
Flask Chemist Documentation

Release 1.7.0

Gabriel Falcão

Aug 02, 2021

CONTENTS:

1	Quick Start	3
1.1	Install	3
1.2	Declaring a model	3
1.3	Creating new records	4
1.4	Querying	5
1.5	Editing active records	5
1.6	Deleting	6
2	Using Flask	7
2.1	A simple application	7
3	Using Alembic	9
3.1	Creating tables	9
3.2	Modifying Columns	9
3.3	Migration template (<code>script.py.mako</code>)	9
4	Advanced Querying	11
4.1	<code>SELECT * FROM table WHERE ...</code>	11
5	Writing “unit” tests	13
6	API Reference	15
7	Release History	19
7.1	Changes in 1.7.0	19
7.2	Changes in 1.6.4	19
8	Indices and tables	21
	Python Module Index	23
	Index	25

A simple, flexible and testable active-record powered by SQLAlchemy.

Supports application-level encryption powered by `nacl.utils.SecretBox`

QUICK START

1.1 Install

```
pip install chemist
```

1.1.1 MariaDB/MySQL

```
pip install chemist[mysql]
pip install chemist[mariadb] # alias to [mysql]
```

1.1.2 Postgres

```
pip install chemist[psycopg2]
pip install chemist[postgres] # alias to [psycopg2]
pip install chemist[postgresql] # alias to [psycopg2]
```

1.2 Declaring a model

```
import bcrypt
from chemist import (
    Model, db, metadata
    set_default_uri,
)

engine = set_default_uri('sqlite:///example.db')

CREDIT_CARD_ENCRYPTION_KEY = b'\xb3\x0f\xcc9\xc3\xb1k#\x95j4\xb3\x1f\x08\x98\xd7~
↪6\xff\xceb\xdc\x17vW\xd7\x90\xcf\x82\x9d\xb7j'

class User(Model):
    table = db.Table(
        'auth_user',
        metadata,
        db.Column('id', db.Integer, primary_key=True),
        db.Column('email', db.String(100), nullable=False, unique=True),
        db.Column('password', db.String(100), nullable=False, unique=True),
```

(continues on next page)

(continued from previous page)

```

        db.Column('created_at', db.DateTime, default=datetime.now),
        db.Column('updated_at', db.DateTime, default=datetime.now),
        db.Column('credit_card', db.String(16)),
    )
    encryption = {
        # transparently encrypt data data in "credit_card" field before storing on DB
        # also transparently decrypt after retrieving data
        'credit_card': CREDIT_CARD_ENCRYPTION_KEY,
    }

    @classmethod
    def create(cls, email, password, **kw):
        email = email.lower()
        password = cls.secretify_password(password)
        return super(User, cls).create(email=email, password=password, **kw)

    def to_dict(self):
        # prevent password and credit-card to be returned in HTTP responses that
        ↪serialize model data
        data = self.serialize()
        data.pop('password', None)
        data.pop('credit_card', None)
        return data

    @classmethod
    def secretify_password(cls, plain):
        return bcrypt.hashpw(plain, bcrypt.gensalt(12))

    def match_password(self, plain):
        return self.password == bcrypt.hashpw(plain, self.password)

    def change_password(self, old_password, new_password):
        right_password = self.match_password(old_password)
        if right_password:
            secret = self.secretify_password(new_password)
            self.set(password=secret)
            self.save()
            return True

        return False

metadata.drop_all()
metadata.create_all()

```

1.3 Creating new records

```

data = {
    "email": "octocat@github.com",
    "password": "1234",
}
created = User.create(**data)

assert created.id == 1

```

(continues on next page)

(continued from previous page)

```
assert created.to_dict() == {
    'id': 1,
}

same_user = User.get_or_create(**data)
assert same_user.id == created.id
```

1.4 Querying

```
user_count = User.count()
user_list = User.all()

github_users = User.find_by(email__contains='github.com')
octocat = User.find_one_by(email='octocat@github.com')

assert octocat == user_list[0]

assert octocat.id == 1

assert user_count == 1
```

1.5 Editing active records

```
octocat = User.find_one_by(email='octocat@github.com')

# modify in memory

octocat.password = 'much more secure'
# or ...
octocat.set(
    password='much more secure',
    email='octocat@gmail.com',
)

# save changes (commit transaction and flush db session)
octocat.save()

# or ...

# modify and save changes in a single call
saved_cat = octocat.update_and_save(
    password='even more secure now',
    email='octocat@protonmail.com',
)
assert saved_cat == octocat
```

1.6 Deleting

```
from chemist import set_default_uri

engine = set_default_uri('sqlite:///example.db')

octocat = User.find_one_by(email='octocat@github.com')

# delete row, commit and flush session
ghost_cat = octocat.delete()

# but the copy in memory still has all the data
assert ghost_cat.id == 1

# resurrecting the cat
octocat = ghost_cat.save()
```

USING FLASK

2.1 A simple application

```
import json
from flask import Flask, request
from chemist import (
    Model, db, metadata,
    set_default_uri,
)

app = Flask(__name__)

metadata = Metadata()
engine = set_default_uri('sqlite:///example.db')

class User(Model):
    table = db.Table('user', metadata,
        db.Column('id', db.Integer, primary_key=True),
        db.Column('email', db.String(100), nullable=False, unique=True),
        db.Column('password', db.String(100), nullable=False, unique=True),
        db.Column('created_at', db.DateTime, default=datetime.now),
        db.Column('updated_at', db.DateTime, default=datetime.now)
    )

@app.post("/user")
def create_user():
    email = request.data.get('email')
    password = request.data.get('password')
    user = User.get_or_create(email=email, password=password)
    return json.dumps(user.to_dict()), 201, {'Content-Type': 'application/json'}

@app.post("/user")
def list_users():
    users = User.all()
    data = json.dumps([u.to_dict() for u in users])
    return json.dumps(data), 201, {'Content-Type': 'application/json'}
```


USING ALEMBIC

3.1 Creating tables

```
def upgrade():
    op.create_table(
        'user',
        sa.Column('id', sa.Integer, primary_key=True, nullable=False),
        sa.Column('uuid', sa.String(32), nullable=False),
        sa.Column('email', sa.String(254)),
        sa.Column('password', sa.String(128)),
        sa.Column('creation_date', sa.DateTime, nullable=False),
        sa.Column('activation_date', sa.DateTime, nullable=True),
        sa.Column('json_data', sa.Text, nullable=True),
        sa.Column('last_access_date', sa.DateTime, nullable=True),
        sa.Column('password_change_date', sa.DateTime, nullable=True),
    )

def downgrade():
    op.drop_table('user')
```

3.2 Modifying Columns

```
def upgrade():
    op.alter_column('user', 'json_data', existing_type=db.Text, type_=db.LargeBinary)

def downgrade():
    op.alter_column('user', 'json_data', existing_type=db.LargeBinary, type_=db.Text)
```

3.3 Migration template (script.py.mako)

```
# -*- coding: utf-8 -*-
# flake8: noqa
"""${message}

Revision ID: ${up_revision}
Revises: ${down_revision}
```

(continues on next page)

(continued from previous page)

```
Create Date: ${create_date}

"""

# revision identifiers, used by Alembic.
revision = ${repr(up_revision)}
down_revision = ${repr(down_revision)}

from datetime import datetime
from alembic import op
import sqlalchemy as db
${imports if imports else ""}

def DefaultForeignKey(field_name, parent_field_name,
                      ondelete='CASCADE', nullable=False, **kw):
    return db.Column(field_name, db.Integer,
                      db.ForeignKey(parent_field_name, ondelete=ondelete),
                      nullable=nullable, **kw)

def PrimaryKey(name='id'):
    return db.Column(name, db.Integer, primary_key=True)

def now():
    return datetime.utcnow()

def upgrade():
    ${upgrades if upgrades else "pass"}

def downgrade():
    ${downgrades if downgrades else "pass"}
```

ADVANCED QUERYING

4.1 SELECT * FROM table WHERE ...

Use the methods `where_one()` and `where_many()` with the same clause accepted by `where()`.

```
from datetime import datetime

from sqlalchemy import asc, desc

from chemist import (
    Model, db,
    get_or_create_engine,
)
from chemist import metadata # use chemist-managed metadata

def generate_uuid():
    return str(uuid.uuid4())

def autonow():
    return datetime.utcnow()

class Task(Model):
    table = db.Table(
        'advanced_querying_task',
        metadata,
        db.Column('id', db.Integer, primary_key=True),
        db.Column('uuid', db.String(36), nullable=False, index=True, default=generate_
↳ uuid),
        db.Column('name', db.UnicodeText, nullable=False),
        db.Column('done_at', db.DateTime),
        db.Column('updated_at', db.DateTime, default=autonow),
    )

    @classmethod
    def list_pending(model, *expressions):
        table = model.table
        order_by = (desc(table.c.updated_at), )
        return model.objects().where_many(
            model.table.c.done_at==None,
            *expressions,
            order_by=order_by,
        )
```

(continues on next page)

(continued from previous page)

```
@classmethod
def get_by_uuid(model, uuid):
    table = model.table
    return model.objects().where_one(model.table.c.uuid==uuid)
```

See also:

`where_one()` and `where_many()` **optionally take** an `order_by=` keyword-argument, which must be a tuple of `asc()` or `desc()` columns.

WRITING “UNIT” TESTS

The example below uses the module `testing.postgresql` to run each test case in an isolated postgres server instance all you need is the postgres binaries available in the host machine.

```
import unittest
import testing.postgresql
from chemist import set_default_uri
from models import User

class UserModelTestCase(unittest.TestCase):
    def setUp(self):
        self.postgresql = testing.postgresql.Postgresql()
        set_default_uri(self.postgresql.url())

    def tearDown(self):
        self.postgresql.stop()

    def test_authentication(self):
        # Given a user with a hardcoded password
        foobar = User.create('foo@bar.com', '123insecure')

        # When I match the password
        matched = foobar.match_password('123insecure')

        # Then it should have matched
        assert matched, f'user {foobar} did not match password 123insecure'

    def test_change_password(self):
        # Given a user with a hardcoded password
        foobar = User.create('foo@bar.com', '123insecure')

        # When I change the password
        changed = foobar.change_password('123insecure', 'newPassword')

        # Then it should have succeeded
        assert matched, f'failed to change password for {foobar}'

        # And should authenticate with the new password
        assert foobar.match_password('newPassword'), (
            f'user {foobar} did not match password newPassword')
```

```
import bcrypt
```

(continues on next page)

(continued from previous page)

```

from chemist import (
    Model, db, metadata
    set_default_uri,
)

engine = set_default_uri('sqlite:///example.db')

CREDIT_CARD_ENCRYPTION_KEY = b'\xb3\x0f\xcc9\xc3\xb1k#\x95j4\xb3\x1f\x08\x98\xd7~
↪6\xff\xce\xdc\x17vW\xd7\x90\xcf\x82\x9d\xb7j'

class User(Model):
    table = db.Table(
        'auth_user',
        metadata,
        db.Column('id', db.Integer, primary_key=True),
        db.Column('email', db.String(100), nullable=False, unique=True),
        db.Column('password', db.String(100), nullable=False, unique=True),
        db.Column('created_at', db.DateTime, default=datetime.now),
        db.Column('updated_at', db.DateTime, default=datetime.now),
        db.Column('credit_card', db.String(16)),
    )
    encryption = {
        # transparently encrypt data data in "credit_card" field before storing on DB
        # also transparently decrypt after retrieving data
        'credit_card': CREDIT_CARD_ENCRYPTION_KEY,
    }

    @classmethod
    def create(cls, email, password, **kw):
        email = email.lower()
        password = cls.secretify_password(password)
        return super(User, cls).create(email=email, password=password, **kw)

    def to_dict(self):
        # prevent password and credit-card to be returned in HTTP responses that_
        ↪serialize model data
        data = self.serialize()
        data.pop('password', None)
        data.pop('credit_card', None)
        return data

    @classmethod
    def secretify_password(cls, plain):
        return bcrypt.hashpw(plain, bcrypt.gensalt(12))

    def match_password(self, plain):
        return self.password == bcrypt.hashpw(plain, self.password)

    def change_password(self, old_password, new_password):
        right_password = self.match_password(old_password)
        if right_password:
            secret = self.secretify_password(new_password)
            self.set(password=secret)
            self.save()
            return True

        return False

```

API REFERENCE

class `chemist.managers.Manager` (*model_klass*, *context*)

all (*limit_by=None*, *offset_by=None*, *order_by=None*)
Returns all existing rows as Model

create (***data*)
Creates a new model and saves it to MySQL

find_by (***kw*)
Find a list of models that could be found in the database and match all the given keyword-arguments

find_one_by (***kw*)
Find a single model that could be found in the database and match all the given keyword-arguments

from_result_proxy (*proxy*, *result*)
Creates a new instance of the model given an instance of `sqlalchemy.engine.ResultProxy`

generate_query (*order_by=None*, *limit_by=None*, *offset_by=None*, ***kw*)
Queries the table with the given keyword-args and optionally a single *order_by* field.

get_or_create (***data*)
Tries to get a model from the database that would match the given keyword-args through `Manager.find_one_by()`. If not found, a new instance is created in the database through `Manager.create()`

query_by (***kwargs*)
This method is used internally and is not consistent with the other ORM methods by not returning a model instance.

total_rows (*field_name=None*, ***where*)
Gets the total number of rows in the table

class `chemist.orm.Context` (*default_uri=None*)

Context is a system component that keeps track of multiple engines, switch between them and manage their lifecycle.

It also provides a `sqlalchemy.MetaData` instance that is automatically bound to

Its purpose is to leverage quickly swapping the engine between “unit” tests.

class `chemist.orm.Monetary`

class `chemist.orm.ORM` (*name*, *bases*, *attrs*)
metaclass for `chemist.models.Model`

class `chemist.models.Model` (*engine=None*, ***data*)
Super-class of active record models.

Example:

```
class BlogPost (Model) :
    table = db.Table(
        'blog_post',
        metadata,
        db.Column('id', db.Integer, primary_key=True),
        db.Column('title', db.Unicode(200), nullable=False),
        db.Column('slug', db.Unicode(200), nullable=False),
        db.Column('content', db.UnicodeText, nullable=False),
    )

    def preprocess(self, data):
        # always derive slug from title
        data['slug'] = slugify(data['title'])
        return data
```

delete()

Deletes the current model from the database (removes a row that has the given model primary key)

get (*name*, *fallback=None*)

Get a field value from the model

initialize()

Dummy method to be optionally overwritten in the subclasses. Gets automatically called once a model instance is constructed.

is_persisted

boolean property that returns **True** if the primary key is set. This property **does not perform I/O against the database**

manager

alias of *chemist.managers.Manager*

post_delete()

called right after executing a deletion. This method can be overwritten by subclasses in order to take any domain-related action

post_save (*transaction*)

called right after executing a save. This method can be overwritten by subclasses in order to take any domain-related action

pre_delete()

called right before executing a deletion. This method can be overwritten by subclasses in order to take any domain-related action

pre_save()

called right before executing a save. This method can be overwritten by subclasses in order to take any domain-related action

preprocess (*data*)

Placeholder for your own custom preprocess method, remember it must return a dictionary.

```
class BlogPost (Model) :
    table = db.Table(
        'blog_post',
        metadata,
        db.Column('id', db.Integer, primary_key=True),
        db.Column('title', db.Unicode(200), nullable=False),
        db.Column('slug', db.Unicode(200), nullable=False),
        db.Column('content', db.UnicodeText, nullable=False),
```

(continues on next page)

(continued from previous page)

```

    )

    def preprocess(self, data):
        # always derive slug from title
        data['slug'] = slugify(data['title'])
        return data

```

refresh()

updates the current record with fresh values retrieved by `find_one_by()` and also returns a brand new instance.

Note: any unsaved changes in the model will be lost upon calling this method.

save(input_engine=None)

Persists the model instance in the DB. It takes care of checking whether it already exists and should be just updated or if a new record should be created.

serialize()

pre-serializes the model, returning a dictionary with key-values.

This method is use by the `to_dict()` and only exists as a separate method so that subclasses overwriting `to_dict` can call `serialize()` rather than `super(SubclassName, self).to_dict()`

set(kw)**

Sets multiple fields, does not perform a save operation

to_dict()

pre-serializes the model, returning a dictionary with key-values.

This method can be overwritten by subclasses at will.

Example:

```

>>> post = BlogPost.create(title='Some Title', content='loren ipsum')
>>> post.to_dict()
{
  'id': 1,
  'title': 'Some Title',
  'slug': 'some-title',
}

```

to_insert_params()

utility method used internally to generate a dict with all the serialized values except primary keys.

Example:

```

>>> post = BlogPost.create(title='Some Title', content='loren ipsum')
>>> post.to_insert_params()
{
  'title': 'Some Title',
  'slug': 'some-title',
}

```

to_json(indent=None, sort_keys=True, **kw)

Grabs the dictionary with the current model state returned by `to_dict` and serializes it to JSON

update_and_save(kw)**

Sets multiple fields then saves them

RELEASE HISTORY

7.1 Changes in 1.7.0

- Use `begin()` after every call to retrieve results

7.2 Changes in 1.6.4

- Forward `order_by` keyword-argument from `ModelManager.all`

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

C

`chemist.managers`, [15](#)

`chemist.models`, [15](#)

`chemist.orm`, [15](#)

A

`all()` (*chemist.managers.Manager method*), 15

C

`chemist.managers` (*module*), 15

`chemist.models` (*module*), 15

`chemist.orm` (*module*), 15

`Context` (*class in chemist.orm*), 15

`create()` (*chemist.managers.Manager method*), 15

D

`delete()` (*chemist.models.Model method*), 16

F

`find_by()` (*chemist.managers.Manager method*), 15

`find_one_by()` (*chemist.managers.Manager method*), 15

`from_result_proxy()` (*chemist.managers.Manager method*), 15

G

`generate_query()` (*chemist.managers.Manager method*), 15

`get()` (*chemist.models.Model method*), 16

`get_or_create()` (*chemist.managers.Manager method*), 15

I

`initialize()` (*chemist.models.Model method*), 16

`is_persisted` (*chemist.models.Model attribute*), 16

M

`manager` (*chemist.models.Model attribute*), 16

`Manager` (*class in chemist.managers*), 15

`Model` (*class in chemist.models*), 15

`Monetary` (*class in chemist.orm*), 15

O

`ORM` (*class in chemist.orm*), 15

P

`post_delete()` (*chemist.models.Model method*), 16

`post_save()` (*chemist.models.Model method*), 16

`pre_delete()` (*chemist.models.Model method*), 16

`pre_save()` (*chemist.models.Model method*), 16

`preprocess()` (*chemist.models.Model method*), 16

Q

`query_by()` (*chemist.managers.Manager method*), 15

R

`refresh()` (*chemist.models.Model method*), 17

S

`save()` (*chemist.models.Model method*), 17

`serialize()` (*chemist.models.Model method*), 17

`set()` (*chemist.models.Model method*), 17

T

`to_dict()` (*chemist.models.Model method*), 17

`to_insert_params()` (*chemist.models.Model method*), 17

`to_json()` (*chemist.models.Model method*), 17

`total_rows()` (*chemist.managers.Manager method*), 15

U

`update_and_save()` (*chemist.models.Model method*), 17