producti_gestio Documentation

Release 0.7.2

Gabriel Hearot

Jun 05, 2018

producti-gestio

CHAPTER 1

producti_gestio

The Main module. A simple REST-API server creator.

CHAPTER 2

producti_gestio package

2.1 Subpackages

2.1.1 producti_gestio

The Main module. A simple REST-API server creator.

2.1.2 producti_gestio.core package

producti_gestio.core

That's the core part of the module. It is used to handle the requests and call the **user-defined** function.

producti_gestio.core.check

producti_gestio.core.check

It checks if a request is valid using Filters.

`producti_gestio.core.check.`**`check`**(*filters: object = None*) → <built-in function callable>

This function is used to check if a request is valid using filters.

It could be used only when the `Server` is using the `function` mode (alias *using decorator*, not `handlers`).

Here an example:

```
from producti_gestio import Server, check, Decorator, Filters

@producti_gestio.Decorator # Using this Decorator, the next function will become␣
↪the server-creator function
def my_server(**kwargs):
```

(continues on next page)

(continued from previous page)

```
    print("Initializing the server")

    @producti_gestio.check(Filters.get) # Using this Decorator, the handler␣
↪function will check for the requirements (using Filters)
    def my_function(**kwargs):
        return {
            'response_code': 200, # The response code
            'response': {
                'ok': True,
                'message': 'Hello world!'
            }
        }

    return my_function

my_server(allow_get=True) # Create and start the server instance using threads

while True:
    pass
```

Parameters

- **func** (*callable*) – The native handler function
- **filters** (*Filter or None*) – A Filter object or a None type
- **filters** – object or None: (Default value = None)

Returns A callable object

Return type callable

producti_gestio.core.request_handler module

producti_gestio.core.request_handler

It handles the requests checking the configuration, then pass all parameters and headers to the **user-defined function**.

A **request handler** must have these requirements:

- Have a `do_GET` function
- Have a `do_POST` function

When a Server instance is started, the *request handler* will get the configuration under `self.configuration` and the handlers under `self.handlers`.

class producti_gestio.core.request_handler.**RequestHandler**(*request*, *client_address*, *server*)

Bases: `http.server.BaseHTTPRequestHandler`

The RequestHandler class is used to handle all requests, after they are checked using the configuration.

It has got two needed methods: **do_GET** and **do_POST**, they will be called by the HTTPServer classes, based on the type of the request.

Actually, **HEAD** and **PUT** request methods are not supported.

configuration = {'allow_get': False, 'allow_post': True, 'debug': False, 'function'

do_GET() → bool
The GET requests handler, it checks if GET is allowed as method and then parse the request and pass it to a defined function and return a response.

Returns True if the request succeeded, False if not or GET is not allowed.

Return type bool

do_POST() → bool
The POST requests handler, it checks if POST is allowed as method and then parse the request and pass it to a defined function and return a response.

Returns True if the request succeeeded, False if not or POST is not allowed.

Return type bool

do_request(*request_infos: dict*) → bool
The `do_request` function is used to get a response from the *handler function* or from one of the *Handlers* and send it to the user.

Parameters **request_infos** (*dict*) – A dictionary that will be passed to the Handler function or the Handler.

Returns True if all went right, otherwise False.

Return type bool

handlers = []

parse_post() → <function NewType.<locals>.new_type at 0x7f96f87c3510>
The POST parameters parser. It checks the self.headers dictionary, its content-type and if it is 'multipart/form-data' or 'application/x-www-form-urlencoded', then parses the POST parameters.

Returns POST parameters in a dictionary, where the keys are the parameter names and the values are their values.

Return type dict

use_handler = False

2.1.3 producti_gestio.decorator package

producti_gestio.decorator module

producti_gestio.decorator

That's the decorator part of the module. It is a simple way to create a new web-server without using directly the producti_gestio.server.server class.

producti_gestio.decorator.wrapper

It defines a new class with decorator-magic-methods, that will be called by the function that points to it.

The **user-defined** function, that's how we'll call the function that will be passed as the **handler function**, will be passed to a new Decorator instance with all of the user-passed parameters, that we'll call **configuration**.

You can also use *producti_gestio.handlers.handler* instead of this `Decorator`.

This decorator is intended to be used when there's one **handler function**. The function that is decorated using this class will become the **server-creator function** (calling it will create the server).

Here an example:

```
import producti_gestio

@producti_gestio.Decorator # Use the Decorator
def my_server(**kwargs):
    print('Creating the server')

    def my_function(**kwargs):
        return {
            'response_code': 200,
            'response': {
                'ok': True,
                'message': 'Hello world!'
            }
        }

    return my_function # Return the handler function

my_server(allow_get=True) # Create and start the server instance
```

You can also use the *producti_gestio.core.check()* function to use `Filters`.

class producti_gestio.decorator.wrapper.**Decorator**(*call: <function NewType.<locals>.new_type at 0x7f96f87c3598>*)

Bases: `object`

The Decorator class that launch automatically the server and uses the user-defined function.

It has got just magic methods.

__call__(***kwargs*) → bool
It launches the server and sets the handler function.

Parameters **kwargs** – The chosen configuration. See `producti_gestio.server.Server`.

Returns True if all went well, False if not

Return type bool

__init__(*call: <function NewType.<locals>.new_type at 0x7f96f87c3598>*)
It sets the server-creator function and creates the instance. Before setting the function, it checks if it is callable or not, and the response depends on that.

Parameters **call** (*handler_function*) – The user-defined function

Raises `NotAFunction` – It throws a NotAFunction exception if the given parameter is not a function

__repr__() → str
It returns a representation of the object.

Returns A representation of the object

Return type str

server = None

2.1.4 producti_gestio.exceptions package

producti_gestio.exceptions module

producti_gestio.exceptions

That's the exceptions part of the module. It is used to throw specific exceptions.

producti_gestio.exceptions.exceptions

It contains all the producti-gestio exceptions.

exception producti_gestio.exceptions.exceptions.**NotABoolean**
Bases: Exception

Exceptions thrown if the given parameter is not a boolean.

exception producti_gestio.exceptions.exceptions.**NotAFunction**
Bases: Exception

Exceptions thrown if the given parameter is not a callable function.

exception producti_gestio.exceptions.exceptions.**NotAString**
Bases: Exception

Exceptions thrown if the given parameter is not a string.

exception producti_gestio.exceptions.exceptions.**NotAnInteger**
Bases: Exception

Exceptions thrown if the given parameter is not an integer.

exception producti_gestio.exceptions.exceptions.**NotDefinedFunction**
Bases: Exception

Exceptions thrown if the function is not defined.

2.1.5 producti_gestio.filters package

producti_gestio.filters

That's the filters part of the module. It is used to filter the requests.

producti_gestio.filters.filters module

producti_gestio.filters.filters

All the filters are defined here using the Filters and the Filter class.

You can use filters by passing them as parameters in *producti_gestio.server.server.Server.on_request()*:

```
from producti_gestio import Server, Filter

my_server = Server(allow_get=True) # Create a server instance

@my_server.on_request(Filter.get) # It will filter the GET requests
def my_function(**kwargs):
    return {
        'response_code': 200, # The response code
        'response': { # The response as a dictionary, it will be encoded in JSON
            'ok': True
        }
    }

my_server.start() # Start the server using threads

while True:
    pass
```

You can also use operators (`&`, `|`, `and`, `or`, `~`)!

You can check our repository for other examples.

class producti_gestio.filters.filters.**Filters**

Bases: `object`

The Filters class is used to define all Filters that could be used.

You can also create your own filter, see *producti_gestio.filters.filters.build()*.

get = <producti_gestio.filters.filters.Get object>

static path(*given_path: str*) → producti_gestio.filters.filter.Filter

It filters requests using a path.

The `Path` always starts with `/`!

Parameters

- **given_path** (*str*) – The path you'd like to check
- **given_path** – str:

Returns A Filter-like object

Return type *Filter*

post = <producti_gestio.filters.filters.Post object>

static regex(*pattern: str*, *flags: int = 0*) → producti_gestio.filters.filter.Filter

It filters requests with a path that match a given RegEx pattern.

The `Path` always starts with `/`!

You can also give a flag (using `re`), here an example:

```
import re
from producti_gestio import Filters, Server

my_server = Server(allow_get=True) # Create a server instance

@my_server.on_request(Filters.regex('^/print', re.IGNORECASE)) # It filters␣
↪all requests that start with print, case-insensitive
def my_function(**kwargs):
```

(continues on next page)

(continued from previous page)

```
    return {
        'response_code': 200, # The response code
        'response': {
            'ok': True,
            'message': repr(kwargs['parameters'])
        }
    }

my_server.start() # Start the server using threads

while True:
    pass
```

Parameters

- **pattern** (*str*) – The RegEx pattern
- **flags** (*int*) – The RegEx flags
- **pattern** – str:
- **flags** – int: (Default value = 0)

Returns A Filter-like object

Return type *Filter*

producti_gestio.filters.filters.**build**(*name: str*, *func: <built-in function callable>*, ***kwargs*) → producti_gestio.filters.filter.Filter

It builds a Filter class using a dictionary.

It is used to create a Filter just giving a name and a function using `type()`.

The provided function must have these parameters:

- _ - The object that is passed (useful when using a compiled function, see *producti_gestio.filters.filters.Filters.regex()*)
- **kwargs** - A dictionary of all useful informations (the same is passed to the *handler function*)

Here an example:

```
from producti_gestio.filters.filters import build
from producti_gestio import Server

my_filter = build('My filter', lambda _, kwargs: bool(kwargs['path'] == '/my_path
↪')) # If the path is '/my_path'
my_server = Server(allow_get=True)

@my_server.on_request(my_filter) # It uses our own filter
def my_function(**kwargs):
    return {
        'response_code': 200, # The response code
        'response': {
            'ok': True,
            'path': kwargs['path']
        }
    }

my_server.start() # Start the server using threads
```

(continues on next page)

(continued from previous page)

```
while True:
    pass
```

Parameters

- **name** (*str*) – The name of the Filter.
- **func** (*callable*) – A callable function that checks the request
- **kwargs** – A dictionary for other methods
- **name** – str:
- **func** – callable:
- ****kwargs** –

Returns A Filter-like object

Return type *Filter*

producti_gestio.filters.filter module

producti_gestio.filters.filter

The Filter class is defined here, it is used to check the request using the filters.

When *producti_gestio.filters.filters.build()* is triggered, a new `Filter` object is created. This object is used to create the *special methods* (or dunder methods because of their ___) for Python operators (`__invert__` for ~, `__and__` for `and` and `&`, `__or__` for `or` and `|`).

It could be useful when using `Filters`. Here an example:

```
from producti_gestio import Server, Filters

my_server = Server(allow_get=True) # Create a server instance

@my_server.on_request(Filters.get & Filters.path('/something')) # It will filter the␣
↪GET requests
def my_function(**kwargs):
    return {
        'response_code': 200, # The response code
        'response': { # The response as a dictionary, it will be encoded in JSON
            'ok': True
        }
    }

my_server.start() # Start the server using threads

while True:
    pass
```

class producti_gestio.filters.filter.**AndFilter**(*base: object*, *other: object*)

Bases: *producti_gestio.filters.filter.Filter*

The AndFilter is used to check two Filters and return their result

__call__(*args: dict*) → bool
It returns the result of two Filters.

Parameters **args** (*dict*) – The dictionary that will be used by filters

Returns True if all the Filters returned True, False if not

Return type bool

__init__(*base: object*, *other: object*)
It initializes the object.

Parameters

- **base** (*object*) – One of the Filters that you'd like to check
- **other** (*object*) – The other Filter

class producti_gestio.filters.filter.**Filter**
Bases: object

The Filter class is used to check the given requests using Filters.

__and__(*other: object*) → object
It returns a AndFilter object.

Parameters **other** (Filter) – The other filter you'd like to check

Returns The AndFilter object

Return type *AndFilter*

__call__(*args: dict*)
It raises a NotImplementedError.

Parameters **args** (*dict*) – The dictionary that will be used by filters

Raises NotImplementedError

__invert__() → object
It returns a InvertFilter object.

Returns The InvertFilter object

Return type *InvertFilter*

__or__(*other: object*) → object
It returns a OrFilter object.

Parameters **other** (Filter) – The other filter you'd like to check

Returns The OrFilter object

Return type *OrFilter*

class producti_gestio.filters.filter.**InvertFilter**(*base: object*)
Bases: *producti_gestio.filters.filter.Filter*

The InvertFilter is used to return an inverted filter

__call__(*args: dict*) → bool
It returns the inverted result of a Filter.

Parameters **args** (*dict*) – The dictionary that will be used by filters

Returns The inverted result of the Filter

Return type bool

__init__(*base: object*)
It initializes the object.

Parameters **base** (*object*) – The Filter that you'd like to check

class producti_gestio.filters.filter.**OrFilter**(*base: object*, *other: object*)
Bases: *producti_gestio.filters.filter.Filter*

The OrFilter is used to check two Filters and see if one of them is True.

__call__(*args: dict*) → bool
It returns the result of two Filters.

Parameters **args** (*dict*) – The dictionary that will be used by filters

Returns True if one of the Filters returned True, False if not

Return type bool

__init__(*base: object*, *other: object*)
It initializes the object.

Parameters

- **base** (*object*) – One of the Filters that you'd like to check
- **other** (*object*) – The other Filter

2.1.6 producti_gestio.handlers package

producti_gestio.handlers module

producti_gestio.handlers

That's the handlers part of the module. It contains handlers that could be used to check and get requests.

producti_gestio.handlers.handler

It contains the main handler that could be used to check and get requests.

class producti_gestio.handlers.handler.**Handler**(*callback: <function NewType.<locals>.new_type at 0x7f96f87c3ea0>*, *filters: producti_gestio.filters.filter.Filter = None*)
Bases: object

It's the main Handler, it could be used to check if the request using filters.

It is used by producti_gestio.server.server.add_handler() to create a new Handler.

Here an example:

```
from producti_gestio import Server
from producti_gestio.handlers.handler import Handler

my_server = Server(allow_get=True) # Create the Server instance.
my_handler = Handler(lambda parameters: bool(parameters['path'] == '/test')) #␣
↪Create the handler
```

(continues on next page)

(continued from previous page)

```
my_server.add_handler(my_handler) # Add the handler to the Server object
my_server.start() # Run the server using threads
```

__init__(*callback: <function NewType.<locals>.new_type at 0x7f96f87c3ea0>*, *filters: producti_gestio.filters.filter.Filter = None*) → None

check(*parameters*) → bool
It checks if the request is valid using filters.

Parameters **parameters** (*dict*) – The request handler parameters

Returns True if the request is valid, False if not

Return type bool

filters = None

2.1.7 producti_gestio.project package

producti_gestio.project module

producti_gestio.project

That's the project part of the module. It is a simple way to generate a new project.

producti_gestio.project.generator

It generates a simple mini-project.

producti_gestio.project.generator.**generate_code**(*name*) → bool
It generates the source code in a directory.

Parameters **name** (*str*) – The directory name

Returns True if all went well.

Return type bool

2.1.8 producti_gestio.server package

producti_gestio.server.server module

producti_gestio.server

That's the server part of the module. It contains the main class: the server class. It could generate a new server based on a configuration and a function.

producti_gestio.server.server

This section contains the *producti_gestio.server.server.Server* class, the main class of the entire library.

It could create the web-server, set the configuration by the init function or using set_* functions.

You can create handlers (using *producti_gestio.server.server.Server.on_request()* or not) and start using it simply using *producti_gestio.server.server.Server.run()* or *producti_gestio.server.server.Server.start()* for who's gonna use Threads.

The `producti_gestio.Server` allows a custom configuration that you can pass when you create a new instance as `kwargs`:

- *allow_get* (`bool`): If you want to allow `GET` requests. **Default is False**
- *allow_post* (`bool`): If you want to allow `POST` requests. **Default is True**
- *function* (`callable`): Your `handler_function`, it won't be considered if you're using Handlers.
- *debug* (`bool`): If you want to get **debugging infos**
- *ip** (`str`): Your `ip`. **Default is 127.0.0.1**
- *port* (`int`): Your `server post`. **Default is 8000**

Here an example:

```
from producti_gestio import Server

my_server = Server(allow_get=True, allow_post=False) # Create the Server instance

def my_function(**kwargs):
    return {
        'response_code': 200,
        'response': {
            'ok': True,
            'message': 'Hello world!'
        }
    }

my_server.set_function(my_function) # Set the handler function
my_server.start() # Run the server using threads

while True:
    pass
```

Or using *producti_gestio.handlers.handler* and *producti_gestio.filters.filter*:

```
from producti_gestio import Server, Filter

my_server = Server(allow_get=True, allow_post=False) # Create the Server instance

@my_server.on_request(Filter.get) # Define the decorator for the function (it uses␣
↪Filters)
def my_function(**kwargs): # Define the handler function
    return {
        'response_code': 200, # The response code
        'response': { # A dict of the response (it will be encoded in JSON)
            'ok': True,
            'message': 'Hello world!'
        }
    }

my_server.start() # Run the server using threads

while True:
    pass
```

class producti_gestio.server.server.**Server**(***conf*)

Bases: object

The main class. It is used to create, edit configuration and launch the web server.

__init__(***conf*)

It defines the configuration by merging the default configuration with the user-chosen configuration.

Parameters **conf** (*configuration*) – The chosen configuration. See *producti_gestio.server.server*.

__repr__() → str

It returns a representation of the object.

Returns A representation of the object

Return type str

add_handler(*handler: object*) → bool

It adds an handler to the server object.

Here an example:

```
from producti_gestio import Server
from producti_gestio.handlers.handler import Handler

my_server = Server(allow_get=True) # Create the Server instance.
my_handler = Handler(lambda parameters: bool(parameters['path'] == '/test'))
↪# Create the handler

my_server.add_handler(my_handler) # Add the handler to the Server object
my_server.start() # Run the server using threads
```

Parameters

- **handler** (Handler) – The Handler
- **handler** – object:

Returns True if all went right, False on fails

Return type bool

on_request(*filters: object = None*) → <built-in function callable>

It adds the function to the handler.

It could be used as a decorator, here an example:

```
from producti_gestio import Server, Filters

my_server = Server(allow_get=True) # Create the Server instance

@my_server.on_request(Filters.get) # Create and add an handler using Filters
der my_function(**kwargs):
    return {
        'response_code': 200, # The response code
        'response' : {
            'ok': True
        }
    }

my_server.start() # Start the server using Threads
```

Parameters

- **filters** (Filter) – The filters you'd like to use
- **filters** – object or None: (Default value = None)

Returns A decorator function

Return type callable

remove_handler (*handler: object*) → bool
It removes an handler from the server object.

Here an example:

```
from producti_gestio import Server
from producti_gestio.handlers.handler import Handler

my_server = Server(allow_get=True) # Create the Server instance.
my_handler = Handler(lambda parameters: bool(parameters['path'] == '/test'))
↪# Create the handler
other_handler = Handler(lambda parameters: bool(parameters['path'] == '/
↪second_test')) # Create the other handler


my_server.add_handler(my_handler) # Add the handler to the Server object
my_server.start() # Start the server using threads

my_server.add_handler(other_handler) # Add the other handler to the Server␣
↪object
my_server.remove_handler(my_handler) # Remove 'my handler' from the Server␣
↪object
```

Parameters

- **handler** (Handler) – The Handler
- **handler** – object:

Returns True if all went right, False on fails

Return type bool

run (*handler_class=<class 'producti_gestio.core.request_handler.RequestHandler'>*)
It launches the server and add the configuration to the Handler class.

Remember: It doesn't use _thread, see *producti_gestio.server.server.Server.start()* for that.

Parameters **handler_class** (*classobj|type, optional*) – The Handler class you would like to use. Default is *producti_gestio.core.request_handler.RequestHandler*.

Raises NotDefinedFunction – It throws a NotDefinedFunction exception if the handler function is not defined.

set_allow_get (*allow_get: bool*) → bool
It sets if the GET method is allowed by the allow_get parameter.

Remember: The *GET* method is **not** allowed on by default!

Here an example:

```
from producti_gestio import Server

my_server = Server() # Create the server instance

@my_server.on_request
def my_function(**kwargs):
    return {
        'response_code': 200, # The Response Code
        'response': {
            'ok': True
        }
    }

my_server.set_allow_get(True) # Allow GET requests
my_server.start() # Start the server using Threads
```

Parameters

- **allow_get** (*bool*) – A boolean of the choice.
- **allow_get** – bool:

Raises NotABoolean – It throws a NotABoolean exception if the given parameter is not a boolean.

set_allow_post (*allow_post: bool*) → bool

It sets if the POST method is allowed by the allow_post parameter.

Remember: The *POST* method is allowed by default!

Here an example:

```
from producti_gestio import Server

my_server = Server() # Create the server instance

@my_server.on_request
def my_function(**kwargs):
    return {
        'response_code': 200, # The Response Code
        'response': {
            'ok': True
        }
    }

my_server.set_allow_post(True) # Allow POST requests
my_server.start() # Start the server using Threads
```

Parameters

- **allow_post** (*bool*) – A boolean of the choice.
- **allow_post** – bool:

Raises NotABoolean – It throws a NotABoolean exception if the given parameter is not a boolean.

set_debug_mode (*debug: bool*) → bool

It sets if the Debug mode is on.

Here an example:

```
from producti_gestio import Server

my_server = Server() # Create the server instance

@my_server.on_request
def my_function(**kwargs):
    return {
        'response_code': 200, # The Response Code
        'response': {
            'ok': True
        }
    }

my_server.set_debug_mode(True) # Enable the debug mode
my_server.start() # Start the server using Threads
```

Parameters

- **debug** (*bool*) – A boolean of the choice.
- **debug** – bool:

Raises `NotABoolean` – It throws a NotABoolean exception if the given parameter is not a boolean.

set_function (*handler_function: <built-in function callable>*) → bool

It sets the function by the handler_function parameter.

Here an example:

```
from producti_gestio import Server

my_server = Server() # Create the server instance

def my_function(**kwargs): # It creates the function
    return {
        'response_code': 200, # The Response Code
        'response': {
            'ok': True
        }
    }

my_server.set_function(my_function) # Set the handler function
my_server.start() # Start the server using Threads
```

Parameters

- **handler_function** (*callable*) – The user-defined function you would like to set.
- **handler_function** – callable:

Raises `NotAFunction` – It throws a NotAFunction exception if the given parameter is not a function.

set_ip (*ip: str*) → bool

It sets the IP by the IP parameter.

Here an example:

```
from producti_gestio import Server

my_server = Server() # Create the server instance

@my_server.on_request
def my_function(**kwargs):
    return {
        'response_code': 200, # The Response Code
        'response': {
            'ok': True
        }
    }

my_server.set_ip('127.0.0.1') # Set the IP
my_server.start() # Start the server using Threads
```

Parameters

- **ip** (*str*) – The IP you chose
- **ip** – str:

Raises `NotAString` – It throws a NotAString exception if the given parameter is not a string.

set_port (*port: int*) → bool
It sets the Port by the Port parameter.

Here an example:

```
from producti_gestio import Server

my_server = Server() # Create the server instance

@my_server.on_request
def my_function(**kwargs):
    return {
        'response_code': 200, # The Response Code
        'response': {
            'ok': True
        }
    }

my_server.set_port(8000) # Set the Port
my_server.start() # Start the server using Threads
```

Parameters

- **port** (*int*) – The Port you chose.
- **port** – int:

Raises `NotAnInteger` – It throws a NotAnInteger exception if the given parameter is not an integer.

shutdown ()
It shutdowns the server.

start() → bool
It starts the server. If thread_activated is True, it starts the server using _thread, else it just calls *producti_gestio.server.server.Server.run()*.

Returns True if all went right, False on fails

Return type bool

stop()
It stops the server, you can re-put it on in by re-calling the run function.

thread_activated = True

2.1.9 producti_gestio.utils package

producti_gestio.utils.arguments_parser module

producti_gestio.utils.arguments_parser

It just parses the given arguments via command line.

producti_gestio.utils.arguments_parser.**parser**(*args: list = ['/home/docs/checkouts/readthedocs.org/user_builds/producti-gestio/envs/stable/bin/sphinx-build', '-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*) → object
It tries to parse sys.argv and processes given parameters.

Parameters **args** – list: (Default value = sys.argv)

Returns A parse_args object of all the parameters.

Return type object

CHAPTER 3

Import the library

To import *producti_gestio*, just use the `import` statement:

```
$ python
Python 3.6.5 (default, Apr 12 2018, 22:45:43)
[GCC 7.3.1 20180312] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import producti_gestio
```

If you don't get an error, all is right! Now you can access all the *producti_gestio* module!

CHAPTER 4

Installation

4.1 From PyPi

Just use **pip**:

```
pip install producti_gestio
```

Or if you want to *upgrade* the package:

```
pip install --upgrade producti_gestio
```

4.2 From Github

4.2.1 Using Pip

Try using that piece of code:

```
pip install git+https://github.com/pyTeens/producti-gestio.git
```

Or if you want to *upgrade* the package

```
pip install --upgrade git+https://github.com/pyTeens/producti-gestio.git
```

4.2.2 Downloading files

In primis (from Latin, "firstable"), **clone** the repository:

```
git clone https://github.com/pyTeens/producti-gestio.git
```

Then, change directory:

```
cd producti-gestio
```

And finally, install the **package**:

```
sudo python3 setup.py install
```

CHAPTER 5

Verification

5.1 Command Line

To **verify** if the installation was successful, open the `Command Line` and type this command:

```
$ python
Python 3.6.5 (default, Apr 12 2018, 22:45:43)
[GCC 7.3.1 20180312] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import producti_gestio
>>> producti_gestio.__version__
'0.7.0'
```

If all went well, your installation if **verified**, else if you get an `ImportError`, try reinstalling it.

CHAPTER 6

Changelog

producti_gestio *Version:* 0.7.2

- Fix installation
- Add dependencies in setup.py
- Automatic reading of requirements.txt in setup.py
- Remove bin/producti-gestio, setup.py now will create the script from producti_gestio/__main__.py

Main Contributors:

- Gabriel Hearot <gabriel@hearot.it>

CHAPTER 7

Welcome to producti gestio!

producti gestio (/producti gestio/, from Latin: *Product management*) is a simple **API like web server** written in Python. It could be used for your **projects** and it is useful for **debugging**. You can start a **web server** in few seconds and *enjoy* your development session.

Using *producti-gestio*, you are allowed to have a *coffee break*, because the entire server is managed by it!

CHAPTER 8

Contents

8.1 Installation

8.2 From PyPi

Just use **pip**:

```
pip install producti_gestio
```

Or if you want to *upgrade* the package:

```
pip install --upgrade producti_gestio
```

8.3 From Github

8.3.1 Using Pip

Try using that piece of code:

```
pip install git+https://github.com/pyTeens/producti-gestio.git
```

Or if you want to *upgrade* the package

```
pip install --upgrade git+https://github.com/pyTeens/producti-gestio.git
```

8.3.2 Downloading files

In primis (from Latin, "firstable"), **clone** the repository:

```
git clone https://github.com/pyTeens/producti-gestio.git
```

Then, change directory:

```
cd producti-gestio
```

And finally, install the **package**:

```
sudo python3 setup.py install
```

8.4 Usage

8.4.1 Import the library

Just use `import` statement:

```
import producti_gestio
```

8.4.2 Create an handler function

You're going to use a **decorator** that will create a **Wrapper**. Here an example:

```
from producti_gestio import Server

my_server = Server(allow_get=True) # Create the server instance

@my_server.on_request
def my_function(**kwargs):
    return {
        'response_code': 200, # The response code (see https://en.wikipedia.org/wiki/
↪List_of_HTTP_status_codes)
        'response': { # A dictionary that will be encoded in JSON
            'ok': True,
            'message': 'Hello world!'
```

(continues on next page)

(continued from previous page)

```
        }
    }
```

Then, if you call `my_server.start()` you'll start the *HTTPServer* (using `Threads`):

And, if you'll surf `127.0.0.1:8000` that will be the **output**:

```
{'ok': True, 'message': 'Hello world!'}
```

8.4.3 Get parameters, headers, etc...

Just look at the `kwargs` parameter. It contains a *dictionary* of all the informations you need.

- **Parameters** -> `kwargs['parameters']`
- **Headers** -> `kwargs['header']`
- **Request type** -> `kwargs['request-type']`
- **Path requested** -> `kwargs['path']`
- **Handler object** -> `kwargs['object']`

8.4.4 Configuration

You can pass your own configuration to the **server-creator function** Here all the keyword arguments you can pass:

- **allow_get** (default is *False*)
- **allow_post** (default is *True*)
- **function** (default is *None*, but it is needed, it's the **handler_function**),
- **debug** (default is *False*, if you want to print the *Traceback* under `error_message` in the JSON response when an Exception is caught)
- **ip** (default is *'127.0.0.1'*)
- **port** (default is *8000*)

8.4.5 Run the server using decorators

Just call the **handler function**:

```
import producti_gestio

@producti_gestio.Decorator
def server_create(*args, **kwargs):
    """
    It create the server and
    launch it
    """
    def handler_function(*args, **kwargs):
        return ({
            'response_code': 200,
            'response': {
```

(continues on next page)

(continued from previous page)

```
                'ok': True,
                'is_meme': True
            }
        })

    return handler_function

if __name__ == '__main__':
    server_create(allow_get=True)
```

8.5 Files

You'll find lots of *not understandable* **directory** and **files**, so here a list and definitions of them:

- **producti_gestio** - *Main directory*
  - **producti_gestio/__init__.py** - *Init file, it included all classes*
  - **producti_gestio/__main__.py** - *It parses and processes all given parameters from the command line*
  - **producti_gestio/core** - *Directory for all important classes such as request_handler*
    - **producti_gestio/core/__init__.py** - *It includes all core classes*
    - **producti_gestio/core/check.py** - *It defines a check function that could be used a decorator for filters*
    - **producti_gestio/core/request_handler.py** - *The Handler of the requests, it passes parameters to the defined Handler function and then it send the JSON response*
  - **producti_gestio/decorator** - *Directory for all help-decorator classes*
    - **producti_gestio/decorator/__init__.py** - *It includes all decorator classes*
    - **producti_gestio/decorator/wrapper.py** - *The Decorator class, it is used to launch the Server class and define the Handler function*
  - **producti_gestio/exceptions** - *Directory for all the exception*
    - **producti_gestio/exceptions/__init__.py** - *It includes all the exceptions*
    - **producti_gestio/exceptions/exceptions.py** - *It defines all the exceptions*
  - **producti_gestio/filters** - *Directory for all the filters*
    - **producti_gestio/filters/__init__.py** - *It includes filter.py and filters.py*
    - **producti_gestio/filters/filter.py** - *It defines the Filter class*
    - **producti_gestio/filters/filters.py** - *It defines all the Filters*
  - **producti_gestio/handlers** - *Directory for the handlers*
    - **producti_gestio/handlers/__init__.py** - *It includes handler.py*
    - **producti_gestio/handlers/handler.py** - *It defines the Handler class*
  - **producti_gestio/project** - *Directory of some project generator tools*
    - **producti_gestio/project/__init__.py** - *It includes the project generator*
    - **producti_gestio/project/generator.py** - *Tools for code auto-generating*
  - **producti_gestio/server** - *The Server directory*

    - **producti_gestio/server/__init__.py** - *It includes all server classes*
    - **producti_gestio/server/server.py** - *The main class, it creates the server*
  - **producti_gestio/utils** - *Directory of some useful tools*
    - **producti_gestio/utils/__init__.py It includes the arguments parser**
    - **producti_gestio/utils/arguments_parser.py** - *It parses and processes the given arguments from the command line*

8.6 How to contribute

In primis ("firstable"), you **must** read the code of conducts and the contributing document, then ask @hearot to enter the **organization** (pyTeens).

Python Module Index

p

Index

Symbols

A

B

C

D

F

G

H

I

N