GeekCMS Documentation

Release 0.3

Zhan Haoxun

June 24, 2016

Contents

GeekCMS is a lightweight **framework** for static site development, with properties as follow:

- Well defined protocols to facilitate the process of development and deployment.
- Plugin-base and pipe-and-filter architecture.
- Strict rules of files organization.
- Implemented in Python3.

CHAPTER 1

Execution Model

From the view of theme(in short, **theme = theme settings + plugins**) developers, GeekCMS is a framework with protocols and helpful libraries, to:

- control the behavior of each individual plugin.
- customize the calling sequence of multiple plugins.

For users, GeekCMS is a theme driven tool, means that without plugins, GeekCMS can do nothing!

There are two kinds of plugin execution procedures defined in GeekCMS:

- **Default procedure**: Procedure consists of several runtime components. A runtime component is an area to place plugins to be executed.
- **Extended procedure**: Besides, there are kinds of behaviors can not be classified into components, for instance, automatically pushing static pages to a remote git repo. Such behaviors could be implemented as independent extended procedures, and triggered by CLI command by user.

1.1 Default Procedure

Default procedure is divided into *nine* runtime components:

1. *pre_load*, *in_load*, *post_load*
2. *pre_process*, *in_process*, *post_process*
3. *pre_write*, *in_write*, *post_write*

which would be sequentially executed by GeekCMS. Each runtime component could contain zero or more plugins, details of that would be covered later.

The components can be classified into three layers, load/process/write. Layers are exectured in order, **load –> process –> write**, which is simple and intuitive. Notice that the distinction of components is vague, for instance, a plugin transforming markdown to html can be placed in *in_load*, *post_load* or even *post_write*, depending on developers understanding of components' semantics. By dividing a layer into three components, theme developer could well control the sequence of plugin execution. Plugin execution order within a layer is a little bit complicated, and would be introduce later.

1.2 Extended Procedure

GeekCMS allow developer to define extended procedure for special usage. For more information: *Extended Procedure Related*.

CHAPTER 2

Protocol Related Classes

GeekCMS defines serveral protocols for plugin registration and data operation. All following classes are defined in *geekcms.protocol*.

2.1 Assets Related

class _BaseAsset (**args*, ***kwargs*)

_BaseAsset is the base class of *BaseResource*, *BaseProduct* and *BaseMessage*. Initialize *_BaseAsset* would Always raise an Exception, since _BaseAsset should not be initialized.

set_owner (*self*, *owenr*)

Set the owner of asset instance. Instance of derived classes of *_BaseAsset* must defined an owner, which should be the name of theme.

classmethod get_manager_with_fixed_owner (*cls*, *owner*)

Return an instance of *ManagerProxyWithOwner*, bound with *cls* and *owner*.

objects

Retrive an instance of *Manager*, which bound with *BaseResource*, *BaseProduct* and *BaseMessage* and its derived classes.

class BaseResource (**args*, ***kwargs*)

class derived from *_BaseAsset*.

class BaseProduct (**args*, ***kwargs*)

class derived from *_BaseAsset*.

class BaseMessage (**args*, ***kwargs*)

class derived from *_BaseAsset*.

BaseResource, *BaseProduct* and *BaseMessage* are base classes for data operations. Developer should derived one of these classes for specific usage. Derived classes should overwrite *__init__* method, since *_BaseAsset* defines a *__init__* method which always raise an Exception.

class Manager (*target_cls*, *data=None*)

The class derives from `UserDict`. *target_cls* is a class derives from *BaseResource*, *BaseProduct* or *BaseMessage*. *data* is a dictionary that used to store instances created by manager. If *data* is `None`, a default dictionary would be initialized.

add (*self*, *item*)

Add instance.

remove (*self*, *item*)

Remove instance.

keys (*self*)
Returns a list contanins all the owner of stored items.

create (*self*, **args*, *owner*, ***kwargs*)
Create and store an instance of `self.target_cls`, with arguments `(*args, **kwargs)`. *owner* is a keyword-only parameter.

filter(self, owner):
Return a list of instances that:

- *owner* is the owner of instance.
- `isinstance(instance, self.target_cls)` is True.

values (*self*)
Return a list of instances that `isinstance(instance, self.target_cls)` is True.

clear (*self*)
Clean up all stored items.

class ManagerProxyWithOwner (*self*, *owner*, *manager*)
Instance of this class would be a proxy of *Manager* with fixed owner. All functions defined in *Manager* are avaliable, except that the parameter *owner* is excluded from functions' parameter list. *manager* is an instance of *Manager*, and *owner* is the owner to be fixed.

2.2 Default Procedure Related

class BasePlugin

classmethod get_manager_bind_with_plugin (*cls*, *other_cls*)
Return an instance of *ManagerProxyWithOwner* bound to class of asset. *other_cls* should be one of *BaseResource*, *BaseProduct* and *BaseMessage* and its derived classes.

run (*self*, *resources=None*, *products=None*, *messages=None*)
The interface that a derived plugin class must overwrite. *run* should be a function that implements plugin's bussiness. The return value of *run* function would be discarded.

resources would be a list contains instances of *BaseResource* or its derived class. *products* would be a list contains instances of *BaseProduct* or its derived class. *messages* would be a list contains instances of *BaseMessage* or its derived class.

All user defined plugin classes should derive from *BasePlugin* and overwrite the *run* function.

Besides, class-level attributes *theme* and *plugin* can be defined for further customization. For exmaple:

```
from geekcms import protocol

class TestPlugin(protocol.BasePlugin):

    theme = 'test_theme'
    plugin = 'test_plugin'

    def run(self):
        pass
```

Explanation of class-level attributes are as follow:

theme Defines the theme that a plugin belongs to. Value of theme that would be used to filter resources, products or messages passed to run method. For instances, suppose class *A*, *B* both definded `theme`

= 'AB', and there is another class C definded `theme = 'C'`. If *A*'s *run* method created some instance of resources owned by 'AB', and *B*, *C* were executed after *A*, then *B*'s *run* function might receive instances of resource created by *A*(or might not, due to the parameter controller) while *C*'s run function would not receive instances created by *A*. This attribute could be omitted, in such case the name of theme's top-level directory would be adapt.

plugin Defines the name of plugin which related to the plugin names of theme settings. This attribute could be omitted, in that case, the class name would be used as the plugin name.

The parameter list of overwrited *run* function is a bit more complicated. Since the bussiness of plugins various a lot, developers might defind *run* function with different parameters, such as:

```
from geekcms import protocol

class TestPlugin(protocol.BasePlugin):

    theme = 'test_theme'

    # accept all assets.
    def run(self, resources, products, messages):
        pass

    # only accept resources.
    def run(self, resources):
        pass

    # accept nothing.
    def run(self):
        pass
```

GeekCMS would detect the signature of run function, with the default order *[resources, products, messages]*. For example, if developer defined a run function with two positional parameters, then GeekCMS would pass instances of resources and products to such function.

For further control of parameter list, developers should consider using decorators defined in *PluginController*.

class PluginController

RESOURCES
String indicating *BaseResource* and its derived classes.

PRODUCTS
String indicating *BaseProduct* and its derived classes.

MESSAGES
String indicating *BaseMessage* and its derived classes.

classmethod accept_owners(*cls*, **owners*)
accept_owners is a decorator for plugin's *run* function. The owner of assets passed into *run* function would be adjusted, with respect to *owners*. *owners* should be a list of avaliable owners.

classmethod accept_parameters(*cls*, **fixed_params*, ***typed_params*)
accept_parameters is a decorator for plugin's *run* function. By decorating, the order and type of parameters would be adjusted, with respect to *fixed_params* and *typed_params*.

fixed_params should be a list of:

- string of *[RESOURCES, PRODUCTS, MESSAGES]*.
- two-element tuple(or list) in the form of *(name, accept_cls)*, in which *name* should be string of *[RESOURCES, PRODUCTS, MESSAGES]* and *accept_cls* should be class of asset.

typed_params is a dictionary with key-value pairs *(name, accept_cls)*, in which *name* should be string of *[RESOURCES, PRODUCTS, MESSAGES]* and *accept_cls* should be class of asset.

Example of *accept_parameters*:

```
from geekcms import protocol
pcl = protocol.PluginController


class DerivedMessage(protocol.BaseMessage):

    def __init__(self):
        pass


class TestPlugin(protocol.BasePlugin):
    theme = 'test_theme'

    # accept only messages of BaseMessage.
    @pcl.accept_parameters(pcl.MESSAGES)
    def run(self, messages):
        pass

    # accept only messages of DerivedMessage.
    @pcl.accept_parameters(
        (pcl.MESSAGES, DerivedMessage),
    )
    def run(self, messages):
        pass

    # accept only messages of DerivedMessage.
    @pcl.accept_parameters(
        **{pcl.MESSAGES: DerivedMessage}
    )
    def run(self, messages):
        pass

    # accept only messages of DerivedMessage.
    @pcl.accept_parameters(
        messages=DerivedMessage,
    )
    def run(self, messages):
        pass
```

Example of *accept_owners*:

```
from geekcms import protocol

pcl = protocol.PluginController

class TestPlugin(protocol.BasePlugin):

    # accept only messages owned by test_theme and another_theme.
    @pcl.accept_parameters(pcl.MESSAGES)
    @pcl.accept_owners('test_theme', 'another_theme')
    def run(self, messages):
        pass
```

2.3 Extended Procedure Related

class `BaseExtendedProcedure`

`get_command_and_explanation` (*self*)
Should be a function returns *(command, explanation)* tuple, with *command* as string to trigger the extended procedure and *explanation* as a brief explanation of the extended procedure. Derived class should overwrite this function.

`get_doc` (*self*)
Should return string that can be parsed by docopt [1]. Derived class should overwrite this function.

`run` (*self*, *args*)
run should be a function that implements plugin's bussiness. *args* is the processed arguments return by docopt. The return value of *run* function would be discarded. Derived class should overwrite this function.

Plugins class of extended procedure should derive from `BaseExtendedProcedure`.

[1] https://github.com/docopt/docopt

CHAPTER 3

Utilities Classes

All following classes are defined in *geekcms.utils*.

3.1 Data Share

class `ShareData`

classmethod `get` (*cls*, *search_key*)
search_key is a key for search *Share* section of theme and project settings.

search_key could be in the pattern of 'theme_name.key'. In such pattern, GeekCMS would lookup the settings file of theme *theme_name* for *key*. If such key do not exist in the settings file of theme *theme_name*, `None` would be return. If found, a string bound with *key* would be return.

Besides, *search_key* could be presented in the pattern of 'key'(with no '.' in the *search_key*). In such case, GeekCMS would lookup all settings files of themes and project, the first match value would be return. If not found, `None` would be return.

Before executing default and extended procedures, GeekCMS would parse all settings file of project and registered themes, and *`ShareData`* would load up all the key-value pairs in *Share* section of the settings files. Since *`ShareData.get`* judges the pattern of *search_key* by finding a 'dot', It's not a good idea to defines a key along with a 'dot'.

3.2 Path Resolve

class `PathResolver`

`project_path`
The path of project.

classmethod `set_project_path` (*cls*, *path*)
Set the *project_path* with *path*.

classmethod `inputs` (*cls*, *, *ensure_exist=False*)
Return the path of *inputs* directory of project. If *ensure_exist* is `True` and the directory of *inputs* do not exist, then an empty directory would be created.

classmethod outputs(*cls*, *, *ensure_exist=False*)
Return the path of *outputs* directory of project. If *ensure_exist* is `True` and the directory of *outputs* do not exist, then an empty directory would be created.

classmethod themes(*cls*, *, *ensure_exist=False*)
Return the path of *themes* directory of project. If *ensure_exist* is `True` and the directory of *themes* do not exist, then an empty directory would be created.

classmethod states(*cls*, *, *ensure_exist=False*)
Return the path of *states* directory of project. If *ensure_exist* is `True` and the directory of *states* do not exist, then an empty directory would be created.

classmethod theme_state(*cls*, *theme_name*, *, *ensure_exist=False*)
Return the path of directory contains state of theme. Such path is generated by joining `cls.states()` and *theme_name*. If *ensure_exist* is `True` and the directory of theme's state do not exist, then an empty directory would be created.

classmethod theme_dir(*cls*, *theme_name*, *, *ensure_exist=False*)
Return the path of directory contains code of theme. Such path is generated by joining `cls.themes()` and *theme_name*. If *ensure_exist* is `True` and the directory of theme's dir do not exist, then an empty directory would be created.

PathResolver can be helpful for development, with which developer could easily get the path of specific directory, and create specific directory if such directory does not exist.

CHAPTER 4

Project Organization

4.1 Structure Of Project

GeekCMS would maintained a directory containing all files required to generate a website, such directory is organized as a *projcet*.

Sturcture of a project is as follow:

```
example_project/
                themes/
                    ...
                states/
                    ...
                inputs/
                    ...
                outputs/
                    ...
                settings
```

Brief explanations of above file and direcroties:

themes

A directory where all the code of theme exists.

states

A directory for themes to place its intermediate data.

inputs

A direcroty contains all input files.

outputs

A directory contains all generated files.

settings

A text file named *project settings*, in which defines registered themes and global shared data.

The names of above file and direcroties is hardcoded in GeekCMS.

4.2 Structure Of Theme

A theme should be organized as a *python package*, structure is as follow:

```
example_project/
                themes/
                       theme_A/
                           __init__.py
                           settings       # theme settings
                           ...
                       theme_B/
                           __init__.py
                           settings       # theme settings
                           ...
                states/
                    ...
                inputs/
                    ...
                outputs/
                    ...
                settings                  # project settings
```

All themes should be placed in *themes* directory. As you can see, there is *settings file* exists in each theme package. Such settings file is named *theme settings*.

4.3 Settings File

GeekCMS's behavior is guided by *project settings* and *theme settings*. Format of *settings* is described in configparser [1].

projcet settings should defines a *RegisterTheme*(case-sensitive) section. Names of themes(the name of theme's directory) to be loaded by GeekCMS should be seperated by whitespaces and set as the value of *themes* key(case-insensitive). Example is as follow:

```
# project settings.
[RegisterTheme]
themes: simple git_upload
```

where directories *simple* and *git_upload* are registered.

themes settings should defines a *RegisterPlugin*(case-sensitive) section. Keys in the section should be one of *[pre_load, in_load, post_load, pre_process, in_process, post_process, pre_write, in_write, post_write]* and *[cli_extend]*. All avaliable keys except *cli_extend* is discussed in *Default Procedure*, and *cli_extend* is a key for registering *extended procedure*. An example for demonstration:

```
# settings of simple.
[RegisterPlugin]

in_load:
        load_inputs_static
        load_article
        load_about
        load_index
        load_theme_static
```

[1] http://docs.python.org/3/library/configparser.html

```
in_process:
        md_to_html << gen_article_page

post_process:
        gen_about_page
        gen_index_page
        gen_time_line_page
        gen_archive_page

pre_write:
        clean

in_write:
        write_static
        write_page

post_write:
        cname

# settings of git_upload
[RegisterPlugin]

cli_extend: GitUploader
```

Both *projcet settings* and *theme settings* can define a *Share* section. Key-value pairs defined in *Share* section can be retrived by *ShareData*. An example for demonstration:

```
# settings of simple.
[Share]
# special pages
index_page: index.html
time_line_page: speical/time_line.html
about_page: speical/about.html
archive_page: speical/archive.html
```

where the value of *index_page* can be retrived by `ShareData.get('simple.index_page')`.

4.4 Plugin Registration

Execution order of plugins within the same *runtime component* is defined by *plugin registration syntax*. The syntax is:

```
runtime_component     ::=   component_name (':'  | '=') [NEWLINE] plugin_relation*
plugin_relation       ::=   binary_relation_expr | unary_relation_expr NEWLINE
binary_relation_expr  ::=   plugin_name (left_relation | right_relation) plugin_name
unary_plugin_expr     ::=   plugin_name [left_relation]
                            | [right_relation] plugin_name
left_relation         ::=   '<<' [decimalinteger]
right_relation        ::=   [decimalinteger] '>>'
component_name        ::=   identifier
plugin_name           ::=   identifier
```

where *identifier*, *decimalinteger* and *NEWLINE* are corresponding to the definitions in Python Lexical Analysis [2].

[2] http://docs.python.org/3/reference/lexical_analysis.html

Semantics:

1. **pre_load: my_loader** register plugin *my_loader* to component *pre_load*.
2. **pre_load: my_loader << my_filter** register plugins *my_loader* and *my_filter* to component *pre_load*, with *my_loader* being executed before *my_filter*.
3. **pre_load: my_filter >> my_loader** has the same meaning as *pre_load: my_loader* << *my_filter*.
4. **pre_load: loader_a <<0 loader_b NEWLINE loader_c <<1 loader_b** the execution order would be *loader_c –> loader_a –> loader_b*. << is equivalent to *<<0*, and << *decimalinteger* is equivalent to *decimalinteger* >>.
5. **pre_load: my_loader <<** means *my_loader* would be executed before the other plugins within a component, unless another relation such as *anther_loader <<1* is established.
6. **pre_load: >> my_filter** reverse meaning of *pre_load: my_loader* <<.

Notice that the *plugin_name* should be presented in the pattern of 'theme_name.plugin_name'. 'theme_name.' can be omitted, as presented in above example, if *plugin_name* points to a plugin exists in current theme directory.

GeekCMS would automatically import the *__init__* module of registered theme packages. Besides writing a *theme settings*, developer should import the module(s) that defines plugin(s) in *__init__*. An example is given for demonstration:

```
# ../git_upload/__init__.py

# necessary!
from . import plugin


# ../git_upload/plugin.py

"""
Usage:
    geekcms gitupload

"""

from datetime import datetime
import subprocess
import os

from geekcms.protocol import BaseExtendedProcedure
from geekcms.utils import PathResolver


class CWDContextManager:

    def __enter__(self):
        os.chdir(PathResolver.outputs())

    def __exit__(self, *args, **kwargs):
        os.chdir(PathResolver.project_path)


class GitUploader(BaseExtendedProcedure):

    def get_command_and_explanation(self):
        return ('gitupload',
                'Automatically commit and push all files of outputs.')
```

```
    def get_doc(self):
        return __doc__

    def run(self, args):
        commit_text = 'GeekCMS Update, {}'.format(
            datetime.now().strftime('%c'),
        )
        commands = [
            ['git', 'add', '--all', '.'],
            ['git', 'commit', '-m', commit_text],
            ['git', 'push'],
        ]
        with CWDContextManager():
            for command in commands:
                subprocess.check_call(command)
```

GeekCMS would automatically loaded `GitUploader` in above example.

CHAPTER 5

Deploy And Download Themes

5.1 Command Line Interface

GeekCMS provides a friendly CLI interface of usage. CLI of GeekCMS is implemented by using docopt [1].

For code sharing, developer could package their codes as a template. A template is organized in as a project.

5.2 Download

If you type `geekcms` in your prompt and current working directory is not a project, then the shell would presents:

```
$ geekcms
Usage:
    geekcms startproject <template_name>
```

Entering `startproject` option with <template_name> would automatically download a directory with the name of <template_name>, which should be an empty project, from GeekCMS-Themes [2] to current working directory:

```
$ ls
$ geekcms startproject simple
A    simple/inputs
A    simple/inputs/about
A    simple/inputs/about/about.md
A    simple/inputs/article
A    simple/inputs/article/test
A    simple/inputs/article/test/codetest.md
A    simple/inputs/article/test/longlong.md
A    simple/inputs/article/test/top-level.md
A    simple/inputs/index
A    simple/inputs/index/welcome.md
A    simple/inputs/static
A    simple/inputs/static/delete it.
A    simple/settings
A    simple/themes
A    simple/themes/git_upload
A    simple/themes/git_upload/__init__.py
A    simple/themes/git_upload/plugin.py
A    simple/themes/git_upload/settings
```

[1] https://github.com/docopt/docopt
[2] https://github.com/haoxun/GeekCMS-Themes

```
A    simple/themes/simple
A    simple/themes/simple/__init__.py
A    simple/themes/simple/assets.py
A    simple/themes/simple/load.py
A    simple/themes/simple/process.py
A    simple/themes/simple/settings
A    simple/themes/simple/static
A    simple/themes/simple/static/css
A    simple/themes/simple/static/css/github.css
A    simple/themes/simple/templates
A    simple/themes/simple/templates/archive.html
A    simple/themes/simple/templates/article.html
A    simple/themes/simple/templates/base.html
A    simple/themes/simple/templates/time_line.html
A    simple/themes/simple/utils.py
A    simple/themes/simple/write.py
Checked out revision 15.
$ ls
simple
```

After downloading such directory, GeekCMS would ensure *project settings* and *themes*, *states*, *inputs*, *outputs* exists, so developer should not consider pushing an empty directory to git repo.

5.3 Deploy

If you want to share your code with the others, just push your code to GeekCMS-Themes [2].

CHAPTER 6

Example Of Development

A simple [1] template is developed for demonstraction. The template simply renders markdown files and generates a static site.

[1] https://github.com/haoxun/GeekCMS-Themes/tree/master/simple

Symbols

A

B

C

G

I

K

M

O

P

R

S

T

V