flask-sqlalchemy-russian Documentation

Выпуск 1

Ilya Flakin

авг. 28, 2017

Оглавление

Перевод документаци Flask-sqlalchemy Flask-SQLAlchemy это расширение для Flask добовляющее поддрежку SQLAlchemy для ваших приложений. Для работы требуется SQLAlchemy 0.6 или выше. Он призван упростить работу во Flask с SQLAlchemy предоставляя удобные стандартные и дополнительные подходы, для упрощения выполнения основных задач.

Глава 1

Руководство пользователя

Данная часть документации покажет с чего начать использование Flask-SQLAlchemy.

Быстрый старт

Flask-SQLAlchemy прост в использовании, удобен в простых приложениях, и легко расширяется для сложных. Для получения дополнительных сведений, просмотрите API докупентацию класса `SQLAlchemy`.

Минимальное приложение

Для общего случая с одним приложением Flask все что вам необходимо сделать это создать ваше Flask приложение, загрузить конфигурацию с выбором БД, затем создать объект `SQLAlchemy` передав ему наше Flask приложение.

После создания, этот объект содержит все функции и инструменты из `sqlalchemy` и `sqlalchemy.orm`. Кроме того он предоставляет класс `Model` который может быть использован для описания модели:

```
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
```

Для создания базы данных, просто импортируйте объект *db* из Python shell и запустите `SQLAlchemy.create_all()` метод для создания таблиц и базы данных:

```
>>> from yourapplication import db
>>> db.create_all()
```

Бах, и база данных создана. Теперь создадим нескольких пользователей:

```
>>> from yourapplication import User
>>> admin = User('admin', 'admin@example.com')
>>> guest = User('guest', 'guest@example.com')
```

Но они не будут занесены в базу данных пока мы не выполним следующее:

```
>>> db.session.add(admin)
>>> db.session.add(guest)
>>> db.session.commit()
```

Доступ к данным в базе данных очень прост:

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

Простые связи

SQLAlchemy подключается к реляционным базам данных, и что в них действительно хорошо - это связи. Как пример, рассмотрим приложение использующее две таблицы, которые имеют свзяи друг с другом:

```
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
```

Для начала созданим некоторые объекты:

```
>>> py = Category('Python')
>>> p = Post('Hello Python!', 'Python is pretty cool', py)
>>> db.session.add(py)
>>> db.session.add(p)
```

Теперь т.к. мы обяъвили `posts` как объект связи со свойством dynamic в backref, он отображается как запрос(прим. пер. не смог сформулировать правильно):

```
>>> py.posts
<sqlalchemy.orm.dynamic.AppenderBaseQuery object at 0x1027d37d0>
```

Он ведет себя как обычный объект запроса, так мы можем выполнить запрос на все сообщения, которые связаны с нашей тестовой категорией “Python”:

```
>>> py.posts.all()
[<Post 'Hello Python!'>]
```

Дорога к просвящению

Главное, что неоходимо знать, по сравнению с обычной SQLAlchemy:

1. `SQLAlchemy` дает вам доступ к следующим вещам:
   - все функции и классы из `sqlalchemy` и `sqlalchemy.orm`
   - предсконфигурированный с ограниченной областью видимости объект сессии называемый `session`
   - `metadata`
   - `engine`
   - методы `SQLAlchemy.create_all()` и `SQLAlchemy.drop_all()` для создания и удаления таблиц в соответсвии с моделями.
   - Базовый класс `Model` являющийся конфигурационной декларативной основой.
2. Класс `Model` декларативный базовый класс ведет себя как обычный класс Python, но имеет атрибут `query`, который может быть использован для запроса к модели. (`Model` и `BaseQuery`)

3. Вы должны выполнить commit сессию, но не должны удалять ее в конце запроса, Flask-SQLAlchemy сделает это за вас.

Оригинал этой страницы

Введение в Contexts

Если вы планируете использовать только одно приложения, тогда можете пропустить эту статью. Просто передайте приложение в конструктор `SQLAlchemy` и можно начинать работать. Однако, если вы хотите использовать больше чем одно приложение или создавать их динамически в функции, тогда вам следует прочитать этот текст.

Если вы объявляете ваше приложени в функции, но объект `SQLAlchemy` глобальны, как потом узнать об объекте создателе? Ответ - функция `init_app()`:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

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

Что она делает - подготавливает ваше приложение для работы с `SQLAlchemy`. Однако при этом не связывает обект `SQLAlchemy` с вашим приложением. Почему бы этого не сделать? Потому что возможно использование больше чем одно приложение.

Как же теперь `SQLAlchemy` узнает о вашем приложении? Вам необходимо установить контекст приложения. Если вы работаете внутри функции Flask, это происходит автоматически. Однако если вы работаете в интерактивной оболочке - вам придется сделать это вручную (смотрите Creating an Application Context).

В двух словах, сделать так:

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

Внутри функции также можно использовать конструкцию with:

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

Некоторые функции внутри Flask-SQLAlchemy также могут принимать объект приложения:

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

Оригинал этой страницы

Конфигурация

Здесь представлены значения конфигурации используемые в Flask-SQLAlchemy. Flask-SQLAlchemy загружает эти значения из вашей конфигурации Flask, которая может быть заданна несколькими путями. Стоит обратить внимание, что некоторые значения не могут быть изменены после создания экземпляра, поэтому убедитесь что они заданны как можно раньше и не меняйте их во время выполнения приложения.

Конфигурационные ключи

Список конфигурационных ключей currently understood by the extension:

SQLALCHEMY_DATABASE_URI	Путь/URI базы данных, который будет использоваться для подключения. Пример: • 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 *Работа с несколькими базами данных и Binds*.
SQLALCHEMY_ECHO	Если установлен в ‘ёTrue‘ё то SQLAlchemy будет выводить все сообщения в stderr, что может быть полезно при отладке.
SQLALCHEMY_RECORD_QUERIES	Может применяться для явного отключения или включения записи запросов. Запись запросов происходит автоматически в режимах debug и testing. Для дополнительной информации смотрите get_debug_queries()
SQLALCHEMY_NATIVE_UNICODE	Может использоваться для отключения встроенной поддержки юникода. 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	Количество секунд по истечению которого соединение автоматически перезапустится. Это необходимо для MySQL, которая по умолчанию удаляет соединения после 8 часов простоя. Если используется MySQL Flask-SQLAlchemy автоматически устанавливает его равным 2 часам.
SQLALCHEMY_MAX_OVERFLOW	Контролирует количество соединений которые могут быть созданы после того как pool набрал свой максимальный размер. When those additional connections are returned to the pool, they are disconnected and discarded.
SQLALCHEMY_TRACK_MODIFICATIONS	Если установлен в True, то Flask-SQLAlchemy будет отслеживать изменения объектов и посылать сигналы. По умолчанию становлен в None, что включает отслеживание но выводит предупреждение, что в будующем будет отчключена по умолчанию. Данная функция требует дополнительную память, и должна быть отключена если не используется.

Добавлено в версии 0.8: Были добавлены конфигурационные ключи SQLALCHEMY_NATIVE_UNICODE, SQLALCHEMY_POOL_SIZE, SQLALCHEMY_POOL_TIMEOUT и SQLALCHEMY_POOL_RECYCLE.

Добавлено в версии 0.12: Был добавлен конфигурационный ключ SQLALCHEMY_BINDS.

Добавлено в версии 0.17: Был добавлен конфигурационный ключ SQLALCHEMY_MAX_OVERFLOW.

Добавлено в версии 2.0: Был добавлен конфигурационный ключ SQLALCHEMY_TRACK_MODIFICATIONS.

Изменено в версии 2.1: SQLALCHEMY_TRACK_MODIFICATIONS выводит предупреждение если не установлен.

Формат connection URI

Для получения полного списка поддерживаемых connection URIs обратитесь к документации SQLAlchemy (Supported Databases). Здесь показаны некоторые общие строки подключения.

SQLAlchemy указывает источник БД (прим. пер. Engine буду называть БД) как URI комбинированный с опциональными ключевыми словами определяющими параметры БД. Формат записи URI:

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

Многие части в строке - необязательны. Если driver не указан то выбирает один из стандартных (убедитесь что *не* включили + в этом случае).

Postgres:

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

MySQL:

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

Oracle:

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

SQLite (обратите внимание на четыре слеша):

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

Использование специальных MetaData и соглашение об именовании

Вы можете построить объект `SQLAlchemy`co специальным объектом :class:`~sqlalchemy.schema.MetaData`. Это позволяет вам, помимо всего прочего, задать собственное ограничение именования. Это имеет важное значение для работы с миграциями баз данных (например с использованием alembic как указанно здесь. Т.к. SQL не определяет соглашений по именованию, нет гарантий совместимости по умолчанию среди реализаций базы данных. Вы можете определить пользовательские соглашения об именах, как это, как это было предложено в документации SQLAlchemy:

```
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)
```

Для дополнительной информации о классе `MetaData`, обратитесь к официальной документации.

Оригинал этой страницы

Декларирование Models

Вообще Flask-SQLAlchemy ведет себя как правильно сконфигурированная описательная база из расширения `declarative`. Мы рекомендуем прочитать документацию SQLAlchemy для полного восприятия. Однако, наиболее распространенные примеры отражены здесь.

Что нужно иметь ввиду:

- Базовый класс для всех ваших моделей называется *db.Model*. Он хранится в экземпляре SQLAlchemy который вы создаете. Смотрите *Быстрый старт* для дополнительной информации.
- Некоторые части требуемые в SQLAlchemy не являются обязательными в Flask-SQLAlchemy. Например имя таблицы устанавливается автоматически, если оно не было заданно. Оно образуется из названия класса с преобразованием имени в нижний регистр, так "CamelCase" преобразуется в "camel_case". Чтобы определить имя таблицы, установите атрибут *__tablename__*.

Простой пример

Очень простой пример:

```
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
```

Используйте `Column` для определения столбца. Имя столбца - это имя атрибута которому оно присвоенно. Если вы хотите использовать другое имя в таблице, вы можете указать необязательный первый аргумент, который является строкой определяющей имя столбца. Первичные ключи отмечаются с помощью `primary_key=True`. Несколько ключей могут быть отмечены как первичные ключи в этом случае они образуют составной первичный ключ.

Тип столбца является первый аргумент в `Column`. Вы можете указать их непосредственно или вызвать их в дальнейшем с дополнительным определением (например указать длинну). Следующие типы наиболее распространены:

Integer	Числовой
String (size)	Числовой с указанием размера
Text	Более длинный текс в юникоде
DateTime	Дата и время выраженные как объект Python `datetime`
Float	Число с плавающей точкой
Boolean	Логическое значение
PickleType	Хранит pickled Python объект
LargeBinary	Хранит большие произвольные бинарные данные

Связь Один-ко-Многим

Связь Один-ко-Многим являются наиболее распространенными. Поскольку связи объявляются, прежде чем они установлены вы можете использовать строки для обозначения классов, которые еще не созданы (например, если *Person* определяет связь к *Address*, которая объявляется позднее в файле).

Связи записываются с помощью функции `relationship()`. Однако внешний ключ должен быть объевлен отдельно с помощью класса `sqlalchemy.schema.ForeignKey`:

```
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'))
```

Что же делает `db.relationship()`? Эта функция возвращает новое свойство, которое может сделать несколько вещей. В данном случае мы сказали ей указывать на класс *Address* и загружать их несколько. Откуда он знает, что будет возвращать более одного адреса? Потому что SQLAlchemy угадывает полезные преднастройки из вашего описания. Если вы хотели бы иметь связь один к одному можно указать `uselist=False` в `relationship()`.

Итак что же изначают параметры *backref* и *lazy*? *backref* простой способ объявить новое свойство для класса *Address*. Вы можете также использовать `my_address.person` чтобы обратиться к человеку из этого адреса. *lazy* определяет, когда SQLAlchemy будет загружать данные из базы данных:

- `'select'` (значение по умолчанию) означет что SQLAlchemy будет загружать данные по мере необходимости в один заход с использованием стандартного оператора выбора.
- `'joined'` говорит SQLAlchemy загружать связи в том же запросе что и родительский используя оператор *JOIN*.
- `'subquery'` работает как `'joined'`, но SQLAlchemy использует подзапрос.
- `'dynamic'` является особено полезным, если у вас есть много элементов. Вместо того чтобы загружать элементы SQLAlchemy будет возвращать объект запроса, который вы можете дополнительно уточнить перед загрузкой элементов. Как правило, это как раз то, в чем вы нуждаетесь, если ожидаете более чем несколько элементов для этих связей.

Как вам определить lazy статус для backrefs? Используйте функцию `backref()`

```
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')
```

Связь Многие-ко-Многим

Если вы хотите использовать связи много-ко-многим вам нужно будет определить вспомогательную таблицу, которая будет использоваться для связей. Для этой вспомогательной таблице строго рекмендуется *не* использовать model, а создать ее самостоятельно:

```
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)
```

Здесь мы задали *Page.tags* как список тэгов загружаемых в один заход, потому что мы не ожидаем большого количества тегов на одной странице. Список страниц на тэг (*Tag.pages*) однако является динамическим backref. Как уже упоминалось выше, это означает, что вы получите объект запроса обратно и можете использовать для выборки в нем (прим.пер. слабо понял последнюю фразу =().

Оригинал этой страницы

Select, Insert, Delete

Теперь, когда вы *объявили модели* пришло время для запроса данных из базы. Мы будем использовать модель определеннкю в глвае *Быстрый старт*.

Вставка записей

Прежде чем запросить какие либо данные, мы должны вставить эти данные в базу данных. Все ваши модели должны иметь конструктор, проверьте, что вы не забыли этого сделать. Конструкторы используются только вами, поэтому все зависит от того как вы их определите.

Вставка данных в базу данных проходит в три этапа:

1. Создание объекта Python
2. Добавление его в сессию
3. Commit сессии

Сессия здесь это не Flask сессия, а Flask-SQLAlchemy. По сути это расширенная версия транзакции базы данных. Как это работает:

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

Отлично, это не сложно. Что здесь произошло? Перед тем как вы добавили объект в сессию, SQLAlchemy не планировал добавлять его в транзакцию. Это хорошо, потому что вы можете на данном этапе отменить изменения. Для примера представьте создание поста настранице, но вы только хотели посмотреть как будет выглядит пост в предварительном просмотре вместо сохраниения его в базу данных.

Вызов Функции `add()` добавляет объект. Она будет вызвать оператор *INSERT* для базы данных, но так как транзакция еще не завершена, вы не получаете идентификатор. Если вы выполните commit, ваш пользователь получет ID:

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

Удаление записей

Удаление записей очень простое, вместо `add()` используйте `delete()`:

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

Запрос записей

Итак, как же нам получить данные обратно из нашей базы данных? Для этой цели Flask-SQLAlchemy предоставляет атрибут `query` для класса `Model`. Когда вы обращаетесь к нему вы получаете новый объект запроса по всем записям. Вы можете использовать метод `filter()` для фильтрации записей перед выполнением выборки `all()` или `first()`. Если вы хотите произвести выборку по первичном ключу используйте `get()`.

Рассмотрим несколько запросов для следующих записей в базе данных:

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

Получить пользователя по имени:

```
>>> peter = User.query.filter_by(username='peter').first()
>>> peter.id
2
>>> peter.email
u'peter@example.org'
```

То же, что и выше, но для не существующего пользователя вернуть *None*:

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

Выбор нескольких пользователей, более сложным выражением:

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

Упорядочить по какому либо полю:

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

Установить лимит на количество возвращаемых значений:

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

Получить пользователя по первичному ключу:

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

Запросы и представления

Если вы создаете функции представлений во Flask, часто бывает удобно вернуть 404 ошибку для отсутствующих данных. Т.к. это очень распространненая идома, Flask-SQLAlchemy предоставляет инструменты для данных целей. Вместо `get()` можно использовать `get_or_404()` и вместо `first()` `first_or_404()`. Они будут вызывать исключение с номером 404 вместо возвращения знаения *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)
```

Оригинал этой страницы

Работа с несколькими базами данных и Binds

Начиная с версии 0.12 Flask-SQLAlchemy может легко подключаться к нескольким базам данных. Для этого неоходимо выполнить преконфигурацию SQLAlchemy для поддержки нескольких “binds” (прим. пер. буду переводить как привязка).

Что такое привязки? В SQLAlchemy гворят привязка это то что может выполнить операторы SQL, и как правило подключения или движок. В Flask-SQLAlchemy под привязками имеют ввиду движки которые создаются автоматически невидимо для вас. Каждый из этих движков затем связывается с коротким ключом (ключ привязки). Этот ключ затем используется во время описания модели, чтобы связать модель с конкретным двигателем.

Елси не один ключ привязки не указан для модели, то используется подключение по умолчанию (как заданно `SQLALCHEMY_DATABASE_URI`).

Пример конфигурации

Следующая конфигурация объявляет поключение к трем базам данных. Одно по умолчанию, а также два других названных *users* (для пользователей) и appmeta (которое подключается к базе данных SQLite с доступом только на чтения к некоторым данным приложение обеспечивает внутри):

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

Создание и удаление таблиц

Методы `create_all()` и `drop_all()` по умолчанию работают на всех привязках, включая которая по умолчанию. Это поведение может быть настроенно в реализации параметра *bind*. Это может быть имя одной привязки, `'__all__'` для указания всех привязок или список привязок. По умолчанию привязка (`SQLALCHEMY_DATABASE_URI`) установлеа в *None*:

```
>>> db.create_all()
>>> db.create_all(bind=['users'])
>>> db.create_all(bind='appmeta')
>>> db.drop_all(bind=None)
```

Ссылки на привязки

Если вы объявляете модель, вы можете указать привязку для нее, используя атрибут `__bind_key__`:

```
class User(db.Model):
    __bind_key__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
```

Внутри ключ привязки сохраняется в словаре таблицы *info* как `'bind_key'`. Это вадно знать, потому что когда вы захотите создать объект таблицы напрямую, вам придется поместить его туда:

```
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'}
)
```

Если вы указали *__bind_key__* в ваших моделях вы можете использовать их именно так, как привыкли. Модель подключается к указанной в ней базе данных.

Оригинал этой страницы

Поддержка сигналов

Подключение к следующим сигналам позволяет принимать уведомления до и после применения изменений в базе данных. Эти изменения отслеживаются только если `SQLALCHEMY_TRACK_MODIFICATIONS` включен в конфигурации.

Добавлено в версии 0.10.

Изменено в версии 2.1: `before_models_committed` срабатывает корректно.

Не рекомендуется, начиная с версии 2.1: Будет отключена по умолчанию в будущих версиях.

`flask_sqlalchemy.models_committed`
: Этот сигнал посылается, когда измененные модели были зафиксированы в базе данных.

  Отправителем является приложение которое запустило изменение. Принимающий получает параметр `changes` со списком кортежей в форме `(model instance, operation)`.

  Инициирующие операции `'insert'`, `'update'`, и `'delete'`.

`flask_sqlalchemy.before_models_committed`
: Этот сигнал работает точно так же, как *models_committed* но запускается перед применением изменений.

Оригинал этой страницы

Глава 2

Спарвка об API

Если вы ищите информацию по конкретным функция, классам или методам, то эта часть документации для вас.

API

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

Configuration

Models

Sessions

Utilities

Оригинал этой страницы

Глава 3

Дополнительные замечания

Просмотреть правовую информацию о проекте Flask.

Содержание модулей Python

f

Алфавитный указатель