Flask-SQLAlchemy Documentation

Release 2.0

May 23, 2015

CONTENTS

Flask-SQLAlchemy is an extension for Flask that adds support for SQLAlchemy to your application. It requires SQLAlchemy 0.6 or higher. It aims to simplify using SQLAlchemy with Flask by providing useful defaults and extra helpers that make it easier to accomplish common tasks.

Part I

USER'S GUIDE

This part of the documentation will show you how to get started in using Flask-SQLAlchemy with Flask.

CHAPTER

ONE

QUICKSTART

Flask-SQLAlchemy is fun to use, incredibly easy for basic applications, and readily extends for larger applications. For the complete guide, checkout the API documentation on the *SQLAlchemy* class.

1.1 A Minimal Application

For the common case of having one Flask application all you have to do is to create your Flask application, load the configuration of choice and then create the *SQLAlchemy* object by passing it the application.

Once created, that object then contains all the functions and helpers from both `sqlalchemy` and `sqlalchemy.orm`. Furthermore it provides a class called *Model* that is a declarative base which can be used to declare models:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__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'>
```

1.2 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!'>]
```

1.3 Road to Enlightenment

The only things you need to know compared to plain SQLAlchemy are:

1. *SQLAlchemy* gives you access to the following things:
   - all the functions and classes from `sqlalchemy` and `sqlalchemy.orm`
   - a preconfigured scoped session called *session*
   - the *metadata*
   - the *engine*
   - a *SQLAlchemy.create_all()* and *SQLAlchemy.drop_all()* methods to create and drop tables according to the models.
   - a *Model* baseclass that is a configured declarative base.

2. The *Model* declarative base class behaves like a regular Python class but has a *query* attribute attached that can be used to query the model. (*Model* and *BaseQuery*)
3. You have to commit the session, but you don't have to remove it at the end of the request, Flask-SQLAlchemy does that for you.

CHAPTER

TWO

INTRODUCTION INTO CONTEXTS

If you are planning on using only one application you can largely skip this chapter. Just pass your application to the *SQLAlchemy* constructor and you're usually set. However if you want to use more than one application or create the application dynamically in a function you want to read on.

If you define your application in a function, but the *SQLAlchemy* object globally, how does the latter learn about the former? The answer is the *init_app()* function:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

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

What it does is prepare the application to work with *SQLAlchemy*. However that does not now bind the *SQLAlchemy* object to your application. Why doesn't it do that? Because there might be more than one application created.

So how does *SQLAlchemy* now really know about your application? You will have to setup an application context. If you are working inside a Flask view function, that automatically happens. However if you are working inside the interactive shell, you will have to do that yourself (see Creating an Application Context).

In a nutshell, do something like this:

```
>>> from yourapp import create_app
>>> app = create_app()
>>> app.app_context().push()
```

Inside scripts it makes also sense to use the with-statement:

```
def my_function():
    with app.app_context():
        user = db.User(...)
```

```
        db.session.add(user)
        db.session.commit()
```

Some functions inside Flask-SQLAlchemy also accept optionally the application to operate on:

```
>>> from yourapp import db, create_app
>>> db.create_all(app=create_app())
```

CHAPTER

THREE

CONFIGURATION

The following configuration values exist for Flask-SQLAlchemy. Flask-SQLAlchemy loads these values from your main Flask config which can be populated in various ways. Note that some of those cannot be modified after the engine was created so make sure to configure as early as possible and to not modify them at runtime.

3.1 Configuration Keys

A list of configuration keys currently understood by the extension:

SQLALCHEMY_DATABASE_URI	The database URI that should be used for the connection. Examples: • sqlite:////tmp/test.db • mysql://username:password@server/db
SQLALCHEMY_BINDS	A dictionary that maps bind keys to SQLAlchemy connection URIs. For more information about binds see *Multiple Databases with Binds*.
SQLALCHEMY_ECHO	If set to *True* SQLAlchemy will log all the statements issued to stderr which can be useful for debugging.
SQLALCHEMY_RECORD_QUERIES	Can be used to explicitly disable or enable query recording. Query recording automatically happens in debug or testing mode. See get_debug_queries() for more information.
SQLALCHEMY_NATIVE_UNICODE	Can be used to explicitly disable native unicode support. This is required for some database adapters (like PostgreSQL on some Ubuntu versions) when used with improper database defaults that specify encoding-less databases.
SQLALCHEMY_POOL_SIZE	The size of the database pool. Defaults to the engine's default (usually 5)
SQLALCHEMY_POOL_TIMEOUT	Specifies the connection timeout for the pool. Defaults to 10.
SQLALCHEMY_POOL_RECYCLE	Number of seconds after which a connection is automatically recycled. This is required for MySQL, which removes connections after 8 hours idle by default. Note that Flask-SQLAlchemy automatically sets this to 2 hours if MySQL is used.
SQLALCHEMY_MAX_OVERFLOW	Controls the number of connections that can be created after the pool reached its maximum size. When those additional connections are returned to the pool, they are disconnected and discarded.
SQLALCHEMY_TRACK_MODIFICATIONS	If set to True, Flask-SQLAlchemy will track modifications of objects and emit signals. The default is None, which enables tracking but issues a warning that it will be disabled by default in the future. This requires extra memory and should be disabled if not needed.

New in version 0.8: The SQLALCHEMY_NATIVE_UNICODE, SQLALCHEMY_POOL_SIZE, SQLALCHEMY_POOL_TIMEOUT and SQLALCHEMY_POOL_RECYCLE configuration keys were added.

New in version 0.12: The SQLALCHEMY_BINDS configuration key was added.

New in version 0.17: The `SQLALCHEMY_MAX_OVERFLOW` configuration key was added.

New in version 2.0: The `SQLALCHEMY_TRACK_MODIFICATIONS` configuration key was added.

Changed in version 2.1: `SQLALCHEMY_TRACK_MODIFICATIONS` will warn if unset.

3.2 Connection URI Format

For a complete list of connection URIs head over to the SQLAlchemy documentation under (Supported Databases). This here shows some common connection strings.

SQLAlchemy indicates the source of an Engine as a URI combined with optional keyword arguments to specify options for the Engine. The form of the URI is:

```
dialect+driver://username:password@host:port/database
```

Many of the parts in the string are optional. If no driver is specified the default one is selected (make sure to *not* include the + in that case).

Postgres:

```
postgresql://scott:tiger@localhost/mydatabase
```

MySQL:

```
mysql://scott:tiger@localhost/mydatabase
```

Oracle:

```
oracle://scott:tiger@127.0.0.1:1521/sidname
```

SQLite (note the four leading slashes):

```
sqlite:////absolute/path/to/foo.db
```

3.3 Using custom MetaData and naming conventions

You can optionally construct the `SQLAlchemy` object with a custom `MetaData` object. This allows you to, among other things, specify a custom constraint naming convention. Doing so is important for dealing with database migrations (for instance using alembic as stated here. Since SQL defines no standard naming conventions, there is no guaranteed nor effective compatibility by default among database implementations. You can define a custom naming convention like this as suggested by the SQLAlchemy docs:

```
from sqlalchemy import MetaData
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy
```

```
convention = {
    "ix": 'ix_%(column_0_label)s',
    "uq": "uq_%(table_name)s_%(column_0_name)s",
    "ck": "ck_%(table_name)s_%(constraint_name)s",
    "fk": "fk_%(table_name)s_%(column_0_name)s_%(referred_table_name)s",
    "pk": "pk_%(table_name)s"
}

metadata = MetaData(naming_convention=convention)
db = SQLAlchemy(app, metadata=metadata)
```

For more info about `MetaData`, check out the official docs on it.

CHAPTER

FOUR

DECLARING MODELS

Generally Flask-SQLAlchemy behaves like a properly configured declarative base from the `declarative` extension. As such we recommend reading the SQLAlchemy docs for a full reference. However the most common use cases are also documented here.

Things to keep in mind:

- The baseclass for all your models is called *db.Model*. It's stored on the SQLAlchemy instance you have to create. See *Quickstart* for more details.
- Some parts that are required in SQLAlchemy are optional in Flask-SQLAlchemy. For instance the table name is automatically set for you unless overridden. It's derived from the class name converted to lowercase and with "CamelCase" converted to "camel_case".

4.1 Simple Example

A very simple example:

```
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
```

Use `Column` to define a column. The name of the column is the name you assign it to. If you want to use a different name in the table you can provide an optional first argument which is a string with the desired column name. Primary keys are marked with `primary_key=True`. Multiple keys can be marked as primary keys in which case they become a compound primary key.

The types of the column are the first argument to `Column`. You can either provide them directly or call them to further specify them (like providing a length). The following types are the most common:

Integer	an integer
String (size)	a string with a maximum length
Text	some longer unicode text
DateTime	date and time expressed as Python `datetime` object.
Float	stores floating point values
Boolean	stores a boolean value
PickleType	stores a pickled Python object
LargeBinary	stores large arbitrary binary data

4.2 One-to-Many Relationships

The most common relationships are one-to-many relationships. Because relationships are declared before they are established you can use strings to refer to classes that are not created yet (for instance if *Person* defines a relationship to *Article* which is declared later in the file).

Relationships are expressed with the `relationship()` function. However the foreign key has to be separately declared with the `sqlalchemy.schema.ForeignKey` class:

```
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50))
    addresses = db.relationship('Address', backref='person',
                                lazy='dynamic')

class Address(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(50))
    person_id = db.Column(db.Integer, db.ForeignKey('person.id'))
```

What does `db.relationship()` do? That function returns a new property that can do multiple things. In this case we told it to point to the *Address* class and load multiple of those. How does it know that this will return more than one address? Because SQLAlchemy guesses a useful default from your declaration. If you would want to have a one-to-one relationship you can pass `uselist=False` to `relationship()`.

So what do *backref* and *lazy* mean? *backref* is a simple way to also declare a new property on the *Address* class. You can then also use `my_address.person` to get to the person at that address. *lazy* defines when SQLAlchemy will load the data from the database:

- `'select'` (which is the default) means that SQLAlchemy will load the data as necessary in one go using a standard select statement.
- `'joined'` tells SQLAlchemy to load the relationship in the same query as the parent using a *JOIN* statement.

- `'subquery'` works like `'joined'` but instead SQLAlchemy will use a subquery.
- `'dynamic'` is special and useful if you have many items. Instead of loading the items SQLAlchemy will return another query object which you can further refine before loading the items. This is usually what you want if you expect more than a handful of items for this relationship.

How do you define the lazy status for backrefs? By using the `backref()` function:

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50))
    addresses = db.relationship('Address',
        backref=db.backref('person', lazy='joined'), lazy='dynamic')
```

4.3 Many-to-Many Relationships

If you want to use many-to-many relationships you will need to define a helper table that is used for the relationship. For this helper table it is strongly recommended to *not* use a model but an actual table:

```
tags = db.Table('tags',
    db.Column('tag_id', db.Integer, db.ForeignKey('tag.id')),
    db.Column('page_id', db.Integer, db.ForeignKey('page.id'))
)

class Page(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    tags = db.relationship('Tag', secondary=tags,
        backref=db.backref('pages', lazy='dynamic'))

class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```

Here we configured *Page.tags* to be a list of tags once loaded because we don't expect too many tags per page. The list of pages per tag (*Tag.pages*) however is a dynamic backref. As mentioned above this means that you will get a query object back you can use to fire a select yourself.

CHAPTER

FIVE

SELECT, INSERT, DELETE

Now that you have *declared models* it's time to query the data from the database. We will be using the model definitions from the *Quickstart* chapter.

5.1 Inserting Records

Before we can query something we will have to insert some data. All your models should have a constructor, so make sure to add one if you forgot. Constructors are only used by you, not by SQLAlchemy internally so it's entirely up to you how you define them.

Inserting data into the database is a three step process:

1. Create the Python object
2. Add it to the session
3. Commit the session

The session here is not the Flask session, but the Flask-SQLAlchemy one. It is essentially a beefed up version of a database transaction. This is how it works:

```
>>> from yourapp import User
>>> me = User('admin', 'admin@example.com')
>>> db.session.add(me)
>>> db.session.commit()
```

Alright, that was not hard. What happens at what point? Before you add the object to the session, SQLAlchemy basically does not plan on adding it to the transaction. That is good because you can still discard the changes. For example think about creating the post at a page but you only want to pass the post to the template for preview rendering instead of storing it in the database.

The `add()` function call then adds the object. It will issue an *INSERT* statement for the database but because the transaction is still not committed you won't get an ID back immediately. If you do the commit, your user will have an ID:

```
>>> me.id
1
```

5.2 Deleting Records

Deleting records is very similar, instead of `add()` use `delete()`:

```
>>> db.session.delete(me)
>>> db.session.commit()
```

5.3 Querying Records

So how do we get data back out of our database? For this purpose Flask-SQLAlchemy provides a *`query`* attribute on your *`Model`* class. When you access it you will get back a new query object over all records. You can then use methods like `filter()` to filter the records before you fire the select with `all()` or `first()`. If you want to go by primary key you can also use `get()`.

The following queries assume following entries in the database:

id	*username*	*email*
1	admin	admin@example.com
2	peter	peter@example.org
3	guest	guest@example.com

Retrieve a user by username:

```
>>> peter = User.query.filter_by(username='peter').first()
>>> peter.id
1
>>> peter.email
u'peter@example.org'
```

Same as above but for a non existing username gives *None*:

```
>>> missing = User.query.filter_by(username='missing').first()
>>> missing is None
True
```

Selecting a bunch of users by a more complex expression:

```
>>> User.query.filter(User.email.endswith('@example.com')).all()
[<User u'admin'>, <User u'guest'>]
```

Ordering users by something:

```
>>> User.query.order_by(User.username)
[<User u'admin'>, <User u'guest'>, <User u'peter'>]
```

Limiting users:

```
>>> User.query.limit(1).all()
[<User u'admin'>]
```

Getting user by primary key:

```
>>> User.query.get(1)
<User u'admin'>
```

5.4 Queries in Views

If you write a Flask view function it's often very handy to return a 404 error for missing entries. Because this is a very common idiom, Flask-SQLAlchemy provides a helper for this exact purpose. Instead of `get()` one can use `get_or_404()` and instead of `first()` `first_or_404()`. This will raise 404 errors instead of returning *None*:

```
@app.route('/user/<username>')
def show_user(username):
    user = User.query.filter_by(username=username).first_or_404()
    return render_template('show_user.html', user=user)
```

CHAPTER

SIX

MULTIPLE DATABASES WITH BINDS

Starting with 0.12 Flask-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`).

6.1 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'
}
```

6.2 Creating and Dropping Tables

The *create_all()* and *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)
```

6.3 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.

CHAPTER

SEVEN

SIGNALLING SUPPORT

Connect to the following signals to get notified before and after changes are committed to the database. These changes are only tracked if `SQLALCHEMY_TRACK_MODIFICATIONS` is enabled in the config.

New in version 0.10.

Changed in version 2.1: `before_models_committed` is triggered correctly.

Deprecated since version 2.1: This will be disabled by default in a future version.

`models_committed`
: This signal is sent when changed models were committed to the database.

  The sender is the application that emitted the changes. The receiver is passed the `changes` parameter with a list of tuples in the form `(model instance, operation)`.

  The operation is one of `'insert'`, `'update'`, and `'delete'`.

`before_models_committed`
: This signal works exactly like *`models_committed`* but is emitted before the commit takes place.

Part II
API REFERENCE

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

CHAPTER

EIGHT

API

This part of the documentation documents all the public classes and functions in Flask-SQLAlchemy.

8.1 Configuration

class `flask.ext.sqlalchemy.SQLAlchemy`(*app=None*, *use_native_unicode=True*, *session_options=None*, *metadata=None*)

This class is used to control the SQLAlchemy integration to one or more Flask applications. Depending on how you initialize the object it is usable right away or will attach as needed to a Flask application.

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

```
app = Flask(__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 = Flask(__name__)
    db.init_app(app)
    return app
```

The difference between the two is that in the first case methods like *create_all()* and *drop_all()* will work all the time but in the second case a `flask.Flask.app_context()` has to exist.

By default Flask-SQLAlchemy will apply some backend-specific settings to improve your experience with them. As of SQLAlchemy 0.6 SQLAlchemy will probe the library for native unicode support. If it detects unicode it will let the library handle that, otherwise do that itself. Sometimes this detection can fail in which case you might want to set *use_native_unicode* (or the

SQLALCHEMY_NATIVE_UNICODE configuration key) to *False*. Note that the configuration key overrides the value you pass to the constructor.

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))
```

You can still use sqlalchemy and sqlalchemy.orm directly, but note that Flask-SQLAlchemy customizations are available only through an instance of this *SQLAlchemy* class. Query classes default to *BaseQuery* for *db.Query*, *db.Model.query_class*, and the default query_class for *db.relationship* and *db.backref*. If you use these interfaces through sqlalchemy and sqlalchemy.orm directly, the default query class will be that of sqlalchemy.

Check types carefully

Don't perform type or *isinstance* checks against *db.Table*, which emulates *Table* behavior but is not a class. *db.Table* exposes the *Table* interface, but is a function which allows omission of metadata.

You may also define your own SessionExtension instances as well when defining your SQLAlchemy class instance. You may pass your custom instances to the *session_extensions* keyword. This can be either a single SessionExtension instance, or a list of SessionExtension instances. In the following use case we use the VersionedListener from the SQLAlchemy versioning examples.:

```
from history_meta import VersionedMeta, VersionedListener

app = Flask(__name__)
db = SQLAlchemy(app, session_extensions=[VersionedListener()])

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

The *session_options* parameter can be used to override session options. If provided it's a dict of parameters passed to the session's constructor.

New in version 0.10: The *session_options* parameter was added.

New in version 0.16: *scopefunc* is now accepted on *session_options*. It allows specifying a custom function which will define the SQLAlchemy session's scoping.

New in version 2.1: The *metadata* parameter was added. This allows for setting custom naming conventions among other, non-trivial things.

Query
: The *BaseQuery* class.

`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*.

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

  Changed in version 0.12: Parameters were added

`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 *`SignallingSession`*.

  New in version 2.0.

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

  Changed in version 0.12: Parameters were added

`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.

  New in version 0.12.

`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`(*metadata=None*)
Creates the declarative base.

`metadata`
Returns the metadata

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

Changed in version 0.12: Parameters were added

8.2 Models

class `flask.ext.sqlalchemy.Model`
Baseclass for custom user models.

`__bind_key__`
Optionally declares the bind to use. *None* refers to the default bind. For more information see *Multiple Databases with Binds*.

`__tablename__`
The name of the table in the database. This is required by SQLAlchemy; however, Flask-SQLAlchemy will set it automatically if a model has a primary key defined. If the `__table__` or `__tablename__` is set explicitly, that will be used instead.

`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 *BaseQuery* is used.

alias of *BaseQuery*

class `flask.ext.sqlalchemy.BaseQuery`(*entities*, *session=None*)
The default query object used for models, and exposed as *Query*. This can be subclassed and replaced for individual models by setting the *query_class* attribute. This is a subclass of a standard SQLAlchemy `Query` class and has all the methods of a standard query as well.

`all`()
Return the results represented by this query as a list. This results in an execution of the underlying query.

`order_by`(**criterion*)
apply one or more ORDER BY criterion to the query and return the newly resulting query.

`limit(`*limit*`)`
Apply a LIMIT to the query and return the newly resulting query.

`offset(`*offset*`)`
Apply an OFFSET to the query and return the newly resulting query.

`first()`
Return the first result of this query or *None* if the result doesn't contain any rows. This results in an execution of the underlying query.

`first_or_404()`
Like *`first()`* but aborts with 404 if not found instead of returning *None*.

`get(`*ident*`)`
Return an instance based on the given primary key identifier, or `None` if not found.

E.g.:

```
my_user = session.query(User).get(5)

some_object = session.query(VersionedFoo).get((5, 10))
```

`get()` is special in that it provides direct access to the identity map of the owning `Session`. If the given primary key identifier is present in the local identity map, the object is returned directly from this collection and no SQL is emitted, unless the object has been marked fully expired. If not present, a SELECT is performed in order to locate the object.

`get()` also will perform a check if the object is present in the identity map and marked as expired - a SELECT is emitted to refresh the object as well as to ensure that the row is still present. If not, `ObjectDeletedError` is raised.

`get()` is only used to return a single mapped instance, not multiple instances or individual column constructs, and strictly on a single primary key value. The originating `Query` must be constructed in this way, i.e. against a single mapped entity, with no additional filtering criterion. Loading options via `options()` may be applied however, and will be used if the object is not yet locally present.

A lazy-loading, many-to-one attribute configured by `relationship()`, using a simple foreign-key-to-primary-key criterion, will also use an operation equivalent to `get()` in order to retrieve the target value from the local identity map before querying the database. See `/orm/loading_relationships` for further details on relationship loading.

Parameters `ident` – A scalar or tuple value representing the primary key. For a composite primary key, the order of identifiers corresponds in most cases to that of the mapped `Table` object's primary key columns. For a `mapper()` that was given the `primary key` argument during construction, the order of identifiers corresponds to the elements present in this collection.

Returns The object instance, or `None`.

`get_or_404`(*ident*)
Like *`get()`* but aborts with 404 if not found instead of returning *None*.

`paginate`(*page=None*, *per_page=None*, *error_out=True*)
Returns *per_page* items from page *page*. By default it will abort with 404 if no items were found and the page was larger than 1. This behavor can be disabled by setting *error_out* to *False*.

If page or per_page are None, they will be retrieved from the request query. If the values are not ints and `error_out` is true, it will abort with 404. If there is no request or they aren't in the query, they default to page 1 and 20 respectively.

Returns an *`Pagination`* object.

8.3 Sessions

class `flask.ext.sqlalchemy.SignallingSession`(*db*, *autocommit=False*, *autoflush=True*, ***options*)
The signalling session is the default session that Flask-SQLAlchemy uses. It extends the default session system with bind selection and modification tracking.

If you want to use a different session you can override the *`SQLAlchemy.create_session()`* function.

New in version 2.0.

`app` **= None**
The application that this session belongs to.

8.4 Utilities

class `flask.ext.sqlalchemy.Pagination`(*query*, *page*, *per_page*, *total*, *items*)
Internal helper class returned by *`BaseQuery.paginate()`*. You can also construct it from any other SQLAlchemy query object if you are working with other libraries. Additionally it is possible to pass *None* as query object in which case the *`prev()`* and *`next()`* will no longer work.

`has_next`
True if a next page exists.

`has_prev`
True if a previous page exists

`items` **= None**
the items for the current page

`iter_pages`(*left_edge=2*, *left_current=2*, *right_current=5*, *right_edge=2*)
Iterates over the page numbers in the pagination. The four parameters control the thresholds how many numbers should be produced from the sides.

Skipped page numbers are represented as *None*. This is how you could render such a pagination in the templates:

```
{% macro render_pagination(pagination, endpoint) %}
  <div class=pagination>
  {%- for page in pagination.iter_pages() %}
    {% if page %}
      {% if page != pagination.page %}
        <a href="{{ url_for(endpoint, page=page) }}">{{ page }}</a>
      {% else %}
        <strong>{{ page }}</strong>
      {% endif %}
    {% else %}
      <span class=ellipsis>...</span>
    {% endif %}
  {%- endfor %}
  </div>
{% endmacro %}
```

`next`(*error_out=False*)
Returns a *Pagination* object for the next page.

`next_num`
Number of the next page

`page` = **None**
the current page number (1 indexed)

`pages`
The total number of pages

`per_page` = **None**
the number of items to be displayed on a page.

`prev`(*error_out=False*)
Returns a *Pagination* object for the previous page.

`prev_num`
Number of the previous page.

`query` = **None**
the unlimited query object that was used to create this pagination object.

`total` = **None**
the total number of items matching the query

`flask.ext.sqlalchemy.get_debug_queries()`
In debug mode Flask-SQLAlchemy will log all the SQL queries sent to the database. This information is available until the end of request which makes it possible to easily ensure that the SQL generated is the one expected on errors or in unittesting. If you don't want to enable the DEBUG mode for your unittests you can also enable the query recording by setting the `'SQLALCHEMY_RECORD_QUERIES'` config variable to *True*. This is automatically enabled if Flask is in testing mode.

The value returned will be a list of named tuples with the following attributes:

statement The SQL statement issued

parameters The parameters for the SQL statement

start_time* / *end_time Time the query started / the results arrived. Please keep in mind that the timer function used depends on your platform. These values are only useful for sorting or comparing. They do not necessarily represent an absolute timestamp.

duration Time the query took in seconds

context A string giving a rough estimation of where in your application query was issued. The exact format is undefined so don't try to reconstruct filename or function name.

Part III

ADDITIONAL NOTES

See Flask's license for legal information governing this project.

CHAPTER

NINE

CHANGELOG

Here you can see the full list of changes between each Flask-SQLAlchemy release.

9.1 Version 2.1

In development, codename Caesium

- Table names are automatically generated in more cases, including subclassing mixins and abstract models.
- Allow using a custom MetaData object.

9.2 Version 2.0

Released on August 29th 2014, codename Bohrium

- Changed how the builtin signals are subscribed to skip non Flask-SQLAlchemy sessions. This will also fix the attribute error about model changes not existing.
- Added a way to control how signals for model modifications are tracked.
- Made the `SignallingSession` a public interface and added a hook for customizing session creation.
- If the `bind` parameter is given to the signalling session it will no longer cause an error that a parameter is given twice.
- Added working table reflection support.
- Enabled autoflush by default.
- Consider `SQLALCHEMY_COMMIT_ON_TEARDOWN` harmful and remove from docs.

9.3 Version 1.0

Released on July 20th 2013, codename Aurum

- Added Python 3.3 support.
- Dropped 2.5 compatibility.
- Various bugfixes
- Changed versioning format to do major releases for each update now.

9.4 Version 0.16

- New distribution format (flask_sqlalchemy)
- Added support for Flask 0.9 specifics.

9.5 Version 0.15

- Added session support for multiple databases

9.6 Version 0.14

- Make relative sqlite paths relative to the application root.

9.7 Version 0.13

- Fixed an issue with Flask-SQLAlchemy not selecting the correct binds.

9.8 Version 0.12

- Added support for multiple databases.
- Expose Flask-SQLAlchemy's BaseQuery as *db.Query*.
- Set default query_class for *db.relation*, *db.relationship*, and *db.dynamic_loader* to Flask-SQLAlchemy's BaseQuery.
- Improved compatibility with Flask 0.7.

9.9 Version 0.11

- Fixed a bug introduced in 0.10 with alternative table constructors.

9.10 Version 0.10

- Added support for signals.
- Table names are now automatically set from the class name unless overriden.
- Model.query now always works for applications directly passed to the SQLAlchemy constructor. Furthermore the property now raises an RuntimeError instead of being None.
- added session options to constructor.
- fixed a broken *__repr__*
- *db.Table* is now a factor function that creates table objects. This makes it possible to omit the metadata.

9.11 Version 0.9

- applied changes to pass the Flask extension approval process.

9.12 Version 0.8

- added a few configuration keys for creating connections.
- automatically activate connection recycling for MySQL connections.
- added support for the Flask testing mode.

9.13 Version 0.7

- Initial public release

INDEX