django-docker-helpers

Release 0.1.12

Mar 18, 2019

Contents

CHAPTER 1

Installation

At the command line:

```
pip install django-docker-helpers
```

CHAPTER 2

Config loader usage

To initialize config loader use this:

```
yml_conf = os.path.join(
    BASE_DIR, 'my_project', 'config',
    os.environ.get('DJANGO_CONFIG_FILE_NAME', 'without-docker.yml')
    )
os.environ.setdefault('YAMLPARSER__CONFIG', yml_conf)

configure = ConfigLoader.from_env(suppress_logs=True, silent=True)
```

Note: You can specify parsers with env variable `CONFIG__PARSERS`. It can be set to, i.e. `EnvironmentParser,RedisParser,YamlParser`. Also you can define config parsers this way:

```
loader = ConfigLoader.from_env(parser_modules=['EnvironmentParser'])
```

Read more about config loader: *from_env()*

Then use `configure` to read a setting from configs:

```
DEBUG = configure('debug', False)
```

All settings are case insensitive:

```
DEBUG = configure('DEBUG', False)
```

You can use nested variable paths (path parts delimiter is comma by default):

```
SECRET_KEY = configure('common.secret_key', 'secret')
```

Strict typing may be added with `coerce_type`:

```
DATABASES = {
    'default': {
        'ENGINE': configure('db.engine', 'django.db.backends.postgresql'),
        'HOST': configure('db.host', 'localhost'),
        'PORT': configure('db.port', 5432, coerce_type=int),

        'NAME': configure('db.name', 'marfa'),
        'USER': configure('db.user', 'marfa'),
        'PASSWORD': configure('db.password', 'marfa'),

        'CONN_MAX_AGE': configure('db.conn_max_age', 60, coerce_type=int)
    }
}
```

Note: You can create your own `coercer`. By default it's equal to `coerce_type`. Example: *`django_docker_helpers.utils.coerce_str_to_bool()`*

CHAPTER 3

Management helpers

3.1 Database

`django_docker_helpers.db.`**`ensure_caches_alive`**(*max_retries=100*, *retry_timeout=5*, *exit_on_failure=True*)

Checks every cache backend alias in `settings.CACHES` until it becomes available. After `max_retries` attempts to reach any backend are failed it returns `False`. If `exit_on_failure` is set it shuts down with `exit(1)`.

It sets the `django-docker-helpers:available-check` key for every cache backend to ensure it's receiving connections. If check is passed the key is deleted.

Parameters

- **`exit_on_failure`** (`bool`) – set to `True` if there's no sense to continue
- **`max_retries`** (*int*) – a number of attempts to reach cache backend, default is `100`
- **`retry_timeout`** (*int*) – a timeout in seconds between attempts, default is `5`

Return type `bool`

Returns `True` if all backends are available `False` if any backend check failed

`django_docker_helpers.db.`**`ensure_databases_alive`**(*max_retries=100*, *retry_timeout=5*, *exit_on_failure=True*)

Checks every database alias in `settings.DATABASES` until it becomes available. After `max_retries` attempts to reach any backend are failed it returns `False`. If `exit_on_failure` is set it shuts down with `exit(1)`.

For every database alias it tries to `SELECT 1`. If no errors raised it checks the next alias.

Parameters

- **`exit_on_failure`** (`bool`) – set to `True` if there's no sense to continue
- **`max_retries`** (*int*) – number of attempts to reach every database; default is `100`
- **`retry_timeout`** (*int*) – timeout in seconds between attempts

Return type bool

Returns True if all backends are available, False if any backend check failed

django_docker_helpers.db.**migrate**(**argv*)
Runs Django migrate command.

Return type bool

Returns always True

django_docker_helpers.db.**modeltranslation_sync_translation_fields**()
Runs modeltranslation's sync_translation_fields manage.py command: execute_from_command_line(['./manage.py', 'sync_translation_fields', '--noinput'])

Return type bool

Returns None if modeltranslation is not specified is INSTALLED_APPS, True if all synced.

3.2 Files

django_docker_helpers.files.**collect_static**()
Runs Django collectstatic command in silent mode.

Return type bool

Returns always True

3.3 Management

django_docker_helpers.management.**create_admin**(*user_config_path='CONFIG.superuser'*)
Creates a superuser from a specified dict/object bundle located at user_config_path. Skips if the specified object contains no email or no username. If a user with the specified username already exists and has no usable password it updates user's password with a specified one.

user_config_path can accept any path to a deep nested object, like dict of dicts, object of dicts of objects, and so on. Let's assume you have this weird config in your settings.py:

```
class MyConfigObject:
    my_var = {
        'user': {
            'username': 'user',
            'password': 'qwe',
            'email': 'no@example.com',
        }
    }
local_config = MyConfigObject()
```

To access the 'user' bundle you have to specify: local_config.my_var.user.

Parameters **user_config_path** (str) – dot-separated path to object or dict, default is 'CONFIG.superuser'

Return type bool

Returns True if user has been created, False otherwise

django_docker_helpers.management.**run_gunicorn**(*application*, *gunicorn_module_name='gunicorn_prod'*)

Runs gunicorn with a specified config.

Parameters

- **application** (WSGIHandler) – Django uwsgi application
- **gunicorn_module_name** (str) – gunicorn settings module name

Returns Application().run()

CHAPTER 4

Utils

`django_docker_helpers.utils._materialize_dict`(*bundle*, *separator='.'*)
Traverses and transforms a given dict `bundle` into tuples of `(key_path, value)`.

Parameters

- **`bundle`** (`dict`) – a dict to traverse
- **`separator`** (`str`) – build paths with a given separator

Return type `Generator[Tuple[str, Any], None, None]`

Returns a generator of tuples `(materialized_path, value)`

Example: >>> list(_materialize_dict({'test': {'path': 1}, 'key': 'val'}, '.')) >>> [('key', 'val'), ('test.path', 1)]

`django_docker_helpers.utils.coerce_str_to_bool`(*val*, *strict=False*)
Converts a given string `val` into a boolean.

Parameters

- **`val`** (`Union[str, int, None]`) – any string representation of boolean
- **`strict`** (`bool`) – raise `ValueError` if `val` does not look like a boolean-like object

Return type `bool`

Returns `True` if `val` is thruthy, `False` otherwise.

Raises **`ValueError`** – if `strict` specified and `val` got anything except `['', 0, 1, true, false, on, off, True, False]`

`django_docker_helpers.utils.dot_path`(*obj*, *path*, *default=None*, *separator='.'*)
Provides an access to elements of a mixed dict/object type by a delimiter-separated path.

```
class O1:
    my_dict = {'a': {'b': 1}}

class O2:
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        self.nested = O1()

class O3:
    final = O2()

o = O3()
assert utils.dot_path(o, 'final.nested.my_dict.a.b') == 1
```

```
True
```

Parameters

- **obj** (object) – object or dict
- **path** (str) – path to value
- **default** (Optional[Any]) – default value if chain resolve failed
- **separator** (str) – . by default

Returns value or default

django_docker_helpers.utils.**dotkey**(*obj*, *path*, *default=None*, *separator='.'*)

Provides an interface to traverse nested dict values by dot-separated paths. Wrapper for dpath.util.get.

Parameters

- **obj** (dict) – dict like {'some': {'value': 3}}
- **path** (str) – 'some.value'
- **separator** – '.' or '/' or whatever
- **default** – default for KeyError

Returns dict value or default value

django_docker_helpers.utils.**env_bool_flag**(*flag_name*, *strict=False*, *env=None*)

Converts an environment variable into a boolean. Empty string (presence in env) is treated as True.

Parameters

- **flag_name** (str) – an environment variable name
- **strict** (bool) – raise ValueError if a flag_name value connot be coerced into a boolean in obvious way
- **env** (Optional[Dict[str, str]]) – a dict with environment variables, default is os.environ

Return type bool

Returns True if flag_name is thruthy, False otherwise.

Raises **ValueError** – if strict specified and val got anything except ['', 0, 1, true, false, True, False]

django_docker_helpers.utils.**is_dockerized**(*flag_name='DOCKERIZED'*, *strict=False*)

Reads env DOCKERIZED variable as a boolean.

Parameters

- **flag_name** (str) – environment variable name

- **strict** (bool) – raise a ValueError if variable does not look like a normal boolean

Returns True if has truthy DOCKERIZED env, False otherwise

django_docker_helpers.utils.**is_production**(*flag_name='PRODUCTION'*, *strict=False*)
Reads env PRODUCTION variable as a boolean.

Parameters

- **flag_name** (str) – environment variable name
- **strict** (bool) – raise a ValueError if variable does not look like a normal boolean

Returns True if has truthy PRODUCTION env, False otherwise

django_docker_helpers.utils.**materialize_dict**(*bundle*, *separator='.'*)
Transforms a given bundle into a *sorted* list of tuples with materialized value paths and values: ('path.to.value', <value>). Output is ordered by depth: the deepest element first.

Parameters

- **bundle** (dict) – a dict to materialize
- **separator** (str) – build paths with a given separator

Return type List[Tuple[str, Any]]

Returns a depth descending and alphabetically ascending sorted list (-deep, asc), the longest first

```
sample = {
    'a': 1,
    'aa': 1,
    'b': {
        'c': 1,
        'b': 1,
        'a': 1,
        'aa': 1,
        'aaa': {
            'a': 1
        }
    }
}
materialize_dict(sample, '/')
[
    ('b/aaa/a', 1),
    ('b/a', 1),
    ('b/aa', 1),
    ('b/b', 1),
    ('b/c', 1),
    ('a', 1),
    ('aa', 1)
]
```

django_docker_helpers.utils.**mp_serialize_dict**(*bundle*, *separator='.'*, *serialize=<function dump>*, *value_prefix='::YAML::\n'*)
Transforms a given bundle into a *sorted* list of tuples with materialized value paths and values: ('path.to.value', b'<some>'). If the <some> value is not an instance of a basic type, it's serialized with serialize callback. If this value is an empty string, it's serialized anyway to enforce correct type if storage backend does not support saving empty strings.

Parameters

- **bundle** (dict) – a dict to materialize
- **separator** (str) – build paths with a given separator
- **serialize** (Optional[Callable]) – a method to serialize non-basic types, default is yaml.dump
- **value_prefix** (str) – a prefix for non-basic serialized types

Return type List[Tuple[str, bytes]]

Returns a list of tuples (mat_path, b'value')

```
sample = {
    'bool_flag': '',  # flag
    'unicode': '',
    'none_value': None,
    'debug': True,
    'mixed': ['ascii', '', 1, {'d': 1}, {'b': 2}],
    'nested': {
        'a': {
            'b': 2,
            'c': b'bytes',
        }
    }
}

result = mp_serialize_dict(sample, separator='/')
assert result == [
    ('nested/a/b', b'2'),
    ('nested/a/c', b'bytes'),
    ('bool_flag', b"::YAML::\n''\n"),
    ('debug', b'true'),
    ('mixed', b'::YAML::\n- ascii\n- '
              b'"\\u044E\\u043D\\u0438\\u043A\\u043E\\u0434"\n- 1\n- '
              b'{d: 1}\n- {b: 2}\n'),
    ('none_value', None),
    ('unicode', b'\xd0\xb2\xd0\xb0\xd1\x81\xd1\x8f')
]
```

django_docker_helpers.utils.**run_env_once**(*f*)

A decorator to prevent manage.py from running code twice for everything. (https://stackoverflow.com/questions/16546652/why-does-django-run-everything-twice)

Parameters **f** (Callable) – function or method to decorate

Return type Callable

Returns callable

django_docker_helpers.utils.**shred**(*key_name*, *value*, *field_names=('password', 'secret', 'pass', 'pwd', 'key', 'token', 'auth', 'cred')*)

Replaces sensitive data in value with * if key_name contains something that looks like a secret.

Parameters

- **field_names** (Iterable[str]) – a list of key names that can possibly contain sensitive data
- **key_name** (str) – a key name to check
- **value** (Any) – a value to mask

Return type Union[Any, str]

Returns an unchanged value if nothing to hide, '*' * len(str(value)) otherwise

django_docker_helpers.utils.**wf**(*raw_str*, *flush=True*, *prevent_completion_polluting=True*, *stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Writes a given raw_str into a stream. Ignores output if prevent_completion_polluting is set and there's no extra sys.argv arguments present (a bash completion issue).

Parameters

- **raw_str** (str) – a raw string to print
- **flush** (bool) – execute flush()
- **prevent_completion_polluting** (bool) – don't write anything if len(sys.argv) <= 1
- **stream** (Textio) – sys.stdout by default

Returns None

CHAPTER 5

Config

5.1 Loader

class django_docker_helpers.config.**ConfigLoader**(*parsers*, *silent=False*, *suppress_logs=False*, *keep_read_records_max=1024*)

Bases: `object`

- provides a single interface to read from specified config parsers in order they present;
- tracks accessed from parsers options;
- prints config options access log in pretty-print way.

Example

```
env = {
    'PROJECT__DEBUG': 'false'
}
parsers = [
    EnvironmentParser(scope='project', env=env),
    RedisParser('my/conf/service/config.yml', host=REDIS_HOST, port=REDIS_PORT),
    YamlParser(config='./tests/data/config.yml', scope='project'),
]
configure = ConfigLoader(parsers=parsers)

DEBUG = configure('debug')  # 'false'
DEBUG = configure('debug', coerce_type=bool)  # False
```

Initialization:

- takes a list of initialized parsers;

- it's supposed to use ONLY unique parsers for `parsers` argument (or you are going to get the same initial arguments for all parsers of the same type in *from_env()*);
- the parsers's order does matter.

Parameters

- **parsers** (`List[`*BaseParser*`]`) – a list of initialized parsers
- **silent** (`bool`) – don't raise exceptions if any read attempt failed
- **suppress_logs** (`bool`) – don't display any exception warnings on screen
- **keep_read_records_max** (`int`) – max capacity queue length

__call__(*variable_path*, *default=None*, *coerce_type=None*, *coercer=None*, *required=False*, ***kwargs*)

A useful shortcut for method *get()*

format_config_read_queue(*use_color=False*, *max_col_width=50*)

Prepares a string with pretty printed config read queue.

Parameters

- **use_color** (`bool`) – use terminal colors
- **max_col_width** (`int`) – limit column width, `50` by default

Return type `str`

Returns

static from_env(*parser_modules=('django_docker_helpers.config.backends.EnvironmentParser', 'django_docker_helpers.config.backends.MPTRedisParser', 'django_docker_helpers.config.backends.MPTConsulParser', 'django_docker_helpers.config.backends.RedisParser', 'django_docker_helpers.config.backends.ConsulParser', 'django_docker_helpers.config.backends.YamlParser')*, *env=None*, *silent=False*, *suppress_logs=False*, *extra=None*)

Creates an instance of *ConfigLoader* with parsers initialized from environment variables.

By default it tries to initialize all bundled parsers. Parsers may be customized with `parser_modules` argument or `CONFIG__PARSERS` environment variable. Environment variable has a priority over the method argument.

Parameters

- **parser_modules** (`Union[List[str], Tuple[str], None]`) – a list of dot-separated module paths
- **env** (`Optional[Dict[str, str]]`) – a dict with environment variables, default is `os.environ`
- **silent** (`bool`) – passed to *ConfigLoader*
- **suppress_logs** (`bool`) – passed to *ConfigLoader*
- **extra** (`Optional[dict]`) – pass extra arguments to *every* parser

Return type *ConfigLoader*

Returns an instance of *ConfigLoader*

Example:

```
env = {
    'CONFIG__PARSERS': 'EnvironmentParser,RedisParser,YamlParser',
    'ENVIRONMENTPARSER__SCOPE': 'nested',
    'YAMLPARSER__CONFIG': './tests/data/config.yml',
    'REDISPARSER__HOST': 'wtf.test',
    'NESTED__VARIABLE': 'i_am_here',
}

loader = ConfigLoader.from_env(env=env)
assert [type(p) for p in loader.parsers] == [EnvironmentParser, RedisParser,␣
↪YamlParser]
assert loader.get('variable') == 'i_am_here', 'Ensure env copied from␣
↪ConfigLoader'

loader = ConfigLoader.from_env(parser_modules=['EnvironmentParser'], env={})
```

get (*variable_path*, *default=None*, *coerce_type=None*, *coercer=None*, *required=False*, ***kwargs*)
Tries to read a `variable_path` from each of the passed parsers. It stops if read was successful and returns a retrieved value. If none of the parsers contain a value for the specified path it returns `default`.

Parameters

- **variable_path** (`str`) – a path to variable in config
- **default** (`Optional[Any]`) – a default value if `variable_path` is not present anywhere
- **coerce_type** (`Optional[Type[+CT_co]]`) – cast a result to a specified type
- **coercer** (`Optional[Callable]`) – perform the type casting with specified callback
- **required** (`bool`) – raise `RequiredValueIsEmpty` if no `default` and no result
- **kwargs** – additional options to all parsers

Returns **the first successfully read** value from the list of parser instances or `default`

Raises **config.exceptions.RequiredValueIsEmpty** – if nothing is read,‘‘required‘‘ flag is set, and there’s no `default` specified

static import_parsers (*parser_modules*)
Resolves and imports all modules specified in `parser_modules`. Short names from the local scope are supported (the scope is `django_docker_helpers.config.backends`).

Parameters **parser_modules** (`Iterable[str]`) – a list of dot-separated module paths

Return type `Generator[Type[`*BaseParser*`], None, None]`

Returns a generator of [probably] *BaseParser*

Example:

```
parsers = list(ConfigLoader.import_parsers([
    'EnvironmentParser',
    'django_docker_helpers.config.backends.YamlParser'
]))
assert parsers == [EnvironmentParser, YamlParser]
```

static load_parser_options_from_env (*parser_class*, *env=None*)
Extracts arguments from `parser_class.__init__` and populates them from environment variables.

Uses `__init__` argument type annotations for correct type casting.

Note: Environment variables should be prefixed with `<UPPERCASEPARSERCLASSNAME>__`.

Parameters

- **parser_class** (Type[*BaseParser*]) – a subclass of *BaseParser*
- **env** (Optional[Dict[str, str]]) – a dict with environment variables, default is os.environ

Return type Dict[str, Any]

Returns parser's __init__ arguments dict mapping

Example:

```
env = {
    'REDISPARSER__ENDPOINT': 'go.deep',
    'REDISPARSER__HOST': 'my-host',
    'REDISPARSER__PORT': '66',
}

res = ConfigLoader.load_parser_options_from_env(RedisParser, env)
assert res == {'endpoint': 'go.deep', 'host': 'my-host', 'port': 66}
```

print_config_read_queue(*use_color=False*, *max_col_width=50*)

Prints all read (in call order) options.

Parameters

- **max_col_width** (int) – limit column width, 50 by default
- **use_color** (bool) – use terminal colors

Returns nothing

class django_docker_helpers.config.**ConfigReadItem**(*variable_path*, *value*, *type*, *is_default*, *parser_name*)

Bases: tuple

Create new instance of ConfigReadItem(variable_path, value, type, is_default, parser_name)

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *variable_path*, *value*, *type*, *is_default*, *parser_name*)

Create new instance of ConfigReadItem(variable_path, value, type, is_default, parser_name)

__repr__()

Return a nicely formatted representation string

_asdict()

Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)

Make a new ConfigReadItem object from a sequence or iterable

_replace(***kwds*)

Return a new ConfigReadItem object replacing specified fields with new values

is_default

Alias for field number 3

parser_name
Alias for field number 4

type
Alias for field number 2

value
Alias for field number 1

variable_path
Alias for field number 0

5.2 Base Parser

class django_docker_helpers.config.backends.base.**BaseParser**(*scope=None*, *config=None*, *nested_delimiter='__'*, *path_separator='.'*, *env=None*)

Bases: `object`

Base class to inherit from in custom parsers.

All `__init__` arguments **MUST** be optional if you need *`from_env()`* automatic parser initializer (it initializes parsers like `parser_class(**parser_options)`).

Since *`ConfigLoader`* can initialize parsers from environment variables it's **recommended** to annotate argument types to provide a correct auto typecast.

`BaseParser` creates a `logger` with name `__class__.__name__`.

`BaseParser` implements generic copying of following arguments without any backend-specific logic inside.

Parameters

- **scope** (`Optional[str]`) – a global prefix to all underlying values
- **config** (`Optional[str]`) – optional config
- **nested_delimiter** (`str`) – optional delimiter for environment backend
- **path_separator** (`str`) – specifies which character separates nested variables, default is `'.'`
- **env** (`Optional[Dict[str, str]]`) – a dict with environment variables, default is `os.environ`

client
Helper property to lazy initialize and cache client. Runs *`get_client()`*.

Returns an instance of backend-specific client

static coerce(*val*, *coerce_type=None*, *coercer=None*)
Casts a type of `val` to `coerce_type` with `coercer`.

If `coerce_type` is bool and no `coercer` specified it uses *`coerce_str_to_bool()`* by default.

Parameters

- **val** (`Any`) – a value of any type
- **coerce_type** (`Optional[Type[+CT_co]]`) – any type

- **coercer** (Optional[Callable]) – provide a callback that takes val and returns a value with desired type

Return type Any

Returns type casted value

get(*variable_path*, *default=None*, *coerce_type=None*, *coercer=None*, ***kwargs*)
Inherited method should take all specified arguments.

Parameters

- **variable_path** (str) – a delimiter-separated path to a nested value
- **default** (Optional[Any]) – default value if there's no object by specified path
- **coerce_type** (Optional[Type[+CT_co]]) – cast a type of a value to a specified one
- **coercer** (Optional[Callable]) – perform a type casting with specified callback
- **kwargs** – additional arguments inherited parser may need

Returns value or default

get_client()
If your backend needs a client, inherit this method and use *client()* shortcut.

Returns an instance of backend-specific client

5.3 Environment Parser

class django_docker_helpers.config.backends.environment_parser.**EnvironmentParser**(*scope=None*, *config=None*, *nested_delimiter=None*, *path_separator=None*, *env=None*)

Bases: *django_docker_helpers.config.backends.base.BaseParser*

Provides a simple interface to read config options from environment variables.

Example:

```
from json import loads as json_load
from yaml import load as yaml_load

env = {
    'MY__VARIABLE': '33',
    'MY__NESTED__YAML__LIST__VARIABLE': '[33, 42]',
    'MY__NESTED__JSON__DICT__VARIABLE': '{"obj": true}',
}

parser = EnvironmentParser(env=env)
assert p.get('my.variable') == '33'

assert p.get('my.nested.yaml.list.variable',
             coerce_type=list, coercer=yaml_load) == [33, 42]
assert p.get('my.nested.json.dict.variable',
             coerce_type=dict, coercer=json_load) == {'obj': True}
```

(continues on next page)

(continued from previous page)

```
parser = EnvironmentParser(env=env, scope='my.nested')
assert parser.get('yaml.list.variable',
                  coerce_type=list, coercer=yaml_load) == [33, 42]
```

Parameters

- **scope** (Optional[str]) – a global namespace-like variable prefix
- **config** (Optional[str]) – not used
- **nested_delimiter** (str) – replace path_separator with an appropriate environment variable delimiter, default is ___
- **path_separator** (str) – specifies which character separates nested variables, default is '.'
- **env** (Optional[Dict[str, str]]) – a dict with environment variables, default is os.environ

__str__()
Return str(self).

get(*variable_path*, *default=None*, *coerce_type=None*, *coercer=None*, ***kwargs*)
Reads a value of variable_path from environment.

If coerce_type is bool and no coercer specified, coerces forced to be *coerce_str_to_bool()*

Parameters

- **variable_path** (str) – a delimiter-separated path to a nested value
- **default** (Optional[Any]) – default value if there's no object by specified path
- **coerce_type** (Optional[Type[+CT_co]]) – cast a type of a value to a specified one
- **coercer** (Optional[Callable]) – perform a type casting with specified callback
- **kwargs** – additional arguments inherited parser may need

Returns value or default

get_client()
If your backend needs a client, inherit this method and use *client()* shortcut.

Returns an instance of backend-specific client

5.4 Yaml Parser

class django_docker_helpers.config.backends.yaml_parser.**YamlParser**(*config=None*, *path_separator='.'*, *scope=None*)

Bases: *django_docker_helpers.config.backends.base.BaseParser*

Provides a simple interface to read config options from Yaml.

Example:

```
p = YamlParser('./tests/data/config.yml', scope='development')
assert p.get('up.down.above') == [1, 2, 3]
```

Parameters

- **config** (Union[str, Textio, None]) – a path to config file, or *TextIO* object
- **path_separator** (str) – specifies which character separates nested variables, default is '.'
- **scope** (Optional[str]) – a global namespace-like variable prefix

Raises **ValueError** – if no config specified

__str__()
Return str(self).

get(*variable_path*, *default=None*, *coerce_type=None*, *coercer=None*, ***kwargs*)
Inherited method should take all specified arguments.

Parameters

- **variable_path** (str) – a delimiter-separated path to a nested value
- **default** (Optional[Any]) – default value if there's no object by specified path
- **coerce_type** (Optional[Type[+CT_co]]) – cast a type of a value to a specified one
- **coercer** (Optional[Callable]) – perform a type casting with specified callback
- **kwargs** – additional arguments inherited parser may need

Returns value or default

get_client()
If your backend needs a client, inherit this method and use *client()* shortcut.

Returns an instance of backend-specific client

5.5 Consul Parser

class django_docker_helpers.config.backends.consul_parser.**ConsulParser**(*endpoint='service'*, *host='127.0.0.1'*, *port=8500*, *scheme='http'*, *verify=True*, *cert=None*, *kv_get_opts=None*, *path_separator='.'*, *inner_parser_class=<class 'django_docker_helpers.conf*

Bases: *django_docker_helpers.config.backends.base.BaseParser*

Reads a whole config bundle from a consul kv key and provides the unified interface to access config options.

It assumes that config in your storage can be parsed with any simple parser, like YamlParser.

Compared to, e.g. *EnvironmentParser* it does not have scope support by design, since endpoint is a good enough scope by itself.

Example:

```
parser = ConsulParser('my/server/config.yml', host=CONSUL_HOST, port=CONSUL_PORT)
parser.get('nested.a.b', coerce_type=int)
```

Parameters

- **endpoint** (str) – specifies a key in consul kv storage, e.g. 'services/mailer/config.yml'
- **host** (str) – consul host, default is '127.0.0.1'
- **port** (int) – consul port, default is 8500
- **scheme** (str) – consul scheme, default is 'http'
- **verify** (bool) – verify certs, default is True
- **cert** (Optional[str]) – path to certificate bundle
- **kv_get_opts** (Optional[Dict[~KT, ~VT]]) – read config bundle with optional arguments to client.kv.get()
- **path_separator** (str) – specifies which character separates nested variables, default is '.'
- **inner_parser_class** (Optional[Type[*BaseParser*]]) – use the specified parser to read config from endpoint key

__str__()
Return str(self).

get(*variable_path*, *default=None*, *coerce_type=None*, *coercer=None*, ***kwargs*)
Reads a value of variable_path from consul kv storage.

Parameters

- **variable_path** (str) – a delimiter-separated path to a nested value
- **default** (Optional[Any]) – default value if there's no object by specified path
- **coerce_type** (Optional[Type[+CT_co]]) – cast a type of a value to a specified one
- **coercer** (Optional[Callable]) – perform a type casting with specified callback
- **kwargs** – additional arguments inherited parser may need

Returns value or default

Raises

- **config.exceptions.KVStorageKeyDoestNotExist** – if specified endpoint does not exists
- **config.exceptions.KVStorageValueIsEmpty** – if specified endpoint does not contain a config

get_client()
If your backend needs a client, inherit this method and use *client()* shortcut.

Returns an instance of backend-specific client

inner_parser
Prepares inner config parser for config stored at endpoint.

Return type *BaseParser*

Returns an instance of *BaseParser*

Raises

- **config.exceptions.KVStorageKeyDoestNotExist** – if specified `endpoint` does not exists
- **config.exceptions.KVStorageValueIsEmpty** – if specified `endpoint` does not contain a config

5.6 Redis Parser

class django_docker_helpers.config.backends.redis_parser.**RedisParser**(*endpoint='service'*, *host='127.0.0.1'*, *port=6379*, *db=0*, *path_separator='.'*, *inner_parser_class=<class 'django_docker_helpers.config.b*, ***redis_options*)

Bases: *django_docker_helpers.config.backends.base.BaseParser*

Reads a whole config bundle from a redis key and provides the unified interface to access config options.

It assumes that config in your storage can be parsed with any simple parser, like `YamlParser`.

Compared to, e.g. *EnvironmentParser* it does not have scope support by design, since `endpoint` is a good enough scope by itself.

Example:

```
parser = RedisParser('my/server/config.yml', host=REDIS_HOST, port=REDIS_PORT)
parser.get('nested.a.b', coerce_type=int)
```

Parameters

- **endpoint** (`str`) – specifies a redis key with serialized config, e.g. `'services/mailer/config.yml'`
- **host** (`str`) – redis host, default is `'127.0.0.1'`
- **port** (`int`) – redis port, default id `6379`
- **db** (`int`) – redis database, default is `0`
- **path_separator** (`str`) – specifies which character separates nested variables, default is `'.'`
- **inner_parser_class** (`Optional[Type[BaseParser]]`) – use the specified parser to read config from `endpoint` key
- **redis_options** – additional options for `redis.Redis` client

__str__()
Return str(self).

get(*variable_path*, *default=None*, *coerce_type=None*, *coercer=None*, ***kwargs*)
Reads a value of `variable_path` from redis storage.

Parameters

- **variable_path** (str) – a delimiter-separated path to a nested value
- **default** (Optional[Any]) – default value if there's no object by specified path
- **coerce_type** (Optional[Type[+CT_co]]) – cast a type of a value to a specified one
- **coercer** (Optional[Callable]) – perform a type casting with specified callback
- **kwargs** – additional arguments inherited parser may need

Returns value or default

Raises **config.exceptions.KVStorageValueIsEmpty** – if specified endpoint does not contain a config

get_client()

If your backend needs a client, inherit this method and use *client()* shortcut.

Returns an instance of backend-specific client

inner_parser

Prepares inner config parser for config stored at endpoint.

Return type *BaseParser*

Returns an instance of *BaseParser*

Raises **config.exceptions.KVStorageValueIsEmpty** – if specified endpoint does not contain a config

5.7 MPT Consul Parser

class django_docker_helpers.config.backends.mpt_consul_parser.**MPTConsulParser**(*scope=None*, *host='127.0.0.1'*, *port=8500*, *scheme='http'*, *verify=True*, *cert=None*, *path_separator='.'*, *consul_path_separato*
object_deserialize_p
object_deserialize=<
*default_yaml_object_*

Bases: *django_docker_helpers.config.backends.base.BaseParser*

Materialized Path Tree Consul Parser.

Compared to, e.g. *ConsulParser* it does not load a whole config file from a single key, but reads every config option from a corresponding variable path.

Example:

```
parser = MPTConsulParser(host=CONSUL_HOST, port=CONSUL_PORT, path_separator='.')
parser.get('nested.a.b')
```

If you want to store your config with separated key paths take *mp_serialize_dict()* helper to materialize your dict.

Parameters

- **scope** (Optional[str]) – a global namespace-like variable prefix
- **host** (str) – consul host, default is '127.0.0.1'
- **port** (int) – consul port, default is 8500
- **scheme** (str) – consul scheme, default is 'http'
- **verify** (bool) – verify certs, default is True
- **cert** – path to certificate bundle
- **path_separator** (str) – specifies which character separates nested variables, default is '.'
- **consul_path_separator** (str) – specifies which character separates nested variables in consul kv storage, default is '/'
- **object_deserialize_prefix** (str) – if object has a specified prefix, it's deserialized with object_deserialize
- **object_deserialize** (Optional[Callable]) – deserializer for complex variables

__str__()
Return str(self).

get(*variable_path*, *default=None*, *coerce_type=None*, *coercer=None*, ***kwargs*)

Parameters

- **variable_path** (str) – a delimiter-separated path to a nested value
- **default** (Optional[Any]) – default value if there's no object by specified path
- **coerce_type** (Optional[Type[+CT_co]]) – cast a type of a value to a specified one
- **coercer** (Optional[Callable]) – perform a type casting with specified callback
- **kwargs** – additional arguments inherited parser may need

Returns value or default

get_client()
If your backend needs a client, inherit this method and use *client()* shortcut.

Returns an instance of backend-specific client

5.8 MPT Redis Parser

class django_docker_helpers.config.backends.mpt_redis_parser.**MPTRedisParser**(*scope=None*, *host='127.0.0.1'*, *port=6379*, *db=0*, *path_separator='.'*, *key_prefix=''*, *object_deserialize_prefi*, *object_deserialize=<fu*, *default_yaml_object_de*, ***redis_options*)

Bases: *django_docker_helpers.config.backends.base.BaseParser*

Materialized Path Tree Redis Parser.

Compared to, e.g. *RedisParser* it does not load a whole config file from a single key, but reads every config option from a corresponding variable path.

Example:

```
parser = MPTRedisParser(host=REDIS_HOST, port=REDIS_PORT)
parser.get('nested.a.b')
parser.get('debug')
```

If you want to store your config with separated key paths take *mp_serialize_dict()* helper to materialize your dict.

Parameters

- **scope** (Optional[str]) – a global namespace-like variable prefix
- **host** (str) – redis host, default is '127.0.0.1'
- **port** (int) – redis port, default id 6379
- **db** (int) – redis database, default is 0
- **path_separator** (str) – specifies which character separates nested variables, default is '.'
- **key_prefix** (str) – prefix all keys with specified one
- **object_deserialize_prefix** (str) – if object has a specified prefix, it's deserialized with object_deserialize
- **object_deserialize** (Optional[Callable]) – deserializer for complex variables
- **redis_options** – additional options for redis.Redis client

__str__()
Return str(self).

get(*variable_path*, *default=None*, *coerce_type=None*, *coercer=None*, ***kwargs*)

Parameters

- **variable_path** (str) – a delimiter-separated path to a nested value

- **default** (Optional[Any]) – default value if there's no object by specified path
- **coerce_type** (Optional[Type[+CT_co]]) – cast a type of a value to a specified one
- **coercer** (Optional[Callable]) – perform a type casting with specified callback
- **kwargs** – additional arguments inherited parser may need

Returns value or default

get_client()

If your backend needs a client, inherit this method and use *client()* shortcut.

Returns an instance of backend-specific client

CHAPTER 6

Reference

6.1 django-docker-helpers

CHAPTER 7

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

7.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.2 Documentation improvements

django-docker-helpers could always use more documentation, whether as part of the official django-docker-helpers docs, in docstrings, or even on the web in blog posts, articles, and such.

7.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/night-crawler/django-docker-helpers/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

7.4 Development

To set up *django-docker-helpers* for local development:

1. Fork django-docker-helpers (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/django-docker-helpers.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

7.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

7.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 8

Authors

- Igor Kalishevsky - https://github.com/night-crawler/

CHAPTER 9

Changelog

9.1 0.1.12 (2018-01-31)

- First release on PyPI.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

B

C

D

E

W

Y