Flask-SQLAlchemy Documentation

Release 2.2.dev

Jul 09, 2017

Contents

Flask-SQLAlchemy is an extension for Flask that adds support for SQLAlchemy to your application. It requires SQLAlchemy 0.8 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.

See the SQLAlchemy documentation to learn how to work with the ORM in depth. The following documentation is a brief overview of the most common tasks, as well as the features specific to Flask-SQLAlchemy.

Part I

USER'S GUIDE

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

CHAPTER 1

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.

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, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
```

```
    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(username='admin', email='admin@example.com')
>>> guest = User(username='guest', email='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:

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

Note how we never defined a `__init__` method on the `User` class? That's because SQLAlchemy adds an implicit constructor to all model classes which accepts keyword arguments for all its columns and relationships. If you decide to override the constructor for any reason, make sure to keep accepting `**kwargs` and call the super constructor with those `**kwargs` to preserve this behavior:

```
class Foo(db.Model):
    # ...
    def __init__(**kwargs):
        super(Foo, self).__init__(**kwargs)
        # do custom stuff
```

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), nullable=False)
    body = db.Column(db.Text, nullable=False)
    pub_date = db.Column(db.DateTime, nullable=False,
        default=datetime.utcnow)

    category_id = db.Column(db.Integer, db.ForeignKey('category.id'),
        nullable=False)
    category = db.relationship('Category',
        backref=db.backref('posts', lazy=True))

    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), nullable=False)

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

First let's create some objects:

```
>>> py = Category(name='Python')
>>> Post(title='Hello Python!', body='Python is pretty cool', category=py)
>>> p = Post(title='Snakes', body='Ssssssss')
>>> py.posts.append(p)
>>> db.session.add(py)
```

As you can see, there is no need to add the `Post` objects to the session. Since the `Category` is part of the session all objects associated with it through relationships will be added too. It does not matter whether `db.session.add()` is called before or after creating these objects. The association can also be done on either side of the relationship - so a post can be created with a category or it can be added to the list of posts of the category.

Let's look at the posts. Accessing them will load them from the database since the relationship is lazy-loaded, but you will probably not notice the difference - loading a list is quite fast:

```
>>> py.posts
[<Post 'Hello Python!'>, <Post 'Snakes'>]
```

While lazy-loading a relationship is fast, it can easily become a major bottleneck when you end up triggering extra queries in a loop for more than a few objects. For this case, SQLAlchemy lets you override the loading strategy on the query level. If you wanted a single query to load all categories and their posts, you could do it like this:

```
>>> from sqlalchemy.orm import joinedload
>>> query = Category.query.options(joinedload('posts'))
>>> for category in query:
...     print category, category.posts
<Category u'Python'> [<Post u'Hello Python!'>, <Post u'Snakes'>]
```

If you want to get a query object for that relationship, you can do so using `with_parent()`. Let's exclude that post about Snakes for example:

```
>>> Post.query.with_parent(py).filter(Post.title != 'Snakes').all()
[<Post 'Hello Python!'>]
```

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 2

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` come to 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()
```

Alternatively, use the with-statement to take care of setup and teardown:

```
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 3

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.

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. Some backends may use a different default timeout value. For more information about timeouts see *Timeouts*.
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.

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
```

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 in conjunction with SQLAlchemy 0.9.2 or higher. Doing so is important for dealing with database migrations (for instance using alembic as stated here. Here's an example, as suggested by the SQLAlchemy docs:

```
from sqlalchemy import MetaData
from flask import Flask
from flask_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.

Timeouts

Certain database backends may impose different inactive connection timeouts, which interferes with Flask-SQLAlchemy's connection pooling.

By default, MariaDB is configured to have a 600 second timeout. This often surfaces hard to debug, production environment only exceptions like `2013: Lost connection to MySQL server during query`.

If you are using a backend (or a pre-configured database-as-a-service) with a lower connection timeout, it is recommended that you set *SQLALCHEMY_POOL_RECYCLE* to a value less than your backend's timeout.

CHAPTER 4

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". To override the table name, set the `__tablename__` class attribute.

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, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)

    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 (optional in some databases, e.g. PostgreSQL)
`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

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 `Address` 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 `ForeignKey` class:

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

class Address(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), nullable=False)
    person_id = db.Column(db.Integer, db.ForeignKey('person.id'),
        nullable=False)
```

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()`.

Since a person with no name or an email address with no address associated makes no sense, `nullable=False` tells SQLAlchemy to create the column as `NOT NULL`. This

is implied for primary key columns, but it's a good idea to specify it for all other columns to make it clear to other people working on your code that you did actually want a nullable column and did not just forget to add it.

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'` / `True` (which is the default, but explicit is better than implicit) means that SQLAlchemy will load the data as necessary in one go using a standard select statement.
- `'joined'` / `False` 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 can be useful if you have many items and always want to apply additional SQL filters to them. Instead of loading the items SQLAlchemy will return another query object which you can further refine before loading the items. Note that this cannot be turned into a different loading strategy when querying so it's often a good idea to avoid using this in favor of `lazy=True`. A query object equivalent to a dynamic `user.addresses` relationship can be created using `Address.query.with_parent(user)` while still being able to use lazy or eager loading on the relationship itself as necessary.

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), nullable=False)
    addresses = db.relationship('Address', lazy='select',
        backref=db.backref('person', lazy='joined'))
```

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'), primary_key=True),
    db.Column('page_id', db.Integer, db.ForeignKey('page.id'), primary_key=True)
)

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

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

Here we configured `Page.tags` to be loaded immediately after loading a Page, but using a separate query. This always results in two queries when retrieving a Page, but when querying for multiple pages you will not get additional queries.

The list of pages for a tag on the other hand is something that's rarely needed. For example, you won't need that list when retrieving the tags for a specific page. Therefore, the backref is set to be lazy-loaded so that accessing it for the first time will trigger a query to get the list of pages for that tag. If you need to apply further query options on that list, you could either switch to the `'dynamic'` strategy - with the drawbacks mentioned above - or get a query object using `Page.query.with_parent(some_tag)` and then use it exactly as you would with the query object from a dynamic relationship.

CHAPTER 5

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.

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
```

Deleting Records

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

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

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
2
>>> 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).all()
[<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'>
```

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 6

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`).

Example Configuration

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

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

Creating and Dropping Tables

The `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)
```

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 7

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.

CHAPTER 8

Customizing

Flask-SQLAlchemy defines sensible defaults. However, sometimes customization is needed. Two major pieces to customize are the Model base class and the default Query class.

Both of these customizations are applied at the creation of the `SQLAlchemy` object and extend to all models derived from its `Model` class.

Model Class

Flask-SQLAlchemy allows defining a custom declarative base, just like SQLAlchemy, that all model classes should extend from. For example, if all models should have a custom `__repr__` method:

```
from flask_sqlalchemy import Model  # this is the default declarative base
from flask_sqlalchemy import SQLAlchemy

class ReprBase(Model):
     def __repr__(self):
         return "<{0} id: {1}>".format(self.__class__.__name__, self.id)

db = SQLAlchemy(model_class=ReprBase)

class MyModel(db.Model):
    ...
```

Note: While not strictly necessary to inherit from `flask_sqlalchemy.Model` it is en-

couraged as future changes may cause incompatibility.

Note: If behavior is needed in only some models, not all, a better strategy is to use a Mixin, as exampled below.

While this particular example is more useful for debugging, it is possible to provide many augmentations to models that would otherwise be achieved with mixins instead. The above example is equivalent to the following:

```
class ReprBase(object):
    def __repr__(self):
        return "<{0} id: {1}>".format(self.__class__.__name__, self.id)

db = SQLAlchemy()

class MyModel(db.Model, ReprBase):
    ...
```

It also possible to provide default columns and properties to all models as well:

```
from flask_sqlalchemy import Model, SQLAlchemy
from sqlalchemy import Column, DateTime
from datetime import datetime

class TimestampedModel(Model):
    created_at = Column(DateTime, default=datetime.utcnow)

db = SQLAlchemy(model_class=TimestampedModel)

class MyModel(db.Model):
    ...
```

All model classes extending from `db.Model` will now inherit a `created_at` column.

Query Class

It is also possible to customize what is availble for use on the special `query` property of models. For example, providing a `get_or` method:

```
from flask_sqlalchemy import BaseQuery, SQLAlchemy

class GetOrQuery(BaseQuery):
    def get_or(self, ident, default=None):
        return self.get(ident) or default

db = SQLAlchemy(query_class=GetOrQuery)
```

And now all queries executed from the special `query` property on Flask-SQLAlchemy models can use the `get_or` method as part of their queries. All relationships defined with `db.relationship` (but not `sqlalchemy.relationship()`) will also be provided with this functionality.

> **Warning:** Unlike a custom `Model` base class, it is required to either inherit from either `flask_sqlalchemy.BaseQuery` or `sqlalchemy.orm.Query()` in order to define a custom query class.

It also possible to define a custom query class for individual relationships as well, by providing the `query_class` keyword in the definition. This works with both `db.relationship` and `sqlalchemy.relationship`:

```
class MyModel(db.Model):
    cousin = db.relationship('OtherModel', query_class=GetOrQuery)
```

Note: If a query class is defined on a relationship, it will take precedence over the query class attached to its corresponding model.

It is also possible to define a specific query class for individual models by overriding the `query_class` class attribute on the model:

```
class MyModel(db.Model):
    query_class = GetOrQuery
```

In this case, the `get_or` method will be only availble on queries orginating from `MyModel.query`.

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 9

API

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

Configuration

class flask_sqlalchemy.**SQLAlchemy**(*app=None*, *use_native_unicode=True*, *session_options=None*, *metadata=None*, *query_class=<class 'flask_sqlalchemy.BaseQuery'>*, *model_class=<class 'flask_sqlalchemy.Model'>*)

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.

The `session_options` parameter, if provided, is a dict of parameters to be passed to the session constructor. See `Session` for the standard options.

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.

New in version 3.0: The *query_class* parameter was added, to allow customisation of the query class, in place of the default of `BaseQuery`.

The *model_class* parameter was added, which allows a custom model class to be used in place of `Model`.

Changed in version 3.0: Utilise the same query class across *session*, *Model.query* and *Query*.

`Query` = None
Default query class used by `Model.query` and other queries. Customize this by passing `query_class` to `SQLAlchemy()`. Defaults to `BaseQuery`.

`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*)
Create a `scoped_session` on the factory from `create_session()`.

An extra key `'scopefunc'` can be set on the `options` dict to specify a custom scope function. If it's not provided, Flask's app context stack identity is used. This will ensure that sessions are created and removed with the request/response cycle, and should be fine in most cases.

Parameters **`options`** – dict of keyword arguments passed to session class in `create_session`

`create_session`(*options*)
Create the session factory used by `create_scoped_session()`.

The factory **must** return an object that SQLAlchemy recognizes as a session, or registering session events may raise an exception.

Valid factories include a `Session` class or a `sessionmaker`.

The default implementation creates a `sessionmaker` for `SignallingSession`.

Parameters **`options`** – dict of keyword arguments passed to session class

`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 spe-

cific 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=None*, *bind=None*)
Returns a specific engine.

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

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

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

make_declarative_base(*model*, *metadata=None*)
Creates the declarative base.

metadata
The metadata associated with `db.Model`.

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

Changed in version 0.12: Parameters were added

Models

class `flask_sqlalchemy.`**Model**
Base class for SQLAlchemy declarative base model.

To define models, subclass `db.Model`, not this class. To customize `db.Model`, subclass this and pass it as `model_class` to `SQLAlchemy()`.

__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
Convenience property to query the database for instances of this model using the current session. Equivalent to `db.session.query(Model)` unless `query_class` has been changed.

query_class = None
Query class used by `query`. Defaults to `SQLAlchemy.Query`, which defaults to `BaseQuery`.

class flask_sqlalchemy.**BaseQuery**(*entities*, *session=None*)
SQLAlchemy `Query` subclass with convenience methods for querying in a web application.

This is the default `query` object used for models, and exposed as `Query`. Override the query class for an individual model by subclassing this and setting `query_class`.

first_or_404()
Like `first()` but aborts with 404 if not found instead of returning `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`.

If no items are found and `page` is greater than 1, or if page is less than 1, it aborts with 404. This behavior can be disabled by passing `error_out=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 aborts with 404. If there is no request or they aren't in the query, they default to 1 and 20 respectively.

Returns a `Pagination` object.

Sessions

class flask_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.

New in version 2.1: The *binds* option was added, which allows a session to be joined to an external transaction.

app = None
The application that this session belongs to.

Utilities

class flask_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_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 **10**

Changelog

Version 2.2

Released on February 27, 2017, codename Dubnium

- Minimum SQLAlchemy version is 0.8 due to use of `sqlalchemy.inspect`.
- Added support for custom `query_class` and `model_class` as args to the `SQLAlchemy` constructor. (#328)
- Allow listening to SQLAlchemy events on `db.session`. (#364)
- Allow `__bind_key__` on abstract models. (#373)
- Allow `SQLALCHEMY_ECHO` to be a string. (#409)
- Warn when `SQLALCHEMY_DATABASE_URI` is not set. (#443)
- Don't let pagination generate invalid page numbers. (#460)
- Drop support of Flask < 0.10. This means the db session is always tied to the app context and its teardown event. (#461)
- Tablename generation logic no longer accesses class properties unless they are `declared_attr`. (#467)

Version 2.1

Released on October 23rd 2015, codename Caesium

- Table names are automatically generated in more cases, including subclassing mixins and abstract models.
- Allow using a custom MetaData object.
- Add support for binds parameter to session.

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.

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.

Version 0.16

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

Version 0.15

- Added session support for multiple databases

Version 0.14

- Make relative sqlite paths relative to the application root.

Version 0.13

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

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.

Version 0.11

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

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.

Version 0.9

- applied changes to pass the Flask extension approval process.

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.

Version 0.7

- Initial public release

Index