zask Documentation

Release 1.9.4

J5

August 24, 2016

Contents

Zask is a framework to work with ZeroRPC. Zask is inspired by Flask, you can consider zask as Flask without `WSGI`, `Jinja2` and `Router` but with ZeroRPC and SqlAlchemy.

CHAPTER 1

Why not Flask

Flask is designed for web application, but Zask is for internal service application.

1.1 Installation

1.1.1 Libzmq

Ensure libzmq installed.

1.1.2 Virtualenv and VE

Virtualenv now is considered as best practice of developping Python project:

```
$ virtualenv .virutualenv
$ . .virtualenv/bin/activate
```

VE is a perfect friend to work with virtualenv. VE will activate any virtualenv in the current path. You can run any python command with a `ve` prefix:

```
$ ve python setup.py develop
$ ve pip install foo
$ ve pip freeze
```

1.1.3 Install from setuptools

Add dependence in your `setup.py`:

```
setup(
    # jump other params
    ...

    install_requires = [
        'zask==1.0.dev'
    ],
    dependency_links=[
        "git+ssh://git@github.com/j-5/zask.git@0.1.dev#egg=zask-1.0.dev"
    ]
)
```

You can use zask in your project after run `ve python setup.py develop`:

```
import zask
```

1.1.4 Build from source code

If you want to dive into zask, run:

```
$ git clone http://github.com/j-5/zask.git
$ cd zask
$ virtualenv .virtualenv
$ ve python setup.py develop
```

1.2 Tutorial

1.2.1 Quickstart

Zask is easy to use:

```
from zask import Zask
from zask.ext.zerorpc import ZeroRPC, access_log

app = Zask(__name__)
rpc = ZeroRPC(app, middlewares=None)

@access_log
class MySrv(rpc.Server):

    def foo(self):
        return "bar"

server = MySrv()
server.bind("tcp://0.0.0.0:8081")
server.run()
```

1.2.2 Debug Mode

If `config['DEBUG']` is `True`, then all the loggers will sent messages to `stdout`, otherwise, if `config['DEUBG']` is `False`, messages will be rewrited to the files. For now, there are two loggers in the zask, `zask logger` and `access logger`.

1.2.3 Zask Logger

Basic Usage:

```
from zask import Zask

app = Zask(__name__)

app.logger.info("info")
app.logger.debug("debug")
app.logger.error("error")
```

```
try:
    raise Exception("exception")
except:
    app.logger.exception("")
```

1.2.4 Access Logger

Generally speaking, you wont use this logger directly. When you use the `access_log` decorator or `ACCESS_LOG_MIDDLEWARE`, it will working automatically.

1.2.5 Zask-SQLAlchemy

If you are not familiar with Flask-SQLAlchemy, the extention improves SqlAlchemy in several ways.

1. Bind with specific framework to make it easy to use
2. Contains all `sqlalchemy` and `sqlalchemy.orm` functions in one object
3. Add an amazing python descriptor to the object, which make it super cool to query data
4. Dynamic database bind depend on multiple bind configures.

As the same reason of why not just use Flask, we can't use Flask-SQLAlchemy directly.

Differents between Flask-SQLAlchemy:

1. Default `scopefunc` is `gevent.getcurrent`
2. No signal session
3. No query record
4. No pagination and HTTP headers, e.g. `get_or_404`
5. No difference between app bound and not bound

But the usage of Zask-SQLAlchemy is quite similar with Flask-SQLAlchemy. So the following of this section is just a copy from Flask-SQLAlchemy.

A Minimal Application

The `SQLAlchemy` provides a class called Model that is a declarative base which can be used to models:

```
from zask import Zask
from zask.ext.sqlalchemy import SQLAlchemy

app = Zask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)

    def __init__(self, username, email):
        self.username = username
        self.email = email
```

```
    def __repr__(self):
        return '<User %r>' % self.username
```

To create the initial database, just import the *db* object from an interactive Python shell and run the `SQLAlchemy.create_all()` method to create the tables and database:

```
>>> from yourapplication import db
>>> db.create_all()
```

Boom, and there is your database. Now to create some users:

```
>>> from yourapplication import User
>>> admin = User('admin', 'admin@example.com')
>>> guest = User('guest', 'guest@example.com')
```

But they are not yet in the database, so let's make sure they are:

```
>>> db.session.add(admin)
>>> db.session.add(guest)
>>> db.session.commit()
```

Accessing the data in database is easy as a pie:

```
>>> users = User.query.all()
[<User u'admin'>, <User u'guest'>]
>>> admin = User.query.filter_by(username='admin').first()
<User u'admin'>
```

Scope Session with ZeroRPC

Session in Zask is separated by greenlet. There is a middleware for clear session automatically:

```
from zask import Zask
from zask.ext.sqlalchemy import SQLAlchemy, SessionMiddleware
from zask.ext.zerorpc import ZeroRPC

app = Zask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)
rpc = ZeroRPC(app)
rpc.register_middleware(SessionMiddleware(db))
```

Simple Relationships

SQLAlchemy connects to relational databases and what relational databases are really good at are relations. As such, we shall have an example of an application that uses two tables that have a relationship to each other:

```
from datetime import datetime


class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80))
    body = db.Column(db.Text)
    pub_date = db.Column(db.DateTime)

    category_id = db.Column(db.Integer, db.ForeignKey('category.id'))
```

```
    category = db.relationship('Category',
        backref=db.backref('posts', lazy='dynamic'))

    def __init__(self, title, body, category, pub_date=None):
        self.title = title
        self.body = body
        if pub_date is None:
            pub_date = datetime.utcnow()
        self.pub_date = pub_date
        self.category = category

    def __repr__(self):
        return '<Post %r>' % self.title


class Category(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50))

    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return '<Category %r>' % self.name
```

First let's create some objects:

```
>>> py = Category('Python')
>>> p = Post('Hello Python!', 'Python is pretty cool', py)
>>> db.session.add(py)
>>> db.session.add(p)
```

Now because we declared *posts* as dynamic relationship in the backref it shows up as query:

```
>>> py.posts
<sqlalchemy.orm.dynamic.AppenderBaseQuery object at 0x1027d37d0>
```

It behaves like a regular query object so we can ask it for all posts that are associated with our test "Python" category:

```
>>> py.posts.all()
[<Post 'Hello Python!'>]
```

Multiple Databases with Binds

Zask-SQLAlchemy can easily connect to multiple databases. To achieve that it preconfigures SQLAlchemy to support multiple "binds".

What are binds? In SQLAlchemy speak a bind is something that can execute SQL statements and is usually a connection or engine. In Flask-SQLAlchemy binds are always engines that are created for you automatically behind the scenes. Each of these engines is then associated with a short key (the bind key). This key is then used at model declaration time to assocate a model with a specific engine.

If no bind key is specified for a model the default connection is used instead (as configured by `SQLALCHEMY_DATABASE_URI`).

Example Configuration

The following configuration declares three database connections. The special default one as well as two others named *users* (for the users) and *appmeta* (which connects to a sqlite database for read only access to some data the application provides internally):

```
SQLALCHEMY_DATABASE_URI = 'postgres://localhost/main'
SQLALCHEMY_BINDS = {
    'users':        'mysqldb://localhost/users',
    'appmeta':      'sqlite:////path/to/appmeta.db'
}
```

Creating and Dropping Tables

The `SQLAlchemy.create_all()` and `SQLAlchemy.drop_all()` methods by default operate on all declared binds, including the default one. This behavior can be customized by providing the *bind* parameter. It takes either a single bind name, `'__all__'` to refer to all binds or a list of binds. The default bind (`SQLALCHEMY_DATABASE_URI`) is named *None*:

```
>>> db.create_all()
>>> db.create_all(bind=['users'])
>>> db.create_all(bind='appmeta')
>>> db.drop_all(bind=None)
```

Referring to Binds

If you declare a model you can specify the bind to use with the `__bind_key__` attribute:

```
class User(db.Model):
    __bind_key__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
```

Internally the bind key is stored in the table's *info* dictionary as `'bind_key'`. This is important to know because when you want to create a table object directly you will have to put it in there:

```
user_favorites = db.Table('user_favorites',
    db.Column('user_id', db.Integer, db.ForeignKey('user.id')),
    db.Column('message_id', db.Integer, db.ForeignKey('message.id')),
    info={'bind_key': 'users'}
)
```

If you specified the *__bind_key__* on your models you can use them exactly the way you are used to. The model connects to the specified database connection itself.

1.2.6 Zask-ZeroRPC

Middleware in zerorpc is designed to provide a flexible way to change the RPC behavior. Zask-ZeroRPC provides a few features by the built-in middlewares.

Configuration Based Middleware

Endpoint middleware

First is the `CONFIG_ENDPOINT_MIDDLEWARE`, which will resolve endpoint according to the zask application configuration. To use that you can setup a ZeroRPC like this:

```
app = Zask(__name__)
app.config['ZERORPC_SOME_SERVICE'] = {
    '1.0': endpoint,
}
rpc = ZeroRPC(app, middlewares=[CONFIG_ENDPOINT_MIDDLEWARE])
```

Then create a server and a client:

```
class Srv(object):
    __version__ = "1.0"
    __service_name__ = "some_service"

    def hello(self):
        return 'world'
server = rpc.Server(Srv())
# don't need bind anymore
# rpc.Server will do that for you
server.run()
client = rpc.Client('some_service', version='1.0')
client.hello()
```

Application will look for `RPC_SOME_SERVICE` config, which is combined the `RPC_` prefix and upper case of `some_service`. You can set a default version to make the client initialization more easier:

```
app.config['ZERORPC_SOME_SERVICE'] = {
    '1.0': endpoint,
    'default': '1.0'
}
client = rpc.Client('some_service')
client.hello()
```

Custom Header Middleware

This is a default middleware.

We want to figure out where the request come from and deploy multiple version for one service. So we have to send the `version` and the `access_key` in the header, and validate the two in the server side:

```
app = Zask(__name__)
app.config['ZERORPC_SOME_SERVICE'] = {
    '2.0': new_endpoint,
    '1.0': old_endpoint,
    'client_keys': ['foo_client_key', 'bar_client_key'],
    'access_key': 'foo_client_key',
    'default': '2.0'
}

# as this is the default middleware
# second parameter can be omitted
rpc = ZeroRPC(app)
```

```
srv = rpc.Server(Srv())
srv.run()
client = rpc.Client('some_service')
```

Request header would be like this:

```
{
    'message_id': message_id,
    'v': 3,
    'service_name': 'some_service',
    'service_version': '2.0',
    'access_key': 'foo_client_key'
}
```

If `access_key` is not within the `client_keys` list of server side configuration, an exception will be raised and returned it back to the client.

But if `client_keys` is set to `None` or not setted, `access_key` will not be validated by the server.

Access Log Middleware

This is a default middleware.

As a RPC system, we want to save the access log for monitoring and analyzing. All the services in one physical machine will share on logfile:

```
'%(access_key)s - [%(asctime)s] - %(message)s'
```

If client don't send `access_key` in the header, `access_key` will leave to `None`:

```
None - [2014-12-18 13:33:16,433] - "MySrv" - "foo" - OK - 1ms
```

Disable Middlewares

The middlewares will be applied to all the servers and clients by default. If you don't want to use the middlewares, just set `middlewares` to `None`:

```
app = Zask(__name__)
rpc = ZeroRPC(app, middlewares=None)
```

Or set a new context to the Server/Client during the runtime:

```
app = Zask(__name__)
rpc = ZeroRPC(app, middlewares=[CONFIG_ENDPOINT_MIDDLEWARE])

default_context = zerorpc.Context().get_instance()
srv = rpc.Server(Srv(), context=default_context)
client = rpc.Client(context=default_context)
```

1.2.7 Default configures

Name	Description
DEBUG	enable/disable debug mode default: `True`
ERROR_LOG	the path for the `zask logger` when `DEBUG` is `False` default: `/tmp/zask.error.log`
ZERORPC_ACCESS_LOG	the path for the `access logger` when `DEBUG` is `False` default: `/tmp/zask.acess.log`
SQLALCHEMY_DATABASE_URI	the main URI for SqlAlchemy default: `sqlite://`
SQLALCHEMY_BINDS	multiple binds mapping. default: `None`
SQLALCHEMY_NATIVE_UNICODE	default: `None`
SQLALCHEMY_ECHO	enable/disable echo SqlAlchemy debug default: `False`
SQLALCHEMY_POOL_SIZE	default: `None`
SQLALCHEMY_POOL_TIMEOUT	default: `None`
SQLALCHEMY_POOL_RECYCLE	default: `3600`
SQLALCHEMY_MAX_OVERFLOW	default: `None`

1.2.8 Best Practices

Configure loader

We have several configure files for different envs, dev and prod for example. We can load config files in a special order:

```
from zask import Zask

app = Zask(__name__)

# which is in the codebase
app.config.from_pyfile("settings.cfg")

# which is for development,
# ignored by codebase
app.config.from_pyfile("dev.setting.cfg", silent=True)

# which is for production,
# deployed by CM tools
app.config.from_pyfile("/etc/foo.cfg", silent=True)

# which is work with supervisord
app.config.from_envvar('CONFIG_PATH', silent=True)

# use logger after configure initialize
app.logger.debug("Config loaded")
```

1.2.9 Developer

Document

Document is powed by Sphinx. First, ensure sphinx is installed to the same environment as source code. Second, run:

```
$ sphinx-apidoc
$ make html
```

Note: update your `sphinx-build` *to the real path in* `Makefile`.

Testing

Similar to documentation, testing module need to be installed to the same environment. You can test one file at a time by run the script:

```
$ python tests/test_config.py
```

Or test all the cases with `tox` and `pytest`:

```
$ tox
```

Visit tox and pytest for more infomation.

1.2.10 API Reference

zask package

Subpackages

zask.ext package

Subpackages

zask.ext.sqlalchemy package

Module contents

zask.ext.sqlalchemy Adds basic SQLAlchemy support to your application. I have not add all the feature, bacause zask is not for web, The other reason is i can't handle all the features right now :P

Differents between Flask-SQLAlchemy:

1. No default `scopefunc` it means that you need define how to separate sessions your self
2. No signal session
3. No query record
4. No pagination and HTTP headers, e.g. `get_or_404`
5. No difference between app bound and not bound

copyright

3. 2015 by the J5.

license BSD, see LICENSE for more details.

copyright

3. 2012 by Armin Ronacher, Daniel Neuhäuser.

license BSD, see LICENSE for more details.

class zask.ext.sqlalchemy.**BindSession**(*db*, *autocommit=False*, *autoflush=True*, ***options*)
Bases: sqlalchemy.orm.session.Session

The BindSession is the default session that Zask-SQLAlchemy uses. It extends the default session system with bind selection. If you want to use a different session you can override the *SQLAlchemy.create_session()* function.

app = None
The application that this session belongs to.

get_bind(*mapper*, *clause=None*)

class zask.ext.sqlalchemy.**Model**
Bases: object

Baseclass for custom user models.

query = None
an instance of *query_class*. Can be used to query the database for instances of this model.

query_class
the query class used. The *query* attribute is an instance of this class. By default a orm.Query is used.

alias of Query

class zask.ext.sqlalchemy.**SQLAlchemy**(*app=None*, *use_native_unicode=True*, *session_options=None*)
Bases: object

This class is used to control the SQLAlchemy integration to one or more Zask applications.

There are two usage modes which work very similarly. One is binding the instance to a very specific Zask application:

```
app = Zask(__name__)
db = SQLAlchemy(app)
```

The second possibility is to create the object once and configure the application later to support it:

```
db = SQLAlchemy()
def create_app():
    app = Zask(__name__)
    db.init_app(app)
    return app
```

This class also provides access to all the SQLAlchemy functions and classes from the sqlalchemy and sqlalchemy.orm modules. So you can declare models like this:

```
class User(db.Model):
    username = db.Column(db.String(80), unique=True)
    pw_hash = db.Column(db.String(80))
```

apply_driver_hacks(*app*, *info*, *options*)
This method is called before engine creation and used to inject driver specific hacks into the options. The *options* parameter is a dictionary of keyword arguments that will then be used to call the sqlalchemy.create_engine() function.

The default implementation provides some saner defaults for things like pool sizes for MySQL and sqlite. Also it injects the setting of *SQLALCHEMY_NATIVE_UNICODE*.

apply_pool_defaults(*app*, *options*)

create_all(*bind='__all__'*, *app=None*)
Creates all tables.

create_scoped_session(*options=None*)
Helper factory method that creates a scoped session. It internally calls *create_session()*.

create_session(*options*)
Creates the session. The default implementation returns a *BindSession*.

drop_all(*bind='__all__'*, *app=None*)
Drops all tables.

engine
Gives access to the engine. If the database configuration is bound to a specific application (initialized with an application) this will always return a database connection. If however the current application is used this might raise a `RuntimeError` if no application is active at the moment.

get_app(*reference_app=None*)
Helper method that implements the logic to look up an application.

get_binds(*app=None*)
Returns a dictionary with a table->engine mapping.

This is suitable for use of sessionmaker(binds=db.get_binds(app)).

get_engine(*app*, *bind=None*)
Returns a specific engine.

get_tables_for_bind(*bind=None*)
Returns a list of all tables relevant for a bind.

init_app(*app*)
This callback can be used to initialize an application for the use with this database setup. Never use a database in the context of an application not initialized that way or connections will leak.

make_connector(*app*, *bind=None*)
Creates the connector for a given state and bind.

make_declarative_base()
Creates the declarative base.

metadata
Returns the metadata

reflect(*bind='__all__'*, *app=None*)
Reflects tables from the database.

class zask.ext.sqlalchemy.**SessionMiddleware**(*db*)
Bases: `object`

server_after_exec(*request_event*, *reply_event*)

server_inspect_exception(*request_event*, *reply_event*, *task_context*, *exc_infos*)

zask.ext.sqlalchemy.**get_state**(*app*)
Gets the state for the application

zask.ext.zerorpc package

Module contents

zask.ext.zerorpc Add zerorpc support to zask.

copyright

3. 2015 by the J5.

license BSD, see LICENSE for more details.

class zask.ext.zerorpc.**AccessLogMiddleware**(*app*)
Bases: object

This can't be used before initialize the logger.

server_after_exec(*request_event*, *reply_event*)

server_before_exec(*request_event*)

server_inspect_exception(*request_event*, *reply_event*, *task_context*, *exc_infos*)

set_class_name(*class_name*)

exception zask.ext.zerorpc.**ClientMissingVersionException**
Bases: exceptions.Exception

class zask.ext.zerorpc.**ConfigCustomHeaderMiddleware**(*app*)
Bases: *zask.ext.zerorpc.ConfigEndpointMiddleware*

Besides resolve the endpoint by service name, add custome header to the client.

Server side will do the validation for the access key and service version.

client_before_request(*event*)

load_task_context(*event_header*)

set_server_version(*version*)

class zask.ext.zerorpc.**ConfigEndpointMiddleware**(*app*)
Bases: *zask.ext.zerorpc.ConfigMiddleware*

Resolve the endpoint by service name.

resolve_endpoint(*endpoint*)

class zask.ext.zerorpc.**ConfigMiddleware**(*app*)
Bases: object

A middleware work with configure of zask application.

This is the base class for all the config based middlewares.

get_access_key(*name*)

get_client_keys(*name*)

get_endpoint(*name*, *version*)

get_version(*name*, *version*)

class zask.ext.zerorpc.**HandleEndpoint**
Bases: object

static decode(*endpoint*)

static encode(*name*, *version*)

exception zask.ext.zerorpc.**MissingAccessKeyException**(*config_name*)
Bases: exceptions.Exception

exception zask.ext.zerorpc.**MissingConfigException**(*config_name*)
Bases: exceptions.Exception

exception zask.ext.zerorpc.**MissingMiddlewareException**(*middleware*)
Bases: exceptions.Exception

Raised when Zask tries to invoke a functionality provided by a specific middleware, but that middleware is not loaded.

exception zask.ext.zerorpc.**NoNameException**
Bases: exceptions.Exception

exception zask.ext.zerorpc.**NoSuchAccessKeyException**(*access_key*)
Bases: exceptions.Exception

exception zask.ext.zerorpc.**NoVersionException**
Bases: exceptions.Exception

class zask.ext.zerorpc.**RequestChainMiddleware**(*app*)
Bases: object

Generate UUID for requests and store in greenlet's local storage

clear_uuid()

client_before_request(*event*)

get_uuid()

server_after_exec(*request_event*, *reply_event*)

server_before_exec(*request_event*)

server_inspect_exception(*request_event*, *reply_event*, *task_context*, *exc_infos*)

set_uuid(*uuid*)

class zask.ext.zerorpc.**RequestEventMiddleware**
Bases: object

Exposes the request_event to the object being passed to Server() via self.get_request_event() from a service endpoint.

server_before_exec(*request_event*)
Injects the request_event into greenlet's local storage context.

exception zask.ext.zerorpc.**VersionNotMatchException**(*access_key*, *request_version*, *server_version*)
Bases: exceptions.Exception

class zask.ext.zerorpc.**ZeroRPC**(*app=None, middlewares=['header', 'uuid', 'access_log', 'event']*)
Bases: object

This is a class used to integrate zerorpc to the Zask application.

ZeroRPC extention provides a few powful middlewares.

Take CONFIG_ENDPOINT_MIDDLEWARE as example, which will resolve endpoint according to the zask application configuration. To use that you can setup a ZeroRPC like this:

```
app = Zask(__name__)
app.config['ZERORPC_SOME_SERVICE'] = {
    '1.0': endpoint,
}
rpc = ZeroRPC(app, middlewares=[CONFIG_ENDPOINT_MIDDLEWARE])
```

Then create a server and a client:

```
class Srv(object):
    __version__ = "1.0"
    __service_name__ = "some_service"

def hello(self):
    return 'world'

client = rpc.Client('some_service', version='1.0')
client.hello()
```

Application will look for `RPC_SOME_SERVICE` config. You can set a default version to make the client initialization more easier:

```
app.config['ZERORPC_SOME_SERVICE'] = {
    '1.0': endpoint,
    '2.0': [ # set list if you have multiple endpoints
        endpoint1,
        endpoint2
    ]
    'default': '1.0'
}
client = rpc.Client('some_service')
client.hello()
```

But if you don't want to use the middlewares, just set `middlewares` to `None`:

```
app = Zask(__name__)
rpc = ZeroRPC(app, middlewares=None)
```

Or set a new context to the Server/Client during the runtime:

```
app = Zask(__name__)
rpc = ZeroRPC(app, middlewares=[CONFIG_ENDPOINT_MIDDLEWARE])

default_context = zerorpc.Context().get_instance()
srv = rpc.Server(Srv(), context=default_context)
client = rpc.Client(context=default_context)
```

init_app(*app*)

Initial the access logger and zerorpc exception handlers.

Parameters **app** – current zask application

register_middleware(*middleware*)

zask.ext.zerorpc.**access_log**(*cls*)

[Deprecated] A decorator for zerorpc server class to generate access logs:

```
@access_log
Class MySrv(Object):

    def foo(self)
        return "bar"
```

Every request from client will create a log:

```
[2014-12-18 13:33:16,433] - None - "MySrv" - "foo" - OK - 1ms
```

Parameters **cls** – the class object

zask.ext.zerorpc.**init_zerorpc**(*app*)
Baskward compatibility.

Module contents

Submodules

zask.config module

zask.config

1. remove useless methods of flask.config
2. update docstring

copyright

3. 2015 by the J5

license BSD, see LICENSE for more details.

copyright

3. 2015 by the Werkzeug Team, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class zask.config.**Config**(*root_path*, *defaults=None*)
Bases: dict

Works exactly like a dict but provides ways to fill it from files or special dictionaries. There are two common patterns to populate the config.

You can fill the config from a config file:

```
app.config.from_pyfile('yourconfig.cfg')
```

Only uppercase keys are added to the config. This makes it possible to use lowercase values in the config file for temporary values that are not added to the config or to define the config keys in the same file that implements the application.

Probably the most interesting way to load configurations is from an environment variable pointing to a file:

```
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

In this case before launching the application you have to set this environment variable to the file you want to use. On Linux and OS X use the export statement:

```
export YOURAPPLICATION_SETTINGS='/path/to/config/file'
```

On windows use *set* instead.

Parameters

- **root_path** – path to which files are read relative from. When the config object is created by the application, this is the application's root_path.
- **defaults** – an optional dictionary of default values

from_envvar(*variable_name*, *silent=False*)
Loads a configuration from an environment variable pointing to a configuration file. This is basically just a shortcut with nicer error messages for this line of code:

```
app.config.from_pyfile(os.environ['YOURAPPLICATION_SETTINGS'])
```

Parameters

- **variable_name** – name of the environment variable
- **silent** – set to `True` if you want silent failure for missing files.

Returns bool. `True` if able to load config, `False` otherwise.

from_object(*obj*)

Updates the values from the given object. An object can be of one of the following two types:

•a string: in this case the object with that name will be imported

•an actual object reference: that object is used directly

Objects are usually either modules or classes.

Just the uppercase variables in that object are stored in the config. Example usage:

```
app.config.from_object('yourapplication.default_config')
from yourapplication import default_config
app.config.from_object(default_config)
```

You should not use this function to load the actual configuration but rather configuration defaults. The actual config should be loaded with *from_pyfile()* and ideally from a location not within the package because the package might be installed system wide.

Parameters **obj** – an import name or object

from_pyfile(*filename*, *silent=False*)

Updates the values in the config from a Python file. This function behaves as if the file was imported as module with the *from_object()* function.

Parameters

- **filename** – the filename of the config. This can either be an absolute filename or a filename relative to the root path.
- **silent** – set to `True` if you want silent failure for missing files.

get_namespace(*namespace*, *lowercase=True*, *trim_namespace=True*)

Returns a dictionary containing a subset of configuration options that match the specified namespace/prefix. Example usage:

```
config['IMAGE_STORE_TYPE'] = 'fs'
config['IMAGE_STORE_PATH'] = '/var/app/images'
config['IMAGE_STORE_BASE_URL'] = 'http://img.website.com'
image_store_config = config.get_namespace('IMAGE_STORE_')
```

The resulting dictionary *image_store* would look like:

```
{
    'type': 'fs',
    'path': '/var/app/images',
    'base_url': 'http://img.website.com'
}
```

This is often useful when configuration options map directly to keyword arguments in functions or class constructors.

Parameters

- **namespace** – a configuration namespace
- **lowercase** – a flag indicating if the keys of the resulting dictionary should be lowercase
- **trim_namespace** – a flag indicating if the keys of the resulting dictionary should not include the namespace

zask.logging module

zask.logging Implements the logging support for Zask.

copyright

3. 2015 by the J5.

license BSD, see LICENSE for more details.

`zask.logging.`**`create_logger`**(*config*)
Creates a logger for the application. Logger's behavior depend on `DEBUG` flag.Furthermore this function also removes all attached handlers in case there was a logger with the log name before.

`zask.logging.`**`debug_handler`**()

`zask.logging.`**`production_handler`**(*config*)

zask.utils module

zask.utils

1. remove useless methods of werkzeug.utils
2. add get_root_path

copyright

3. 2015 by the J5.

license BSD, see LICENSE for more details.

copyright

3. 2015 by the Werkzeug Team, see AUTHORS for more details.

license BSD, see LICENSE for more details.

exception `zask.utils.`**`ImportStringError`**(*import_name*, *exception*)
Bases: `exceptions.ImportError`

Provides information about a failed *`import_string()`* attempt.

`exception` = None
Wrapped exception.

`import_name` = None
String in dotted notation that failed to be imported.

`zask.utils.`**`get_root_path`**(*import_name*)

`zask.utils.`**`import_string`**(*import_name*, *silent=False*)
Imports an object based on a string. This is useful if you want to use import paths as endpoints or something similar. An import path can be specified either in dotted notation (`xml.sax.saxutils.escape`) or with a colon as object delimiter (`xml.sax.saxutils:escape`).

If *silent* is True the return value will be *None* if the import fails.

Parameters

- **import_name** – the dotted name for the object to import.
- **silent** – if set to *True* import errors are ignored and *None* is returned instead.

Returns imported object

Module contents

class zask.**LocalContext**
Bases: object

Store data in local greenlet context.

> Gevent also allows you to specify data which is local to the greenlet context. > Internally, this is implemented as a global lookup which addresses a private namespace keyed by the greenlet's getcurrent() value.

See http://sdiehl.github.io/gevent-tutorial/#thread-locals for information.

get_request_cxt()

class zask.**Zask**(*import_name*)
Bases: object

logger

CHAPTER 2

References

- ZeroRPC website

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Z

A

B

C

D

E

F

G

H

I

L

M

N

P

Q

R

S

V

Z