# flask-sqlalchemy-booster Documentation

***Release 0.1.0***

**Surya Sankar**

January 03, 2016

Contents

Flask-SQLAlchemy-Booster is a collection of enhancements to the Flask-SQLAlchemy library.

**It replaces the Model class with a subclass that adds**

1. Additional querying methods and
2. Easily configurable `todict` methods and `tojson` methods for serializing objects.

It also provides some decorators and utility functions which can be used to easily generate JSON responses.

CHAPTER 1

# Features

- Fully compatible with code written for Flask-SQLAlchemy. It will just transparently replace it with additional features.
- Simple api for most common querying operations:

```
>>> user = User.first()
>>> user2 = User.last()
>>> newcust = Customer.find_or_create(name="Alex", age=21, email="al@h.com")
```

- JSON response functions which can be dynamically configured via the GET request params allowing you to do things like:

```
GET /api/customers?city~=Del&expand=shipments.country,user&sort=desc&limit=5
```

CHAPTER 2

# Contents

## 2.1 Installation

Install via pip:

```
$ pip install Flask-SQLAlchemy-Booster
```

Or you can clone the public repository:

```
$ git clone git@github.com:inkmonk/flask-sqlalchemy-booster.git
```

and then run:

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

## 2.2 How To Use

Since it just subclasses Flask-SQLAlchemy's Model class, the usage is entirely similar.

Set up Flask-SQLAlchemy related configuration keys to set up the database.

Then create a db instance like this:

```
from flask.ext.sqlalchemy_booster import FlaskSQLAlchemyBooster
db = FlaskSQLAlchemyBooster()
```

You can then subclass the `db.Model` class to create your model classes:

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True, unique=True)
    email = db.Column(db.String(100), unique=True)
    password = db.Column(db.String(100))
    name = db.Column(db.String(100))
    active = db.Column(db.Boolean())

class Order(db.Model):
    id = db.Column(db.Integer, primary_key=True, unique=True)
user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
```

## 2.3 Using the Querying API

This module provides a set of commonly used methods - for CRUD operations on models. Usage is very simple

To get the first instance obeying some conditions:

```
customer = Customer.first(state="Rajasthan")

first_cust = Customer.first()
```

To get the last instance obeying some conditions:

```
customer = Customer.last(city="Delhi")

last_cust = Customer.last()
```

## 2.4 API

### 2.4.1 Core

**class** flask_sqlalchemy_booster.core.**FlaskSQLAlchemyBooster**(***kwargs*)
Bases: flask_sqlalchemy.SQLAlchemy

Sets the Model class to ModelBooster, providing all the methods defined on ModelBooster.

#### Examples

```
>>> db = FlaskSQLAlchemyBooster()
```

```
>>> class User(db.Model):
        id = db.Column(db.Integer, primary_key=True, unique=True)
        email = db.Column(db.String(100), unique=True)
        password = db.Column(db.String(100))
        name = db.Column(db.String(100))
        contact_number = db.Column(db.String(20))
```

```
>>> User.all()
```

```
>>> u = User.first()
```

```
>>> u.todict()
```

**class** flask_sqlalchemy_booster.core.**QueryPropertyWithModelClass**(*sa*)
Bases: flask_sqlalchemy._QueryProperty

Subclassed to add the cls attribute to a query instance.

This is useful in instances when we need to find the class of the model being queried when provided only with a query object

### 2.4.2 ModelBooster

This is the db.Model class that you will use as the super class for your Models. It has two sets of methods defined on it, apart from the ones already defined by FlaskSQLAlchemy.

**class** `flask_sqlalchemy_booster.queryable_mixin.`**`QueryableMixin`**

Contains all querying methods. Used for common ORM operations

**`_no_overwrite_`**

*list* – The list of attributes that should not be overwritten.

**classmethod `add`**(*model*, *commit=True*)

Adds a model instance to session and commits the transaction.

**Parameters** **`model`** – The instance to add.

#### Examples

```
>>> customer = Customer.new(name="hari", email="hari@gmail.com")
```

```
>>> Customer.add(customer)
hari@gmail.com
```

**classmethod `add_all`**(*models*, *commit=True*, *check_type=False*)

Batch method for adding a list of model instances to the db in one get_or_404.

**Parameters**

- **`models`** (*list*) – A list of the instances to add.
- **`commit`** (*bool, optional*) – Defaults to True. If False, the transaction won't get committed.
- **`check_type`** (*bool, optional*) – If True, each instance is type checked and exception is thrown if it is not an instance of the model. By default, False.

**Returns** A list of `Model` instances

**Return type** list

**classmethod `all`**(**criterion*, ***kwargs*)

Returns all the instances which fulfil the filtering criterion and kwargs if any given.

#### Examples

```
>>> Tshirt.all()
[tee1, tee2, tee4, tee5]
```

>> Tshirt.all(reverse=True, limit=3) [tee5, tee4, tee2]

>> Tshirt.all(color="Red") [tee4, tee2]

**classmethod `build`**(***kwargs*)

Similar to create. But the transaction is not committed

**Parameters** **`**kwargs`** – The keyword arguments for the constructor

**Returns** A model instance which has been added to db session. But session transaction has not been committed yet.

**classmethod `build_all`**(*list_of_kwargs*)

Similar to `create_all`. But transaction is not committed.

**`commit`**()

Commits a transaction.

**classmethod count** (**criterion*, ***kwargs*)
Returns a count of the instances meeting the specified filter criterion and kwargs.

#### Examples

```
>>> User.count()
500
```

```
>>> User.count(country="India")
300
```

```
>>> User.count(User.age > 50, country="India")
39
```

**classmethod create** (***kwargs*)
Initializes a new instance, adds it to the db and commits the transaction.

**Parameters** ****kwargs** – The keyword arguments for the init constructor.

#### Examples

```
>>> user = User.create(name="Vicky", email="vicky@h.com")
>>> user.id
35
```

**classmethod create_all** (*list_of_kwargs*)
Batch method for creating a list of instances

**Parameters** **list_of_kwargs** (*list of dicts*) – hereA list of dicts where each dict denotes the keyword args that you would pass to the create method separately

#### Examples

```
>>> Customer.create_all([
... {'name': 'Vicky', 'age': 34, 'user_id': 1},
... {'name': 'Ron', 'age': 40, 'user_id': 1, 'gender': 'Male'}])
```

**delete** (*commit=True*)
Deletes a model instance.

#### Examples

```
>>> customer.delete()
```

**classmethod filter** (**criterion*, ***kwargs*)
Same as SQLAlchemy's filter. Additionally this accepts two special keyword arguments `limit` and `reverse` for limiting the results and reversing the order respectively.

**Parameters** ****kwargs** – filter parameters

#### Examples

```
>>> user = User.filter(User.email=="new@x.com")
```

```
>>> shipments = Order.filter(Order.price < 500, limit=3, reverse=True)
```

**classmethod** **filter_by**(***kwargs*)

Same as SQLAlchemy's filter_by. Additionally this accepts two special keyword arguments `limit` and `reverse` for limiting the results and reversing the order respectively.

**Parameters** ****kwargs** – filter parameters

#### Examples

```
>>> user = User.filter_by(email="new@x.com")
```

```
>>> shipments = Shipment.filter_by(country="India", limit=3, reverse=True)
```

**classmethod** **find_or_build**(***kwargs*)

Checks if an instance already exists in db with these kwargs else returns a new, saved instance of the service's model class.

**Parameters** ****kwargs** – instance parameters

**classmethod** **find_or_build_all**(*list_of_kwargs*)

Similar to `find_or_create_all`. But transaction is not committed.

**classmethod** **find_or_create**(***kwargs*)

Checks if an instance already exists by filtering with the kwargs. If yes, returns that instance. If not, creates a new instance with kwargs and returns it

**Parameters**

- ****kwargs** – The keyword arguments which are used for filtering and initialization.
- **keys** (*list, optional*) – A special keyword argument. If passed, only the set of keys mentioned here will be used for filtering. Useful when we want to 'find' based on a subset of the keys and create with all the keys

#### Examples

```
>>> customer = Customer.find_or_create(
...     name="vicky", email="vicky@h.com", country="India")
>>> customer.id
45
>>> customer1 = Customer.find_or_create(
...     name="vicky", email="vicky@h.com", country="India")
>>> customer1==customer
True
>>> customer2 = Customer.find_or_create(
...     name="vicky", email="vicky@h.com", country="Russia")
>>> customer2==customer
False
>>> customer3 = Customer.find_or_create(
...      name="vicky", email="vicky@h.com", country="Russia",
...      keys=['name', 'email'])
```

```
>>> customer3==customer
True
```

**classmethod** **find_or_create_all** (*list_of_kwargs*, *keys=[]*)

Batch method for querying for a list of instances and creating them if required

**Parameters**

- **list_of_kwargs** (*list of dicts*) – A list of dicts where each dict denotes the keyword args that you would pass to the create method separately
- **keys** (*list, optional*) – A list of keys to use for the initial finding step. Matching is done only on these attributes.

**Examples**

```
>>> Customer.find_or_create_all([
... {'name': 'Vicky', 'email': 'vicky@x.com', 'age': 34},
... {'name': 'Ron', 'age': 40, 'email': 'ron@x.com',
... 'gender': 'Male'}], keys=['name', 'email'])
```

**classmethod** **first** (**criterion*, ***kwargs*)

Returns the first instance found of the model class filtered by the specified criterion and/or key word arguments. Return None if no result found.

**Examples**

```
>>> will = User.first(name="Will")
```

**classmethod** **get** (*keyval*, *key='id'*, *user_id=None*)

Fetches a single instance which has value `keyval` for the attribute `key`.

**Parameters**

- **keyval** – The value of the attribute.
- **key** (*str, optional*) – The attribute to search by. By default, it is 'id'.

**Returns** A model instance if found. Else None.

**Examples**

```
>>> User.get(35)
user35@i.com
```

```
>>> User.get('user35@i.com', key='email')
user35@i.com
```

**classmethod** **get_all** (*keyvals*, *key='id'*, *user_id=None*)

Works like a map function from keyvals to instances.

**Parameters**

- **keyvals** (*list*) – The list of values of the attribute.
- **key** (*str, optional*) – The attribute to search by. By default, it is 'id'.

**Returns** A list of model instances, in the same order as the list of keyvals.

**Return type** list

#### Examples

```
>>> User.get_all([2,5,7, 8000, 11])
user2@i.com, user5@i.com, user7@i.com, None, user11@i.com
```

```
>>> User.get_all(['user35@i.com', 'user5@i.com'], key='email')
user35@i.com, user5@i.com
```

**classmethod get_and_setattr**(*id*, ***kwargs*)
Returns an updated instance of the service's model class.

**Parameters**

- **model** – the model to update
- ****kwargs** – update parameters

**classmethod get_and_update**(*id*, ***kwargs*)
Returns an updated instance of the service's model class.

**Parameters**

- **model** – the model to update
- ****kwargs** – update parameters

**classmethod get_or_404**(*id*)
Same as Flask-SQLAlchemy's `get_or_404`.

**classmethod last**(**criterion*, ***kwargs*)
Returns the last instance matching the criterion and/or keyword arguments.

#### Examples

last_male_user = User.last(gender="male")

**classmethod new**(***kwargs*)
Returns a new, unsaved instance of the model class.

**classmethod one**(**criterion*, ***kwargs*)
Similar to `first`. But throws an exception if no result is found.

#### Examples

```
>>> user = User.one(name="here")
```

**Raises** `NoResultFound` – No row was found for one()

**save**()
Saves a model instance to db.

#### Examples

```
>>> customer = Customer.new(name="hari")
>>> customer.save()
```

**update**(***kwargs*)
Updates an instance.

**Parameters** ****kwargs** – Arbitrary keyword arguments. Column names are keywords and their new values are the values.

#### Examples

```
>>> customer.update(email="newemail@x.com", name="new")
```

**classmethod update_all**(**criterion*, ***kwargs*)
Batch method for updating all instances obeying the criterion

**Parameters**

- ***criterion** – SQLAlchemy query criterion for filtering what instances to update
- ****kwargs** – The parameters to be updated

#### Examples

```
>>> User.update_all(active=True)
```

```
>>> Customer.update_all(Customer.country=='India', active=True)
```

The second example sets active=True for all customers with country India.

**classmethod update_or_build_all**(*list_of_kwargs*, *keys=[]*)
Batch method for updating a list of instances and creating them if required

**Parameters**

- **list_of_kwargs** (*list of dicts*) – A list of dicts where each dict denotes the keyword args that you would pass to the create method separately
- **keys** (*list, optional*) – A list of keys to use for the initial finding step. Matching is done only on these attributes.

#### Examples

```
>>> Customer.update_or_create_all([
... {'name': 'Vicky', 'email': 'vicky@x.com', 'age': 34},
... {'name': 'Ron', 'age': 40, 'email': 'ron@x.com',
... 'gender': 'Male'}], keys=['name', 'email'])
```

**classmethod update_or_create**(***kwargs*)
Checks if an instance already exists by filtering with the kwargs. If yes, updates the instance with new kwargs and returns that instance. If not, creates a new instance with kwargs and returns it.

**Parameters**

- ****kwargs** – The keyword arguments which are used for filtering and initialization.

- **keys** (*list, optional*) – A special keyword argument. If passed, only the set of keys mentioned here will be used for filtering. Useful when we want to 'filter' based on a subset of the keys and create with all the keys.

#### Examples

```
>>> customer = Customer.update_or_create(
...     name="vicky", email="vicky@h.com", country="India")
>>> customer.id
45
>>> customer1 = Customer.update_or_create(
...     name="vicky", email="vicky@h.com", country="India")
>>> customer1==customer
True
>>> customer2 = Customer.update_or_create(
...     name="vicky", email="vicky@h.com", country="Russia")
>>> customer2==customer
False
>>> customer3 = Customer.update_or_create(
...      name="vicky", email="vicky@h.com", country="Russia",
...      keys=['name', 'email'])
>>> customer3==customer
True
```

**classmethod** **update_or_create_all**(*list_of_kwargs*, *keys=[]*)

Batch method for updating a list of instances and creating them if required

**Parameters**

- **list_of_kwargs** (*list of dicts*) – A list of dicts where each dict denotes the keyword args that you would pass to the create method separately
- **keys** (*list, optional*) – A list of keys to use for the initial finding step. Matching is done only on these attributes.

#### Examples

```
>>> Customer.update_or_create_all([
... {'name': 'Vicky', 'email': 'vicky@x.com', 'age': 34},
... {'name': 'Ron', 'age': 40, 'email': 'ron@x.com',
... 'gender': 'Male'}], keys=['name', 'email'])
```

**class** flask_sqlalchemy_booster.dictizable_mixin.**DictizableMixin**

Methods for converting Model instance to dict and json.

**_attrs_to_serialize_**

*list of str* – The columns which should be serialized as a part of the output dictionary

**_key_modifications_**

*dict of str,str* – A dictionary used to map the display names of columns whose original name we want to be modified in the json

**_rels_to_serialize_**

*list of tuple of str* – A list of tuples. The first element of the tuple is the relationship that is to be serialized. The second element it the name of the attribute in the related model, the value of which is to be used as the representation

**_rels_to_expand_**
*list of str* – A list of relationships to expand. You can specify nested relationships by placing dots.

**_group_listrels_by_**
*dict of str, list of str* – A dictionary representing how to hierarchially group a list like relationship. The relationship fields are the keys and the list of the attributes based on which they are to be grouped are the values.

**serialize_attrs**(**args*)
Converts and instance to a dictionary with only the specified attributes as keys

**Parameters** ***args** (*list*) – The attributes to serialize

#### Examples

```
>>> customer = Customer.create(name="James Bond", email="007@mi.com",
                               phone="007", city="London")
>>> customer.serialize_attrs('name', 'email')
{'name': u'James Bond', 'email': u'007@mi.com'}
```

**to_serializable_dict**(*attrs_to_serialize=None*, *rels_to_expand=None*, *rels_to_serialize=None*, *key_modifications=None*)
An alias for `todict`

**todict**(*attrs_to_serialize=None*, *rels_to_expand=None*, *rels_to_serialize=None*, *group_listrels_by=None*, *key_modifications=None*)
Converts an instance to a dictionary form

**Parameters**

- **attrs_to_serialize** (*list of str*) – The columns which should be serialized as a part of the output dictionary
- **key_modifications** (*dict of str,str*) – A dictionary used to map the display names of columns whose original name we want to be modified in the json
- **rels_to_serialize** (*list of tuple of str*) – A list of tuples. The first element of the tuple is the relationship that is to be serialized. The second element it the name of the attribute in the related model, the value of which is to be used as the representation
- **rels_to_expand** (*list of str*) – A list of relationships to expand. You can specify nested relationships by placing dots.
- **group_listrels_by** (*dict of str, list of str*) – A dictionary representing how to hierarchially group a list like relationship. The relationship fields are the keys and the list of the attributes based on which they are to be grouped are the values.

### 2.4.3 Responses

flask_sqlalchemy_booster.responses.**as_list**(*func*)
A decorator used to return a JSON response of a list of model objects. It expects the decorated function to return a list of model instances. It then converts the instances to dicts and serializes them into a json response

#### Examples

```
>>> @app.route('/api')
... @as_list
... def list_customers():
...     return Customer.all()
```

flask_sqlalchemy_booster.responses.**as_obj**(*func*)

A decorator used to return a JSON response with a dict representation of the model instance. It expects the decorated function to return a Model instance. It then converts the instance to dicts and serializes it into a json response

#### Examples

```
>>> @app.route('/api/shipments/<id>')
... @as_obj
... def get_shipment(id):
...     return Shipment.get(id)
```

flask_sqlalchemy_booster.responses.**as_processed_list**(*func*)

A decorator used to return a JSON response of a list of model objects. It differs from `as_list` in that it accepts a variety of querying parameters and can use them to filter and modify the results. It expects the decorated function to return either Model Class to query or a SQLAlchemy filter which exposes a subset of the instances of the Model class. It then converts the instances to dicts and serializes them into a json response

#### Examples

```
>>> @app.route('/api/customers')
... @as_processed_list
... def list_all_customers():
...     return Customer
```

```
>>> @app.route('/api/editors')
... @as_processed_list
... def list_editors():
...     return User.filter(role='editor')
```

flask_sqlalchemy_booster.responses.**jsoned**(*struct*, *wrap=True*, *meta=None*, *struct_key='result'*)

Provides a json dump of the struct

**Parameters**

- **struct** – The data to dump
- **wrap** (*bool, optional*) – Specify whether to wrap the struct in an enclosing dict
- **struct_key** (*str, optional*) – The string key which will contain the struct in the result dict
- **meta** (*dict, optional*) – An optional dictonary to merge with the output dictionary.

#### Examples

```
>>> jsoned([3,4,5])
... '{"status": "success", "result": [3, 4, 5]}'
```

```
    >>> jsoned([3,4,5], wrap=False)
    ... '[3, 4, 5]'
```

flask_sqlalchemy_booster.responses.**serializable_list**(*olist*, *attrs_to_serialize=None*, *rels_to_expand=None*, *group_listrels_by=None*, *rels_to_serialize=None*, *key_modifications=None*, *groupby=None*, *keyvals_to_merge=None*, *preserve_order=False*)

Converts a list of model instances to a list of dictionaries using their `todict` method.

**Parameters**

- **olist** (*list*) – The list of instances to convert
- **attrs_to_serialize** (*list, optional*) – To be passed as an argument to the `todict` method
- **rels_to_expand** (*list, optional*) – To be passed as an argument to the `todict` method
- **group_listrels_by** (*dict, optional*) – To be passed as an argument to the `todict` method
- **rels_to_serialize** (*list, optional*) – To be passed as an argument to the `todict` method
- **key_modifications** (*dict, optional*) – To be passed as an argument to the `todict` method
- **groupby** (*list, optional*) – An optional list of keys based on which the result list will be hierarchially grouped ( and converted

  into a dict)
- **keyvals_to_merge** (*list of dicts, optional*) – A list of parameters to be merged with each dict of the output list

flask_sqlalchemy_booster.responses.**serialized_list**(*olist*, ***kwargs*)

Misnamed. Should be deprecated eventually.

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

## Symbols



## A



## B



## C



## D



## F



## G