for shared objects:

```
from groundwork.patterns import GwSharedObjectsPattern

class MyPlugin(GwSharedObjectsPattern):
    def __init__(self, app, **kwargs)
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

        self.my_shared_object = {"name": "shared"
                                 "name2": "object"}

    def activate(self):
        self.shared_objects.register(name="my_shared_object",
                                     description="A shared object of My Plugin",
                                     obj=self.my_shared_object)
```

3.10.2 Get/Access a shared object

There are 2 functions, to access a shared object:

- *get()*
- *access()*

`get()` returns the complete shared object including registered meta data like name, description and plugin. It may also return a dictionary of shared objects, if no name was given. The search is performed on plugin level only, so there is no possibility to access shared objects of other plugins via `get()`

`access()` returns the object only, without any meta data. It can be used to access a single shared object only. A **name** must be given and the search is performed on application level:

```
from groundwork.patterns import GwSharedObjectsPattern

class MyPlugin(GwSharedObjectsPattern):
    ...

    def activate(self):
        self.shared_objects.register(name="my_shared_object",
                                     description="A shared object of My Plugin",
                                     obj=self.my_shared_object)

class MyPlugin2(GwSharedObjectsPattern):
    ...

    def some_function(self):
        # Will work
        obj = self.shared_objects.access("my_shared_object")

        # The following will not work as "my_shared_object" was not registered by␣
↪this plugin
        # get() only works on plugin level!
        shared_object = self.shared_objects.get("my_shared_object")

        # But if access to shared object meta data is needed, you can use the␣
↪application to get it.
        shared_object = self.app.shared_objects.get(name="my_shared_object")
        obj = shared_object.obj
```

3.10.3 Unregister

Use *unregister()* to unregister a shared object:

```
...
def deactivate(self):
    self.shared_objects.unregister("my_shared_object")
```

Warning: Unregistration of a shared object may be tricky, as other plugins may have already stored a reference to this object. Therefore as a plugin developer do not store an external shared object in your own plugin class. Try to safely request it via *access()* every time you need access on it.

3.11 Documents

Documents are used to describe functions and usage of a plugin to an end-user.

Their output is independent, so that plugins can collect them and create documentations in different formats, like console output, html pages or whatever is needed.

groundwork documents support Jinja and rst. Based on this, they are not static and can be easily used to document dynamic behaviors of an application. For instance to provide a list of available commands.

3.11.1 Live example

groundwork provides an easy console viewer for its basic_app. It is part of the *GwDocumentsInfo* plugin.

After installation of groundwork, simply type the following in a console window:

```
groundwork doc
```

Use **N** and **P** to navigate. **X** to quit.

3.11.2 Registration

To register a document, a plugin must inherit from *GwDocumentsPattern* and call the *register()* function:

```
from groundwork.patterns import GwDocumentsPattern

class My_Plugin(GwDocumentsPattern):
    def __init__(self, app, **kwargs):
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self):
        my_content = """
        My Plugin
        =========
        Application name: {{app.name}}
        Plugin name: {{plugin.name}}
        """
        self.documents.register(name="my_document",
                                content=my_content,
                                description="Provides information about 'My Plugin'")
```

3.11.3 Unregister document

To unregister a document, you must use *unregister()*:

```
...
def deactivate(self):
    self.documents.unregister("my_document")
```

3.11.4 Using Jinja and RST

Jinja and rst are powerful, wide used and well documented libraries for creating intelligent and beautiful documents.

Jinja

Jinja is template engine and allows a developer to use variables and loops inside a text document (besides a lot of more awesome stuff).

groundwork provides the application object as `app` and the plugin object, which has registered the document, as `plugin` to each template:

```
# JINJA template

Application name: {{ app.name }}

{% if app.plugins.get()|count > 5 %}
    Wohooow, we have a lot of plugins!
{% else %}
    Ok, we have some plugins.
{% endif %}

{# get() provides a dict, so we use items() to iterate over it #}
{% for key, plugin in app.plugins.get().items() %}
    name: plugin.name
{% endfor %}
```

The template engine must be executed by the plugin, which provides a viewer to these documents. And the execution should be done directly before the document gets presented to the user.

rst

Restructured Text is used to give your document some sort of a layout. For instance add titles and chapters, make some words strong and add some links.

rst is so generic, that it can be used to build pdf documents, html webpages, epub (an ebook format) and much more.

A famous rst based documentation framework is Sphinx

For a quick introduction, please read Quick reStructuredText.

3.11.5 Developing a document viewer

A viewer for the groundwork documents must care about the following functions:

1. Render the Jinja template string.
2. Transform rst-content to the needed output.

Step 1: Render Jinja

Step 1 can be done using the Jinja template and its from_string() command:

```
from jinja2 import Environment

...  # App initialisation, plugin activation, ...

document = my_app.documents.get("my example document")
rendered_doc = Environment().from_string(document.content).render(app=my_app,␣
↪plugin=document.plugin))
```

It is important to provide 2 parameters to the jinja template:

- **app**: the current application object
- **plugin**: the plugin, which has registered the current document

Step 2: Transform rst

The second step depends on the needed output format. You will find a wide range of rst supports for different programming languages. A good starting point is a list of rst supporting libraries and tools in this stackoverflow answer.

However, the following example will make *html* from an already rendered, rst structured document content:

```
from docutils.core import publish_parts

...  # App initialisation, plugin activation, jinja rendering, ...

output = publish_parts(rendered_doc, writer_name="html")['html_body']
```

`publish_parts()` renders the rst string and provides several groups of html areas. Based on this it is very easy to get the complete html tree or the body content only. Which would be really helpful, if a document should be integrated into an already existing html frame.

Supported areas are: body_prefix, fragment, html_subtitle, header, version, meta, stylesheet, subtitle, html_head, body_pre_docinfo, head, html_body, body, html_prolog, title, docinfo, html_title, whole, body_suffix, head_prefix, footer, encoding.

For details of `publish_parts()` and its supported part names, please take a look into the official documentation.

3.11.6 Sphinx support

Sphinx is a documentation builder, which takes static, rst based files and generates websites, PDFs and more out of it. For instance, this documentation is using sphinx.

As sphinx supports physical files on a hard disk only, it can not integrate with groundwork documents directly.

Luckily the groundwork plugin *`GwDocumentsInfo`* provides the command `doc_write` to store the content of all registered documents of an application in a directory.

Before it writes the files, the command will give you an overview about what will happen and asks for a final confirmation.

Examples:

```
# On a command line

groundwork doc_write ../temp            # Writes rst documents to given, relative␣
↪path.

groundwork doc_write /home/user/temp    # Writes rst documents to the given, absolute␣
↪path.

groundwork doc_write ./temp -h          # Writes HTML documents.

groundwork doc_write ./temp -o          # Does not exit, if given directory is not␣
↪empty.

groundwork doc_write ./temp -q          # Does not ask for final confirmation. Most␣
↪needed by automation scripts.

groundwork doc_write ./temp -o -q -h    # All options together...
```

After export, you can use the generated rst files as normal input files for sphinx. For instance you can add them to a `.. toctree::` of your index.rst.

Note: The output filename of a document is the document name in lowercase. Also all whitespaces are removed. For instance: “My Great Document” becomes “mygreatdocument.rst”

3.12 Threads

Threads are used to allow functions to run in background and in parallel to the application, so that these functions do not block the execution of the app.

This is very helpful when you have long-running tasks (e.g. file operations) but your app must still be able to response to user input very quickly (like a running webserver).

3.12.1 Registering threads

To register threads, a plugin must inherit from *GwThreadsPattern* and use the function *register()*.

```
from groundwork.patterns import GwThreadsPattern

class MyPlugin(GwThreadsPattern):
    def __init__(self, app, **kwargs)
        self.name = "My Plugin"
        super().__init__(app, **kwargs)

    def activate(self):
        my_thread = self.threads.register(name="my_thread",
                                          description="run something",
                                          function=self.my_thread)

        my_thread.run()

    def my_thread(self, plugin, **kwargs):
        print("Yehaaa")
```

The registered function must have two arguments: `self` and `plugin`.

As the function gets not executed in the context of the plugin class, but in the context of a threading class, `self` can not help to get access to your plugin.

Therefore we need the argument `plugin`, which contains the plugin, which has registered the thread.

3.12.2 Thread status and response

Because threads are running in parallel to the normal execution, you can not simply catch the response value of `my_thread.run()`. Following code does **not** work:

```
response = my_thread.run()    # response will be None, because my_thread is still␣
↪running
```

Instead you have to wait and monitor the thread by your own:

```
while my_thread.running:
    pass  # Do nothing
response = my_thread.response
```

But again, this code would block your application.

Another approach would be to let your thread-function send a *signal* as last action. Now you are able to define a *receiver*, which can catch the response.

3.13 Recipes

Recipes are used to generate directories and files based on given user input.

They are most used to speed up the set up of

- New Python packages
- New groundwork projects
- New groundwork applications, plugins or patterns.
- New own projects

Besides folder structures and needed files, they can also be used to provide project/company specific values for some preconfiguration. These values may be:

- Contact details of project leader or IT administrators.
- Common links to general documentation like IT security rules or project handbook.
- Source and integrations of corporate designs like css files or office templates.
- Company wide used libraries and their ready-to-use integration
- Configurations for external IT services, like continuous integration systems and bug trackers.
- Whatever is needed...

Recipes makes it possible to start relevant coding in less than 30 seconds after a new project was set up. Without missing any rules, designs, integrations, checks or whatever is required for the current project.

Note: groundwork recipes are based on cookiecutter and supports every function of it. To get a deep understandig of what is possible with groundwork recipes, you should take a look into cookiecutter's documentation as well.

3.13.1 Workflow

So, what happens if a recipe gets executed? Here is the workflow:

1. Run *groundwork recipe_build gw_package*. **gw_package** is provided by groundwork, but can be replaced by any other recipe.
2. The user gets asked on command line interface for some variable inputs.
3. The recipe gets executed and uses the user's input to create folders and files with input related names.
4. In most cases the input is also used to become part of some files. For instance a README file may contain the author's name after generation.

3.13.2 List available recipes

groundwork knows all available recipes of a groundwork application. And if this app has loaded the *Gw Recipe Builder* plugin, it provides the command *recipe_list* to get a list of all registered recipes.

The groundwork application itself already has some usable recipes. Just execute the following to get a complete list:

```
groundwork recipe_list
```

recipe gw_package

At least you can see a recipe called **gw_package**. This recipe creates a ready-to-use, groundwork based Python package. Including an example groundwork application and plugin, configured sphinx project for documentation, configured test cases with pytest and a test environment based on tox, travis support, ...

3.13.3 Building/Execute a recipe

To build/execute a recipe simply open a command line interface and move to the directory, where the initial recipe folder shall be created. Then execute:

```
groundwork recipe_build RECIPE_NAME

# For instance
groundwork recipe_build gw_package
```

Based on the recipe, you may get asked some questions, which mostly affects the naming of files and directories.

After the last question is answered, groundwork executes the recipe and everything gets created. After this there should be a new folder inside your current working directory.

3.13.4 Creating own recipes

Registration

Own recipes must be registered by a plugin, which needs to give the following data during registration:

- name of the recipe
- absolute path of the recipe directory
- description of the recipe
- final words, which will be printed after an recipe was executed (optional)

See the following code from the RecipeBuilder plugin to get an example:

```
class GwRecipesBuilder(GwCommandsPattern, GwRecipesPattern):
    def __init__(self, *args, **kwargs):
        self.name = self.__class__.__name__
        super().__init__(*args, **kwargs)

    def activate(self):
        ...
        self.recipes.register("gw_package",
                              os.path.abspath(os.path.join(os.path.dirname(__file__),
↪"../recipes/gw_package")),
```

```
                              description="Groundwork basic package. Includes places␣
↪for "
                                          "apps, plugins, patterns and recipes.",
                              final_words="Recipe Installation is done.\n\n"
                                          "For installation run: 'python setup.py␣
↪develop' \n"
                                          "For documentation run: 'make html' inside␣
↪doc folder "
                                          "(after installation!)\n\n"
                                          "For more information, please take a look␣
↪into the README file "
                                          "to know how to go on.\n"
                                          "For help visit: https://groundwork.
↪readthedocs.io\n\n"
                                          "Have fun with your groundwork package.")
```

Structure

A recipe must follow the rules of cookiecutter. Therefore it needs to have the following structure:

```
/
|-- cookiecutter.json
|
|-- {{ cookiecutter.project_name}}
|   |
|   |-- other directories/files, which will be copied.
|
|-- other directories/files, which will NOT be copied
```

Note: It is important to have a **cookiecutter.json** file, as well as a single root-directory, which name is surrounded by {{ }}.

cookiecutter.json

The **cookiecutter.json** file is used as configuration file and must hold a json string, which defines all needed parameters for the recipe setup.

All these parameters can be used and access in directory / file names as well as in file content.

Structure

The following example for a **cookiecutter.json** file comes from the RecipeBuilder plugin:

```
{
 "full_name": "My Name",
 "github_user" : "{{cookiecutter.full_name.lower().replace(' ', '_') }}",
 "email": "{{cookiecutter.github_user}}@provider.com",
 "project_name": "My Package",
 "project_slug": "{{ cookiecutter.project_name.lower().replace(' ', '_') }}",
 "github_project_name": "{{cookiecutter.project_slug}}",
 "project_app": "{{cookiecutter.project_slug}}_app",
 "project_plugin": "{{cookiecutter.project_slug}}_plugin",
```

```
 "project_short_description": "Package for hosting groundwork apps and plugins like {
↪{cookiecutter.project_app}} or {{cookiecutter.project_plugin}}.",
 "test_folder": "tests",
 "test_prefix": "test_",
 "version": "0.1.0",
 "license": ["MIT license", "BSD license", "ISC license", "Apache Software License 2.0
↪", "GNU General Public License v3", "Not open source"]
}
```

Usage

The parameters from the configuration files are all accessible by using **{{cookiecutter.PARAMETER}}**, wherever you want to use this value:

- Directory names
- File names
- File content
- cookiecutter.json

Note: As the parameters are also accessible in the **cookiecutter.json** file, you are free to manipulate an input and use it as default value for the next parameter. For instance: The project name can be used as python package name, by removing all whitespaces and make it lowercase. Example: "project_package": {{ cookiecutter.project_name.lower().replace(' ', '_') }}".

Using Jinja

Jinja statements can be used to manipulate/modify inputs or make decisions out of them. For instance: Based on the chosen license, the content of a file called *LICENSE* could be changed by:

```
{% if cookiecutter.license == MIT %}
Using MTI license

{% else if cookiecutter.license == BSD %}
Using BSD license

{% else %}
Using a private license

{% endif %}
```

3.14 Packaging and Installation

The distribution of *plugins*, *patterns* and even *applications* can easily be done by using solutions from Python's packaging ecosystem. For packaging the library setuptools is recommend. For package installation, the wide known tool pip should be used.

3.14.1 Create a package

You can add as many plugins, patterns and even applications to a single python package as you like.

All you need is a file called **setup.py** in the root folder of your package. Example:

```
from setuptools import setup, find_packages

setup(
    name='my_package',
    version="0.1,
    url='http://my_package.readthedocs.org',
    license='MIT',
    author='me',
    author_email='me@me.com',
    description="My awesome package for doing hot stuff",
    long_description="A longer description of my awesome package",
    packages=find_packages(exclude=['ez_setup', 'examples', 'tests']),
    include_package_data=True,
    platforms='any',
    install_requires=["groundwork", "Another_Package"],
    entry_points={
        'console_scripts': ["my_app my_package.my_app:start_app"],
        'groundwork.plugin': [
            'my_plugin = my_package.my_plugin:MyPlugin',
            'my_plugin_2" = my_package.my_plugin_2:MyPlugin2']
    }
)
```

For more details, take a look into the Developer's Guide of setuptools or read one of the many tutorials about setup.py on the internet.

entry_points

Entry points are used to "advertise" Python objects for use by other distributions.

groundwork uses them to find plugins in installed packages, without the need to use some hard coded imports like `from another_package import PluginX`.

Therefore groundwork knows all packaged plugins, which are available in system path of the currently used python interpreter. These plugins can be used by activation, without any need to register or import them:

```
from groundwork import App

my_app = App()

my_app.activate("GwPluginsInfo")
# GwPluginsInfo is provided by an entry_point inside the groundwork package
```

Note: During activation, the plugins are identified by their names. So the plugin name must be known, which is not necessarily the plugin class name.

The entry_point of a plugin must provide a class, which inherits directly or indirectly from *GwBasePattern*:

```
# setup.py from groundwork package
```

```
from setuptools import setup, find_packages

setup(
    name='groundwork',
    ...
    entry_points={
       'groundwork.plugin': [
            'gw_plugins_info = groundwork.plugins.gw_plugins_info:GwPluginsInfo',
        ]
    }
)
```

Package structure

The following structure is recommended for packaging multiple plugins, patters and applications:

```
my_package
|
|-- setup.py
|
|-- my_package
|   |
|   |-- applications
|   |   |-- my_app
|   |       |-- my_app.py
|   |
|   |-- patterns
|   |   |-- my_pattern
|   |       |-- my_pattern.py
|   |
|   |-- plugins
|       |-- my_plugin
|       |   |-- my_plugin.py
|       |
|       |-- my_plugin_2
|           |-- my_plugin_2.py
|
|-- docs
|   |-- index.rst
|   |-- my_app.rst
|   |-- my_pattern.rst
|   |-- my_plugin.rst
|   |-- my_plugin_2.rst
|
|-- tests
    |-- test_my_app.py
    |-- test_my_pattern.py
    |-- test_my_plugin.py
    |-- test_my_plugin_2.py
```

3.14.2 Install a package

Local packages

If you store your package locally and do not use PyPI for distribution, you need to use your **setup.py** file for all installation scenarios.

During development

During development it is recommend to install a package in development mode on your current virtual environment:

```
python setup.py develop
```

This lets you make changes on your code without the need to reinstall your package after each code change. This must be done only, if you make some changes to the **setup.py** file.

Final installation

To finally install your package inside the current used python environment, use **install**:

```
python setup.py install
```

This will copy all files to your python environment and new changes on your plugin do not have any impact on the installed package.

PyPi

PyPI can be used to share your package globally and allows users to use pip for installation:

```
pip install my_package
```

The usage of PyPi is already explained in some great tutorials. A short selection:

- Python Packaging: Publishing on PyPi
- Peter Downs: How to submit a package to PyPI

3.15 Additional Packages for groundwork

3.15.1 groundwork-database

groundwork-database provides plugins and patterns to store and manage database, database tables and the stored data.

It is based on SQLAlchemy and therefore supports a bunch of common sql based databases.

Visit https://groundwork-database.readthedocs.io for more information.

3.15.2 groundwork-web

groundwork-web supports the creation and management of web applications. It manages servers, routes, contexts and more.

Technically it is based on Flask and all known flask extensions can be used to extend its functionality.

It also has some patterns to easily create admin or REST interfaces based on given database tables.

Visit https://groundwork-web.readthedocs.io for more information.

3.16 Contribute

3.16.1 Running tests

Before tests can be executed, you have to install the test dependencies of groundwork:

```
pip install -r test-requirements.txt
```

Then to run groundwork's own tests, open a command line interface, change to groundwork/tests and run:

```
py.test --flake8
```

pytest and Flake8

groundwork is using pytest for its tests.

For these tests, the following plugins are recommended:

- pytest-flake8
- pytest-sugar

pytest and exception chains

pytest seems to show the traceback of last raised exception only. In some cases this is not really helpful, as the location of last raised exception may not be the place, where you need to fix something.

E.g. if a plugin raises an exception during plugin activation, the pluginmanager will catch this and raises it's own exception. pytest will only guide you to the pluginmanager, but not to the plugin activation routine itself.

groundwork raises exceptions always with the "from e" statement (e.g. raise Exception("Ohh no") from e). A normal python traceback would show this exception chain. pytest unluckily does not, if it is not configured to do so.

To activate the default python traceback, start pytest with the following parameter:

```
py.test --tb=native
```

Deviations from common standards

Maximum line length

The code of groundwork is written with a maximum line length of 120 characters per line. This value is also used for flake8 configuration in the file *setup.cfg*.

3.16.2 Documentation

groundwork is using sphinx for documentation building.

To build the documentation you need to have all documentation requirements installed:

```
pip install -r doc-requirements.txt
```

Then just run the following inside groundwork/docs to get a html documentation:

```
make html
```

groundwork sphinx theme

groundwork has its own theme for sphinx html documentations. It's free and was created to give groundwork related packages a common look.

Code and some instructions can be found inside the github project of gw-sphinx-themes.

CHAPTER 4

API Reference

4.1 API

4.1.1 Application Object

class groundwork.**App**(*config_files=None*, *plugins=None*, *strict=False*)

Application object for a groundwork app. Loads configurations, configures logs, initialises and activates plugins and provides managers.

Performed steps during start up:

1. load configuration
2. configure logs
3. get valid groundwork plugins
4. activate configured plugins

Parameters

- **config_files** (*list of str*) – List of config files, which shall be loaded
- **plugins** (Plugin-Classes, based on *GwBasePattern*) – List of plugins, which shall be registered
- **strict** – If true, Exceptions are thrown, if a plugin can not be initialised or activated.

_configure_logging(*logger_dict=None*)

Configures the logging module with a given dictionary, which in most cases was loaded from a configuration file.

If no dictionary is provided, it falls back to a default configuration.

See Python docs for more information.

Parameters **logger_dict** – dictionary for logger.

config = None
Instance of *ConfigManager*. Used to load different configuration files and create a common configuration object.

log = None
logging object for sending log messages. Example:

```
from groundwork import App
my_app = App()
my_app.log.debug("Send debug message")
my_app.log.error("Send error....")
```

name = None
Name of the application. Is configurable by parameter "APP_NAME" of a configuration file.

path = None
Absolute application path. Is configurable by parameter "APP_PATH" of a configuration file. If not given, the current working directory is taken. The path is used to calculate absolute paths for tests, documentation and much more.

plugins = None
Instance of *PluginManager*- Provides functions to load, activate and

signals = None
Instance of *SignalsApplication*. Provides functions to register and fire

4.1.2 SignalsApplication

class groundwork.signals.**SignalsApplication**(*app*)
Signal and Receiver management class on application level. This class is initialised once per groundwork application object.

Provides functions to register and send signals. And to connect receivers to signals.

Parameters **app** (*GwApp*) – The groundwork application object

connect(*receiver*, *signal*, *function*, *plugin*, *description=''*, *sender=None*)
Connect a receiver to a signal

Parameters

- **receiver** (*str*) – Name of the receiver
- **signal** (*str*) – Name of the signal. Must already be registered!
- **function** – Callable functions, which shall be executed, of signal is send.
- **plugin** – The plugin objects, which connects one of its functions to a signal.
- **description** – Description of the reason or use case, why this connection is needed. Used for documentation.
- **sender** – If set, only signals from this sender will be send to ths receiver.

disconnect(*receiver*)
Disconnect a receiver from a signal. Signal and receiver must exist, otherwise an exception is thrown.

Parameters **receiver** – Name of the receiver

get(*signal=None*, *plugin=None*)
Get one or more signals.

Parameters

- **signal** (*str*) – Name of the signal
- **plugin** (GwBasePattern) – Plugin object, under which the signals where registered

get_receiver(*receiver=None*, *plugin=None*)
Get one or more receivers.

Parameters

- **receiver** (*str*) – Name of the signal
- **plugin** (GwBasePattern) – Plugin object, under which the signals where registered

receivers = None
Dictionary of registered receivers. Dictionary key is the registered receiver name. Value is an instance of *Receiver*.

register(*signal*, *plugin*, *description=''*)
Registers a new signal.

Parameters

- **signal** – Unique name of the signal
- **plugin** – Plugin, which registers the new signal
- **description** – Description of the reason or use case, why this signal is needed. Used for documentation.

send(*signal*, *plugin*, ***kwargs*)
Sends a signal for the given plugin.

Parameters

- **signal** (*str*) – Name of the signal
- **plugin** (GwBasePattern) – Plugin object, under which the signals where registered

signals = None
Dictionary of registered signals. Dictionary key is the registered signal name. Value is an instance of *Signal*.

unregister(*signal*)
Unregisters an existing signal

Parameters **signal** – Name of the signal

class groundwork.signals.**Signal**(*name*, *plugin*, *namespace*, *description=''*)
Groundwork signal class. Used to store name, description and plugin.

This information is mostly used to generated overviews about registered signals and their send history.

Parameters

- **name** (*str*) – Name of the signal
- **namespace** – Namespace of the signal. There is one per groundwork app.
- **description** (*str*) – Additional description for the signal
- **plugin** (GwBasePattern) – The plugin, which registered this signal

class groundwork.signals.**Receiver**(*name*, *signal*, *function*, *plugin*, *namespace*, *description=''*, *sender=None*)
Subscriber class, which stores information for documentation purposes.

Parameters

- **name** (*str*) – Name of the Subscriber
- **signal** (*str*) – Signal name(s)
- **namespace** – Namespace of the signal. There is one per groundwork app.
- **function** – Callable function, which gets executed, if signal is sent.
- **plugin** (GwBasePattern) – Plugin object, which registered the subscriber
- **description** (*str*) – Additional description about the subscriber.

4.1.3 Configuration

ConfigManager

class groundwork.configuration.configmanager.**ConfigManager**(*config_files=[]*)

Loads different configuration files and sets their attributes as attributes of its own instance.

A configuration file must be an importable python file.

Only uppercase attributes are loaded. Everything else is ignored. Example:

```
import os
APP_NAME = "My APP"                     # Is used
APP_PATH = os.path.abspath(".")         # Is used
app_test = "test"                       # Is NOT used
MY_OWN_VAR = "nice"                     # Is used
```

config = None

An instance of *Config* for storing all configuration parameters.

load(*config_files*)

Creates a configuration instance from class *Config* from all files in self.files and set the dictionary items as attributes of of this instance.

Returns Instance of *Config*

Config

class groundwork.configuration.configmanager.**Config**

Stores all configuration parameters and handles access to it.

Example:: import groundwork my_app = groundwork.App(config_files=[“my_config.py”]) param = my_app.config.get(“MY_PARAM”, default=”Not set”)

my_app.config.set(“MY_PARAM_2, value=12345) param_2 = my_app.config.get(“MY_PARAM_2”)

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

Returns an existing configuration parameter. If not available, the default value is used.

Parameters

- **name** – Name of the configuration parameter
- **default** – Default value, if parameter is not set

set(*name*, *value*, *overwrite=False*)

Sets a new value for a given configuration parameter.

If it already exists, an Exception is thrown. To overwrite an existing value, set overwrite to True.

Parameters

- **name** – Unique name of the parameter
- **value** – Value of the configuration parameter
- **overwrite** (*boolean*) – If true, an existing parameter of *name* gets overwritten without warning or exception.

4.1.4 PluginManagers

The pluginmanager module cares about the management of plugin status and their changes between status.

There are two manager classes for managing plugin related objects.

- PluginManager: Cares about initialised Plugins, which can be activated and deactivated.
- PluginClassManager: Cares about plugin classes, which are used to create plugins.

A plugin class can be reused for several plugins. The only thing to care about is the naming of a plugin. This plugin name must be unique inside a groundwork app and can be set during plugin initialisation/activation.

PluginManager

class groundwork.pluginmanager.**PluginManager**(*app*)

PluginManager for searching, initialising, activating and deactivating groundwork plugins.

_register_initialisation(*plugin_instance*)

Internal functions to perform registration actions after plugin load was successful.

activate(*plugins=[]*)

Activates given plugins.

This calls mainly plugin.activate() and plugins register needed resources like commands, signals or documents.

If given plugins have not been initialised, this is also done via `_load()`.

Parameters **plugins** (*list of strings*) – List of plugin names

classes = None

Instance of *PluginClassManager*. Handles the registration of plugin classes, which can be used to create new plugins during runtime.

deactivate(*plugins=[]*)

Deactivates given plugins.

A given plugin must be activated, otherwise it is ignored and no action takes place (no signals are fired, no deactivate functions are called.)

A deactivated plugin is still loaded and initialised and can be reactivated by calling *activate()* again. It is also still registered in the *PluginManager* and can be requested via *get()*.

Parameters **plugins** (*list of strings*) – List of plugin names

exist(*name*)

Returns True if plugin exists. :param name: plugin name :return: boolean

get(*name=None*)

Returns the plugin object with the given name. Or if a name is not given, the complete plugin dictionary is returned.

Parameters **name** – Name of a plugin

Returns None, single plugin or dictionary of plugins

initialise(*clazz*, *name=None*)

initialise_by_names(*plugins=None*)
Initialises given plugins, but does not activate them.

This is needed to import and configure libraries, which are imported by used patterns, like GwFlask.

After this action, all needed python modules are imported and configured. Also the groundwork application object is ready and contains functions and objects, which were added by patterns, like app.commands from GwCommandsPattern.

The class of a given plugin must already be registered in the *PluginClassManager*.

Parameters **plugins** (*list of strings*) – List of plugin names

is_active(*name*)
Returns True if plugin exists and is active. If plugin does not exist, it returns None

Parameters **name** – plugin name

Returns boolean or None

PluginClassManager

class groundwork.pluginmanager.**PluginClassManager**(*app*)
Manages the plugin classes, which can be used to initialise and activate new plugins.

Loads all plugin classes from entry_point "groundwork.plugin" automatically during own initialisation. Provides functions to register new plugin classes during runtime.

_get_plugins_by_entry_points()
Registers plugin classes, which are in sys.path and have an entry_point called 'groundwork.plugin'. :return: dict of plugin classes

exist(*name*)
Returns True if plugin class exists. :param name: plugin name :return: boolean

get(*name=None*)
Returns the plugin class object with the given name. Or if a name is not given, the complete plugin dictionary is returned.

Parameters **name** – Name of a plugin

Returns None, single plugin or dictionary of plugins

register(*classes=[]*)
Registers new plugins.

The registration only creates a new entry for a plugin inside the _classes dictionary. It does not activate or even initialise the plugin.

A plugin must be a class, which inherits directly or indirectly from GwBasePattern.

Parameters **classes** (*list*) – List of plugin classes

register_class(*clazz*, *name=None*, *entrypoint_name=None*, *distribution_path=None*, *distribution_key=None*, *distribution_version=None*)

4.1.5 Plugin Patterns

GwBasePattern

gw_base_pattern provides all basic classes and functions, which are needed by any kind of groundwork plugin or pattern.

It mostly cares about the correct activation and deactivation. Including sending signals to inform other patterns or plugins about status changes of a plugin.

class groundwork.patterns.gw_base_pattern.**GwBasePattern**(*app*, *name=None*, **args*, ***kwargs*)

Base pattern class for all plugins and patterns.

Usage:

```
from groundwork.patterns import GwBasePattern

class MyPlugin(GwBasePattern):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def activate():
        self.signals.register("MySignal", "My description about signal)

    def deactivate():
        self.signals.unregister("MySignal")
```

Parameters

- **app** (*groundwork.App.*) – groundwork application object, for which the plugin shall be initialised.
- **name** – Unique name. Normally set by plugin.

activate()
Must be overwritten by the plugin class itself.

app = None
groundwork application instance. Access it inside a plugin via `self.app`.

deactivate()
Must be overwritten by the plugin class itself.

log = None
A logger, especially created for this plugin. Usage inside a plugin: `self.log.warn("WARNING!!")`.

The logger name is the same as the plugin name. Therefor it is possible to configure the application logging to show log messages of a specif plugin only. See *Logging*

needed_plugins = None
Variable for storing dependencies to other plugins. Tuple must contains needed plugin names. needed_plugins = (“MyPlugin”, “MyPlugin2)

signals = None
Instance of *SignalsPlugin*. Provides functions to register and manage signals and retrievers.

All action takes place in the context of this plugin. For instance a `self.signals.get()` will return signals of this plugin only. To get all signals of an application, please use `self.app.signals.get()`.

class groundwork.patterns.gw_base_pattern.**SignalsPlugin**(*plugin*)
Signal and Receiver management class on plugin level. This class gets initiated once per plugin.

Mostly delegates function calls to the `groundwork.signals.SignalListApplication` instance on application level.

Parameters **plugin** (GwBasePattern) – The plugin, which wants to use signals

connect(*receiver*, *signal*, *function*, *description*, *sender=None*)
Connect a receiver to a signal

Parameters

- **receiver** (*str*) – Name of the receiver
- **signal** (*str*) – Name of the signal. Must already be registered!
- **function** – Callable functions, which shall be executed, of signal is send.
- **description** – Description of the reason or use case, why this connection is needed. Used for documentation.

disconnect(*receiver*)
Disconnect a receiver from a signal. Receiver must exist, otherwise an exception is thrown.

Parameters **receiver** – Name of the receiver

get(*signal=None*)
Returns a single signal or a dictionary of signals for this plugin.

get_receiver(*receiver=None*)
Returns a single receiver or a dictionary of receivers for this plugin.

register(*signal*, *description*)
Registers a new signal. Only registered signals are allowed to be send.

Parameters

- **signal** – Unique name of the signal
- **description** – Description of the reason or use case, why this signal is needed. Used for documentation.

send(*signal*, ***kwargs*)
Sends a signal for the given plugin.

Parameters **signal** (*str*) – Name of the signal

GwCommandPattern

class groundwork.patterns.gw_commands_pattern.**GwCommandsPattern**(**args*, ***kwargs*)
Bases: *groundwork.patterns.gw_base_pattern.GwBasePattern*

Adds a commandline interface to a groundwork app and allows plugins to register own commands.

The functionality is based on click: http://click.pocoo.org/5/

To register command parameters, you have to create instances of click.Option or click.Argument manually and add them to the register-parameter "params"

Example

```
from groundwork import GwCommandsPattern
from click import Option

class MyPlugin(GwCommandsPattern)

    def activate(self):
        self.commands.register(command="my_command",
                               description="Help for my command",
                               params=[Option(("--test", "-t"), help="Some
                               dummy text")])

    def my_command(self, my_test):
            print("Command executed! my_test=%s" % my_test)
```

For a complete list of configurable options, please take a look into the related click documentation of Option and Argument

Starting the command line interface

Groundwork does not start automatically the command line interface. This step must be done by the application developer. Example

```
from groundwork import GwApp

gw_app = GwApp(plugins=["MyCommandPlugin"])
gw_app.activate(plugins=["MyCommandPlugin"])
gw_app.commands.start_cli()
```

activate()
Must be overwritten by the plugin class itself.

commands = None
Instance of *CommandsListPlugin*. Provides functions to register and manage commands for a command line interface.

deactivate()
Must be overwritten by the plugin class itself.

GwCommandsPattern.**commands**

class groundwork.patterns.gw_commands_pattern.**CommandsListPlugin**(*plugin*)

get(*name=None*)
Returns commands, which can be filtered by name.

Parameters **name** (*str*) – name of the command

Returns None, single command or dict of commands

register(*command*, *description*, *function*, *params=[]*)
Registers a new command for a plugin.

Parameters

- **command** – Name of the command
- **description** – Description of the command. Is used as help message on cli
- **function** – function reference, which gets invoked if command gets called.
- **params** – list of click options and arguments

Returns command object

unregister(*command*)
Unregisters an existing command, so that this command is no longer available on the command line interface. This function is mainly used during plugin deactivation.

Parameters **command** – Name of the command

class groundwork.patterns.gw_commands_pattern.**CommandsListApplication**(*app*)

get(*name=None*, *plugin=None*)
Returns commands, which can be filtered by name or plugin.

Parameters

- **name** (*str*) – name of the command
- **plugin** (*instance of GwBasePattern*) – plugin object, which registers the commands

Returns None, single command or dict of commands

register(*command*, *description*, *function*, *params=[]*, *plugin=None*)
Registers a new command, which can be used on a command line interface (cli).

Parameters

- **command** – Name of the command
- **description** – Description of the command. Is used as help message on cli
- **function** – function reference, which gets invoked if command gets called.
- **params** – list of click options and arguments
- **plugin** – the plugin, which registered this command

Returns command object

start_cli(**args*, ***kwargs*)
Start the command line interface for the application.

Parameters

- **args** – arguments
- **kwargs** – keyword arguments

Returns none

unregister(*command*)
Unregisters an existing command, so that this command is no longer available on the command line interface.

This function is mainly used during plugin deactivation.

Parameters **command** – Name of the command

GwSharedObjectsPattern

class groundwork.patterns.gw_shared_objects_pattern.**GwSharedObjectsPattern**(**args*, ***kwargs*)

Bases: *groundwork.patterns.gw_base_pattern.GwBasePattern*

Pattern, which provides access to shared object functionality.

Use shared objects to provide access for other plugins to objects, which are created by this plugin. Or use it to get access to objects provided by other plugins.

Common use cases are stores, which handle business logic and database abstraction. E.g. a user store.

Provided function:

- self.shared_objects.register()
- self.shared_objects.unregister()
- self.shared_objects.get()

activate()

deactivate()

GwSharedObjectsPattern.**shared_objects**

class groundwork.patterns.gw_shared_objects_pattern.**SharedObjectsListPlugin**(*plugin*)

Cares about plugin function for shared objects on plugin level.

The class mainly directs most function calls to the ShareObjectApplication class, which is initiated on application level.

access(*name*)

Returns the object of the shared_object, if the given name has been registered. The search is done on application level, so registered shared objects form other plugins can be access.

Parameters **name** – Name of the shared object

Returns object, whatever it may be...

get(*name=None*)

Returns requested shared objects, which were registered by the current plugin.

If access to objects of other plugins are needed, use *access()* or perform get on application level:

```
my_app.shared_objects.get(name="...")
```

Parameters **name** (*str or None*) – Name of a request shared object

register(*name*, *description*, *obj*)

Registers a new shared object.

Parameters

- **name** (*str*) – Unique name for shared object
- **description** (*str*) – Description of shared object
- **obj** (*any type*) – The object, which shall be shared

unregister(*shared_object*)

Unregisters an already registered shared object.

Parameters **shared_object** (*str*) – name of the shared object

class groundwork.patterns.gw_shared_objects_pattern.**SharedObjectsListApplication**(*app*)

Cares about shared objects on application level.

access(*name*)

Returns the object of the shared_object, if the given name has been registered.

Unlike *get()*, which returns the complete instance of a shared object, including name, description, plugin, access() returns the object only (without any meta data).

Parameters **name** – Name of the shared object

Returns object, whatever it may be...

get(*name=None*, *plugin=None*)
Returns requested shared objects.

Parameters

- **name** (*str or None*) – Name of a request shared object
- **plugin** (*GwBasePattern instance or None*) – Plugin, which has registered the requested shared object

register(*name*, *description*, *obj*, *plugin*)
Registers a new shared object.

Parameters

- **name** (*str*) – Unique name for shared object
- **description** (*str*) – Description of shared object
- **obj** (*any type*) – The object, which shall be shared
- **plugin** – Plugin, which registers the new shared object

unregister(*shared_object*)
Unregisters an existing shared object, so that this shared object is no longer available.

This function is mainly used during plugin deactivation.

Parameters **shared_object** – Name of the shared_object

GwDocumentsPattern

class groundwork.patterns.gw_documents_pattern.**GwDocumentsPattern**(**args*, ***kwargs*)

Bases: *groundwork.patterns.gw_base_pattern.GwBasePattern*

Documents can be collected by other Plugins to present their content inside user documentation, online help, console output or whatever.

Please see *Documents* for more details.

activate()
Must be overwritten by the plugin class itself.

deactivate()
Must be overwritten by the plugin class itself.

documents = None
Stores an instance of *DocumentsListPlugin*

class groundwork.patterns.gw_documents_pattern.**DocumentsListPlugin**(*plugin*)
Stores and handles documents.

These documents are used for real-time and offline documentation of a groundwork application.

The content of a document must be string, which is can contain jinja and rst syntax.

Plugins, which want to generate a documentation out of all documents, must render this content (jinja render_template) and transform the rst by the own (e.g. by using rst2html).

Please see *Documents* for more details.

get(*name=None*)

register(*name*, *content*, *description=None*)
Register a new document.

Parameters

- **content** (*str*) – Content of this document. Jinja and rst are supported.
- **name** – Unique name of the document for documentation purposes.
- **description** – Short description of this document

unregister(*document*)

class groundwork.patterns.gw_documents_pattern.**DocumentsListApplication**(*app*)

get(*document=None*, *plugin=None*)
Get one or more documents.

Parameters

- **document** (*str*) – Name of the document
- **plugin** (GwBasePattern) – Plugin object, under which the document was registered

register(*name*, *content*, *plugin*, *description=None*)
Registers a new document.

Parameters

- **content** (*str*) – Content of the document
- **name** – Unique name of the document for documentation purposes.
- **plugin** (GwBasePattern) – Plugin object, under which the documents where registered

unregister(*document*)
Unregisters an existing document, so that this document is no longer available.

This function is mainly used during plugin deactivation.

Parameters **document** – Name of the document

GwThreadsPattern

class groundwork.patterns.gw_threads_pattern.**GwThreadsPattern**(**args*, ***kwargs*)
Bases: *groundwork.patterns.gw_base_pattern.GwBasePattern*

Threads can be created and started to perform tasks in the background and in parallel to the main application.

Please see *Threads* for more details.

activate()
Must be overwritten by the plugin class itself.

deactivate()
Must be overwritten by the plugin class itself.

threads = None
Stores an instance of *ThreadsListPlugin*

class groundwork.patterns.gw_threads_pattern.**ThreadsListPlugin**(*plugin*)
Stores and handles threads.

Please see *Threads* for more details.

get(*name=None*)

register(*name*, *function*, *description=None*)
Register a new thread.

Parameters

- **function** (*function*) – Function, which gets called for the new thread
- **name** – Unique name of the thread for documentation purposes.
- **description** – Short description of the thread

unregister(*thread*)

class groundwork.patterns.gw_threads_pattern.**ThreadsListApplication**(*app*)

get(*thread=None*, *plugin=None*)
Get one or more threads.

Parameters

- **thread** (*str*) – Name of the thread
- **plugin** (GwBasePattern) – Plugin object, under which the thread was registered

register(*name*, *function*, *plugin*, *description=None*)
Registers a new document.

Parameters

- **function** (*function*) – Function, which gets called for the new thread
- **name** – Unique name of the thread for documentation purposes.
- **plugin** (GwBasePattern) – Plugin object, under which the threads where registered
- **description** – Short description of the thread

unregister(*thread*)
Unregisters an existing thread, so that this thread is no longer available.

This function is mainly used during plugin deactivation.

Parameters **thread** – Name of the thread

class groundwork.patterns.gw_threads_pattern.**Thread**(*name*, *function*, *plugin*, *description=None*)
Groundwork thread class. Used to store name, function and plugin.

This information is mostly used to generated overviews about registered threads.

Parameters

- **name** (*str*) – Name of the thread
- **function** (*function*) – Function, which gets called inside the thread
- **plugin** (GwBasePattern) – The plugin, which registered this thread
- **description** – short description of this thread

response = None
Stores the function return value, if thread has finished

run(***kwargs*)
Runs the thread

Parameters **kwargs** – dictionary of keyword arguments

Returns

running = None
True, if thread is running. Otherwise its False.

thread = None
Thread base class. Type is threading.Thread

time_end = None
datetime object of the ending moment

time_start = None
datetime object of the starting moment

class groundwork.patterns.gw_threads_pattern.**ThreadWrapper**(*thread*)
Wrapper class, which inherits from threading.Thread and performs some useful tasks before and after the provided functions gets executed.

run(***kwargs*)

GwRecipePattern

Groundwork recipe pattern.

Provides function to register, get and build recipes.

Recipes are used create directories and files based on a given template and some user input. It is mostly used to speed up the set up of new python packages, groundwork applications or projects.

Based on cookiecutter: https://github.com/audreyr/cookiecutter/

class groundwork.patterns.gw_recipes_pattern.**GwRecipesPattern**(**args*, ***kwargs*)

activate()
Must be overwritten by the plugin class itself.

deactivate()
Must be overwritten by the plugin class itself.

recipes = None
Stores an instance of *RecipesListPlugin*

class groundwork.patterns.gw_recipes_pattern.**RecipesListPlugin**(*plugin*)
Cares about the recipe management on plugin level. Allows to register, get and build recipes in the context of the current plugin.

Parameters **plugin** – plugin, which shall be used as contxt.

build(*recipe*)
Builds a recipe

Parameters **recipe** – Name of the recipe to build.

get(*name=None*)
Gets a list of all recipes, which are registered by the current plugin. If a name is provided, only the requested recipe is returned or None.

Param name: Name of the recipe

register(*name*, *path*, *description*, *final_words=None*)
Registers a new recipe in the context of the current plugin.

Parameters

- **name** – Name of the recipe
- **path** – Absolute path of the recipe folder
- **description** – A meaningful description of the recipe
- **final_words** – A string, which gets printed after the recipe was build.

unregister(*recipe*)
Unregister a recipe of the current plugin.

Parameters **recipe** – Name of the recipe.

class groundwork.patterns.gw_recipes_pattern.**RecipesListApplication**(*app*)
Cares about the recipe management on application level. Allows to register, get and build recipes.

Parameters **app** – groundwork application instance

build(*recipe*, *plugin=None*)
Execute a recipe and creates new folder and files.

Parameters

- **recipe** – Name of the recipe
- **plugin** – Name of the plugin, to which the recipe must belong.

get(*recipe=None*, *plugin=None*)
Get one or more recipes.

Parameters

- **recipe** (*str*) – Name of the recipe
- **plugin** (GwBasePattern) – Plugin object, under which the recipe was registered

register(*name*, *path*, *plugin*, *description=None*, *final_words=None*)
Registers a new recipe.

unregister(*recipe*)
Unregisters an existing recipe, so that this recipe is no longer available.

This function is mainly used during plugin deactivation.

Parameters **recipe** – Name of the recipe

class groundwork.patterns.gw_recipes_pattern.**Recipe**(*name*, *path*, *plugin*, *description=''*, *final_words=''*)
A recipe is an existing folder, which will be handled by the underlying cookiecutter library as template folder.

Parameters

- **name** – Name of the recipe
- **path** – Absolute path to the recipe folder
- **plugin** – Plugin which registers the recipe

- **description** – Meaningful description of the recipe
- **final_words** – String, which gets printed after a recipe was successfully build.

build(*output_dir=None*, ***kwargs*)

Buildes the recipe and creates needed folder and files. May ask the user for some parameter inputs.

Parameters **output_dir** – Path, where the recipe shall be build. Default is the current working directory

Returns location of the installed recipe

4.1.6 Plugins

GwDocumentsInfo

class groundwork.plugins.gw_documents_info.**GwDocumentsInfo**(**args*, ***kwargs*)

Bases: *groundwork.patterns.gw_commands_pattern.GwCommandsPattern*, *groundwork.patterns.gw_documents_pattern.GwDocumentsPattern*

Provides a little documentation viewer for all registered documents. Accessible via **app doc**.

Presents also an overview about all registered documents of an application. Accessible via **app doc_list**.

GwPluginInfo

class groundwork.plugins.gw_plugins_info.**GwPluginsInfo**(**args*, ***kwargs*)

Bases: *groundwork.patterns.gw_commands_pattern.GwCommandsPattern*, *groundwork.patterns.gw_documents_pattern.GwDocumentsPattern*

Collects information about plugins, which are registered at the current application.

Collected information are accessible via command line or via a generated document during documentation generation (Additional plugin needed)

GwSignalInfo

class groundwork.plugins.gw_signals_info.**GwSignalsInfo**(**args*, ***kwargs*)

Bases: *groundwork.patterns.gw_commands_pattern.GwCommandsPattern*, *groundwork.patterns.gw_documents_pattern.GwDocumentsPattern*

GwCommandsIInfo

class groundwork.plugins.gw_commands_info.**GwCommandsInfo**(**args*, ***kwargs*)

Bases: *groundwork.patterns.gw_documents_pattern.GwDocumentsPattern*, *groundwork.patterns.gw_commands_pattern.GwCommandsPattern*

Provides documents for giving an overview about registered commands.

GwRecipesBuilder

class groundwork.plugins.gw_recipes_builder.**GwRecipesBuilder**(**args*, ***kwargs*)

Bases: *groundwork.patterns.gw_commands_pattern.GwCommandsPattern*, *groundwork.patterns.gw_recipes_pattern.GwRecipesPattern*

Provides commands for listing and building recipes via command line interface.

Provided commands:

- recipe_list
- recipe_build

Provides also the recipe **gw_package**, which can be used to setup a groundwork related python package. Content of the package:

- setup.py: Preconfigured and ready to use.
- groundwork package structure: Directories for applications, patterns, plugins and recipes.
- Simple, runnable example of a groundwork application and plugins.
- usable test, supported by py.test and tox.
- expandable documentation, supported by sphinx and the groundwork sphinx template.
- .gitignore

This code is hardly based on Cookiecutter's main.py file: https://github.com/audreyr/cookiecutter/blob/master/cookiecutter/main.py

CHAPTER 5

Changelog

5.1 Changelog

5.1.1 0.1.13

Minor update, no functional changes

- added Python 3.3 support to Tox and Travis
- added test cases for logging
- added test cases for click integration
- some fixes in the documentation

5.1.2 0.1.12

Start of the change log.

Python Module Index

g

Index

Symbols

A

B

C

D

E

G

I

L

N

P

R

S

T

U