Flask-Restless Documentation

Release 0.16.0

February 12, 2016

Contents

Flask-Restless provides simple generation of ReSTful APIs for database models defined using SQLAlchemy (or Flask-SQLAlchemy). The generated APIs send and receive messages in JSON format.

Part I
USER'S GUIDE

CHAPTER 1

Downloading and installing Flask-Restless

Flask-Restless can be downloaded from its page on the Python Package Index. The development version can be downloaded from its page at GitHub. However, it is better to install with `pip` (hopefully in a virtual environment provided by `virtualenv`):

```
pip install Flask-Restless
```

Flask-Restless requires Python version 2.6, 2.7, or 3.3. Python 3.2 is not supported by Flask and therefore cannot be supported by Flask-Restless.

Flask-Restless has the following dependencies (which will be automatically installed if you use `pip`):

- Flask version 0.10 or greater
- SQLAlchemy version 0.8 or greater
- mimerender version 0.5.2 or greater
- python-dateutil version strictly greater than 2.0
- Flask-SQLAlchemy, *only if* you want to define your models using Flask-SQLAlchemy (which we highly recommend)

CHAPTER 2

Quickstart

For the restless:

```
import flask
import flask.ext.sqlalchemy
import flask.ext.restless

# Create the Flask application and the Flask-SQLAlchemy object.
app = flask.Flask(__name__)
app.config['DEBUG'] = True
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = flask.ext.sqlalchemy.SQLAlchemy(app)

# Create your Flask-SQLALchemy models as usual but with the following two
# (reasonable) restrictions:
#   1. They must have a primary key column of type sqlalchemy.Integer or
#      type sqlalchemy.Unicode.
#   2. They must have an __init__ method which accepts keyword arguments for
#      all columns (the constructor in flask.ext.sqlalchemy.SQLAlchemy.Model
#      supplies such a method, so you don't need to declare a new one).
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode, unique=True)
    birth_date = db.Column(db.Date)


class Computer(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode, unique=True)
    vendor = db.Column(db.Unicode)
    purchase_time = db.Column(db.DateTime)
    owner_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    owner = db.relationship('Person', backref=db.backref('computers',
                                                         lazy='dynamic'))


```

```
# Create the database tables.
db.create_all()

# Create the Flask-Restless API manager.
manager = flask.ext.restless.APIManager(app, flask_sqlalchemy_db=db)

# Create API endpoints, which will be available at /api/<tablename> by
# default. Allowed HTTP methods can be specified as well.
manager.create_api(Person, methods=['GET', 'POST', 'DELETE'])
manager.create_api(Computer, methods=['GET'])

# start the flask loop
app.run()
```

You may find this example at `examples/quickstart.py` in the source distribution; you may also view it online at GitHub.

Further examples can be found in the `examples/` directory in the source distribution or on the web.

CHAPTER 3

Creating API endpoints

To use this extension, you must have defined your database models using either SQLAlchemy or Flask-SQLALchemy.

The basic setup for Flask-SQLAlchemy is the same. First, create your `flask.Flask` object, `flask.ext.sqlalchemy.SQLAlchemy` object, and model classes as usual but with the following two (reasonable) restrictions on models:

1. They must have a primary key column of type `sqlalchemy.Integer` or type `sqlalchemy.Unicode`.
2. They must have an `__init__` method which accepts keyword arguments for all columns (the constructor in `flask.ext.sqlalchemy.SQLAlchemy.Model` supplies such a method, so you don't need to declare a new one).

```
import flask
import flask.ext.sqlalchemy

app = flask.Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = flask.ext.sqlalchemy.SQLAlchemy(app)

class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode, unique=True)
    birth_date = db.Column(db.Date)
    computers = db.relationship('Computer',
                                backref=db.backref('owner',
                                                   lazy='dynamic'))


class Computer(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode, unique=True)
    vendor = db.Column(db.Unicode)
    owner_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    purchase_time = db.Column(db.DateTime)
```

```
db.create_all()
```

If you are using pure SQLAlchemy:

```
from flask import Flask
from sqlalchemy import Column, Date, DateTime, Float, Integer, Unicode
from sqlalchemy import ForeignKey
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import backref, relationship
from sqlalchemy.orm import scoped_session, sessionmaker

app = Flask(__name__)
engine = create_engine('sqlite:////tmp/testdb.sqlite', convert_unicode=True)
Session = sessionmaker(autocommit=False, autoflush=False, bind=engine)
mysession = scoped_session(Session)

Base = declarative_base()
Base.metadata.bind = engine

class Computer(Base):
    __tablename__ = 'computer'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, unique=True)
    vendor = Column(Unicode)
    buy_date = Column(DateTime)
    owner_id = Column(Integer, ForeignKey('person.id'))

class Person(Base):
    __tablename__ = 'person'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, unique=True)
    age = Column(Float)
    other = Column(Float)
    birth_date = Column(Date)
    computers = relationship('Computer',
                             backref=backref('owner', lazy='dynamic'))

Base.metadata.create_all()
```

Warning: Attributes of these entities must not have a name containing two underscores. For example, this class definition is no good:

```
class Person(db.Model):
    __mysecretfield = db.Column(db.Unicode)
```

This restriction is necessary because the search feature (see *Making search queries*) uses double underscores as a separator. This may change in future versions.

Second, instantiate a `flask.ext.restless.APIManager` object with the `Flask` and `SQLAlchemy` objects:

```
import flask.ext.restless

manager = flask.ext.restless.APIManager(app, flask_sqlalchemy_db=db)
```

Or if you are using pure SQLAlchemy, specify the session you created above instead:

```
manager = flask.ext.restless.APIManager(app, session=mysession)
```

Third, create the API endpoints which will be accessible to web clients:

```
person_blueprint = manager.create_api(Person,
                                      methods=['GET', 'POST', 'DELETE'])
computer_blueprint = manager.create_api(Computer)
```

Note that you can specify which HTTP methods are available for each API endpoint. There are several more customization options; for more information, see *Customizing the ReSTful interface*.

Due to the design of Flask, these APIs must be created before your application handles any requests. The return value of `APIManager.create_api()` is the blueprint in which the endpoints for the specified database model live. The blueprint has already been registered on the `Flask` application, so you do *not* need to register it yourself. It is provided so that you can examine its attributes, but if you don't need it then just ignore it:

```
manager.create_api(Person, methods=['GET', 'POST', 'DELETE'])
manager.create_api(Computer)
```

If you wish to create the blueprint for the API without registering it (for example, if you wish to register it later in your code), use the `APIManager.create_api_blueprint()` method instead:

```
blueprint = manager.create_api_blueprint(Person, methods=['GET', 'POST'])
# later...
app.register_blueprint(blueprint)
```

By default, the API for `Person`, in the above code samples, will be accessible at `http://<host>:<port>/api/person`, where the `person` part of the URL is the value of `Person.__tablename__`:

```
>>> import json  # import simplejson as json, if on Python 2.5
>>> import requests  # python-requests is installable from PyPI...
>>> newperson = {'name': u'Lincoln', 'age': 23}
>>> r = requests.post('/api/person', data=json.dumps(newperson),
...                   headers={'content-type': 'application/json'})
>>> r.status_code, r.headers['content-type'], r.data
(201, 'application/json', '{"id": 1}')
>>> newid = json.loads(r.data)['id']
>>> r = requests.get('/api/person/%s' % newid,
...                  headers={'content-type': 'application/json'})
```

```
>>> r.status_code, r.headers['content-type']
(200, 'application/json')
>>> r.data
{
  "other": null,
  "name": "Lincoln",
  "birth_date": null,
  "age": 23.0,
  "computers": [],
  "id": 1
}
```

If the primary key is a `Unicode` instead of an `Integer`, the instances will be accessible at URL endpoints like `http://<host>:<port>/api/person/foo` instead of `http://<host>:<port>/api/person/1`.

3.1 Initializing the Flask application after creating the API manager

Instead of providing the Flask application at instantiation time, you can initialize the Flask application after instantiating the `APIManager` object by using the `APIManager.init_app()` method. If you do this, you will need to provide the Flask application object using the `app` keyword argument to the `APIManager.create_api()` method:

```
from flask import Flask
from flask.ext.restless import APIManager
from flask.ext.sqlalchemy import SQLAlchemy

app = Flask(__name__)
db = SQLAlchemy(app)
manager = APIManager(flask_sqlalchemy_db=db)

# later...

manager.init_app(app)
manager.create_api(Person, app=app)
```

You can also use this approach to initialize multiple Flask applications with a single instance of `APIManager`. For example:

```
from flask import Flask
from flask.ext.restless import APIManager
from flask.ext.sqlalchemy import SQLAlchemy

# Create two Flask applications, both backed by the same database.
app1 = Flask(__name__)
app2 = Flask(__name__ + '2')
```

```
app1.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
app2.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app1)

# Create the Flask-SQLAlchemy models.
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode, unique=True)
    birth_date = db.Column(db.Date)
    computers = db.relationship('Computer',
                                backref=db.backref('owner',
                                                   lazy='dynamic'))


class Computer(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode, unique=True)
    vendor = db.Column(db.Unicode)
    owner_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    purchase_time = db.Column(db.DateTime)


# Create the database tables.
db.create_all()

# Create the APIManager and initialize it with the different Flask objects.
manager = APIManager(flask_sqlalchemy_db=db)
manager.init_app(app1)
manager.init_app(app2)

# When creating each API, you need to specify which Flask application
# should be handling these requests.
manager.create_api(Person, app=app1)
manager.create_api(Computer, app=app2)
```

Changed in version 0.16.0: The `APIManager.init_app()` method behaved incorrectly before version 0.16.0. From that version on, you must provide the Flask application when you call `APIManager.create_api()` after having performed the delayed initialization described in this section.

Customizing the ReSTful interface

4.1 HTTP methods

By default, the `APIManager.create_api()` method creates a read-only interface; requests with HTTP methods other than GET will cause a response with 405 Method Not Allowed. To explicitly specify which methods should be allowed for the endpoint, pass a list as the value of keyword argument `methods`:

```
apimanager.create_api(Person, methods=['GET', 'POST', 'DELETE'])
```

This creates an endpoint at `/api/person` which responds to GET, POST, and DELETE methods, but not to other ones like PUT or PATCH.

The recognized HTTP methods and their semantics are described below (assuming you have created an API for an entity `Person`). All endpoints which respond with data respond with serialized JSON strings.

`GET /api/person`
: Returns a list of all `Person` instances.

`GET /api/person/`(**int:** *id*)
: Returns a single `Person` instance with the given `id`.

`GET /api/person?q=<searchjson>`
: Returns a list of all `Person` instances which match the search query specified in the query parameter `q`. For more information on searching, see *Making search queries*.

`DELETE /api/person/`(**int:** *id*)
: Deletes the person with the given `id` and returns 204 No Content.

`DELETE /api/person`
: This is only available if the `allow_delete_many` keyword argument is set to `True` when calling the `create_api()` method. For more information, see *Enable bulk patching or deleting*.

Deletes all instances of `Person` that match the search query provided in the `q` URL query paremeter. For more information on search parameters, see *Making search queries*.

`POST /api/person`
Creates a new person in the database and returns its `id`. The initial attributes of the `Person` are read as JSON from the body of the request. For information about the format of this request, see *Format of requests and responses*.

`PATCH /api/person/`(**int:** *id*)
Updates the attributes of the `Person` with the given `id`. The attributes are read as JSON from the body of the request. For information about the format of this request, see *Format of requests and responses*.

`PATCH /api/person`
This is only available if the `allow_patch_many` keyword argument is set to `True` when calling the `create_api()` method. For more information, see allowpatch-many.

Updates the attributes of all `Person` instances. The attributes are read as JSON from the body of the request. For information about the format of this request, see *Format of requests and responses*.

`PUT /api/person`

`PUT /api/person/`(**int:** *id*)
Aliases for `PATCH /api/person` and `PATCH /api/person/(int:id)`.

4.2 API prefix

To create an API at a different prefix, use the `url_prefix` keyword argument:

```
apimanager.create_api(Person, url_prefix='/api/v2')
```

Then your API for `Person` will be available at `/api/v2/person`.

4.3 Collection name

By default, the name of the collection which appears in the URLs of the API will be the name of the table that backs your model. If your model is a SQLAlchemy model, this will be the value of its `__tablename__` attribute. If your model is a Flask-SQLAlchemy model, this will be the lowercase name of the model with camel case changed to all-lowercase with underscore separators. For example, a class named `MyModel` implies a collection name of `'my_model'`. Furthermore, the URL at which this collection is accessible by default is `/api/my_model`.

To provide a different name for the model, provide a string to the *collection_name* keyword argument of the `APIManager.create_api()` method:

```
apimanager.create_api(Person, collection_name='people')
```

Then the API will be exposed at `/api/people` instead of `/api/person`.

4.4 Specifying one of many primary keys

If your model has more than one primary key (one called `id` and one called `username`, for example), you should specify the one to use:

```
manager.create_api(User, primary_key='username')
```

If you do this, Flask-Restless will create URLs like `/api/user/myusername` instead of `/api/user/137`.

4.5 Enable bulk patching or deleting

By default, a `PATCH /api/person` request (note the missing ID) will cause a 405 Method Not Allowed response. By setting the `allow_patch_many` keyword argument of the `APIManager.create_api()` method to be `True`, `PATCH /api/person` requests will patch the provided attributes on all instances of `Person`:

```
apimanager.create_api(Person, methods=['PATCH'], allow_patch_many=True)
```

If search parameters are provided via the `q` query parameter as described in *Making search queries*, only those instances matching the search will be patched.

Similarly, to allow bulk deletions, set the `allow_delete_many` keyword argument to be `True`.

4.6 Capturing validation errors

By default, no validation is performed by Flask-Restless; if you want validation, implement it yourself in your database models. However, by specifying a list of exceptions raised by your backend on validation errors, Flask-Restless will forward messages from raised exceptions to the client in an error response.

A reasonable validation framework you might use for this purpose is SQLAlchemy Validation. You can also use the `validates()` decorator that comes with SQLAlchemy.

For example, if your validation framework includes an exception called `ValidationError`, then call the `APIManager.create_api()` method with the `validation_exceptions` keyword argument:

```
from cool_validation_framework import ValidationError
apimanager.create_api(Person, validation_exceptions=[ValidationError])
```

Note: Currently, Flask-Restless expects that an instance of a specified validation error will have a `errors` attribute, which is a dictionary mapping field name to error description (note: one error per field). If you have a better, more general solution to this problem, please visit our issue tracker.

Now when you make POST and PATCH requests with invalid fields, the JSON response will look like this:

```
HTTP/1.1 400 Bad Request

{ "validation_errors":
    {
      "age": "Must be an integer",
    }
}
```

Currently, Flask-Restless can only forward one exception at a time to the client.

4.7 Exposing evaluation of SQL functions

If the `allow_functions` keyword argument is set to `True` when creating an API for a model using `APIManager.create_api()`, then an endpoint will be made available for `GET /api/eval/person` which responds to requests for evaluation of functions on all instances the model.

For information about the request and response formats for this endpoint, see *Function evaluation*.

4.8 Specifying which columns are provided in responses

By default, all columns of your model will be exposed by the API. If the `include_columns` keyword argument is an iterable of strings, *only* columns with those names (that is, the strings represent the names of attributes of the model which are `Column` objects) will be provided in JSON responses for GET requests.

For example, if your models are defined like this (using Flask-SQLAlchemy):

```
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode, unique=True)
    birth_date = db.Column(db.Date)
    computers = db.relationship('Computer')
```

and you want your JSON responses to include only the values of the `name` and `birth_date` columns, create your API with the following arguments:

```
apimanager.create_api(Person, include_columns=['name', 'birth_date'])
```

Now requests like `GET /api/person/1` will return JSON objects which look like this:

```
{"name": "Jeffrey", "birth_date": "1999-12-31"}
```

The `exclude_columns` keyword argument works similarly; it forces your JSON responses to include only the columns *not* specified in `exclude_columns`. For example:

```
apimanager.create_api(Person, exclude_columns=['name', 'birth_date'])
```

will produce responses like:

```
{"id": 1, "computers": [{"id": 1, "vendor": "Apple", "model": "MacBook"}]}
```

In this example, the `Person` model has a one-to-many relationship with the `Computer` model. To specify which columns on the related models will be included or excluded, include a string of the form `'<relation>.<column>'`, where `<relation>` is the name of the relationship attribute of the model and `<column>` is the name of the column on the related model which you want to be included or excluded. For example:

```
includes = ['name', 'birth_date', 'computers', 'computers.vendor']
apimanager.create_api(Person, include_columns=includes)
```

will produce responses like:

```
{
  "name": "Jeffrey",
  "birth_date": "1999-12-31",
  "computers": [{"vendor": "Apple"}]
}
```

An attempt to include a field on a related model without including the relationship field has no effect:

```
includes = ['name', 'birth_date', 'computers.vendor']
apimanager.create_api(Person, include_columns=includes)
```

```
{"name": "Jeffrey", "birth_date": "1999-12-31"}
```

To include the return value of an arbitrary method defined on a model, use the `include_methods` keyword argument. This argument must be an iterable of strings representing methods with no arguments (other than `self`) defined on the model for which the API will be created:

```
class Person(Base):
    __tablename__ = 'person'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode)
    age = Column(Integer)

    def name_and_age(self):
        return "%s (aged %d)" % (self.name, self.age)
```

```
include_methods = ['name_and_age']
manager.create_api(Person, include_methods=['name_and_age'])
```

A response to a GET request will then look like this:

```
{
  "id": 1,
  "name": "Paul McCartney",
  "age": 64,
  "name_and_age": "Paul McCartney (aged 64)"
}
```

4.9 Server-side pagination

To set the default number of results returned per page, use the `results_per_page` keyword argument to the `APIManager.create_api()` method. The default number of results per page is ten. The client can override the number of results per page by using a query parameter in its GET request; see *Pagination*.

To set the maximum number of results returned per page, use the `max_results_per_page` keyword argument. Even if `results_per_page` > `max_results_per_page`, at most `max_results_per_page` will be returned. The same is true if the client specifies `results_per_page` as a query argument; `max_results_per_page` provides an upper bound.

If `max_results_per_page` is set to anything but a positive integer, the client will be able to specify arbitrarily large page sizes. If, further, `results_per_page` is set to anything but a positive integer, pagination will be disabled by default, and any GET request which does not specify a page size in its query parameters will get a response with all matching results.

> **Attention:** Disabling pagination can result in large responses!

For example, to set each page to include only two results:

```
apimanager.create_api(Person, results_per_page=2)
```

Then a request to `GET /api/person` will return a JSON object which looks like this:

```
{
  "num_results": 6,
  "total_pages": 3,
  "page": 1,
  "objects": [
    {"name": "Jeffrey", "id": 1},
    {"name": "John", "id": 2}
  ]
}
```

For more information on using pagination in the client, see *Pagination*.

4.10 Request preprocessors and postprocessors

To apply a function to the request parameters and/or body before the request is processed, use the `preprocessors` keyword argument. To apply a function to the response data after the request is processed (immediately before the response is sent), use the `postprocessors` keyword argument. Both `preprocessors` and `postprocessors` must be a dictionary which maps HTTP method names as strings (with exceptions as described below) to a list of functions. The specified functions will be applied in the order given in the list.

Since GET and PATCH (and PUT) requests can be made not only on individual instances of the model but also the entire collection of instances, you must separately specify which functions to apply in the individual case and which to apply in the collection case. For example:

```
# Define pre- and postprocessor functions as described below.
def pre_get_single(**kw): pass
def pre_get_many(**kw): pass
def post_patch_many(**kw): pass
def pre_delete(**kw): pass

# Create an API for the Person model.
manager.create_api(Person,
                   # Allow GET, PATCH, and POST requests.
                   methods=['GET', 'PATCH', 'DELETE'],
                   # Allow PATCH requests modifying the whole collection.
                   allow_patch_many=True,
                   # A list of preprocessors for each method.
                   preprocessors={
                       'GET_SINGLE': [pre_get_single],
                       'GET_MANY': [pre_get_many],
                       'DELETE': [pre_delete]
                       },
                   # A list of postprocessors for each method.
                   postprocessors={
                       'PATCH_MANY': [post_patch_many]
                       }
                   )
```

As introduced in the above example, the dictionary keys for the *preprocessors* and *postprocessors* can be one of the following strings:

- `'POST'` for requests to post a new instance of the model.
- `'GET_SINGLE'` for requests to get a single instance of the model.
- `'GET_MANY'` for requests to get multiple instances of the model.

- `'PATCH_SINGLE'` or `'PUT_SINGLE'` for requests to patch a single instance of the model.
- `'PATCH_MANY'` or `'PUT_MANY'` for requests to patch multiple instances of the model.
- `'DELETE_SINGLE'` for requests to delete an instance of the model.
- `'DELETE_MANY'` for requests to delete multiple instances of the model.

Note: Since PUT requests are handled by the PATCH handler, any preprocessors or postprocessors specified for the PUT method will be applied on PATCH requests *after* the preprocessors or postprocessors specified for the PATCH method.

The preprocessors and postprocessors for each type of request accept different arguments. Most of them should have no return value (more specifically, any returned value is ignored). The return value from each of the `GET_SINGLE`, `PATCH_SINGLE`, and `DELETE_SINGLE` preprocessors is interpreted as a value with which to replace `instance_id`, the variable containing the value of the primary key of the requested instance of the model. For example, if a request for `GET /api/person/1` encounters a preprocessor (for `GET_SINGLE`) that returns the integer `8`, Flask-Restless will continue to process the request as if it had received `GET /api/person/8`. (If multiple preprocessors are specified for a single HTTP method and each one has a return value, Flask-Restless will only remember the value returned by the last preprocessor function.)

Those preprocessors and postprocessors that accept dictionaries as parameters can (and should) modify their arguments *in-place*. That means the changes made to, for example, the `result` dictionary will be seen by the Flask-Restless view functions and ultimately returned to the client.

The arguments to the preprocessor and postprocessor functions will be provided as keyword arguments, so you should always add `**kw` as the final argument when defining a preprocessor or postprocessor function. This way, you can specify only the keyword arguments you need when defining your functions. Furthermore, if a new version of Flask-Restless changes the API, you can update Flask-Restless without breaking your code.

Changed in version 0.16.0: Replaced `DELETE` with `DELETE_MANY` and `DELETE_SINGLE`.

New in version 0.13.0: Functions provided as postprocessors for `GET_MANY` and `PATCH_MANY` requests receive the `search_params` keyword argument, so that both preprocessors and postprocessors have access to this information.

- GET for a single instance:

```
    def get_single_preprocessor(instance_id=None, **kw):
        """Accepts a single argument, `instance_id`, the primary key of the
        instance of the model to get.

        """
        pass
```

```
    def get_single_postprocessor(result=None, **kw):
        """Accepts a single argument, `result`, which is the dictionary
        representation of the requested instance of the model.

        """
        pass
```

and for the collection:

```
    def get_many_preprocessor(search_params=None, **kw):
        """Accepts a single argument, `search_params`, which is a dictionary
        containing the search parameters for the request.

        """
        pass

    def get_many_postprocessor(result=None, search_params=None, **kw):
        """Accepts two arguments, `result`, which is the dictionary
        representation of the JSON response which will be returned to the
        client, and `search_params`, which is a dictionary containing the
        search parameters for the request (that produced the specified
        `result`).

        """
        pass
```

- PATCH (or PUT) for a single instance:

```
    def patch_single_preprocessor(instance_id=None, data=None, **kw):
        """Accepts two arguments, `instance_id`, the primary key of the
        instance of the model to patch, and `data`, the dictionary of fields
        to change on the instance.

        """
        pass

    def patch_single_postprocessor(result=None, **kw):
        """Accepts a single argument, `result`, which is the dictionary
        representation of the requested instance of the model.

        """
        pass
```

and for the collection:

```
    def patch_many_preprocessor(search_params=None, data=None, **kw):
        """Accepts two arguments: `search_params`, which is a dictionary
        containing the search parameters for the request, and `data`, which
        is a dictionary representing the fields to change on the matching
        instances and the values to which they will be set.

        """
```

```
        pass

    def patch_many_postprocessor(query=None, data=None, search_params=None,
                                 **kw):
        """Accepts three arguments: `query`, which is the SQLAlchemy query
        which was inferred from the search parameters in the query string,
        `data`, which is the dictionary representation of the JSON response
        which will be returned to the client, and `search_params`, which is a
        dictionary containing the search parameters for the request.

        """
        pass
```

- POST:

```
    def post_preprocessor(data=None, **kw):
        """Accepts a single argument, `data`, which is the dictionary of
        fields to set on the new instance of the model.

        """
        pass

    def post_postprocessor(result=None, **kw):
        """Accepts a single argument, `result`, which is the dictionary
        representation of the created instance of the model.

        """
        pass
```

- DELETE for a single instance:

```
    def delete_single_preprocessor(instance_id=None, **kw):
        """Accepts a single argument, `instance_id`, which is the primary key
        of the instance which will be deleted.

        """
        pass

    def delete_postprocessor(was_deleted=None, **kw):
        """Accepts a single argument, `was_deleted`, which represents whether
        the instance has been deleted.

        """
        pass
```

and for the collection:

```
    def delete_many_preprocessor(search_params=None, **kw):
        """Accepts a single argument, `search_params`, which is a dictionary
        containing the search parameters for the request.

        """
```

```
        pass

    def delete_many_postprocessor(result=None, search_params=None, **kw):
        """Accepts two arguments: `result`, which is the dictionary
        representation of which is the dictionary representation of the JSON
        response which will be returned to the client, and `search_params`,
        which is a dictionary containing the search parameters for the
        request.

        """
        pass
```

Note: For more information about search parameters, see *Making search queries*, and for more information about request and response formats, see *Format of requests and responses*.

In order to halt the preprocessing or postprocessing and return an error response directly to the client, your preprocessor or postprocessor functions can raise a `ProcessingException`. If a function raises this exception, no preprocessing or postprocessing functions that appear later in the list specified when the API was created will be invoked. For example, an authentication function can be implemented like this:

```
def check_auth(instance_id=None, **kw):
    # Here, get the current user from the session.
    current_user = ...
    # Next, check if the user is authorized to modify the specified
    # instance of the model.
    if not is_authorized_to_modify(current_user, instance_id):
        raise ProcessingException(description='Not Authorized',
                                  code=401)
manager.create_api(Person, preprocessors=dict(GET_SINGLE=[check_auth]))
```

The `ProcessingException` allows you to specify an HTTP status code for the generated response and an error message which the client will receive as part of the JSON in the body of the response.

4.10.1 Universal preprocessors and postprocessors

New in version 0.13.0.

The previous section describes how to specify a preprocessor or postprocessor on a per-API (that is, a per-model) basis. If you want a function to be executed for *all* APIs created by a `APIManager`, you can use the `preprocessors` or `postprocessors` keyword arguments in the constructor of the `APIManager` class. These keyword arguments have the same format as the corresponding ones in the `APIManager.create_api()` method as described above. Functions specified in this way are prepended to the list of preprocessors or postprocessors specified in the `APIManager.create_api()` method.

This may be used, for example, if all POST requests require authentication:

```
from flask import Flask
from flask.ext.restless import APIManager
from flask.ext.restless import ProcessingException
from flask.ext.login import current_user
from mymodels import User
from mymodels import session

def auth_func(*args, **kw):
    if not current_user.is_authenticated():
        raise ProcessingException(description='Not authenticated!', code=401)

app = Flask(__name__)
api_manager = APIManager(app, session=session,
                         preprocessors=dict(POST=[auth_func]))
api_manager.create_api(User)
```

4.10.2 Preprocessors for collections

When the server receives, for example, a request for `GET /api/person`, Flask-Restless interprets this request as a search with no filters (that is, a search for all instances of `Person` without exception). In other words, `GET /api/person` is roughly equivalent to `GET /api/person?q={}`. Therefore, if you want to filter the set of `Person` instances returned by such a request, you can create a preprocessor for a GET request to the collection endpoint that *appends filters* to the `search_params` keyword argument. For example:

```
def preprocessor(search_params=None, **kw):
    # This checks if the preprocessor function is being called before a
    # request that does not have search parameters.
    if search_params is None:
        return
    # Create the filter you wish to add; in this case, we include only
    # instances with ``id`` not equal to 1.
    filt = dict(name='id', op='neq', val=1)
    # Check if there are any filters there already.
    if 'filters' not in search_params:
        search_params['filters'] = []
    # *Append* your filter to the list of filters.
    search_params['filters'].append(filt)

apimanager.create_api(Person, preprocessors=dict(GET_MANY=[preprocessor]))
```

4.11 Custom queries

In cases where it is not possible to use preprocessors or postprocessors (*Request preprocessors and postprocessors*) efficiently, you can provide a custom `query` attribute

to your model instead. The attribute can either be a SQLAlchemy query expression or a class method that returns a SQLAlchemy query expression. Flask-Restless will use this `query` attribute internally, however it is defined, instead of the default `session.query(Model)` (in the pure SQLAlchemy case) or `Model.query` (in the Flask-SQLAlchemy case). Flask-Restless uses a query during most GET and PATCH requests to find the model(s) being requested.

You may want to use a custom query attribute if you want to reveal only certain information to the client. For example, if you have a set of people and you only want to reveal information about people from the group named "students", define a query class method this way:

```
class Group(Base):
    __tablename__ = 'group'
    id = Column(Integer, primary_key=True)
    groupname = Column(Unicode)
    people = relationship('Person')

class Person(Base):
    __tablename__ = 'person'
    id = Column(Integer, primary_key=True)
    group_id = Column(Integer, ForeignKey('group.id'))
    group = relationship('Group')

    @classmethod
    def query(cls):
        original_query = session.query(cls)
        condition = (Group.groupname == 'students')
        return original_query.join(Group).filter(condition)
```

Then requests to, for example, `GET /api/person` will only reveal instances of `Person` who also are in the group named "students".

4.11.1 Requiring authentication for some methods

If you want certain HTTP methods to require authentication, use preprocessors:

```
from flask import Flask
from flask.ext.restless import APIManager
from flask.ext.restless import NO_CHANGE
from flask.ext.restless import ProcessingException
from flask.ext.login import current_user
from mymodels import User

def auth_func(*args, **kwargs):
    if not current_user.is_authenticated():
        raise ProcessingException(description='Not authenticated!', code=401)
    return True

app = Flask(__name__)
```

```
api_manager = APIManager(app)
api_manager.create_api(User, preprocessors=dict(GET_SINGLE=[auth_func],
                                                GET_MANY=[auth_func]))
```

For a more complete example using Flask-Login, see the `examples/server_configurations/authentication` directory in the source distribution, or view it online at GitHub.

4.11.2 Enabling Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is a protocol that allows JavaScript HTTP clients to make HTTP requests across Internet domain boundaries while still protecting against cross-site scripting (XSS) attacks. If you have access to the HTTP server that serves your Flask application, I recommend configuring CORS there, since such concerns are beyond the scope of Flask-Restless. However, in case you need to support CORS at the application level, you should create a function that adds the necessary HTTP headers after the request has been processed by Flask-Restless (that is, just before the HTTP response is sent from the server to the client) using the `flask.Blueprint.after_request()` method:

```
from flask import Flask
from flask.ext.restless import APIManager

def add_cors_headers(response):
    response.headers['Access-Control-Allow-Origin'] = 'example.com'
    response.headers['Access-Control-Allow-Credentials'] = 'true'
    # Set whatever other headers you like...
    return response

app = Flask(__name__)
manager = APIManager(app)
blueprint = manager.create_api(Person)
blueprint.after_request(add_cors_headers)
```

CHAPTER 5

Making search queries

Clients can make GET requests on individual instances of a model (for example, `GET /api/person/1`) and on collections of all instances of a model (`GET /api/person`). To get all instances of a model that meet some criteria, clients can make GET requests with a query parameter specifying a search. The search functionality in Flask-Restless is relatively simple, but should suffice for many cases.

5.1 Quick examples

The following are some quick examples of creating search queries with different types of clients. Find more complete documentation in subsequent sections. In these examples, each client will search for instances of the model `Person` whose names contain the letter "y".

Using the Python requests library:

```
import requests
import json

url = 'http://127.0.0.1:5000/api/person'
headers = {'Content-Type': 'application/json'}

filters = [dict(name='name', op='like', val='%y%')]
params = dict(q=json.dumps(dict(filters=filters)))

response = requests.get(url, params=params, headers=headers)
assert response.status_code == 200
print(response.json())
```

Using jQuery:

```
var filters = [{"name": "id", "op": "like", "val": "%y%"}];
$.ajax({
  url: 'http://127.0.0.1:5000/api/person',
```

```
  data: {"q": JSON.stringify({"filters": filters})},
  dataType: "json",
  contentType: "application/json",
  success: function(data) { console.log(data.objects); }
});
```

Using curl:

```
curl \
  -G \
  -H "Content-type: application/json" \
  -d "q={\"filters\":[{\"name\":\"name\",\"op\":\"like\",\"val\":\"%y%\"}]}" \
  http://127.0.0.1:5000/api/person
```

The `examples/` directory has more complete versions of these examples.

5.2 Query format

The query parameter `q` must be a JSON string. It can have the following mappings, all of which are optional:

`filters` A list of objects of one of the following forms:

```
{"name": <fieldname>, "op": <operatorname>, "val": <argument>}
```

or:

```
{"name": <fieldname>, "op": <operatorname>, "field": <fieldname>}
```

In the first form, `<operatorname>` is one of the strings described in the *Operators* section, the first `<fieldname>` is the name of the field of the model to which to apply the operator, `<argument>` is a value to be used as the second argument to the given operator. In the second form, the second `<fieldname>` is the field of the model that should be used as the second argument to the operator.

`<fieldname>` may alternately specify a field on a related model, if it is a string of the form `<relationname>__<fieldname>`.

If the field name is the name of a relation and the operator is `"has"` or `"any"`, the `"val"` argument can be a dictionary with the arguments representing another filter to be applied as the argument for `"has"` or `"any"`.

The returned list of matching instances will include only those instances that satisfy all of the given filters.

`disjunction` A Boolean that specifies whether the list of filters should be treated as a disjunction or a conjunction. If this is `true`, the response will include all instances of the model that match *any* of the filters. If this is `false` the response will include all instances of the model that match *all* of the filters. This will be treated as `false` if not specified by the client (in other words, the default is conjunction).

`limit` A positive integer which specifies the maximum number of objects to return.

offset A positive integer which specifies the offset into the result set of the returned list of instances.

order_by A list of objects of the form:

```
{"field": <fieldname>, "direction": <directionname>}
```

where `<fieldname>` is a string corresponding to the name of a field of the requested model and `<directionname>` is either `"asc"` for ascending order or `"desc"` for descending order.

`<fieldname>` may alternately specify a field on a related model, if it is a string of the form `<relationname>__<fieldname>`.

group_by A list of objects of the form:

```
{"field": <fieldname>}
```

where `<fieldname>` is a string corresponding to the name of a field of the requested model.

`<fieldname>` may alternately specify a field on a related model, if it is a string of the form `<relationname>__<fieldname>`.

single A Boolean representing whether a single result is expected as a result of the search. If this is `true` and either no results or multiple results meet the criteria of the search, the server responds with an error message.

If a filter is poorly formatted (for example, `op` is set to `'=='` but `val` is not set), the server responds with 400 Bad Request.

5.3 Operators

The operator strings recognized by the API incude:

- `==`, `eq`, `equals`, `equals_to`
- `!=`, `neq`, `does_not_equal`, `not_equal_to`
- `>`, `gt`, `<`, `lt`
- `>=`, `ge`, `gte`, `geq`, `<=`, `le`, `lte`, `leq`
- `in`, `not_in`
- `is_null`, `is_not_null`
- `like`
- `has`
- `any`

These correspond to SQLAlchemy column operators as defined here.

5.4 Examples

Consider a `Person` model available at the URL `/api/person`, and suppose all of the following requests are `GET /api/person` requests with query parameter `q`.

5.4.1 Attribute greater than a value

On request:

```
GET /api/person?q={"filters":[{"name":"age","op":"ge","val":10}]} HTTP/1.1
Host: example.com
```

the response will include only those `Person` instances that have `age` attribute greater than or equal to 10:

```
HTTP/1.1 200 OK

{
  "num_results": 8,
  "total_pages": 3,
  "page": 2,
  "objects":
  [
    {"id": 1, "name": "Jeffrey", "age": 24},
    {"id": 2, "name": "John", "age": 13},
    {"id": 3, "name": "Mary", "age": 18}
  ]
}
```

5.4.2 Disjunction of filters

On request:

```
GET /api/person?q={"filters":[{"name":"age","op":"lt","val":10},{"name":"age","op":"gt","val":20
Host: example.com
```

the response will include only those `Person` instances that have `age` attribute either less than 10 or greater than 20:

```
HTTP/1.1 200 OK

{
  "num_results": 3,
  "total_pages": 1,
  "page": 1,
  "objects":
  [
    {"id": 4, "name": "Abraham", "age": 9},
    {"id": 5, "name": "Isaac", "age": 25},
```

```
    {"id": 6, "name": "Job", "age": 37}
  ]
}
```

5.4.3 Attribute between two values

On request:

```
GET /api/person?q={"filters":[{"name":"age","op":"ge","val":10},{"name":"age","op":"le","val":20
Host: example.com
```

the response will include only those `Person` instances that have age attribute between 10 and 20, inclusive:

```
HTTP/1.1 200 OK

{
  "num_results": 6,
  "total_pages": 3,
  "page": 2,
  "objects":
  [
    {"id": 2, "name": "John", "age": 13},
    {"id": 3, "name": "Mary", "age": 18}
  ]
}
```

5.4.4 Expecting a single result

On request:

```
GET /api/person?q={"filters":[{"name":"id","op":"eq","val":1}],"single":true} HTTP/1.1
Host: example.com
```

the response will include only the sole `Person` instance with `id` equal to 1:

```
HTTP/1.1 200 OK

{"id": 1, "name": "Jeffrey", "age": 24}
```

In the case that the search would return no results or more than one result, an error response is returned instead:

```
GET /api/person?q={"filters":[{"name":"age","op":"ge","val":10}],"single":true} HTTP/1.1
Host: example.com
```

```
HTTP/1.1 400 Bad Request

{"message": "Multiple results found"}
```

```
GET /api/person?q={"filters":[{"name":"id","op":"eq","val":-1}],"single":true} HTTP/1.1
Host: example.com
```

```
HTTP/1.1 404 Bad Request

{"message": "No result found"}
```

5.4.5 Comparing two attributes

On request:

```
GET /api/person?q={"filters":[{"name":"age","op":"ge","field":"height"}]} HTTP/1.1
Host: example.com
```

the response will include only those `Person` instances that have age attribute greater than or equal to the value of the `height` attribute:

```
HTTP/1.1 200 OK

{
  "num_results": 6,
  "total_pages": 3,
  "page": 2,
  "objects":
  [
    {"id": 1, "name": "John", "age": 80, "height": 65},
    {"id": 2, "name": "Mary", "age": 73, "height": 60}
  ]
}
```

5.4.6 Comparing attribute of a relation

On request:

```
GET /api/person?q={"filters":[{"name":"computers__manufacturer","op":"any","val":"Apple"}],"sing
Host: example.com
```

response will include only those `Person` instances that are related to any `Computer` model that is manufactured by Apple:

```
HTTP/1.1 200 OK

{
  "num_results": 6,
  "total_pages": 3,
  "page": 2,
  "objects":
    {
      "id": 1,
```

```
      "name": "John",
      "computers": [
        { "id": 1, "manufacturer": "Dell", "model": "Inspiron 9300"},
        { "id": 2, "manufacturer": "Apple", "model": "MacBook"}
      ]
    },
    {
      "id": 2,
      "name": "Mary",
      "computers": [
        { "id": 3, "manufacturer": "Apple", "model": "iMac"}
      ]
    }
  ]
}
```

5.4.7 Using has and any

Use the has and any operators to search for instances by fields on related instances. For example, you can search for all Person instances that have a related Computer with a certain ID number by using the any operator. For another example, you can search for all Computer instances that have an owner with a certain name by using the has operator. In general, use the any operator if the relation is a list of objects and use the has operator if the relation is a single object. For more information, see the SQLAlchemy documentation.

On request:

```
GET /api/person?q={"filters":[{"name":"computers","op":"any","val":{"name":"id","op":"gt","val":
Host: example.com
```

the response will include only those Person instances that have a related Computer instance with id field of value greater than 1:

```
HTTP/1.1 200 OK

{
  "num_results": 6,
  "total_pages": 3,
  "page": 2,
  "objects":
  [
    {"id": 1, "name": "John", "age": 80, "height": 65, "computers": [...]},
    {"id": 2, "name": "Mary", "age": 73, "height": 60, "computers": [...]}
  ]
}
```

On request:

```
GET /api/computers?q={"filters":[{"name":"owner","op":"has","val":{"name":"vendor","op":"ilike",
Host: example.com
```

the response will include only those `Computer` instances that have an owner with `name` field that includes `'John'`:

```
HTTP/1.1 200 OK

{
  "num_results": 6,
  "total_pages": 3,
  "page": 2,
  "objects":
  [
    {"id": 1, "name": "pluto", vendor="Apple", ...},
    {"id": 2, "name": "jupiter", vendor="Dell", ...}
  ]
}
```

Format of requests and responses

Requests and responses are all in JSON format, so the mimetype is `application/json`. Ensure that requests you make that require a body (PATCH and POST requests) have the header `Content-Type: application/json`; if they do not, the server will respond with a 415 Unsupported Media Type.

Suppose we have the following Flask-SQLAlchemy models (the example works with pure SQLALchemy just the same):

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

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)

class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode, unique=True)
    birth_date = db.Column(db.Date)
    computers = db.relationship('Computer',
                                backref=db.backref('owner',
                                                   lazy='dynamic'))

class Computer(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode, unique=True)
    vendor = db.Column(db.Unicode)
    owner_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    purchase_time = db.Column(db.DateTime)
```

Also suppose we have registered an API for these models at `/api/person` and `/api/computer`, respectively.

Note: API endpoints do not have trailing slashes. A request to, for example,

`/api/person/` will result in a 404 Not Found response.

Note: For all requests that would return a list of results, the top-level JSON object is a mapping from `"objects"` to the list. JSON lists are not sent as top-level objects for security reasons. For more information, see this.

`GET /api/person`

Gets a list of all `Person` objects.

Sample response:

```
HTTP/1.1 200 OK

{
  "num_results": 8,
  "total_pages": 3,
  "page": 2,
  "objects": [{"id": 1, "name": "Jeffrey", "age": 24}, ...]
}
```

`GET /api/person?q=<searchjson>`

Gets a list of all `Person` objects which meet the criteria of the specified search. For more information on the format of the value of the `q` parameter, see *Making search queries*.

Sample response:

```
HTTP/1.1 200 OK

{
   "num_results": 8,
   "total_pages": 3,
   "page": 2,
   "objects": [{"id": 1, "name": "Jeffrey", "age": 24}, ...]
 }
```

`GET /api/person/`(`int:` *id*)

Gets a single instance of `Person` with the specified ID.

Sample response:

```
HTTP/1.1 200 OK

{"id": 1, "name": "Jeffrey", "age": 24}
```

`GET /api/person/`(`int:` *id*)`/computers`

Gets a list of all `Computer` objects which are owned by the `Person` object with the specified ID.

Sample response:

```
HTTP/1.1 200 OK

{
  "num_results": 2,
  "total_pages": 1,
  "page": 1,
  "objects": [{"id": 1, "vendor": "Apple", "name": "MacBook", ...}, ...]
}
```

DELETE /api/person/(int: *id*)
Deletes the instance of `Person` with the specified ID.

Sample response:

```
HTTP/1.1 204 No Content
```

DELETE /api/person/(int: *id*)**/computers/**
int: *id* Removes the instance of `Computer` with the specified ID from the `computers` collection of the instance of `Person` with the specified ID. This is essentially a shortcut to using a `PATCH /api/person/(int:id)/computers` request with a `remove` parameter in the body of the request.

Sample response:

```
HTTP/1.1 204 No Content
```

POST /api/person
Creates a new person with initial attributes specified as a JSON string in the body of the request.

Sample request:

```
POST /api/person HTTP/1.1
Host: example.com

{"name": "Jeffrey", "age": 24}
```

Sample response:

```
HTTP/1.1 201 Created

{
  "id": 1,
  "name": "Jeffrey",
  "age" 24,
  "computers": []
}
```

The server will respond with 400 Bad Request if the request specifies a field which does not exist on the model.

To create a new person which includes a related list of **new** computer instances via a one-to-many relationship, a request must take the following form.

Sample request:

```
POST /api/person HTTP/1.1
Host: example.com

{
  "name": "Jeffrey",
  "age": 24,
  "computers":
    [
      {"manufacturer": "Dell", "model": "Inspiron"},
      {"manufacturer": "Apple", "model": "MacBook"}
    ]
}
```

Sample response:

```
HTTP/1.1 201 Created

{
  "id": 1,
  "name": "Jeffrey",
  "age": 24,
  "computers":
    [
      {"id": 1, "manufacturer": "Dell", "model": "Inspiron"},
      {"id": 2, "manufacturer": "Apple", "model": "MacBook"}
    ]
}
```

> **Warning:** The response does not denote that new instances have been created for the `Computer` models.

To create a new person which includes a single related **new** computer instance (via a one-to-one relationship), a request must take the following form.

Sample request:

```
POST /api/person HTTP/1.1
Host: example.com

{
  "name": "Jeffrey",
  "age": 24,
  "computer": {"manufacturer": "Dell", "model": "Inspiron"}
}
```

Sample response:

```
HTTP/1.1 201 Created

{
```

```
  "name": "Jeffrey",
  "age": 24,
  "id": 1,
  "computer": {"id": 1, "manufacturer": "Dell", "model": "Inspiron"}
}
```

> **Warning:** The response does not denote that a new `Computer` instance has been created.

To create a new person which includes a related list of **existing** computer instances via a one-to-many relationship, a request must take the following form.

Sample request:

```
POST /api/person HTTP/1.1
Host: example.com

{
  "name": "Jeffrey",
  "age": 24,
  "computers": [ {"id": 1}, {"id": 2} ]
}
```

Sample response:

```
HTTP/1.1 201 Created

{
  "id": 1,
  "name": "Jeffrey",
  "age": 24,
  "computers":
    [
      {"id": 1, "manufacturer": "Dell", "model": "Inspiron"},
      {"id": 2, "manufacturer": "Apple", "model": "MacBook"}
    ]
}
```

To create a new person which includes a single related **existing** computer instance (via a one-to-one relationship), a request must take the following form.

Sample request:

```
POST /api/person HTTP/1.1
Host: example.com

{
  "name": "Jeffrey",
  "age": 24,
  "computer": {"id": 1}
}
```

Sample response:

```
HTTP/1.1 201 Created

{
  "name": "Jeffrey",
  "age": 24,
  "id": 1,
  "computer": {"id": 1, "manufacturer": "Dell", "model": "Inspiron"}
}
```

PATCH /api/person

PUT /api/person

Sets specified attributes on every instance of `Person` which meets the search criteria described in the `q` parameter.

The JSON object specified in the body of a PATCH request to this endpoint may include a mapping from *q* to the parameters for a search, as described in *Making search queries*. If no *q* key exists, then all instances of the model will be patched.

`PUT /api/person` is an alias for `PATCH /api/person`, because the latter is more semantically correct but the former is part of the core HTTP standard.

The response will return a JSON object which specifies the number of instances in the `Person` database which were modified.

Sample request:

Suppose the database contains exactly three people with the letter "y" in his or her name.

```
PATCH /api/person HTTP/1.1
Host: example.com

{
  "age": 1,
  "q": {"filters": [{"name": "name", "op": "like", "val": "%y%"}]}
}
```

Sample response:

```
HTTP/1.1 201 Created

{"num_modified": 3}
```

PATCH /api/person/(int: *id*)

PUT /api/person/(int: *id*)

Sets specified attributes on the instance of `Person` with the specified ID number. `PUT /api/person/1` is an alias for `PATCH /api/person/1`, because the latter is more semantically correct but the former is part of the core HTTP standard.

Sample request:

```
PATCH /api/person/1 HTTP/1.1
Host: example.com

{"name": "Foobar"}
```

Sample response:

```
HTTP/1.1 200 OK

{"id": 1, "name": "Foobar", "age": 24}
```

The server will respond with 400 Bad Request if the request specifies a field which does not exist on the model.

To add a list of existing objects to a one-to-many relationship, a request must take the following form.

Sample request:

```
PATCH /api/person/1 HTTP/1.1
Host: example.com

{ "computers":
  {
    "add": [ {"id": 1} ]
  }
}
```

Sample response:

```
HTTP/1.1 200 OK

{
  "id": 1,
  "name": "Jeffrey",
  "age": 24,
  "computers": [ {"id": 1, "manufacturer": "Dell", "model": "Inspiron"} ]
}
```

To add a list of new objects to a one-to-many relationship, a request must take the following form.

Sample request:

```
PATCH /api/person/1 HTTP/1.1
Host: example.com

{ "computers":
  {
    "add": [ {"manufacturer": "Dell", "model": "Inspiron"} ]
  }
}
```

Warning: The response does not denote that a new instance has been created for the `Computer` model.

Sample response:

```
HTTP/1.1 200 OK

{
  "id": 1,
  "name": "Jeffrey",
  "age": 24,
  "computers": [ {"id": 1, "manufacturer": "Dell", "model": "Inspiron"} ]
}
```

Similarly, to add a new or existing instance of a related model to a one-to-one relationship, a request must take the following form.

Sample request:

```
PATCH /api/person/1 HTTP/1.1
Host: example.com

{ "computers":
  {
    "add": {"id": 1}
  }
}
```

Sample response:

```
HTTP/1.1 200 OK

{
  "id": 1,
  "name": "Jeffrey",
  "age": 24,
  "computers": [ {"id": 1, "manufacturer": "Dell", "model": "Inspiron"} ]
}
```

To remove an existing object (without deleting that object from its own database) from a one-to-many relationship, a request must take the following form.

Sample request:

```
PATCH /api/person/1 HTTP/1.1
Host: example.com

{ "computers":
  {
    "remove": [ {"id": 2} ]
  }
}
```

Sample response:

```
HTTP/1.1 200 OK

{
  "id": 1,
  "name": "Jeffrey",
  "age": 24,
  "computers": [
    {"id": 1, "manufacturer": "Dell", "model": "Inspiron 9300"},
    {"id": 3, "manufacturer": "Apple", "model": "MacBook"}
  ]
}
```

To remove an existing object from a one-to-many relationship and additionally delete it from its own database, a request must take the following form.

Sample request:

```
PATCH /api/person/1 HTTP/1.1
Host: example.com

{ "computers":
  {
    "remove": [ {"id": 2, "__delete__": true} ]
  }
}
```

> **Warning:** The response does not denote that the instance was deleted from its own database.

Sample response:

```
HTTP/1.1 200 OK

{
  "id": 1,
  "name": "Jeffrey",
  "age": 24,
  "computers": [
    {"id": 1, "manufacturer": "Dell", "model": "Inspiron 9300"},
    {"id": 3, "manufacturer": "Apple", "model": "MacBook"}
  ]
}
```

To set the value of a one-to-many relationship to contain either existing or new instances of the related model, a request must take the following form.

Sample request:

```
PATCH /api/person/1 HTTP/1.1
Host: example.com
```

```
{ "computers":
    [
      {"id": 1},
      {"id": 3},
      {"manufacturer": "Lenovo", "model": "ThinkPad"}
    ]
}
```

Sample response:

```
HTTP/1.1 200 OK

{
  "id": 1,
  "name": "Jeffrey",
  "age": 24,
  "computers": [
    {"id": 1, "manufacturer": "Dell", "model": "Inspiron 9300"},
    {"id": 3, "manufacturer": "Apple", "model": "MacBook"}
    {"id": 4, "manufacturer": "Lenovo", "model": "ThinkPad"}
  ]
}
```

To set the value of a one-to-many relationship *and* update fields on existing instances of the related model, a request must take the following form.

Suppose the `Person` instance looked like this before the sample PATCH request below:

```
HTTP/1.1 200 OK

{
  "id": 1,
  "name": "Jeffrey",
  "age": 24,
  "computers": [
    {"id": 1, "manufacturer": "Apple", "model": "MacBook"}
  ]
}
```

Sample request:

```
PATCH /api/person/1 HTTP/1.1
Host: example.com

{ "computers":
    [
      {"id": 1, "manufacturer": "Lenovo", "model": "ThinkPad"}
    ]
}
```

Sample response:

```
HTTP/1.1 200 OK

{
  "id": 1,
  "name": "Jeffrey",
  "age": 24,
  "computers": [
    {"id": 1, "manufacturer": "Lenovo", "model": "ThinkPad"}
  ]
}
```

The changes reflected in this response have been made to the `Computer` instance with ID 1.

6.1 Date and time fields

Flask-Restless will automatically parse and convert date and time strings into the corresponding Python objects. Flask-Restless also understands intervals (also known as durations), if you specify the interval as an integer representing the number of seconds that the interval spans.

If you want the server to set the value of a date or time field of a model as the current time (as measured at the server), use one of the special strings `"CURRENT_TIMESTAMP"`, `"CURRENT_DATE"`, or `"LOCALTIMESTAMP"`. When the server receives one of these strings in a request, it will use the corresponding SQL function to set the date or time of the field in the model.

6.2 Errors and error messages

Most errors return 400 Bad Request. A bad request, for example, will receive a response like this:

```
HTTP/1.1 400 Bad Request

{"message": "Unable to decode data"}
```

If your request triggers a SQLAlchemy `DataError`, `IntegrityError`, or `ProgrammingError`, the session will be rolled back.

6.3 Function evaluation

If the `allow_functions` keyword argument is set to `True` when creating an API for a model using `APIManager.create_api()`, then an endpoint will be made available for `GET /api/eval/person` which responds to requests for evaluation of functions on *all* instances the model.

Sample request:

```
GET /api/eval/person?q={"functions": [{"name": "sum", "field": "age"}, {"name": "avg", "field":
```

The format of the response is

```
HTTP/1.1 200 OK

{"sum__age": 100, "avg_height": 68}
```

If no functions are specified in the request, the response will contain the empty JSON object, `{}`.

Note: The functions whose names are given in the request will be evaluated using SQLAlchemy's func object.

Example

To get the total number of rows in the query (that is, the number of instances of the requested model), use `count` as the name of the function to evaluate, and `id` for the field on which to evaluate it:

Request:

```
GET /api/eval/person?q={"functions": [{"name": "count", "field": "id"}]} HTTP/1.1
```

Response:

```
HTTP/1.1 200 OK

{"count__id": 5}
```

6.4 JSONP callbacks

Add a `callback=myfunc` query parameter to the request URL on any GET requests (including endpoints for function evaluation) to have the JSON data of the response wrapped in the Javascript function `myfunc`. This can be used to circumvent some cross domain scripting security issues. For example, a request like this:

```
GET /api/person/1?callback=foo HTTP/1.1
```

will produce a response like this:

```
HTTP/1.1 200 OK

foo({"meta": ..., "data": ...})
```

Then in your Javascript code, write the function `foo` like this:

```
function foo(response) {
  var meta, data;
  meta = response.meta;
  data = response.data;
  // Do something cool here...
}
```

The metadata includes the status code and the values of the HTTP headers, including the Link headers parsed in JSON format. For example, a link that looks like this:

```
Link: <url1>; rel="next", <url2>; rel="foo"; bar="baz"
```

will look like this in the JSON metadata:

```
[
  {"url": "url1", "rel": "next"},
  {"url": "url2", "rel": "foo", "bar": "baz"}
]
```

The mimetype of a JSONP response is `application/javascript` instead of the usual `application/json`, because the payload of such a response is not valid JSON.

6.5 Pagination

Responses to GET requests are paginated by default, with at most ten objects per page. To request a specific page, add a `page=N` query parameter to the request URL, where `N` is a positive integer (the first page is page one). If no page query parameter is specified, the first page will be returned.

In order to specify the number of results per page, add the query parameter `results_per_page=N` where `N` is a positive integer. If `results_per_page` is greater than the maximum number of results per page as configured by the server (see *Server-side pagination*), then the query parameter will be ignored.

In addition to the `"objects"` list, the response JSON object will have a `"page"` key whose value is the current page, a `"num_pages"` key whose value is the total number of pages into which the set of matching instances is divided, and a `"num_results"` key whose value is the total number of instances which match the requested search. For example, a request to `GET /api/person?page=2` will result in the following response:

```
HTTP/1.1 200 OK

{
  "num_results": 8,
  "page": 2,
  "num_pages": 3,
  "objects": [{"id": 1, "name": "Jeffrey", "age": 24}, ...]
}
```

If pagination is disabled (by setting `results_per_page=None` in `APIManager.create_api()`, for example), any `page` key in the query parameters will be ignored, and the response JSON will include a `"page"` key which always has the value `1`.

Note: As specified in in *Query format*, clients can receive responses with `limit` (a maximum number of objects in the response) and `offset` (the number of initial objects to skip in the response) applied. It is possible, though not recommended, to use pagination in addition to `limit` and `offset`. For simple clients, pagination should be fine.

Part II

API REFERENCE

CHAPTER 7

API

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

class flask.ext.restless.**APIManager**(*app=None*, ***kw*)

Provides a method for creating a public ReSTful JSON API with respect to a given `Flask` application object.

The `Flask` object can be specified in the constructor, or after instantiation time by calling the `init_app()` method. In any case, the application object must be specified before calling the `create_api()` method.

app is the `flask.Flask` object containing the user's Flask application.

session is the `sqlalchemy.orm.session.Session` object in which changes to the database will be made.

flask_sqlalchemy_db is the `flask.ext.sqlalchemy.SQLAlchemy` object with which *app* has been registered and which contains the database models for which API endpoints will be created.

If *flask_sqlalchemy_db* is not `None`, *session* will be ignored.

For example, to use this class with models defined in pure SQLAlchemy:

```
from flask import Flask
from flask.ext.restless import APIManager
from sqlalchemy import create_engine
from sqlalchemy.orm.session import sessionmaker

engine = create_engine('sqlite:////tmp/mydb.sqlite')
Session = sessionmaker(bind=engine)
mysession = Session()
app = Flask(__name__)
apimanager = APIManager(app, session=mysession)
```

and with models defined with Flask-SQLAlchemy:

```
from flask import Flask
from flask.ext.restless import APIManager
from flask.ext.sqlalchemy import SQLAlchemy

app = Flask(__name__)
db = SQLALchemy(app)
apimanager = APIManager(app, flask_sqlalchemy_db=db)
```

init_app(*app*, *session=None*, *flask_sqlalchemy_db=None*, *preprocessors=None*, *postprocessors=None*)

Stores the specified `flask.Flask` application object on which API endpoints will be registered and the `sqlalchemy.orm.session.Session` object in which all database changes will be made.

session is the `sqlalchemy.orm.session.Session` object in which changes to the database will be made.

flask_sqlalchemy_db is the `flask.ext.sqlalchemy.SQLAlchemy` object with which *app* has been registered and which contains the database models for which API endpoints will be created.

If *flask_sqlalchemy_db* is not `None`, *session* will be ignored.

This is for use in the situation in which this class must be instantiated before the `Flask` application has been created.

To use this method with pure SQLAlchemy, for example:

```
from flask import Flask
from flask.ext.restless import APIManager
from sqlalchemy import create_engine
from sqlalchemy.orm.session import sessionmaker

apimanager = APIManager()

# later...

engine = create_engine('sqlite:////tmp/mydb.sqlite')
Session = sessionmaker(bind=engine)
mysession = Session()
app = Flask(__name__)
apimanager.init_app(app, session=mysession)
```

and with models defined with Flask-SQLAlchemy:

```
from flask import Flask
from flask.ext.restless import APIManager
from flask.ext.sqlalchemy import SQLAlchemy

apimanager = APIManager()

# later...
```

```
app = Flask(__name__)
db = SQLALchemy(app)
apimanager.init_app(app, flask_sqlalchemy_db=db)
```

postprocessors and *preprocessors* must be dictionaries as described in the section *Request preprocessors and postprocessors*. These preprocessors and postprocessors will be applied to all requests to and responses from APIs created using this APIManager object. The preprocessors and postprocessors given in these keyword arguments will be prepended to the list of processors given for each individual model when using the `create_api_blueprint()` method (more specifically, the functions listed here will be executed before any functions specified in the `create_api_blueprint()` method). For more information on using preprocessors and postprocessors, see *Request preprocessors and postprocessors*.

New in version 0.13.0: Added the *preprocessors* and *postprocessors* keyword arguments.

create_api(**args*, ***kw*)

Creates and registers a ReSTful API blueprint on the `flask.Flask` application specified in the constructor of this class.

The positional and keyword arguments are passed directly to the `create_api_blueprint()` method, so see the documentation there.

This is a convenience method for the following code:

```
blueprint = apimanager.create_api_blueprint(*args, **kw)
app.register_blueprint(blueprint)
```

Changed in version 0.6: The blueprint creation has been moved to `create_api_blueprint()`; the registration remains here.

create_api_blueprint(*model*, *app=None*, *methods=frozenset(['GET'])*, *url_prefix='/api'*, *collection_name=None*, *allow_patch_many=False*, *allow_delete_many=False*, *allow_functions=False*, *exclude_columns=None*, *include_columns=None*, *include_methods=None*, *validation_exceptions=None*, *results_per_page=10*, *max_results_per_page=100*, *post_form_preprocessor=None*, *preprocessors=None*, *postprocessors=None*, *primary_key=None*)

Creates and returns a ReSTful API interface as a blueprint, but does not register it on any `flask.Flask` application.

The endpoints for the API for `model` will be available at `<url_prefix>/<collection_name>`. If *collection_name* is `None`, the lowercase name of the provided model class will be used instead, as accessed by `model.__tablename__`. (If any black magic was performed on `model.__tablename__`, this will be reflected in the endpoint URL.) For more information, see *Collection name*.

This function must be called at most once for each model for which you wish to create a ReSTful API. Its behavior (for now) is undefined if called more than once.

This function returns the `flask.Blueprint` object which handles the endpoints for the model. The returned `Blueprint` has already been registered with the `Flask` application object specified in the constructor of this class, so you do *not* need to register it yourself.

model is the SQLAlchemy model class for which a ReSTful interface will be created. Note this must be a class, not an instance of a class.

app is the `Flask` object on which we expect the blueprint created in this method to be eventually registered. If not specified, the Flask application specified in the constructor of this class is used.

methods specify the HTTP methods which will be made available on the ReSTful API for the specified model, subject to the following caveats:

- If GET is in this list, the API will allow getting a single instance of the model, getting all instances of the model, and searching the model using search parameters.
- If PATCH is in this list, the API will allow updating a single instance of the model, updating all instances of the model, and updating a subset of all instances of the model specified using search parameters.
- If DELETE is in this list, the API will allow deletion of a single instance of the model per request.
- If POST is in this list, the API will allow posting a new instance of the model per request.

The default set of methods provides a read-only interface (that is, only GET requests are allowed).

collection_name is the name of the collection specified by the given model class to be used in the URL for the ReSTful API created. If this is not specified, the lowercase name of the model will be used.

url_prefix the URL prefix at which this API will be accessible.

If *allow_patch_many* is `True`, then requests to `/api/` will attempt to patch the attributes on each of the instances of the model which match the specified search query. This is `False` by default. For information on the search query parameter `q`, see *Making search queries*.

If *allow_delete_many* is `True`, then requests to `/api/` will attempt to delete each instance of the model that matches the specified search query. This is `False` by default. For information on the search query parameter `q`, see *Making search queries*.

validation_exceptions is the tuple of possible exceptions raised by validation of your database models. If this is specified, validation errors will be cap-

tured and forwarded to the client in JSON format. For more information on how to use validation, see *Capturing validation errors*.

If *allow_functions* is `True`, then requests to `/api/eval/` will return the result of evaluating SQL functions specified in the body of the request. For information on the request format, see *Function evaluation*. This if `False` by default. Warning: you must not create an API for a model whose name is `'eval'` if you set this argument to `True`.

If either *include_columns* or *exclude_columns* is not `None`, exactly one of them must be specified. If both are not `None`, then this function will raise a `IllegalArgumentError`. *exclude_columns* must be an iterable of strings specifying the columns of *model* which will *not* be present in the JSON representation of the model provided in response to GET requests. Similarly, *include_columns* specifies the *only* columns which will be present in the returned dictionary. In other words, *exclude_columns* is a blacklist and *include_columns* is a whitelist; you can only use one of them per API endpoint. If either *include_columns* or *exclude_columns* contains a string which does not name a column in *model*, it will be ignored.

If *include_columns* is an iterable of length zero (like the empty tuple or the empty list), then the returned dictionary will be empty. If *include_columns* is `None`, then the returned dictionary will include all columns not excluded by *exclude_columns*.

If *include_methods* is an iterable of strings, the methods with names corresponding to those in this list will be called and their output included in the response.

See *Specifying which columns are provided in responses* for information on specifying included or excluded columns on fields of related models.

results_per_page is a positive integer which represents the default number of results which are returned per page. Requests made by clients may override this default by specifying `results_per_page` as a query argument. *max_results_per_page* is a positive integer which represents the maximum number of results which are returned per page. This is a "hard" upper bound in the sense that even if a client specifies that greater than *max_results_per_page* should be returned, only *max_results_per_page* results will be returned. For more information, see *Server-side pagination*.

Deprecated since version 0.9.2: The *post_form_preprocessor* keyword argument is deprecated in version 0.9.2. It will be removed in version 1.0. Replace code that looks like this::

    manager.create_api(Person, post_form_preprocessor=foo)

with code that looks like this:

```
manager.create_api(Person, preprocessors=dict(POST=[foo]))
```

See *Request preprocessors and postprocessors* for more information and examples.

post_form_preprocessor is a callback function which takes POST input parameters loaded from JSON and enhances them with other key/value pairs. The example use of this is when your `model` requires to store user identity and for security reasons the identity is not read from the post parameters (where malicious user can tamper with them) but from the session.

preprocessors is a dictionary mapping strings to lists of functions. Each key is the name of an HTTP method (for example, `'GET'` or `'POST'`). Each value is a list of functions, each of which will be called before any other code is executed when this API receives the corresponding HTTP request. The functions will be called in the order given here. The *postprocessors* keyword argument is essentially the same, except the given functions are called after all other code. For more information on preprocessors and postprocessors, see *Request preprocessors and postprocessors*.

primary_key is a string specifying the name of the column of *model* to use as the primary key for the purposes of creating URLs. If the *model* has exactly one primary key, there is no need to provide a value for this. If *model* has two or more primary keys, you must specify which one to use.

New in version 0.16.0: Added the *app* and *allow_delete_many* keyword arguments.

New in version 0.13.0: Added the *primary_key* keyword argument.

New in version 0.10.2: Added the *include_methods* keyword argument.

Changed in version 0.10.0: Removed *authentication_required_for* and *authentication_function* keyword arguments.

Use the *preprocesors* and *postprocessors* keyword arguments instead. For more information, see *Requiring authentication for some methods*.

New in version 0.9.2: Added the *preprocessors* and *postprocessors* keyword arguments.

New in version 0.9.0: Added the *max_results_per_page* keyword argument.

New in version 0.7: Added the *exclude_columns* keyword argument.

New in version 0.6: This functionality was formerly in `create_api()`, but the blueprint creation and registration have now been separated.

New in version 0.6: Added the *results_per_page* keyword argument.

New in version 0.5: Added the *include_columns* and *validation_exceptions* keyword argument.

New in version 0.4: Added the *allow_functions*, *allow_patch_many*, *authentication_required_for*, *authentication_function*, and *collection_name* keyword arguments.

New in version 0.4: Force the model name in the URL to lowercase.

class flask.ext.restless.**ProcessingException**(*description=''*, *code=400*, **args*, ***kwargs*)

Raised when a preprocessor or postprocessor encounters a problem.

This exception should be raised by functions supplied in the `preprocessors` and `postprocessors` keyword arguments to `APIManager.create_api`. When this exception is raised, all preprocessing or postprocessing halts, so any processors appearing later in the list will not be invoked.

code is the HTTP status code of the response supplied to the client in the case that this exception is raised. *description* is an error message describing the cause of this exception. This message will appear in the JSON object in the body of the response to the client.

Part III

ADDITIONAL INFORMATION

CHAPTER 8

Similar projects

If Flask-Restless doesn't work for you, here are some similar Python packages that intend to simplify the creation of ReSTful APIs (in various combinations of Web frameworks and database backends):

- Eve
- Flask-Peewee
- Flask-RESTful
- simpleapi
- Tastypie

CHAPTER 9

Copyright and license

Flask-Restless is copyright 2011 Lincoln de Sousa and copyright 2012, 2013, 2014, 2015 Jeffrey Finkelstein, and is dual-licensed under the following two copyright licenses:

- the GNU Affero General Public License, either version 3 or (at your option) any later version
- the 3-clause BSD License

For more information, see the files `LICENSE.AGPL` and `LICENSE.BSD` in top-level directory of the source distribution.

The artwork for Flask-Restless is copyright 2012 Jeffrey Finkelstein. The couch logo is licensed under the Creative Commons Attribute-ShareAlike 3.0 license. The original image is a scan of a (now public domain) illustration by Arthur Hopkins in a serial edition of "The Return of the Native" by Thomas Hardy published in October 1878. The couch logo with the "Flask-Restless" text is licensed under the Flask Artwork License.

The documentation is licensed under the Creative Commons Attribute-ShareAlike 3.0 license.

CHAPTER **10**

Changelog

Here you can see the full list of changes between each Flask-Restless release. Numbers following a pound sign (#) refer to GitHub issues.

Note: As of version 0.13.0, Flask-Restless supports Python 2.6, 2.7, and 3. Before that, it supported Python 2.5, 2.6, and 2.7.

Note: As of version 0.6, Flask-Restless supports both pure SQLAlchemy and Flask-SQLAlchemy models. Before that, it supported only Elixir models.

10.1 Version 0.16.0

Released on February 3, 2015.

- #237: allows bulk delete of model instances via the `allow_delete_many` keyword argument.
- #313, #389: `APIManager.init_app()` now can be correctly used to initialize multiple Flask applications.
- #327, #391: allows ordering searches by fields on related instances.
- #353: allows search queries to specify `group_by` directives.
- #365: allows preprocessors to specify return values on GET requests.
- #385: makes the *include_methods* keywords argument respect model properties.

10.2 Version 0.15.1

Released on January 2, 2015.

- #367: catch `IntegrityError`, `DataError`, and `ProgrammingError` exceptions in all view methods.
- #374: import `sqlalchemy.Column` from `sqlalchemy` directly, instead of `sqlalchemy.sql.schema`

10.3 Version 0.15.0

Released on October 30, 2014.

- #320: detect settable hybrid properties instead of raising an exception.
- #350: allows exclude/include columns to be specified as SQLAlchemy column objects in addition to strings.
- #356: rollback the SQLAlchemy session on a failed PATCH request.
- #368: adds missing documentation on using custom queries (see *Custom queries*)

10.4 Version 0.14.2

Released on September 2, 2014.

- #351, #355: fixes bug in getting related models from a model with hybrid properties.

10.5 Version 0.14.1

Released on August 26, 2014.

- #210: lists some related projects in the documentation.
- #347: adds automated build testing for PyPy 3.
- #354: renames `is_deleted` to `was_deleted` when providing keyword arguments to postprocessor for DELETE method in order to match documentation.

10.6 Version 0.14.0

Released on August 12, 2014.

- Fixes bug where primary key specified by user was not being checked in some POST requests and some search queries.
- #223: documents CORS example.
- #280: don't expose raw SQL in responses on database errors.

- #299: show error message if search query tests for `NULL` using comparison operators.
- #315: check for query object being `None`.
- #324: DELETE should only return 204 No Content if something is actuall deleted.
- #325: support `null` inside `has` search operators.
- #328: enable automatic testing for Python 3.4.
- #333: enforce limit in `helpers.count()`.
- #338: catch validation exceptions when attempting to update relations.
- #339: use user-specified primary key on PATCH requests.
- #344: correctly encodes Unicode fields in responses.

10.7 Version 0.13.1

Released on April 21, 2014.

- #304: fixes mimerender bug due to how Python 3.4 handles decorators.

10.8 Version 0.13.0

Released on April 6, 2014.

- Allows universal preprocessors or postprocessors; see *Universal preprocessors and postprocessors*.
- Allows specifying which primary key to use when creating endpoint URLs.
- Requires SQLAlchemy version 0.8 or greater.
- #17: use Flask's `flask.Request.json` to parse incoming JSON requests.
- #29: replace custom `jsonify_status_code` function with built-in support for `return jsonify(), status_code` style return statements (new in Flask 0.9).
- #51: Use mimerender to render dictionaries to JSON format.
- #247: adds support for making POST requests to dictionary-like association proxies.
- #249: returns 404 Not Found if a search reveals no matching results.
- #254: returns 404 Not Found if no related field exists for a request with a related field in the URL.
- #256: makes search parameters available to postprocessors for GET and PATCH requests that access multiple resources.
- #263: Adds Python 3.3 support; drops Python 2.5 support.

- #267: Adds compatibility for legacy Microsoft Internet Explorer versions 8 and 9.
- #270: allows the `query` attribute on models to be a callable.
- #282: order responses by primary key if no order is specified.
- #284: catch `DataError` and `ProgrammingError` exceptions when bad data are sent to the server.
- #286: speed up paginated responses by using optimized count() function.
- #293: allows `sqlalchemy.Time` fields in JSON responses.

10.9 Version 0.12.1

Released on December 1, 2013.

- #222: on POST and PATCH requests, recurse into nested relations to get or create instances of related models.
- #246: adds pysqlite to test requirements.
- #260: return a single object when making a GET request to a relation sub-URL.
- #264: all methods now execute postprocessors after setting headers.
- #265: convert strings to dates in related models when making POST requests.

10.10 Version 0.12.0

Released on August 8, 2013.

- #188: provides metadata as well as normal data in JSONP responses.
- #193: allows DELETE requests to related instances.
- #215: removes Python 2.5 tests from Travis configuration.
- #216: don't resolve Query objects until pagination function.
- #217: adds missing indices in format string.
- #220: fix bug when checking attributes on a hybrid property.
- #227: allows client to request that the server use the current date and/or time when setting the value of a field.
- #228 (as well as #212, #218, #231): fixes issue due to a module removed from Flask version 0.10.

10.11 Version 0.11.0

Released on May 18, 2013.

- Requests that require a body but don't have `Content-Type: application/json` will cause a 415 Unsupported Media Type response.
- Responses now have `Content-Type: application/json`.
- #180: allow more expressive `has` and `any` searches.
- #195: convert UUID objects to strings when converting an instance of a model to a dictionary.
- #202: allow setting hybrid properties with expressions and setters.
- #203: adds the `include_methods` keyword argument to `APIManager.create_api()`, which allows JSON responses to include the result of calling arbitrary methods of instances of models.
- #204, 205: allow parameters in `Content-Type` header.

10.12 Version 0.10.1

Released on May 8, 2013.

- #115: change `assertEqual()` methods to `assert` statements in tests.
- #184, #186: Switch to nose for testing.
- #197: documents technique for adding filters in processors when there are none initially.

10.13 Version 0.10.0

Released on April 30, 2013.

- #2: adds basic GET access to one level of relationship depth for models.
- #113: interpret empty strings for date fields as `None` objects.
- #115: use Python's built-in assert statements for testing
- #128: allow disjunctions when filtering search queries.
- #130: documentation and examples now more clearly show search examples.
- #135: added support for hybrid properties.
- #139: remove custom code for authentication in favor of user-defined pre- and postprocessors (this supercedes the fix from #154).

- #141: relax requirement for version of python-dateutil to be not equal to 2.0 if using Python version 2.6 or 2.7.
- #146: preprocessors now really execute before other code.
- #148: adds support for SQLAlchemy association proxies.
- #154 (*this fix is irrelevant due to #139*): authentication function now may raise an exception instead of just returning a Boolean.
- #157: POST requests now receive a response containing all fields of the created instance.
- #162: allow pre- and postprocessors to indicate that no change has occurred.
- #164, #172, and #173: PATCH requests update fields on related instances.
- #165: fixed bug in automatic exposing of URLs for related instances.
- #170: respond with correct HTTP status codes when a query for a single instance results in none or multiple instances.
- #174: allow dynamically loaded relationships for automatically exposed URLs of related instances.
- #176: get model attribute instead of column name when getting name of primary key.
- #182: allow POST requests that set hybrid properties.
- #152: adds some basic server-side logging for exceptions raised by views.

10.14 Version 0.9.3

Released on February 4, 2013.

- Fixes incompatibility with Python 2.5 try/except syntax.
- #116: handle requests which raise `IntegrityError`.

10.15 Version 0.9.2

Released on February 4, 2013.

- #82, #134, #136: added request pre- and postprocessors.
- #120: adds support for JSON-P callbacks in GET requests.

10.16 Version 0.9.1

Released on January 17, 2013.

- #126: fix documentation build failure due to bug in a dependency.
- #127: added “ilike” query operator.

10.17 Version 0.9.0

Released on January 16, 2013.

- Removed ability to provide a `Session` class when initializing `APIManager`; provide an instance of the class instead.
- Changes some dynamically loaded relationships used for testing and in examples to be many-to-one instead of the incorrect one-to-many. Versions of SQLAlchemy after 0.8.0b2 raise an exception when the latter is used.
- #105: added ability to set a list of related model instances on a model.
- #107: server responds with an error code when a PATCH or POST request specifies a field which does not exist on the model.
- #108: dynamically loaded relationships should now be rendered correctly by the `views._to_dict()` function regardless of whether they are a list or a single object.
- #109: use sphinxcontrib-issuetracker to render links to GitHub issues in documentation.
- #110: enable `results_per_page` query parameter for clients, and added `max_results_per_page` keyword argument to `APIManager.create_api()`.
- #114: fix bug where string representations of integers were converted to integers.
- #117: allow adding related instances on PATCH requests for one-to-one relationships.
- #123: PATCH requests to instances which do not exist result in a 404 Not Found response.

10.18 Version 0.8.0

Released on November 19, 2012.

- #94: `views._to_dict()` should return a single object instead of a list when resolving dynamically loaded many-to-one relationships.
- #104: added `num_results` key to paginated JSON responses.

10.19 Version 0.7.0

Released on October 9, 2012.

- Added working `include` and `exclude` functionality to the `views._to_dict()` function.
- Added `exclude_columns` keyword argument to `APIManager.create_api()`.
- #79: attempted to access attribute of `None` in constructor of `APIManager`.
- #83: allow POST requests with one-to-one related instances.
- #86: allow specifying include and exclude for related models.
- #91: correctly handle POST requests to nullable `DateTime` columns.
- #93: Added a `total_pages` mapping to the JSON response.
- #98: GET requests to the function evaluation endpoint should not have a data payload.
- #101: `exclude` in `views._to_dict()` function now correctly excludes requested fields from the returned dictionary.

10.20 Version 0.6

Released on June 20, 2012.

- Added support for accessing model instances via arbitrary primary keys, instead of requiring an integer column named `id`.
- Added example which uses curl as a client.
- Added support for pagination of responses.
- Fixed issue due to symbolic link from `README` to `README.md` when running `pip bundle foobar Flask-Restless`.
- Separated API blueprint creation from registration, using `APIManager.create_api()` and `APIManager.create_api_blueprint()`.
- Added support for pure SQLAlchemy in addition to Flask-SQLAlchemy.
- #74: Added `post_form_preprocessor` keyword argument to `APIManager.create_api()`.
- #77: validation errors are now correctly handled on PATCH requests.

10.21 Version 0.5

Released on April 10, 2012.

- Dual-licensed under GNU AGPLv3+ and 3-clause BSD license.
- Added capturing of exceptions raised during field validation.

- Added `examples/separate_endpoints.py`, showing how to create separate API endpoints for a single model.
- Added `include_columns` keyword argument to `create_api()` method to allow users to specify which columns of the model are exposed in the API.
- Replaced Elixir with Flask-SQLAlchemy. Flask-Restless now only supports Flask-SQLAlchemy.

10.22 Version 0.4

Released on March 29, 2012.

- Added Python 2.5 and Python 2.6 support.
- Allow users to specify which HTTP methods for a particular API will require authentication and how that authentication will take place.
- Created base classes for test cases.
- Moved the `evaluate_functions` function out of the `flask_restless.search` module and corrected documentation about how function evaluation works.
- Added *allow_functions* keyword argument to `create_api()`.
- Fixed bug where we weren't allowing PUT requests in `create_api()`.
- Added `collection_name` keyword argument to `create_api()` to allow user provided names in URLs.
- Added `allow_patch_many` keyword argument to `create_api()` to allow enabling or disabling the PATCH many functionality.
- Disable the PATCH many functionality by default.

10.23 Version 0.3

Released on March 4, 2012.

- Initial release in Flask extension format.

Index