GetWebFilesInator Documentation

Release 0.0.1

ellethee <luca800@gmail.com>

Jun 09, 2017

Contents:

GetWebFilesInator (gwfi) is a simple file downloader to keep your javascript and css libraries up to date. Is not perfect but is easily configurable and for simplest tasks can be more suitable than a real dependency manager.

All you need is to know were your libraries are and a yaml o json configuration file.

CHAPTER 1

Requirements

- ArgParseInator (pip install ArgParseInator)
- PyYAML (pip install PyYAML)
- requests (pip install requests)

Configuration

The configuration file is really simple. It's divided in 6 sections.

Main configuration

Here you can define all variable or list etc you need for your custom downloaders.

Anyway there are few properties which are default or used by default downloaders.

- github_username: user name for github authentication (optional).
- github_password: password for github authentication (optional).
- keep: keeps temporary folder for debug propose.
- cache_path: cache folder for configuration else it uses .cache folder in the script path

Paths

In the paths section you can define all the destinations path for the downloaded files. The values will be expanded so you can use environment vars and references to other paths.

example:

```
paths:
  source: "~/src/django-site/"
  static: "{source}/static"
  js: "{static}/js"
  css: "{static}/css"
```

will be expanded to

```
paths:
    source: "/home/username/src/django"
    static: "/home/username/src/django/static"
    js: "/home/username/src/django/static/js"
    css: "/home/username/src/django/static/css"
```

Note:

- The **static** path must be present in the paths mapping.
- You can use the **{gwfi}** in paths to refer to the script path.

Downloaders

In this section you can define custom downloaders.

example:

```
# paths mapping
paths:
  source: "~/src/django-site/"
  static: "{source}/static"
  downloaders: "~/src/python/gwfi_downloaders"

# add a custom downloader.
downloaders:
  - [test_downloader, "{downloaders}/test_downloader.py"]
```

Note: see **'Custom Downloaders'_**

Tests

Simple tests definition, using regex, to guess the destination path when not specified.

It's a list of lists where the first element is the regex and the second is the destination path.

```
# define our tests using regex to guess destination for files.
tests:
  # if the filename has locale or locales in his path the destination is {locales}
  - ['(?i)\/locales?\/.*\.js$', "{locales}"]
  # if the filename ends with .js the destination is {js}
  - ['(?i)\.js$', "{js}"]
  # if the filename ends with .css the destination is {css}
  - ['(?i)\.css$', "{css}"]
```

```
  # if the filename ends with woff,woff2,otf,eot the destination is {fonts}
  - ['(?i)\.(woff2?|ttf|otf|eot)$', "{fonts}"]
```

Note: If no destination is specified and all the tests files **gwfi** try to get the path from extension.

For example if a I have a **testme.lib** file and I have mapped the path **lib: "{static}/js/lib"** the destination will be **{lib}**

Source Blocks

The source block are the part of html that will be inserted in the template for each file

The default source blocks are:

```
<!-- for js files -->
<script type="text/javascript" src="{}"></script>

<!-- for css files -->
<link rel="stylesheet" href="{}">
```

you can map your source block with the extension name and the block text

```
blocks:
  img: '<img src="{}" alt="void">'
```

Templates

You can define a list of templates which will be use to create html files with links to the relative static path for files.

The gwfi template block syntax is very simple

gwfi_block_{type} {path} [[.property::]{pattern}]

- type - the *Source Blocks* mapping key
- path - the path, relative to the {static} reference (see *Paths*)
- pattern - optional regex for granular filtering.

As you can see the pattern can be write in simple and complex way.

- simple: just the regex pattern it will be used to filter the **target** property (filename complete with destination path)
- complex: the property name and the regex pattern wrote in the form **.property_name::pattern**

```
1 <!-- Core Scripts - Include with every page -->
2 <!-- gwfi_block_js js jquery -->
3 <!-- gwfi_block_js js .group::django-->
4 <!-- gwfi_block_js js -->
5 <!-- gwfi_block_js js/locales -->
6 <!-- gwfi_block_css css -->
```

In the example above in **line 2** we filter all the filenames that contains `jquery` and in the **line 3** we filter all the items that have the property `group` which contains `django`.

Will generate this html file

```
<!-- Core Scripts - Include with every page -->
<!-- gwfi js jquery -->
<script type="text/javascript" src="/static/js/jquery-3.2.1.min.js"></script>
<!-- /gwfi js jquery -->
<!-- gwfi js  -->
<script type="text/javascript" src="/static/js/bootstrap.min.js"></script>
<script type="text/javascript" src="/static/js/pnotify.custom.min.js"></script>
<!-- /gwfi js  -->
<!-- gwfi js/locales  -->

<!-- /gwfi js/locales  -->
<!-- gwfi js .group::django -->
<script type="text/javascript" src="/static/js/fullcalendar.min.js"></script>
<script type="text/javascript" src="/static/js/locales/fullcalendar/it.js"></script>
<script type="text/javascript" src="/static/js/moment-with-locales.min.js"></script>
<!-- /gwfi js .group::django -->
<!-- gwfi css  -->
<link rel="stylesheet" href="/static/css/bootstrap.min.css">
<link rel="stylesheet" href="/static/css/bootstrap-theme.min.css">
<link rel="stylesheet" href="/static/css/fullcalendar.min.css">
<link rel="stylesheet" href="/static/css/pnotify.custom.min.css">
<!-- /gwfi css  -->
```

Files

Files section in where we define what to download which files will be copied and so on...

A file element can have many attribute it depends on what you want to do and on what downloader will be used.

Attributes for default downloaders

For the defaults downloaders (download, local and github related) theses are the attributes.

- get_type: which downloader need to get this file
- url: where to download the file/lib/zip the downloaders can use this to guess what kind of get_type if not specified.
- filename: the filename can be used with url to guess the get_type if not specified.
- rename: rename the file.
- name: to filter (in case of github get_type)
- filter: a regex to filter (in case of github get_type)
- dest: destination for the file or files.
- **files: list of sub files to process**
  - filename: name of the file.
  - dest: destination for the file

Examples

For theses examples i assume we have a configuration like this to avoid problems with paths to guess destination:

```
paths:
  static: "~/src/django-site/static"
  js: "{static}/js"
  css: "{static}/css"
  fonts: "{static}/fonts"

tests:
  - ['(?i)\.js$', "{js}"]
  - ['(?i)\.css$', "{css}"]
  - ['(?i)\.(woff2?|ttf|otf|eot)$', "{fonts}"]
```

Local file

Just copy a local file.

```
files:
  - filename: "~/src/js/palette.js"
```

Single file download

Directly download a single file from url and copy it as is. We define only the *comlplete* url the get_type and the action to take is guessed by the downloaders.

```
files:
  - url: https://code.jquery.com/jquery-3.2.1.min.js
```

Single file download with get query string

Passing the query string to the url and use the filename to tell how to save the result.

```
files:
  - url: https://sciactive.com/pnotify/buildcustom.php?mode=js&min=true&
↪modules=desktop-buttons-nonblock-animate-confirm-callbacks-history&cff=.js
    filename: pnotify.custom.min.js
```

GitHub raw download

Download a raw file from github. We define *user/repository* as url and the filename of the file to download.

```
files:
  - url: moment/moment
    filename: min/moment-with-locales.min.js
```

GitHub latest release

Try to get latest release or the master.zip if can't valid assets. We define *user/repository* as url and use the *files* list to tell which files to download.

```
- url: 'twbs/bootstrap'
  files:
    - filename: js/bootstrap.min.js
    - filename: css/bootstrap.min.css
    - filename: css/bootstrap-theme.min.css
    - filename: fonts/*.woff*   # note we can use wildcards
```

Custom downloaders

GetWebFilesInator comes with some default **Downloader** to manage local files, direct downloads and GitHub latest release and GitHub raw download.

Anyway you can write your own downloader, and is quite easy.

Downloader Object

class `getwebfilesinator.downloaders.downloaders_defaults.`**`Downloader`**(*client*)

Class for create downloaders for GetWebFilesInator

`cfg` = **None**

The configuration object.

`client` = **None**

The client object.

`download`(*sfile*)

Process a sfile

Parameters **`sfile`** (*Edo*) – a Edo object (which is a dict with attributes)

Return type any.

This method can return:

- `False`: if we want to stop processing the file.
- `True` or `None`: if we want the client guess the action from the filename.
- `'zip'` or `'plain'`: to force the action to take.
- a `callable` which accepts `sfile` and `cfg` as parameters if we want to pass the action to take.

`get_type` = **''**

Name of the get_type used to map the downloader.

`guess_priority` = **0**

Gives a priority for the `guess_type` type method

`guess_type`(*sfile*)

Guess the get_type

Parameters **`sfile`** (*Edo*) – the sfile to test.

Return type self or None

Returns `None` if the test on sfile fails else returns `self`

Examples

Moaning downloader

A silly example which do nothing, except to moan for his poor situation.

```
from logging import getLogger
from getwebfilesinator.downloaders.downloaders_defaults import Downloader
log = getLogger(__name__)

class DownloaderTest(Downloader):
    """Test downloader"""

    # Our get type so we can map the Downloader
    get_type = 'test_download'
    # Actually we don't need a guess_priority in this case, we cant omit this

    def download(self, sfile):
        """Download the file"""
        # just log something..
        log.info(":'( I'm a test downloader. No actions for me :'(")
        # and return False so the file will not be processed
        return False

    def guess_type(self, sfile):
        """Guess retrieve type"""
        # just verify is the url is 'imatest'
        if sfile.url == 'imatest':
            return self
```

Greet (action included)

This silly downloader uses a personal action to bypass the standard client behavior.

```
from logging import getLogger
from getwebfilesinator.downloaders.downloaders_defaults import Downloader
log = getLogger(__name__)

class DownloaderGreet(Downloader):
    """
    Downloader with action included

    our url must start with greet://
    """

    get_type = 'greet'

    def guess_type(self, sfile):
        """Guess the type"""
        # the url must starts with ``greet://``
        if sfile.url.startswith('greet://'):
            return self

    def download(self, sfile):
        """Process sfile"""
        # OK retrieve our greet part.
        sfile.greet = sfile.url[8:]
```

```
        # and return our action, this will be executed by the client.
        return self.greet

    def greet(self, sfile, cfg):
        """ Greeting action """
        # just greet.
        log.info("Now we are greeting %s", sfile.greet)
```

Note: I made this script for a personal need is not intended for professional use.

Todo

Implement middlewares.

Index

C

D

G