apicore Documentation

Release 1.0

dev@meez.io

Feb 11, 2018

Contents

Set of core libraries for building REST API and Microservices based on Flask.

The code is open source, release under MIT and written in Python 3.

```
pip install apicore
```

CHAPTER 1

Features

- Cross-origin resource sharing (CORS) ready
- Data caching with redis server or direct in-memory
- Configuration file loader
- A simple Logger
- Raise exception conform to HTTP status codes
- OpenAPI 3.0 specification embedded with Swagger UI

CHAPTER 2

Example

```
#!/usr/bin/env python

from apicore import api, Logger, config, Http409Exception

Logger.info("Starting {} API Server...".format(config.app_name))


@api.route('/error/')
def error():
    """
    summary: Raise an execption
    responses:
      409:
          description: Conflict
    """
    raise Http409Exception()


if __name__ == "__main__":
    # api is an instance of API which inherit from Flask
    api.debug = config.debug
    api.run()
```

CHAPTER 3

Configuration

Configuration is set in `conf/config.yaml` file (see *apicore.config.Config*).

Name	Default value	Description
app_name	"My App"	Application's name.
debug	True	Active debug mode.
prefix	""	Add a prefix to all URL paths (ie: "/api").
redis	None	Redis server used for caching data : redis://:password@localhost:6379/0. If not set use in-memory.
swagger_ui	"/"	Relative URL path to embedded Swagger UI (`prefix` + `swagger_ui`).
specs_login	None	Login to access API Specification (`openapi.json`), no Authentification by default.
specs_pwd	None	Password to access API Specification.

CHAPTER 4

OpenAPI 3.0

- See specification for syntax.
- Document route's methods with Operation Object using yaml syntax.
- Document your API in `conf/openapi.yaml` file.
- Access your documentation through a python dictionary : `api.oas.specs`.
- Your spec is available at `http[s]://<hostname>/openapi.json`.
- Default path to `http[s]://<hostname>/` to see your spec with Swagger UI (set `swagger_ui` in `conf/config.yaml` to change path)
- Full exemple :

```
@api.route('/sellers/<idseller>/', methods=['GET', 'PUT'])
def seller(idseller):
    """
    description: "Path Item Object" level here, only common_responses is added to␣
↪OpenAPi specification. Next level are "Operation Object".
    parameters:
      - name: idseller
        in: path
        description: uuid of seller
        required: true
        type: string
        format: uuid
    common_responses:
        400:
         description: Invalid request
        401:
          description: Authentification required
        403:
          description: Ressource access denied
        500:
          description: Server internal error
    ---
```

```
    tags:
      - profile
    summary: Find a seller profile by ID
    responses:
      200:
        description: Success
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Seller'
      404:
        description: Ressource does not exist
      406:
        description: Nothing to send maching Access-* headers
  ---
    tags:
      - profile
    summary: Update seller profile
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Seller'
        required: true
    responses:
      200:
        description: Success
  """
  pass


  print(api.oas.spec)
```

CHAPTER 5

APIs

5.1 api

`api` is the application, instance of *apicore.api.API* inherited from `flask.Flask`. It handle Cross-origin resource sharing (CORS) and JSON responde message (instead of HTML).

class apicore.api.**API**(*import_name*)

add_template_filter(*f*, *name=None*)
Register a custom template filter. Works exactly like the *template_filter()* decorator.

Parameters **name** – the optional name of the filter, otherwise the function name will be used.

add_template_global(*f*, *name=None*)
Register a custom template global function. Works exactly like the *template_global()* decorator.

New in version 0.10.

Parameters **name** – the optional name of the global function, otherwise the function name will be used.

add_template_test(*f*, *name=None*)
Register a custom template test. Works exactly like the *template_test()* decorator.

New in version 0.10.

Parameters **name** – the optional name of the test, otherwise the function name will be used.

add_url_rule(*rule*, *endpoint=None*, *view_func=None*, ***options*)
Connects a URL rule. Works exactly like the `route()` decorator. If a view_func is provided it will be registered with the endpoint.

Basically this example:

```
@app.route('/')
def index():
    pass
```

Is equivalent to the following:

```
def index():
    pass
app.add_url_rule('/', 'index', index)
```

If the view_func is not provided you will need to connect the endpoint to a view function like so:

```
app.view_functions['index'] = index
```

Internally `route()` invokes *add_url_rule()* so if you want to customize the behavior via subclassing you only need to change this method.

For more information refer to url-route-registrations.

Changed in version 0.2: *view_func* parameter added.

Changed in version 0.6: `OPTIONS` is added automatically as method.

Parameters

- **rule** – the URL rule as string
- **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint
- **view_func** – the function to call when serving a request to the provided endpoint
- **options** – the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

after_request(*f*)

Register a function to be run after each request.

Your function must take one parameter, an instance of *response_class* and return a new response object or the same (see *process_response()*).

As of Flask 0.7 this function might not be executed at the end of the request in case an unhandled exception occurred.

app_context()

Binds the application only. For as long as the application is bound to the current context the `flask.current_app` points to that application. An application context is automatically created when a request context is pushed if necessary.

Example usage:

```
with app.app_context():
    ...
```

New in version 0.9.

app_ctx_globals_class

alias of `_AppCtxGlobals`

authenticate()

Sends a 401 response that enables basic auth

auto_find_instance_path()
Tries to locate the instance path if it was not provided to the constructor of the application class. It will basically calculate the path to a folder named `instance` next to your main file or the package.

New in version 0.8.

before_first_request(*f*)
Registers a function to be run before the first request to this instance of the application.

The function will be called without any arguments and its return value is ignored.

New in version 0.8.

before_request(*f*)
Registers a function to run before each request.

The function will be called without any arguments. If the function returns a non-None value, it's handled as if it was the return value from the view and further request handling is stopped.

config_class
alias of `Config`

context_processor(*f*)
Registers a template context processor function.

create_global_jinja_loader()
Creates the loader for the Jinja2 environment. Can be used to override just the loader and keeping the rest unchanged. It's discouraged to override this function. Instead one should override the *jinja_loader()* function instead.

The global loader dispatches between the loaders of the application and the individual blueprints.

New in version 0.7.

create_jinja_environment()
Creates the Jinja2 environment based on `jinja_options` and *select_jinja_autoescape()*. Since 0.7 this also adds the Jinja2 globals and filters after initialization. Override this function to customize the behavior.

New in version 0.5.

Changed in version 0.11: `Environment.auto_reload` set in accordance with `TEMPLATES_AUTO_RELOAD` configuration option.

create_url_adapter(*request*)
Creates a URL adapter for the given request. The URL adapter is created at a point where the request context is not yet set up so the request is passed explicitly.

New in version 0.6.

Changed in version 0.9: This can now also be called without a request object when the URL adapter is created for the application context.

dispatch_request()
Does the request dispatching. Matches the URL and returns the return value of the view or error handler. This does not have to be a response object. In order to convert the return value to a proper response object, call *make_response()*.

Changed in version 0.7: This no longer does the exception handling, this code was moved to the new *full_dispatch_request()*.

do_teardown_appcontext(*exc=<object object>*)
Called when an application context is popped. This works pretty much the same as *do_teardown_request()* but for the application context.

New in version 0.9.

do_teardown_request(*exc=<object object>*)
Called after the actual request dispatching and will call every as *teardown_request()* decorated function. This is not actually called by the `Flask` object itself but is always triggered when the request context is popped. That way we have a tighter control over certain resources under testing environments.

Changed in version 0.9: Added the *exc* argument. Previously this was always using the current exception information.

endpoint(*endpoint*)
A decorator to register a function as an endpoint. Example:

```
@app.endpoint('example.endpoint')
def example():
    return "example"
```

Parameters **endpoint** – the name of the endpoint

errorhandler(*code_or_exception*)
A decorator that is used to register a function given an error code. Example:

```
@app.errorhandler(404)
def page_not_found(error):
    return 'This page does not exist', 404
```

You can also register handlers for arbitrary exceptions:

```
@app.errorhandler(DatabaseError)
def special_exception_handler(error):
    return 'Database connection failed', 500
```

You can also register a function as error handler without using the *errorhandler()* decorator. The following example is equivalent to the one above:

```
def page_not_found(error):
    return 'This page does not exist', 404
app.error_handler_spec[None][404] = page_not_found
```

Setting error handlers via assignments to `error_handler_spec` however is discouraged as it requires fiddling with nested dictionaries and the special case for arbitrary exception types.

The first `None` refers to the active blueprint. If the error handler should be application wide `None` shall be used.

New in version 0.7: Use *register_error_handler()* instead of modifying `error_handler_spec` directly, for application wide error handlers.

New in version 0.7: One can now additionally also register custom exception types that do not necessarily have to be a subclass of the `HTTPException` class.

Parameters **code_or_exception** – the code as integer for the handler, or an arbitrary exception

finalize_request(*rv*, *from_error_handler=False*)
Given the return value from a view function this finalizes the request by converting it into a response and invoking the postprocessing functions. This is invoked for both normal request dispatching as well as error handlers.

Because this means that it might be called as a result of a failure a special safe mode is available which can be enabled with the *from_error_handler* flag. If enabled, failures in response processing will be logged and otherwise ignored.

Internal

full_dispatch_request()
Dispatches the request and on top of that performs request pre and postprocessing as well as HTTP exception catching and error handling.

New in version 0.7.

get_send_file_max_age(*filename*)
Provides default cache_timeout for the `send_file()` functions.

By default, this function returns `SEND_FILE_MAX_AGE_DEFAULT` from the configuration of `current_app`.

Static file functions such as `send_from_directory()` use this function, and `send_file()` calls this function on `current_app` when the given cache_timeout is `None`. If a cache_timeout is given in `send_file()`, that timeout is used; otherwise, this method is called.

This allows subclasses to change the behavior when sending files based on the filename. For example, to set the cache timeout for .js files to 60 seconds:

```
class MyFlask(flask.Flask):
    def get_send_file_max_age(self, name):
        if name.lower().endswith('.js'):
            return 60
        return flask.Flask.get_send_file_max_age(self, name)
```

New in version 0.9.

got_first_request
This attribute is set to `True` if the application started handling the first request.

New in version 0.8.

handle_exception(*e*)
Default exception handling that kicks in when an exception occurs that is not caught. In debug mode the exception will be re-raised immediately, otherwise it is logged and the handler for a 500 internal server error is used. If no such handler exists, a default 500 internal server error message is displayed.

New in version 0.3.

handle_http_exception(*e*)
Handles an HTTP exception. By default this will invoke the registered error handlers and fall back to returning the exception as response.

New in version 0.3.

handle_url_build_error(*error*, *endpoint*, *values*)
Handle `BuildError` on `url_for()`.

handle_user_exception(*e*)
This method is called whenever an exception occurs that should be handled. A special case are `HTTPExceptions` which are forwarded by this function to the *handle_http_exception()* method. This function will either return a response value or reraise the exception with the same traceback.

New in version 0.7.

has_static_folder
This is `True` if the package bound object's container has a folder for static files.

New in version 0.5.

init_jinja_globals()
Deprecated. Used to initialize the Jinja2 globals.

New in version 0.5.

Changed in version 0.7: This method is deprecated with 0.7. Override *create_jinja_environment()* instead.

inject_url_defaults(*endpoint*, *values*)
Injects the URL defaults for the given endpoint directly into the values dictionary passed. This is used internally and automatically called on URL building.

New in version 0.7.

iter_blueprints()
Iterates over all blueprints by the order they were registered.

New in version 0.11.

jinja_env
The Jinja2 environment used to load templates.

jinja_environment
alias of `Environment`

jinja_loader
The Jinja loader for this package bound object.

New in version 0.5.

json_decoder
alias of `JSONDecoder`

json_encoder
alias of `JSONEncoder`

log_exception(*exc_info*)
Logs an exception. This is called by *handle_exception()* if debugging is disabled and right before the handler is called. The default implementation logs the exception as error on the *logger*.

New in version 0.8.

logger
A `logging.Logger` object for this application. The default configuration is to log to stderr if the application is in debug mode. This logger can be used to (surprise) log messages. Here some examples:

```
app.logger.debug('A value for debugging')
app.logger.warning('A warning occurred (%d apples)', 42)
app.logger.error('An error occurred')
```

New in version 0.3.

make_config(*instance_relative=False*)
Used to create the config attribute by the Flask constructor. The *instance_relative* parameter is passed in from the constructor of Flask (there named *instance_relative_config*) and indicates if the config should be relative to the instance path or the root path of the application.

New in version 0.8.

make_default_options_response()
This method is called to create the default OPTIONS response. This can be changed through subclassing to change the default behavior of OPTIONS responses.

New in version 0.7.

make_null_session()
Creates a new instance of a missing session. Instead of overriding this method we recommend replacing the session_interface.

New in version 0.7.

make_response(*rv*)
Converts the return value from a view function to a real response object that is an instance of *response_class*.

The following types are allowed for *rv*:

response_class	the object is returned unchanged
str	a response object is created with the string as body
unicode	a response object is created with the string encoded to utf-8 as body
a WSGI function	the function is called as WSGI application and buffered as response object
tuple	A tuple in the form (response, status, headers) or (response, headers) where *response* is any of the types defined here, *status* is a string or an integer and *headers* is a list or a dictionary with header values.

Parameters **rv** – the return value from the view function

Changed in version 0.9: Previously a tuple was interpreted as the arguments for the response object.

make_shell_context()
Returns the shell context for an interactive shell for this application. This runs all the registered shell context processors.

New in version 0.11.

name
The name of the application. This is usually the import name with the difference that it's guessed from the run file if the import name is main. This name is used as a display name when Flask needs the name of the application. It can be set and overridden to change the value.

New in version 0.8.

open_instance_resource(*resource*, *mode='rb'*)
Opens a resource from the application's instance folder (instance_path). Otherwise works like *open_resource()*. Instance resources can also be opened for writing.

Parameters

- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is 'rb'.

open_resource(*resource*, *mode='rb'*)
Opens a resource from the application's resource folder. To see how this works, consider the following folder structure:

```
/myapplication.py
/schema.sql
/static
    /style.css
/templates
    /layout.html
    /index.html
```

If you want to open the `schema.sql` file you would do the following:

```
with app.open_resource('schema.sql') as f:
    contents = f.read()
    do_something_with(contents)
```

Parameters

- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is 'rb'.

open_session(*request*)
Creates or opens a new session. Default implementation stores all session data in a signed cookie. This requires that the `secret_key` is set. Instead of overriding this method we recommend replacing the `session_interface`.

Parameters **request** – an instance of *request_class*.

preprocess_request()
Called before the actual request dispatching and will call each *before_request()* decorated function, passing no arguments. If any of these functions returns a value, it's handled as if it was the return value from the view and further request handling is stopped.

This also triggers the *url_value_preprocessor()* functions before the actual *before_request()* functions are called.

preserve_context_on_exception
Returns the value of the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration value in case it's set, otherwise a sensible default is returned.

New in version 0.7.

process_response(*response*)
Can be overridden in order to modify the response object before it's sent to the WSGI server. By default this will call all the *after_request()* decorated functions.

Changed in version 0.5: As of Flask 0.5 the functions registered for after request execution are called in reverse order of registration.

Parameters **response** – a *response_class* object.

Returns a new response object or the same, has to be an instance of *response_class*.

propagate_exceptions
Returns the value of the `PROPAGATE_EXCEPTIONS` configuration value in case it's set, otherwise a sensible default is returned.

New in version 0.7.

raise_routing_exception(*request*)
Exceptions that are recording during routing are reraised with this method. During debug we are not

reraising redirect requests for non `GET`, `HEAD`, or `OPTIONS` requests and we're raising a different error instead to help debug situations.

Internal

register_blueprint(*blueprint*, ***options*)
Registers a blueprint on the application.

New in version 0.7.

register_error_handler(*code_or_exception*, *f*)
Alternative error attach function to the *errorhandler()* decorator that is more straightforward to use for non decorator usage.

New in version 0.7.

request_class
alias of `Request`

request_context(*environ*)
Creates a `RequestContext` from the given environment and binds it to the current context. This must be used in combination with the `with` statement because the request is only bound to the current context for the duration of the `with` block.

Example usage:

```
with app.request_context(environ):
    do_something_with(request)
```

The object returned can also be used without the `with` statement which is useful for working in the shell. The example above is doing exactly the same as this code:

```
ctx = app.request_context(environ)
ctx.push()
try:
    do_something_with(request)
finally:
    ctx.pop()
```

Changed in version 0.3: Added support for non-with statement usage and `with` statement is now passed the ctx object.

Parameters **environ** – a WSGI environment

response_class
alias of `Response`

run(*host=None*, *port=None*, *debug=None*, ***options*)
Runs the application on a local development server.

Do not use `run()` in a production setting. It is not intended to meet security and performance requirements for a production server. Instead, see deployment for WSGI server recommendations.

If the `debug` flag is set the server will automatically reload for code changes and show a debugger in case an exception happened.

If you want to run the application in debug mode, but disable the code execution on the interactive debugger, you can pass `use_evalex=False` as parameter. This will keep the debugger's traceback screen active, but disable code execution.

It is not recommended to use this function for development with automatic reloading as this is badly supported. Instead you should be using the **flask** command line script's `run` support.

Keep in Mind

Flask will suppress any server error with a generic error page unless it is in debug mode. As such to enable just the interactive debugger without the code reloading, you have to invoke *run()* with `debug=True` and `use_reloader=False`. Setting `use_debugger` to `True` without being in debug mode won't catch any exceptions because there won't be any to catch.

Changed in version 0.10: The default port is now picked from the `SERVER_NAME` variable.

Parameters

- **host** – the hostname to listen on. Set this to `'0.0.0.0'` to have the server available externally as well. Defaults to `'127.0.0.1'`.
- **port** – the port of the webserver. Defaults to `5000` or the port defined in the `SERVER_NAME` config variable if present.
- **debug** – if given, enable or disable debug mode. See `debug`.
- **options** – the options to be forwarded to the underlying Werkzeug server. See `werkzeug.serving.run_simple()` for more information.

save_session(*session*, *response*)
Saves the session if it needs updates. For the default implementation, check *open_session()*. Instead of overriding this method we recommend replacing the `session_interface`.

Parameters

- **session** – the session to be saved (a `SecureCookie` object)
- **response** – an instance of *response_class*

select_jinja_autoescape(*filename*)
Returns `True` if autoescaping should be active for the given template name. If no template name is given, returns *True*.

New in version 0.5.

send_static_file(*filename*)
Function used internally to send static files from the static folder to the browser.

New in version 0.5.

shell_context_processor(*f*)
Registers a shell context processor function.

New in version 0.11.

should_ignore_error(*error*)
This is called to figure out if an error should be ignored or not as far as the teardown system is concerned. If this function returns `True` then the teardown handlers will not be passed the error.

New in version 0.10.

static_folder
The absolute path to the configured static folder.

teardown_appcontext(*f*)
Registers a function to be called when the application context ends. These functions are typically also called when the request context is popped.

Example:

```
ctx = app.app_context()
ctx.push()
...
ctx.pop()
```

When `ctx.pop()` is executed in the above example, the teardown functions are called just before the app context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Since a request context typically also manages an application context it would also be called when you pop a request context.

When a teardown function was called because of an exception it will be passed an error object.

The return values of teardown functions are ignored.

New in version 0.9.

teardown_request(*f*)

Register a function to be run at the end of each request, regardless of whether there was an exception or not. These functions are executed when the request context is popped, even if not an actual request was performed.

Example:

```
ctx = app.test_request_context()
ctx.push()
...
ctx.pop()
```

When `ctx.pop()` is executed in the above example, the teardown functions are called just before the request context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Generally teardown functions must take every necessary step to avoid that they will fail. If they do execute code that might fail they will have to surround the execution of these code by try/except statements and log occurring errors.

When a teardown function was called because of a exception it will be passed an error object.

The return values of teardown functions are ignored.

Debug Note

In debug mode Flask will not tear down a request on an exception immediately. Instead it will keep it alive so that the interactive debugger can still access it. This behavior can be controlled by the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration variable.

---

template_filter(*name=None*)

A decorator that is used to register custom template filter. You can specify a name for the filter, otherwise the function name will be used. Example:

```
@app.template_filter()
def reverse(s):
    return s[::-1]
```

Parameters **name** – the optional name of the filter, otherwise the function name will be used.

template_global(*name=None*)

A decorator that is used to register a custom template global function. You can specify a name for the global function, otherwise the function name will be used. Example:

```
@app.template_global()
def double(n):
    return 2 * n
```

New in version 0.10.

Parameters **name** – the optional name of the global function, otherwise the function name will be used.

template_test(*name=None*)

A decorator that is used to register custom template test. You can specify a name for the test, otherwise the function name will be used. Example:

```
@app.template_test()
def is_prime(n):
    if n == 2:
        return True
    for i in range(2, int(math.ceil(math.sqrt(n))) + 1):
        if n % i == 0:
            return False
    return True
```

New in version 0.10.

Parameters **name** – the optional name of the test, otherwise the function name will be used.

test_client(*use_cookies=True*, ***kwargs*)

Creates a test client for this application. For information about unit testing head over to testing.

Note that if you are testing for assertions or exceptions in your application code, you must set `app.testing = True` in order for the exceptions to propagate to the test client. Otherwise, the exception will be handled by the application (not visible to the test client) and the only indication of an AssertionError or other exception will be a 500 status code response to the test client. See the `testing` attribute. For example:

```
app.testing = True
client = app.test_client()
```

The test client can be used in a `with` block to defer the closing down of the context until the end of the `with` block. This is useful if you want to access the context locals for testing:

```
with app.test_client() as c:
    rv = c.get('/?vodka=42')
    assert request.args['vodka'] == '42'
```

Additionally, you may pass optional keyword arguments that will then be passed to the application's `test_client_class` constructor. For example:

```
from flask.testing import FlaskClient

class CustomClient(FlaskClient):
    def __init__(self, *args, **kwargs):
        self._authentication = kwargs.pop("authentication")
        super(CustomClient,self).__init__( *args, **kwargs)
```

```
app.test_client_class = CustomClient
client = app.test_client(authentication='Basic ....')
```

See `FlaskClient` for more information.

Changed in version 0.4: added support for `with` block usage for the client.

New in version 0.7: The *use_cookies* parameter was added as well as the ability to override the client to be used by setting the `test_client_class` attribute.

Changed in version 0.11: Added ***kwargs* to support passing additional keyword arguments to the constructor of `test_client_class`.

test_request_context(**args*, ***kwargs*)
Creates a WSGI environment from the given values (see `werkzeug.test.EnvironBuilder` for more information, this function accepts the same arguments).

trap_http_exception(*e*)
Checks if an HTTP exception should be trapped or not. By default this will return `False` for all exceptions except for a bad request key error if `TRAP_BAD_REQUEST_ERRORS` is set to `True`. It also returns `True` if `TRAP_HTTP_EXCEPTIONS` is set to `True`.

This is called for all HTTP exceptions raised by a view function. If it returns `True` for any exception the error handler for this exception is not called and it shows up as regular exception in the traceback. This is helpful for debugging implicitly raised HTTP exceptions.

New in version 0.8.

try_trigger_before_first_request_functions()
Called before each request and will ensure that it triggers the `before_first_request_funcs` and only exactly once per application instance (which means process usually).

Internal

update_template_context(*context*)
Update the template context with some commonly used variables. This injects request, session, config and g into the template context as well as everything template context processors want to inject. Note that the as of Flask 0.6, the original values in the context will not be overridden if a context processor decides to return a value with the same key.

Parameters **context** – the context as a dictionary that is updated in place to add extra variables.

url_defaults(*f*)
Callback function for URL defaults for all view functions of the application. It's called with the endpoint and values and should update the values passed in place.

url_rule_class
alias of `Rule`

url_value_preprocessor(*f*)
Registers a function as URL value preprocessor for all view functions of the application. It's called before the view functions are called and can modify the url values provided.

wsgi_app(*environ*, *start_response*)
The actual WSGI application. This is not implemented in *__call__* so that middlewares can be applied without losing a reference to the class. So instead of doing this:

```
app = MyMiddleware(app)
```

It's a better idea to do this instead:

```
app.wsgi_app = MyMiddleware(app.wsgi_app)
```

Then you still have the original application object around and can continue to call methods on it.

Changed in version 0.7: The behavior of the before and after request callbacks was changed under error conditions and a new callback was added that will always execute at the end of the request, independent on if an error occurred or not. See callbacks-and-errors.

Parameters

- **environ** – a WSGI environment
- **start_response** – a callable accepting a status code, a list of headers and an optional exception context to start the response

5.2 cache

`cache` is an instance of *apicore.cache.Cache*

class apicore.cache.**Cache**

Cache for the application. To use by importing the instance :

exemple:

```
from apicore import cache

key = "my_data"
date = {"color": "orange", "flag": True}
cache.set(key, data)
print(cache.get(key))
```

Note: If `redis` URI is configured the cache is store in redis server, otherwise in-memory is used and all the cached data are lost after restarting the instance.

---

delete(*key*)

Parameters **key** (*str*) – the key referencing the data to remove

get(*key*)

Parameters **key** (*str*) – the key referencing the data

set(*key*, *value*, *expire=None*)

Parameters

- **key** (*str*) – the key referencing the data
- **value** – the data to store in cache
- **expire** (*integer*) – Expire at a given timestamp in seconde.

5.3 config

`config` is an instance of *apicore.config.Config*

class apicore.config.**Config**(*configFile='conf/config.yaml'*)
Manage configuration values. To use by importing the instance :

exemple:

```
from apicore import config

print(config.server_name)
```

isDefined(*name*)
Check whether configuration directive is defined or not

Parameters **string** (*str*) – Name of configuration directive

Return boolean True is directive is defined

load(*confFile='config.yaml'*)
Load config file from filesystem.

Parameters **string** (*str*) – Path to config file.

5.4 Exceptions

class apicore.**Http400Exception**(*description=None*, *verbose=False*)
Create a 400 *Bad Request* exception.

Parameters

- **infos** (*str*) – A message describing the error.
- **verbose** (*boolean*) – True to send infos in HTTP response.

class apicore.**Http401Exception**(*description=None*, *verbose=False*)
Create a 401 *Unauthorized* exception.

Parameters

- **infos** (*str*) – A message describing the error.
- **verbose** (*boolean*) – True to send infos in HTTP response.

class apicore.**Http402Exception**(*description=None*, *verbose=False*)

class apicore.**Http403Exception**(*description=None*, *verbose=False*)
Create a 403 *Forbidden* exception.

Parameters

- **infos** (*str*) – A message describing the error.
- **verbose** (*boolean*) – True to send infos in HTTP response.

class apicore.**Http404Exception**(*description=None*, *verbose=False*)
Create a 404 *Not Found* exception.

Parameters

- **infos** (*str*) – A message describing the error.
- **verbose** (*boolean*) – True to send infos in HTTP response.

class apicore.**Http406Exception**(*description=None*, *verbose=False*)
Create a 406 *Not Acceptable* exception.

Parameters

- **infos** (*str*) – A message describing the error.
- **verbose** (*boolean*) – True to send infos in HTTP response.

class apicore.**Http409Exception**(*description=None*, *verbose=False*)
Create a 409 *Conflict* exception.

Parameters

- **infos** (*str*) – A message describing the error.
- **verbose** (*boolean*) – True to send infos in HTTP response.

class apicore.**Http500Exception**(*description=None*, *verbose=False*)
Create a 500 *Internal Server Error* exception.

Parameters

- **infos** (*str*) – A message describing the error.
- **verbose** (*boolean*) – True to send infos in HTTP response.

class apicore.**Http501Exception**(*description=None*, *verbose=False*)
Create a 501 *Not Implemented* exception.

Parameters

- **infos** (*str*) – A message describing the error.
- **verbose** (*boolean*) – True to send infos in HTTP response.

5.5 Lang

class apicore.**Lang**

static best_match(*available_languages*, *default=None*)
Determine best language from list of available languages and from Accept-Language Header.

Parameters

- **available_languages** (*list*) – List of available languages to match with.
- **default** (*str*) – Language returned when no match found.

Return str The best matching language or None if no matching.

5.6 Logger

class apicore.**Logger**

static error(*string*)
Print error message to stderr.

Parameters **string** (*str*) – message to print.

static info(*string*)
Print information message to stdout.

Parameters **string** (*str*) – message to print.

Todo:

- i18n HTTP response messages.
- Add namespace for cache
- Configure using command line argument and environnement variables which override configuration file and making it optional.
- Use API Specification and json schemas to validate JSON data
- Access Control Policies engine
- MongoDB helpers
- Extensible notification system (using mail, Firebase, SMS, . . .)

Index

L

M

N

O

P

R

S

T

U

W