Skeletor Documentation

Release 0.1.1

David Aguilar

November 06, 2015

Contents

skeletor is a Python library for reusable SQLAlchemy utilities. It contains modules that can be considered a minimalist "application skeleton", thus "skeletor".

Contents

CHAPTER 1

Usage

CHAPTER 2

Download & Install

The easiest way to get skeletor is via PyPi with pip:

```
$ pip install -U skeletor
```

You can also download or *clone* the latest code and install from source:

```
$ python setup.py install
```

CHAPTER 3

API Reference

3.1 skeletor API documenation

Contents

3.1.1 skeletor.core – Skeleton core components

skeletor.core.compat – Compatibility adapters

skeletor.core.json – JSON utilities

skeletor.core.json.**dumps**(*obj*, ***kwargs*)
Serialize an object into a JSON string

skeletor.core.json.**loads**(*json_str*)
Load an object from a JSON string

skeletor.core.json.**read**(*path*)
Read JSON objects; return an empty dict on error

skeletor.core.json.**write**(*obj*, *path*)
Write object as JSON to path

creates directories if needed

skeletor.core.log – Logging

class skeletor.core.log.**ConsoleHandler**(*combine_stderr=False*)
Pass logging.INFO to stdout, everything else to stderr

class skeletor.core.log.**Formatter**(*fmt='%(levelno)s: %(msg)s'*)
Custom formatter to allow a custom output format per level

skeletor.core.log.**decorator**(*force=False*)
Return a text decorator

skeletor.core.log.**init**(*verbose*, *combine_stderr=False*)
Initialize logging

Parameters **verbose** – Enables debugging output

skeletor.core.log.**logger**(*name=None*)
Return a module-scope logger

skeletor.core.util – Utilities

skeletor.core.util.**deep_update**(*a*, *b*)
Allow piece-wise overriding of dictionaries

skeletor.core.util.**import_string**(*modstr*)
Resolve a package.module.variable string

skeletor.core.version – Library version

3.1.2 skeletor.util – Reusable utilities

skeletor.util.config – Configuration file readers

class skeletor.util.config.**Config**(*path*, *environ=None*)
Read configuration values from a file

value(*key*, *env=None*)
Return a configured value

class skeletor.util.config.**JSONConfig**(*path*, *environ=None*)
Read configuration values from a JSON file

skeletor.util.decorators – Function decorators for managing contexts

Decorators to simplify resource allocation

Many methods will require access to an initialized resource. The *acquire_context()* decorator provides a live resource context to its decorated function as the first argument.

Callers can override the decorator-provided resource by passing in *context*=... as a keyword argument at the call site, which is needed when the resource has been pre-allocated and needs to be reused between calls.

class skeletor.util.decorators.**Context**
Base class for custom contexts

acquire()
Called once iff the context is constructed by the context manager

error()
Called when a exiting a context via an exception

release()
Called at the end of a context iff constructed by a manager

success()
Called when exiting a context successfully

class skeletor.util.decorators.**DefaultFactory**
Creates contexts

static filter_kwargs(*kwargs*)
Filter context arguments out from the function's kwargs

class skeletor.util.decorators.**acquire_context**(*default_factory*, *default_contextmgr=None*, *decorator=None*, **args*, ***kwargs*)
A decorator to provide methods with a live resource context

This decorator takes arguments, and thus the only time the to-be-decorated method is available is post-construction during the decoration process. This happens in __call__, and it is called once.

Methods inside a class can pass "decorator=staticmethod" if they want a static method.

The blind *args* and *kwargs* are passed to the *default_factory* when contexts are constructed.

class skeletor.util.decorators.**bind**(**args*, ***kwargs*)
A decorator to bind function parameters

class skeletor.util.decorators.**bindfunc**(*decorator=None*)
Allows fn.bind(foo=bar) when decorated on a decorator

This is a decorator decorator, which takes a decorator as input and returns another decorator when applied, and grants decorators the ability to do *mydecorator.bind(foo=bar)* and have it work without having to repeat *foo=bar* everywhere.

bind(**args*, ***kwargs*)
bind() stashes arguments

skeletor.util.decorators.**passthrough_decorator**(*f*)
Return a function as-is, unchanged

This is the default decorator used when creating contexts.

skeletor.util.string – String utilities

Functions for formatting various data strings

skeletor.util.string.**decode**(*string*)
decode(encoded_string) returns an unencoded unicode string

skeletor.util.string.**encode**(*string*)
encode(unencoded_string) returns a string encoded in utf-8

skeletor.util.string.**expand_path**(*path*, *environ=None*)
Resolve $VARIABLE and ~user paths

skeletor.util.string.**expandvars**(*path*, *environ=None*)
Like os.path.expandvars, but operates on a provided dictionary

os.environ is used when *environ* is None.

```
>>> expandvars('$foo', environ={'foo': 'bar'}) == 'bar'
True
```

```
>>> environ = {'foo': 'a', 'bar': 'b'}
>>> expandvars('$foo:$bar:${foo}${bar}', environ=environ)  == 'a:b:ab'
True
```

3.1.3 skeletor.db – SQLAlchemy powertools

skeletor.db.context – Default database context

skeletor.db.decorators – Decorators for providing contexts

Decorators to simplify database access

skeletor database decorators makes it easy to provide access to a common schema or other database resource. These are referred to as contexts.

Contexts manage a sqlalchemy session and provide it to their decorated functions through their *context* keyword argument.

Contexts are created by a factory provided by the skeletor library, `skeletor.db.context.DatabaseFactory`. In order to customize how the context is created, the default factory supports a *creator* keyword argument that is called with a single *commit=True/False* argument to signify whether the returned context should acquire a transaction.

skeletor provides decorators for read-only queries and transactional mutators. Mutators will construct contexts with the keyword argument *commit=True*. An internal context manager catches exceptions created in the wrapped functions and automatically rolls back mutator transactions when an exception occurs. Upon completion of a mutator function, the transaction is committed.

When multiple mutator functions need to be chained then the *context* keyword argument must be passed along when calling out to other decorated context functions. e.g. *some_func(context=context)* will ensure that the context is reused and passed through as-is rather than having to construct a new context for that call. This is a good thing to do in general when calling other wrapped functions since it will reuse the existing session instead of creating a new one for that call.

The expected use case is that you can use these decorators while binding them to a custom creator function. First, we'll define a simple schema.

```
from sqlachemy import Column, Integer, String

from skeletor.db import schema, context, decorators

class Schema(schema.Schema):
    # A schema with a single users table: this is __not__ an ORM

    def __init__(self):
        schema.Schema.__init__(self)
        self.add_table('users',
                       Column('id', Integer,
                              autoincrement=True, primary_key=True),
```

```
                        Column('name', String),
                        Column('email', String, unique=True))

    @staticmethod
    def get():
        return Schema().bind_url('sqlite:///:memory:')
```

Next, we'll provide a `creator()` function that will wrap our custom schema in a `skeletor.db.context.DatabaseContext`. This class provides the automatic commit/rollback logic.

```
def creator(commit=False):
    return context.DatabaseContext(context=Schema.get())
```

To use the our custom creator with a free-form function we decorate it and supply the decorator with our custom `creator()`.

```
@decorators.query.bind(creator=creator)
def get_users(context=None):
    return context.users.select().execute().fetchall()
```

The end result is that callers do not need to worry about managing contexts. It will be automatically supplied by the decorator.

```
# Somewhere else
users = get_users() #  no arguments required!
```

When `get_users()` is called, no context was provided so the provided creator will be used to construct a context. `creator()` constructs a Schema object that defines the tables, binds it a URL, and returns the schema. That schema is effectively what is seen by `get_users()`.

This extends to mutator functions. Mutators are automatically run within a transaction which is committed when the function finishes, or rolls back if an exception occurs.

```
@decorators.mutator.bind(creator=creator)
def create_user(context=None):
    table = context.users
    args = dict(name='hello', email='world')
    return table.insert(args).execute()

create_user()
```

If you supply the context explicitly then no commit/rollback is performed. You must manually manage the context.

```
context = creator(commit=True)
try:
    create_user(context=context)
    context.commit()
except:
    context.rollback()
```

`skeletor.db.decorators.`**`query`**`()`
Provide a read-only database context to a wrapped function

`skeletor.db.decorators.`**`mutator`**`()`
Provide a transactional database context to a wrapped function

`skeletor.db.decorators.`**`staticmethod_query`**`()`
Provide a read-only database context to a wrapped function

Returns a staticmethod.

skeletor.db.decorators.**staticmethod_mutator**()
Provide a transactional database context to a wrapped function

Returns a staticmethod.

skeletor.db.decorators.**classmethod_query**()
Provide a read-only database context to a wrapped function

Returns a classmethod.

skeletor.db.decorators.**classmethod_mutator**()
Provide a transactional database context to a wrapped function

Returns a classmethod.

skeletor.db.schema – Schema definitions

skeletor.db.sql – SQLAlchemy helpers and utilities

skeletor.db.sql.**delete**(*table*, *operator=<function and_>*, *where=<function where>*, ***values*)
Delete rows from a table based on the filter values

skeletor.db.sql.**eq_column**(*table*, *column*, *value*)
column == value

skeletor.db.sql.**fetchone**(*table*, *where_expr*)
Select one row from a table filtered by a *where* expression

skeletor.db.sql.**ilike_column**(*table*, *column*, *value*)
column ILIKE value

skeletor.db.sql.**iwhere**(*table*, *operator=<function and_>*, ***values*)
Return a *where* expression to combine column ILIKE value criteria

skeletor.db.sql.**reduce_filters**(*table*, *fn*, *operator=<function and_>*, ***values*)
Return a *where* expression to combine column=value criteria

skeletor.db.sql.**select_all**(*table*, *where=<function where>*, *operator=<function and_>*, ***values*)
Select all rows filtered by *values* column=value criteria

skeletor.db.sql.**select_one**(*table*, *where=<function where>*, *operator=<function and_>*, ***values*)
Select one row filtered by *values* column=value criteria

skeletor.db.sql.**update**(*table*, *table_id*, ***values*)
Update a specific row's values by ID

skeletor.db.sql.**update_values**(*table*, *where_expr*, ***values*)
Return an update().values(...) expression for the given table

skeletor.db.sql.**where**(*table*, *operator=<function and_>*, ***values*)
Return a *where* expression to combine column=value criteria

skeletor.db.table – Query SQLAlchemy tables

class skeletor.db.table.**ScopedLogger**(*client*, *logger=None*)
Prefix log messages with the calling code's class name

class skeletor.db.table.**Table**(*table*, *logger=None*, *verbose=False*)
Run simple queries against specific tables

3.1.4 skeletor.cli – Command-line utilities

skeletor.cli.options – Common argparse.ArgumentParser options

skeletor.cli.options.**verbose**(*parser*, *options=('-v', '–verbose')*, *help='increase verbosity'*)
Add -v and –verbose options to an ArgumentParser

Parameters

- **parser** – an argparse.ArgumentParser instance
- **options** – overrides the default *-v, –verbose* options.
- **help** – overrides the default help text.

skeletor.cli.menu – Terminal menus

class skeletor.cli.menu.**Menu**(*title*, *prompt='? '*, *errormsg='invalid response, please try again'*, *fmt='%(key)s ... %(description)s'*, *underline='='*, *get_input=None*, *logger=None*, *logger_name=None*, *case_sensitive=False*, *args=None*, *kwargs=None*)
Display and interact with a callback-based menu

args = None
Allows callbacks to accept custom arguments

kwargs = None
Allows callbacks to accept custom keyword arguments

present(*options*)
Present a menu of options

Parameters **options** – a tuple of *(option, (description, callback))* where *options* is a string that is expected to be returned get_input, *description* is a description of the options, and *callback* is either a callable or Menu.QUIT to signify exiting the menu.

e.g.

```
from skeletor.core import log
from skeletor.cli.menu import Menu

log.init(True)

def hello():
    print('oh hai')

options = (
    ('c', ('say hello', hello)),
    ('q', ('quit', Menu.QUIT)),
)
Menu('example menu').present(options)
```

skeletor.cli.menu.**confirm**(*user_msg*, *default=True*, *empty_chooses=True*, *get_input=None*)
Confirm an action with a prompt

Parameters

- **user_msg** – the text to display.
- **default** – the value to return when no input is given.

- **empty_chooses** – when True, empty values choose the default.

Returns bool whether "yes" was chosen.

CHAPTER 4

Contributing

4.1 Contributing to skeletor

We welcome contributions from everyone. Please fork this project on github.

4.1.1 Get the Code

```
git clone git://github.com/davvid/skeletor.git
```

4.1.2 Run the Test Suite

All tests must pass before code can be pulled into the master branch. If you are contributing an addition or a change in behavior, we ask that you document the change in the form of test cases.

In order to run the test cases you will need to install some dependencies into your test environment. We recomment using tox To simplify the process of setting up a test environment.

To run the suite, simply invoke *make test* or use tox to run the unittests across multiple Python versions:

```
$ make test
nosetests --with-doctest  skeletor tests
..........................
----------------------------------------------------------------------
Ran 29 tests in 0.544s
```

4.1.3 Generate Documentation

You do not need to generate the documentation when contributing, though, if you are interested, you can generate the docs yourself. The following requires Sphinx:

```
cd docs
make html
```

If you wish to browse the documentation, use Python's `SimpleHTTPServer` to host them at http://localhost:8000:

```
cd build/html
python -m SimpleHTTPServer
```

CHAPTER 5

Contact

Follow the project on Github, submit pull requests, and file issues. Please try to report bugs in the form of a test case that can be added to the test suite.

CHAPTER 6

Authors

- David Aguilar davvid -at- gmail.com - https://github.com/davvid

CHAPTER 7

Change Log

7.1 Change Log

7.1.1 Version 0.0.1 - January 2015

- The initial skeletor release contains powerful decorators and utility classes for use with SQLAlchemy databases.
- The core things needed to support a databse-backed application are configuration, logging, and database context/session management.
- Configuration is used for getting credentials since they should not live alongside the library and application code.
- Logging is needed for reporting database and application errors.
- Database context management is needed to streamline the work needed to establish database connections and transactions.
- Skeletor's database decorators allow you to tag a function as a "mutator", which means that calling it without an existing context will create a context, start a transation, and call into the function. If the function raises any exceptions then the transaction is automatically rolled back by the database context manager.

CHAPTER 8

License

skeletor is provided under a New BSD license,

S

T

U

V

W