# Flask-Restless

# Flask-Restless Documentation

*Release 1.0.0b2.dev*

Mar 25, 2017

# Contents

**Flask-Restless** provides simple generation of ReSTful APIs for database models defined using SQLAlchemy (or Flask-SQLAlchemy). The generated APIs satisfy the requirements of the JSON API specification.

This is the documentation for version 1.0.0b2. See also the the most recent stable version documentation and the development version documentation

> **Warning:** This is a "beta" version, so there may be more bugs than usual.

CHAPTER 1

# User's guide

How to use Flask-Restless in your own projects. Much of the documentation in this chapter assumes some familiarity with the terminology and interfaces of the JSON API specification.

## Downloading and installing Flask-Restless

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

```
pip install Flask-Restless
```

Flask-Restless supports all Python versions that Flask supports, which currently include versions 2.6, 2.7, 3.3, 3.4, and 3.5.

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
- python-dateutil version strictly greater than 2.2

Flask-SQLAlchemy is supported but not required.

# Quickstart

For the restless:

```
import flask
import flask_sqlalchemy
import flask_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_sqlalchemy.SQLAlchemy(app)


# Create your Flask-SQLALchemy models as usual but with the following
# restriction: they must have an __init__ method that accepts keyword
# arguments for all columns (the constructor in
# flask_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)
    birth_date = db.Column(db.Date)


class Article(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.Unicode)
    published_at = db.Column(db.DateTime)
    author_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    author = db.relationship(Person, backref=db.backref('articles',
                                                        lazy='dynamic'))


# Create the database tables.
db.create_all()

# Create the Flask-Restless API manager.
manager = flask_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(Article, 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. Further examples can be found in the `examples/` directory in the source distribution or on the web

## Creating API endpoints

To use this extension, you must have defined your database models using either SQLAlchemy or Flask-SQLALchemy. The basic setup in either case is nearly the same.

If you have defined your models with Flask-SQLAlchemy, first, create your `Flask` object, `SQLAlchemy` object, and model classes as usual but with one additional restriction: each model must have a primary key column of type either `Integer` or `Unicode`.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

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


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


class Article(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    author_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    author = db.relationship(Person, backref=db.backref('articles'))

db.create_all()
```

If you are using pure SQLAlchemy:

```
from flask import Flask
from sqlalchemy import Column, 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 Person(Base):
    id = Column(Integer, primary_key=True)


class Article(Base):
    id = Column(Integer, primary_key=True)
    author_id = Column(Integer, ForeignKey('person.id'))
    author = relationship(Person, backref=backref('articles'))

Base.metadata.create_all()
```

Second, instantiate an `APIManager` object with the `Flask` and `SQLAlchemy` objects:

```
from flask_restless import APIManager

manager = APIManager(app, flask_sqlalchemy_db=db)
```

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

```
manager = APIManager(app, session=mysession)
```

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

```
person_blueprint = manager.create_api(Person, methods=['GET', 'POST'])
article_blueprint = manager.create_api(Article)
```

You can specify which HTTP methods are available for each API endpoint. In this example, the client can fetch and create people, but only fetch articles (the default if no methods are specified). There are many options for customizing the endpoints created at this step; 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:

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

If you wish to create the blueprint for the API without registering it (for example, if you wish to register it manually later in your code), use the `create_api_blueprint()` method instead. You *must* provide an additional positional argument, *name*, to this method:

```
blueprint = manager.create_api_blueprint('person', Person, methods=methods)
# later...
someapp.register_blueprint(blueprint)
```

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

```
>>> import json
>>> # The python-requests library is installable from PyPI.
>>> import requests
>>> # Let's create a new person resource with the following fields.
>>> newperson = {'type': 'person', 'name': u'Lincoln', 'age': 23}
>>> # Our requests must have the appropriate JSON API headers.
>>> headers = {'Content-Type': 'application/vnd.api+json',
...            'Accept': 'application/vnd.api+json'}
>>> # Assume we have a Flask application running on localhost.
>>> r = requests.post('http://localhost/api/person',
...                   data=json.dumps(newperson), headers=headers)
>>> r.status_code
201
>>> document = json.loads(r.data)
>>> dumps(document, indent=2)
{
  "data": {
    "id": "1",
    "type": "person",
    "relationships": {
      "articles": {
        "data": [],
        "links": {
          "related": "http://localhost/api/person/1/articles",
          "self": "http://localhost/api/person/1/relationships/articles"
        }
      },
    },
    "links": {
      "self": "http://localhost/api/person/1"
    }
  }
  "meta": {},
  "jsonapi": {
    "version": "1.0"
  }
}
>>> newid = document['data']['id']
>>> r = requests.get('/api/person/{0}'.format(newid), headers=headers)
>>> r.status_code
200
>>> document = loads(r.data)
>>> dumps(document, indent=2)
{
  "data": {
```

```
    "id": "1",
    "type": "person",
    "relationships": {
      "articles": {
        "data": [],
        "links": {
          "related": "http://localhost/api/person/1/articles",
          "self": "http://localhost/api/person/1/relationships/articles"
        }
      },
    },
    "links": {
      "self": "http://localhost/api/person/1"
    }
  }
  "meta": {},
  "jsonapi": {
    "version": "1.0"
  }
}
```

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`.

## Deferred API registration

If you only wish to create APIs on a single Flask application and have access to the Flask application before you create the APIs, you can provide a Flask application as an argument to the constructor of the `APIManager` class, as described above. However, if you wish to create APIs on multiple Flask applications or if you do not have access to the Flask application at the time you create the APIs, you can use the `APIManager.init_app()` method.

If a `APIManager` object is created without a Flask application,

```
manager = APIManager(session=session)
```

then you can create your APIs without registering them on a particular Flask application:

```
manager.create_api(Person)
manager.create_api(Article)
```

Later, you can call the `init_app()` method with any `Flask` objects on which you would like the APIs to be available:

```
app1 = Flask('app1')
app2 = Flask('app2')
```

```
manager.init_app(app1)
manager.init_app(app2)
```

The manager creates and stores a blueprint each time `create_api()` is invoked, and registers those blueprints each time `init_app()` is invoked. (The name of each blueprint will be a `uuid.UUID`.)

Changed in version 1.0.0: The behavior of the `init_app()` method was strange and incorrect before version 1.0.0. It is best not to use earlier versions.

# Requests and responses

Requests and responses are all in the JSON API format, so each request must include an Accept header whose value is `application/vnd.api+json` and any request that contains content must include a Content-Type header whose value is `application/vnd.api+json`. If they do not, the client will receive an error response.

This section of the documentation assumes some familiarity with the JSON API specification.

## Fetching resources and relationships

This section described fetching resources and relationships via GET requests.

### Function evaluation

*This section describes behavior that is not part of the JSON API specification.*

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

If the client specifies the `functions` query parameter, it must be a percent-encoded list of *function objects*, as described below.

A *function object* is a JSON object. A function object must be of the form

```
{"name": <function_name>, "field": <field_name>}
```

where `<function_name>` is the name of a SQL function as provided by SQLAlchemy's `func` object.

For example, to get the average age of all people in the database,

```
GET /api/eval/person?functions=[{"name":"avg","field":"age"}] HTTP/1.1
Host: example.com
Accept: application/json
```

The response will be a JSON object with a single element, `data`, containing a list of the results of all the function evaluations requested by the client, in the same order as in the `functions` query parameter. For example, to get the sum and the average ages of all people in the database, the request

```
GET /api/eval/person?functions=[{"name":"avg","field":"age"},{"name":"sum","field
↪":"age"}] HTTP/1.1
Host: example.com
Accept: application/json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/json

[15.0, 60.0]
```

---

**Example**

To get the total number of resources in the collection (that is, the number of instances of the model), you can use the function object

```
{"name": "count", "field": "id"}
```

Then the request

```
GET /api/eval/person?functions=[{"name":"count","field":"id"}] HTTP/1.1
Host: example.com
Accept: application/json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "data": [42]
}
```

---

The function evaluation endpoint also respects filtering query parameters. Specifically, filters are applied to the model *before* the function evaluation is performed, so you can apply a function to a subset of resources. See *Filtering* for more information.

Changed in version 1.0.0b2: Adds ability to use filters in function evaluation.

### Inclusion of related resources

*For more information on client-side included resources, see* Inclusion of Related Resources *in the JSON API specification.*

By default, no related resources will be included in a compound document on requests that would return data. For the client to request that the response includes related resources in a compound document, use the `include` query parameter. For example, to fetch a single resource and include all resources related to it, the request

```
GET /api/person/1?include=articles HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "id": "1",
    "links": {
      "self": "http://example.com/api/person/1"
    },
    "relationships": {
      "articles": {
        "data": [
          {
            "id": "1",
            "type": "article"
          }
        ],
        "links": {
          "related": "http://example.com/api/person/1/articles",
          "self": "http://example.com/api/person/1/relationships/articles"
        }
      }
    },
    "type": "person"
  }
  "included": [
    {
      "id": "1",
      "links": {
        "self": "http://example.com/api/article/1"
      },
      "relationships": {
        "author": {
          "data": {
            "id": "1",
            "type": "person"
          },
          "links": {
            "related": "http://example.com/api/article/1/author",
            "self": "http://example.com/api/article/1/relationships/author"
```

```
          }
        }
      },
      "type": "article"
    }
  ]
}
```

To specify a default set of related resources to include when the client does not specify any *include* query parameter, use the `includes` keyword argument to the `APIManager.create_api()` method.

### Specifying which fields appear in responses

*For more information on client-side sparse fieldsets, see* Sparse Fieldsets *in the JSON API specification.*

> **Warning:** The server-side configuration for specifying which fields appear in resource objects as described in this section is simplistic; a better way to specify which fields are included in your responses is to use a Python object serialization library and specify custom serialization and deserialization functions as described in *Custom serialization*.

By default, all fields of your model will be exposed by the API. A client can request that only certain fields appear in the resource object in a response to a GET request by using the `only` query parameter. On the server side, you can specify which fields appear in the resource object representation of an instance of the model by setting the `only`, `exclude` and `additional_attributes` keyword arguments to the `APIManager.create_api()` method.

If `only` is an iterable of column names or actual column attributes, only those fields will appear in the resource object that appears in responses to fetch instances of this model. If instead `exclude` is specified, all fields except those specified in that iterable will appear in responses. If `additional_attributes` is an iterable of column names, the values of these attributes will also appear in the response; this is useful if you wish to see the value of some attribute that is not a column or relationship.

> **Attention:** The `type` and `id` elements will always appear in the resource object, regardless of whether the server or the client tries to exclude them.

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)
```

```
    birthday = db.Column(db.Date)
    articles = db.relationship('Article')

    # This class attribute is not a column.
    foo = 'bar'

class Article(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    author_id = db.Column(db.Integer, db.ForeignKey('person.id'))
```

and you want your resource objects to include only the values of the `name` and `birthday` columns, create your API with the following arguments:

```
apimanager.create_api(Person, only=['name', 'birthday'])
```

Now a request like

```
GET /api/person/1 HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "id": "1",
    "links": {
      "self": "http://example.com/api/person/1"
    },
    "attributes": {
      "birthday": "1969-07-20",
      "name": "foo"
    },
    "type": "person"
  }
}
```

If you want your resource objects to *exclude* the `birthday` and `name` columns:

```
apimanager.create_api(Person, exclude=['name', 'birthday'])
```

Now the same request yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
```

```
    "id": "1",
    "links": {
      "self": "http://example.com/api/person/1"
    }
    "relationships": {
      "articles": {
        "data": [],
        "links": {
          "related": "http://example.com/api/person/1/articles",
          "self": "http://example.com/api/person/1/links/articles"
        }
      },
    },
    "type": "person"
  }
}
```

If you want your resource objects to include the value for the class attribute `foo`:

```
apimanager.create_api(Person, additional_attributes=['foo'])
```

Now the same request yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "attributes": {
      "birthday": "1969-07-20",
      "foo": "bar",
      "name": "foo"
    },
    "id": "1",
    "links": {
      "self": "http://example.com/api/person/1"
    }
    "relationships": {
      "articles": {
        "data": [],
        "links": {
          "related": "http://example.com/api/person/1/articles",
          "self": "http://example.com/api/person/1/links/articles"
        }
      }
    },
    "type": "person"
  }
}
```

### Sorting

Clients can sort according to the sorting protocol described in the Sorting section of the JSON API specification. Sorting by a nullable attribute will cause resources with null attributes to appear first. The client can request case-insensitive sorting by setting the query parameter `ignorecase=1`.

Clients can also request grouping by using the `group` query parameter. For example, if your database has two people with name `'foo'` and two people with name `'bar'`, a request like

```
GET /api/person?group=name HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "attributes": {
        "name": "foo",
      },
      "id": "1",
      "links": {
        "self": "http://example.com/api/person/1"
      },
      "relationships": {
        "articles": {
          "data": [],
          "links": {
            "related": "http://example.com/api/person/1/articles",
            "self": "http://example.com/api/person/1/relationships/articles"
          }
        }
      },
      "type": "person"
    },
    {
      "attributes": {
        "name": "bar",
      },
      "id": "3",
      "links": {
        "self": "http://example.com/api/person/3"
      },
      "relationships": {
        "articles": {
```

```
          "data": [],
          "links": {
            "related": "http://example.com/api/person/3/articles",
            "self": "http://example.com/api/person/3/relationships/articles"
          }
        }
      },
      "type": "person"
    },
  ],
  "links": {
    "first": "http://example.com/api/person?group=name&page[number]=1&
↪page[size]=10",
    "last": "http://example.com/api/person?group=name&page[number]=1&page[size]=10
↪",
    "next": null,
    "prev": null,
    "self": "http://example.com/api/person?group=name"
  },
  "meta": {
    "total": 2
  }
}
```

Pagination

Pagination works as described in the JSON API specification, via the `page[number]` and `page[size]` query parameters. Pagination respects sorting, grouping, and filtering. The first page is page one. If no page number is specified by the client, the first page will be returned. By default, pagination is enabled and the page size is ten. If the page size specified by the client is greater than the maximum page size as configured on the server, then the query parameter will be ignored.

To set the default page size for collections of resources, use the `page_size` keyword argument to the `APIManager.create_api()` method. To set the maximum page size that the client can request, use the `max_page_size` argument. Even if `page_size` is greater than `max_page_size`, at most `max_page_size` resources will be returned in a page. If `max_page_size` is set to `0`, the client will be able to specify arbitrarily large page sizes. If, further, `page_size` is set to `0`, pagination will be disabled by default, and any GET request that does not specify a page size in its query parameters will get a response with all matching results.

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

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

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

Then a GET request to `/api/person?page[number]=2` would yield the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "id": "3",
      "type": "person",
      "attributes": {
        "name": "John"
      }
    }
    {
      "id": "4",
      "type": "person",
      "attributes": {
        "name": "Paul"
      }
    }
  ],
  "links": {
    "first": "http://example.com/api/person?page[number]=1&page[size]=2",
    "last": "http://example.com/api/person?page[number]=3&page[size]=2",
    "next": "http://example.com/api/person?page[number]=3&page[size]=2",
    "prev": "http://example.com/api/person?page[number]=1&page[size]=2",
    "self": "http://example.com/api/person"
  },
  "meta": {
    "total": 6
  }
}
```

Filtering

Requests that would normally return a collection of resources can be filtered so that only a subset of the resources are returned in a response. If the client specifies the `filter[objects]` query parameter, it must be a URL encoded JSON list of *filter objects*, as described below.

## Quick client examples for filtering

The following are some quick examples of making filtered GET requests from different types of clients. More complete documentation is in subsequent sections. In these

examples, each client will filter by 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 = {'Accept': 'application/vnd.api+json'}

filters = [dict(name='name', op='like', val='%y%')]
params = {'filter[objects]': json.dumps(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({
  data: {"filter[objects]": JSON.stringify(filters)},
  headers: {
    "Accept": JSONAPI_MIMETYPE
  },
  success: function(data) { console.log(data.objects); },
  url: 'http://127.0.0.1:5000/api/person'
});
```

Using curl:

```
curl \
  -G \
  -H "Accept: application/vnd.api+json" \
  -d "filter[objects]=[{\"name\":\"name\",\"op\":\"like\",\"val\":\"%y%\"}]" \
  http://127.0.0.1:5000/api/person
```

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

## Filter objects

A *filter object* is a JSON object. Filter objects are defined recursively as follows. A filter object may be of the form

```
{"name": <field_name>, "op": <unary_operator>}
```

where `<field_name>` is the name of a field on the model whose instances are being fetched and `<unary_operator>` is the name of one of the unary operators supported by Flask-Restless. For example,

```
{"name": "birthday", "op": "is_null"}
```

A filter object may be of the form

```
{"name": <field_name>, "op": <binary_operator>, "val": <argument>}
```

where `<binary_operator>` is the name of one of the binary operators supported by Flask-Restless and `<argument>` is the second argument to that binary operator. For example,

```
{"name": "age", "op": "gt", "val": 23}
```

A filter object may be of the form

```
{"name": <field_name>, "op": <binary_operator>, "field": <field_name>}
```

The `field` element indicates that the second argument to the binary operator should be the value of that field. For example, to filter by resources that have a greater width than height,

```
{"name": "width", "op": "gt", "field": "height"}
```

A filter object may be of the form

```
{"name": <relation_name>, "op": <relation_operator>, "val": <filter_object>}
```

where `<relation_name>` is the name of a relationship on the model whose resources are being fetched, `<relation_operator>` is either `"has"`, for a to-one relationship, or `"any"`, for a to-many relationship, and `<filter_object>` is another filter object. For example, to filter person resources by only those people that have authored an article dated before January 1, 2010,

```
{
  "name": "articles",
  "op": "any",
  "val": {
    "name": "date",
    "op": "lt",
    "val": "2010-01-01"
  }
}
```

For another example, to filter article resources by only those articles that have an author of age at most fifty,

```
{
  "name": "author",
  "op": "has",
  "val": {
    "name": "age",
```

```
    "op": "lte",
    "val": 50
  }
}
```

A filter object may be a conjunction ("and"), disjunction ("or"), or negation ("not") of other filter objects:

```
{"or": [<filter_object>, <filter_object>, ...]}
```

or

```
{"and": [<filter_object>, <filter_object>, ...]}
```

or

```
{"not": <filter_object>}
```

For example, to filter by resources that have width greater than height, and length of at least ten,

```
{
  "and": [
    {"name": "width", "op": "gt", "field": "height"},
    {"name": "length", "op": "lte", "val": 10}
  ]
}
```

How are filter objects used in practice? To get a response in which only those resources that meet the requirements of the filter objects are returned, clients can make requests like this:

```
GET /api/person?filter[objects]=[{"name":"age","op":"<","val":18}] HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

## Operators

Flask-Restless understands the following operators, which correspond to the appropriate SQLAlchemy column operators.

- `==`, `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`, `ilike`, `not_like`
- `has`
- `any`

Flask-Restless also understands the PostgreSQL network address operators `<<`, `<<=`, `>>`, `>>=`, `<>`, and `&&`.

**Warning:** If you use a percent sign in the argument to the `like` operator (for example, `%somestring%`), make sure it is percent-encoded, otherwise the server may interpret the first few characters of that argument as a percent-encoded character when attempting to decode the URL.

### Custom operators

You can use the `register_operator()` function to extend the set of known operators:

```
from flask_restless import register_operator

# Create a custom "greater than" implementation.
register_operator('my_gt', lambda x, y: x - y > 0)
```

Then the client makes a request with a filter object whose `op` element is the name of this operator:

```
GET /api/person?filter[objects]=[{"name":"age","op":"my_gt","val":18}] HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

You can also override existing operators by setting the name of your operator to be the name of a existing operator; the built-in operators are listed in the *previous section*:

```
register_operator('gt', lambda x, y: x - y > 0)
```

### Simpler filtering

Flask-Restless also supports a simpler form of filtering as described in the JSON API filtering recommendation. For filtering by the foreign key of a to-one relationship, use a request of the form

```
GET /api/comments?filter[post]=1,2&filter[author]=12 HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

Flask-Restless will automatically determine the correct query corresponding to the given to-one relationships.

You can also filter by attribute:

```
GET /api/person?filter[age]=21 HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

---

**Implementation note**

Each of these simple filters is converted to the more complex filter object representation as described in the preceding sections and appended to the list of filter objects computed from the request query parameters.

---

## Requiring singleton collections

If a client wishes a request for a collection to yield a response with a singleton collection, the client can use the `filter[single]` query parameter. The value of this parameter must be either `1` or `0`. If the value of this parameter is `1` and the response would yield a collection of either zero or more than two resources, the server instead responds with 404 Not Found.

For example, a request like

```
GET /api/person?filter[single]=1&filter[objects]=[{"name":"id","op":"eq","val":1}
↪] HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "id": "1",
    "type": "person",
    "links": {
      "self": "http://example.com/api/person/1"
    }
  },
  "links": {
    "self": "http://example.com/api/person?filter[single]=1&filter[objects]=[{\
↪"name\":\"id\",\"op\":\"eq\",\"val\":1}]"
  },
}
```

But a request like

```
GET /api/person?filter[single]=1 HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

would yield an error response if there were more than one `Person` instance in the database.

## Filter object examples

### Attribute greater than a value

On request

```
GET /api/person?filter[objects]=[{"name":"age","op":"gt","val":18}] HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

the response will include only those `Person` instances that have `age` attribute greater than or equal to 18:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "attributes": {
        "age": 19
      },
      "id": "2",
      "links": {
        "self": "http://example.com/api/person/2"
      },
      "type": "person"
    },
    {
      "attributes": {
        "age": 29
      },
      "id": "5",
      "links": {
        "self": "http://example.com/api/person/5"
      },
      "type": "person"
    },
  ],
  "links": {
```

```
    "self": "/api/person?filter[objects]=[{\"name\":\"age\",\"op\":\"gt\",\"val\
↪":18}]"
  },
  "meta": {
    "total": 2
  }
}
```

### Arbitrary Boolean expression of filters

On request

```
GET /api/person?filter[objects]=[{"or":[{"name":"age","op":"lt","val":10},{"name":
↪"age","op":"gt","val":20}]}] HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

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
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "attributes": {
        "age": 9
      },
      "id": "1",
      "links": {
        "self": "http://example.com/api/person/1"
      },
      "type": "person"
    },
    {
      "attributes": {
        "age": 25
      },
      "id": "3",
      "links": {
        "self": "http://example.com/api/person/3"
      },
      "type": "person"
    }
  ],
  "links": {
    "self": "/api/person?filter[objects]=[{\"or\":[{\"name\":\"age\",\"op\":\"lt\
↪",\"val\":10},{\"name\":\"age\",\"op\":\"gt\",\"val\":20}]}]"
```

```
  },
  "meta": {
    "total": 2
  }
}
```

### Comparing two attributes

On request

```
GET /api/box?filter[objects]=[{"name":"width","op":"ge","field":"height"}] HTTP/1.
↪1
Host: example.com
Accept: application/vnd.api+json
```

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

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "attributes": {
        "height": 10,
        "width": 20
      }
      "id": "1",
      "links": {
        "self": "http://example.com/api/box/1"
      },
      "type": "box"
    },
    {
      "attributes": {
        "height": 15,
        "width": 20
      }
      "id": "2",
      "links": {
        "self": "http://example.com/api/box/2"
      },
      "type": "box"
    }
  ],
  "links": {
    "self": "/api/box?filter[objects]=[{\"name\":\"width\",\"op\":\"ge\",\"field\
↪":\"height\"}]"
```

```
  },
  "meta": {
    "total": 100
  }
}
```

#### Using has and any

On request

```
GET /api/person?filter[objects]=[{"name":"articles","op":"any","val":{"name":"date
↪","op":"lt","val":"2010-01-01"}}] HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

the response will include only those people that have authored an article dated before January 1, 2010 (assume in the example below that at least one of the article linkage objects refers to an article that has such a date):

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "id": "1",
      "links": {
        "self": "http://example.com/api/person/1"
      },
      "relationships": {
        "articles": {
          "data": [
            {
              "id": "1",
              "type": "article"
            },
            {
              "id": "2",
              "type": "article"
            }
          ],
          "links": {
            "related": "http://example.com/api/person/1/articles",
            "self": "http://example.com/api/person/1/relationships/articles"
          }
        }
      },
      "type": "person"
    }
```

```
  ],
  "links": {
    "self": "/api/person?filter[objects]=[{\"name\":\"articles\",\"op\":\"any\",\
↪"val\":{\"name\":\"date\",\"op\":\"lt\",\"val\":\"2010-01-01\"}}]"
  },
  "meta": {
    "total": 1
  }
}
```

On request

```
GET /api/article?filter[objects]=[{"name":"author","op":"has","val":{"name":"age",
↪"op":"lte","val":50}}] HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

the response will include only those articles that have an author of age at most fifty (assume in the example below that the author linkage objects refers to a person that has such an age):

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "id": "1",
      "links": {
        "self": "http://example.com/api/article/1"
      },
      "relationships": {
        "author": {
          "data": {
            "id": "7",
            "type": "person"
          },
          "links": {
            "related": "http://example.com/api/article/1/author",
            "self": "http://example.com/api/article/1/relationships/author"
          }
        }
      },
      "type": "article"
    }
  ],
  "links": {
    "self": "/api/article?filter[objects]=[{\"name\":\"author\",\"op\":\"has\",\
↪"val\":{\"name\":\"age\",\"op\":\"lte\",\"val\":50}}]"
  },
```

```
  "meta": {
    "total": 1
  }
}
```

### Basic fetching

For the purposes of concreteness in this section, suppose we have executed the following code on the server:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_restless import APIManager

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)

class Article(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.Unicode)
    author_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    author = db.relationship(Person, backref=db.backref('articles'))

db.create_all()
manager = APIManager(app, flask_sqlalchemy_db=db)
manager.create_api(Person)
manager.create_api(Article)
```

By default, all columns and relationships will appear in the resource object representation of an instance of your model. See *Specifying which fields appear in responses* for more information on specifying which values appear in responses.

To fetch a collection of resources, the request

```
GET /api/person HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
```

```
  "data": [
    {
      "attributes": {
        "name": "John"
      },
      "id": "1",
      "links": {
        "self": "http://example.com/api/person/1"
      },
      "relationships": {
        "articles": {
          "data": [],
          "links": {
            "related": "http://example.com/api/person/1/articles",
            "self": "http://example.com/api/person/1/relationships/articles"
          }
        }
      },
      "type": "person"
    }
  ],
  "links": {
    "first": "http://example.com/api/person?page[number]=1&page[size]=10",
    "last": "http://example.com/api/person?page[number]=1&page[size]=10",
    "next": null,
    "prev": null,
    "self": "http://example.com/api/person"
  },
  "meta": {
    "total": 1
  }
}
```

To fetch a single resource, the request

```
GET /api/person/1 HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "attributes": {
      "name": "John"
    },
    "id": "1",
    "links": {
```

```
      "self": "http://example.com/api/person/1"
    },
    "relationships": {
      "articles": {
        "data": [],
        "links": {
          "related": "http://example.com/api/person/1/articles",
          "self": "http://example.com/api/person/1/relationships/articles"
        }
      }
    },
    "type": "person"
  }
}
```

To fetch a resource from a to-one relationship, the request

```
GET /api/article/1/author HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "attributes": {
      "name": "John"
    },
    "id": "1",
    "links": {
      "self": "http://example.com/api/person/1"
    },
    "relationships": {
      "articles": {
        "data": [
          {
            "id": "1",
            "type": "article"
          }
        ],
        "links": {
          "related": "http://example.com/api/person/1/articles",
          "self": "http://example.com/api/person/1/relationships/articles"
        }
      }
    },
    "type": "person"
  }
```

```
}
```

To fetch a resource from a to-many relationship, the request

```
GET /api/person/1/articles HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "attributes": {
        "title": "Once upon a time"
      },
      "id": "2",
      "links": {
        "self": "http://example.com/api/articles/2"
      },
      "relationships": {
        "author": {
          "data": {
            "id": "1",
            "type": "person",
          },
          "links": {
            "related": "http://example.com/api/articles/2/author",
            "self": "http://example.com/api/articles/2/relationships/author"
          }
        }
      },
      "type": "article"
    }
  ],
  "links": {
    "first": "http://example.com/api/person/1/articles?page[number]=1&
↪page[size]=10",
    "last": "http://example.com/api/person/1/articles?page[number]=1&page[size]=10
↪",
    "next": null,
    "prev": null,
    "self": "http://example.com/api/person/1/articles"
  },
  "meta": {
    "total": 1
  }
}
```

To fetch a single resource from a to-many relationship, the request

```
GET /api/person/1/articles/2 HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "attributes": {
      "title": "Once upon a time"
    },
    "id": "2",
    "links": {
      "self": "http://example.com/api/articles/2"
    },
    "relationships": {
      "author": {
        "data": {
          "id": "1",
          "type": "person"
        },
        "links": {
          "related": "http://example.com/api/articles/2/author",
          "self": "http://example.com/api/articles/2/relationships/author"
        }
      }
    },
    "type": "article"
  }
}
```

To fetch the link object for a to-one relationship, the request

```
GET /api/article/1/relationships/author HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "id": "1",
```

```
    "type": "person"
  }
}
```

To fetch the link objects for a to-many relationship, the request

```
GET /api/person/1/relationships/articles HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "id": "1",
      "type": "article"
    },
    {
      "id": "2",
      "type": "article"
    }
  ]
}
```

## Creating resources

For the purposes of concreteness in this section, suppose we have executed the following code on the server:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_restless import APIManager

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)

db.create_all()
manager = APIManager(app, flask_sqlalchemy_db=db)
manager.create_api(Person, methods=['POST'])
```

To create a new resource, the request

```
POST /api/person HTTP/1.1
Host: example.com
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "person",
    "attributes": {
      "name": "foo"
    }
  }
}
```

yields the response

```
HTTP/1.1 201 Created
Location: http://example.com/api/person/1
Content-Type: application/vnd.api+json

{
  "data": {
    "attributes": {
      "name": "foo"
    },
    "id": "1",
    "jsonapi": {
      "version": "1.0"
    },
    "links": {
      "self": "http://example.com/api/person/bd34b544-ad39-11e5-a2aa-4cbb58b9ee34"
    },
    "meta": {},
    "type": "person"
  }
}
```

To create a new resource with a client-generated ID (if enabled by setting `allow_client_generated_ids` to `True` in `APIManager.create_api()`), the request

```
POST /api/person HTTP/1.1
Host: example.com
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "person",
    "id": "bd34b544-ad39-11e5-a2aa-4cbb58b9ee34",
```

```
    "attributes": {
      "name": "foo"
    }
  }
}
```

yields the response

```
HTTP/1.1 201 Created
Location: http://example.com/api/person/bd34b544-ad39-11e5-a2aa-4cbb58b9ee34
Content-Type: application/vnd.api+json

{
  "data": {
    "attributes": {
      "name": "foo"
    },
    "id": "bd34b544-ad39-11e5-a2aa-4cbb58b9ee34",
    "links": {
      "self": "http://example.com/api/person/bd34b544-ad39-11e5-a2aa-4cbb58b9ee34"
    },
    "meta": {},
    "jsonapi": {
      "version": "1.0"
    },
    "type": "person"
  }
}
```

The server always responds with 201 Created and a complete resource object on a request with a client-generated ID.

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

## Deleting resources

For the purposes of concreteness in this section, suppose we have executed the following code on the server:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_restless import APIManager

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)
```

```
db.create_all()
manager = APIManager(app, flask_sqlalchemy_db=db)
manager.create_api(Person, methods=['DELETE'])
```

To delete a resource, the request

```
DELETE /api/person/1 HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields a 204 No Content response.

## Updating resources

For the purposes of concreteness in this section, suppose we have executed the following code on the server:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_restless import APIManager

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)

class Article(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    author_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    author = db.relationship(Person, backref=db.backref('articles'))

db.create_all()
manager = APIManager(app, flask_sqlalchemy_db=db)
manager.create_api(Person, methods=['PATCH'])
manager.create_api(Article)
```

To update an existing resource, the request

```
PATCH /api/person/1 HTTP/1.1
Host: example.com
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
```

```
    "type": "person",
    "id": 1,
    "attributes": {
      "name": "foo"
    }
  }
}
```

yields a 204 No Content response.

If you set the `allow_to_many_replacement` keyword argument of `APIManager.create_api()` to `True`, you can replace a to-many relationship entirely by making a request to update a resource. To update a to-many relationship, the request

```
PATCH /api/person/1 HTTP/1.1
Host: example.com
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "person",
    "id": 1,
    "relationships": {
      "articles": {
        "data": [
          {
            "id": "1",
            "type": "article"
          },
          {
            "id": "2",
            "type": "article"
          }
        ]
      }
    }
  }
}
```

yields a 204 No Content response.

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

## Updating relationships

For the purposes of concreteness in this section, suppose we have executed the following code on the server:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_restless import APIManager

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)

class Article(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    author_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    author = db.relationship(Person, backref=db.backref('articles'))

db.create_all()
manager = APIManager(app, flask_sqlalchemy_db=db)
manager.create_api(Person, methods=['PATCH'])
manager.create_api(Article)
```

To update a to-one relationship, the request

```
PATCH /api/articles/1/relationships/author HTTP/1.1
Host: example.com
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "person",
    "id": 1
  }
}
```

yields a 204 No Content response.

To update a to-many relationship (if enabled by setting `allow_to_many_replacement` to `True` in `APIManager.create_api()`), the request

```
PATCH /api/people/1/relationships/articles HTTP/1.1
Host: example.com
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": [
    {
      "type": "article",
      "id": 1
```

```
    },
    {
      "type": "article",
      "id": 2
    }
  ]
}
```

yields a 204 No Content response.

To add to a to-many relationship, the request

```
POST /api/person/1/relationships/articles HTTP/1.1
Host: example.com
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": [
    {
      "type": "article",
      "id": 1
    },
    {
      "type": "article",
      "id": 2
    }
  ]
}
```

yields a 204 No Content response.

To remove from a to-many relationship, the request

```
DELETE /api/person/1/links/articles HTTP/1.1
Host: example.com
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": [
    {
      "type": "article",
      "id": 1
    },
    {
      "type": "article",
      "id": 2
    }
  ]
}
```

yields a 204 No Content response.

To remove from a to-many relationship (if enabled by setting `allow_delete_from_to_many_relationships` to `True` in `APIManager.create_api()`), the request

```
DELETE /api/person/1/relationships/articles HTTP/1.1
Host: example.com
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": [
    {
      "type": "article",
      "id": 1
    },
    {
      "type": "article",
      "id": 2
    }
  ]
}
```

yields a 204 No Content response.

## Schema at root endpoint

A GET request to the root endpoint responds with a valid JSON API document whose `meta` element contains a `modelinfo` object, which itself contains one member for each resource object exposed by the API. Each element in `modelinfo` contains information about that resource. For example, a request like

```
GET /api HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

yields the response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": null,
  "jsonapi": {
    "version": "1.0"
  },
  "included": [],
  "links": {},
  "meta": {
```

```
    "modelinfo": {
      "article": {
        "primarykey": "id",
        "url": "http://example.com/api/article"
      },
      "person": {
        "primarykey": "id",
        "url": "http://example.com/api/person"
      }
    }
  }
}
```

### Resource ID must be a string

As required by the JSON API, the ID (and type) of a resource must be a string in request and response documents. This does *not* mean that the primary key in the database must be a string, only that it will appear as a string in communications between the client and the server. For more information, see the Identification section of the JSON API specification.

### Trailing slashes in URLs

API endpoints do not have trailing slashes. A GET request to, for example, `/api/person/` will result in a 404 Not Found response.

### 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 units 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.

### Errors and error messages

Flask-Restless returns the error responses required by the JSON API specification, and most other server errors yield a 400 Bad Request. Errors are included in the `errors` element in the top-level JSON document in the response body.

If a request triggers a `sqlalchemy.exc.SQLAlchemyError` (or any subclass of that exception, including `DataError`, `IntegrityError`, `ProgrammingError`, etc.), the session will be rolled back

## JSONP callbacks

Flask-Restless responds to JavaScript clients that request JSONP responses. Add a `callback=myfunc` query parameter to the request URL on any request that yields a response that contains content (including endpoints for function evaluation; see *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.

The Content-Type of a JSONP response is `application/javascript` instead of `application/vnd.api+json` because the payload of such a response is not valid JSON API.

For example, a request like this:

```
GET /api/person/1?callback=foo HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

will produce a response like this:

```
HTTP/1.1 200 OK
Content-Type: application/javascript

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

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

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

## JSON API extensions

Flask-Restless does not yet support the in-development JSON API extension system.

## 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_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_blueprint('mypersonapi', Person)
blueprint.after_request(add_cors_headers)
app.register_blueprint(blueprint)
```

# Customizing the ReSTful interface

This section describes how to use the keyword arguments to the `create_api()` method to customize the interface created by Flask-Restless.

## Custom serialization

New in version 0.17.0.

Changed in version 1.0.0b1: Transitioned from function-based serialization to class-based serialization.

Flask-Restless provides serialization and deserialization that work with the JSON API specification. If you wish to have more control over the way instances of your models are converted to Python dictionary representations, you can specify custom serialization by providing it to `APIManager.create_api()` via the `serializer_class` keyword argument. Similarly, to provide a deserializer that converts a Python dictionary representation to an instance of your model, use the `deserializer_class` keyword argument. However, if you provide a serializer that fails to produce resource objects that satisfy the JSON API specification, your client will receive non-compliant responses!

Your serializer classes must be a subclass of `DefaultSerializer` and can override the `serialize()` and `serialize_many()` methods to provide custom serialization. These methods take an instance or instances as input and return a dictionary representing a JSON API document. Each also accepts an `only` keyword argument, indicating the sparse fieldsets requested by the client:

```
from flask_restless import DefaultSerializer

class MySerializer(DefaultSerializer):

    def serialize(self, instance, only=None):
        super_serialize = super(DefaultSerializer, self).serialize
        document = super_serialize(instance, only=only)
        # Make changes to the document here...
        ...
        return document

    def serialize_many(self, instances, only=None):
        super_serialize = super(DefaultSerializer, self).serialize_many
        document = super_serialize(instances, only=only)
        # Make changes to the document here...
        ...
        return document
```

`instance` is an instance of a SQLAlchemy model, `instances` is a list of instances, and the `only` argument is a list; only the fields (that is, the attributes and relationships) whose names appear as strings in *only* should appear in the returned dictionary. The only exception is that the keys `'id'` and `'type'` must always appear, regardless of whether they appear in *only*. The function must return a dictionary representation of the resource object.

Flask-Restless also provides functional access to the default serialization, via the `simple_serialize()` and `simple_serialize_many()` functions, which return the result of the built-in default serialization.

For deserialization, define your custom deserialization class like this:

```
from flask_restless import DefaultDeserializer

class MyDeserializer(DefaultDeserializer):

    def deserialize(self, document):
        return Person(...)
```

`document` is a dictionary representation of the *complete* incoming JSON API document, where the `data` element contains the primary resource object or objects. The function must return an instance of the model that has the requested fields. If you override the constructor, it must take two positional arguments, *session* and *model*.

Your code can raise a `SerializationException` when overriding the `DefaultSerializer.serialize()` method, and similarly a `DeserializationException` in the `DefaultDeserializer.deserialize()` method; Flask-Restless will automatically catch those exceptions and format a JSON API error response. If you wish to collect multiple exceptions (for example, if several fields of a resource provided to the `deserialize()` method fail validation) you can raise a `MultipleExceptions` exception, providing a list of other serialization or deserialization exceptions at instantiation time.

**Note:** If you wish to write your own serialization functions, we **strongly suggest** using a Python object serialization library instead of writing your own serialization functions. This is also likely a better approach than specifying which columns to include or exclude (*Inclusion of related resources*) or preprocessors and postprocessors (*Request preprocessors and postprocessors*).

For example, if you create schema for your database models using Marshmallow, then you use that library's built-in serialization functions as follows:

```
class PersonSchema(Schema):
    id = fields.Integer()
    name = fields.String()

    def make_object(self, data):
        return Person(**data)

class PersonSerializer(DefaultSerializer):

    def serialize(self, instance, only=None):
        person_schema = PersonSchema(only=only)
        return person_schema.dump(instance).data

    def serialize_many(self, instances, only=None):
        person_schema = PersonSchema(many=True, only=only)
        return person_schema.dump(instances).data


class PersonDeserializer(DefaultDeserializer):

    def deserialize(self, document):
        person_schema = PersonSchema()
        return person_schema.load(instance).data

    # # JSON API doesn't currently allow bulk creation of resources. When
    # # it does, either in the specification or in an extension, this is
    # # how you would implement it.
    # def deserialize_many(self, document):
    #     person_schema = PersonSchema(many=True)
    #     return person_schema.load(instance).data

manager = APIManager(app, session=session)
manager.create_api(Person, methods=['GET', 'POST'],
                   serializer_class=PersonSerializer,
                   deserializer_class=PersonDeserializer)
```

For a complete version of this example, see the `examples/server_configurations/custom_serialization.py` module in the source distribution, or view it online.

Per-model serialization

The correct serialization function will be used for each type of SQLAlchemy model for which you invoke `APIManager.create_api()`. For example, if you create two APIs, one for `Person` objects and one for `Article` objects,

```
manager.create_api(Person, serializer=person_serializer)
manager.create_api(Article, serializer=article_serializer)
```

and then make a request like

```
GET /api/article/1?include=author HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

then Flask-Restless will use the `article_serializer` function to serialize the primary data (that is, the top-level `data` element in the response document) and the `person_serializer` to serialize the included `Person` resource.

## 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.

There are many different routes on which you can apply preprocessors and postprocessors, depending on HTTP method type, whether the client is accessing a resource or a relationship, whether the client is accessing a collection or a single resource, etc.

This table states the preprocessors that apply to each type of endpoint.

| preprocessor name | applies to URLs like... |
|---|---|
| GET_COLLECTION | /api/person |
| GET_RESOURCE | /api/person/1 |
| GET_RELATION | /api/person/1/articles |
| GET_RELATED_RESOURCE | /api/person/1/articles/2 |
| DELETE_RESOURCE | /api/person/1 |
| POST_RESOURCE | /api/person |
| PATCH_RESOURCE | /api/person/1 |
| GET_RELATIONSHIP | /api/person/1/relationships/articles |
| DELETE_RELATIONSHIP | /api/person/1/relationships/articles |
| POST_RELATIONSHIP | /api/person/1/relationships/articles |
| PATCH_RELATIONSHIP | /api/person/1/relationships/articles |

This table states the postprocessors that apply to each type of endpoint.

| postprocessor name | applies to URLs like... |
|---|---|
| GET_COLLECTION | /api/person |
| GET_RESOURCE | /api/person/1 |
| GET_TO_MANY_RELATION | /api/person/1/articles |
| GET_TO_ONE_RELATION | /api/articles/1/author |
| GET_RELATED_RESOURCE | /api/person/1/articles/2 |
| DELETE_RESOURCE | /api/person/1 |
| POST_RESOURCE | /api/person |
| PATCH_RESOURCE | /api/person/1 |
| GET_TO_MANY_RELATIONSHIP | /api/person/1/relationships/articles |
| GET_TO_ONE_RELATIONSHIP | /api/articles/1/relationships/author |
| GET_RELATIONSHIP | /api/person/1/relationships/articles |
| DELETE_RELATIONSHIP | /api/person/1/relationships/articles |
| POST_RELATIONSHIP | /api/person/1/relationships/articles |
| PATCH_RELATIONSHIP | /api/person/1/relationships/articles |

Each type of preprocessor or postprocessor requires different arguments. For preprocessors:

| preprocessor name | keyword arguments |
|---|---|
| GET_COLLECTION | filters, sort, group_by, single |
| GET_RESOURCE | resource_id |
| GET_RELATION | resource_id, relation_name, filters, sort, group_by, single |
| GET_RELATED_RESOURCE | resource_id, relation_name, related_resource_id |
| DELETE_RESOURCE | resource_id |
| POST_RESOURCE | data |
| PATCH_RESOURCE | resource_id, data |
| GET_RELATIONSHIP | resource_id, relation_name |
| DELETE_RELATIONSHIP | resource_id, relation_name |
| POST_RELATIONSHIP | resource_id, relation_name, data |
| PATCH_RELATIONSHIP | resource_id, relation_name, data |

For postprocessors:

| postprocessor name | keyword arguments |
|---|---|
| GET_COLLECTION | result, filters, sort, group_by, single |
| GET_RESOURCE | result |
| GET_TO_MANY_RELATION | result, filters, sort, group_by, single |
| GET_TO_ONE_RELATION | result |
| GET_RELATED_RESOURCE | result |
| DELETE_RESOURCE | was_deleted |
| POST_RESOURCE | result |
| PATCH_RESOURCE | result |
| GET_TO_MANY_RELATIONSHIP | result, filters, sort, group_by, single |
| GET_TO_ONE_RELATIONSHIP | result |
| DELETE_RELATIONSHIP | was_deleted |
| POST_RELATIONSHIP | none |
| PATCH_RELATIONSHIP | none |

How can one use these tables to create a preprocessor or postprocessor? If you want to create a preprocessor that will be applied on GET requests to `/api/person`, first define a function that accepts the keyword arguments you need, and has a `**kw` argument for any additional keyword arguments (and any new arguments that may appear in future versions of Flask-Restless):

```
def fetch_preprocessor(filters=None, sort=None, group_by=None, single=None,
                       **kw):
    # Here perform any application-specific code...
```

Next, instruct these preprocessors to be applied by Flask-Restless by using the `preprocessors` keyword argument to `APIManager.create_api()`. The value of this argument must be a dictionary in which each key is a string containing a processor name and each value is a list of functions to be applied for that request:

```
preprocessors = {'GET_COLLECTION': [fetch_preprocessor]}
manager.create_api(Person, preprocessors=preprocessors)
```

For preprocessors for endpoints of the form `/api/person/1`, a returned value will be interpreted as the resource ID for the request. (Remember, as described in *Resource ID must be a string*, the returned ID must be a string.) For example, if a preprocessor for a GET request to `/api/person/1` returns the string `'foo'`, then Flask-Restless will behave as if the request were originally for the URL `/api/person/foo`. For preprocessors for endpoints of the form `/api/person/1/articles` or `/api/person/1/relationships/articles`, the function can return either one value, in which case the resource ID will be replaced with the return value, or a two-tuple, in which case both the resource ID and the relationship name will be replaced. Finally, for preprocessors for endpoints of the form `/api/person/1/articles/2`, the function can return one, two, or three values; if three values are returned, the resource ID, the relationship name, and the related resource ID are all replaced. (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.

---

**Note:** For more information about the `filters` and `single` keyword arguments, see *Filtering*. For more information about `sort` and `group_by` keyword arguments, see *Sorting*.

---

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(resource_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(detail='Not Authorized', status=401)
manager.create_api(Person, preprocessors=dict(GET_SINGLE=[check_auth]))
```

The `ProcessingException` allows you to specify as keyword arguments to the constructor the elements of the JSON API error object. If no arguments are provided, the error is assumed to have status code 400 Bad Request.

### 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_restless import APIManager
from flask_restless import ProcessingException
from flask_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(detail='Not authenticated', status=401)

app = Flask(__name__)
preprocessors = {'POST_RESOURCE': [auth_func]}
api_manager = APIManager(app, session=session, preprocessors=preprocessors)
api_manager.create_api(User)
```

#### Preprocessors for collections

When the server receives, for example, a GET request for `/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, a GET request to `/api/person` is roughly equivalent to the same request to `/api/person?filter[objects]=[]`. Therefore, if you want to filter the set of `Person` instances returned by such a request, you can create a `GET_COLLECTION` preprocessor that *appends filters* to the `filters` keyword argument. For example:

```
def preprocessor(filters=None, **kw):
    # This checks if the preprocessor function is being called before a
    # request that does not have search parameters.
    if filters 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)
    # *Append* your filter to the list of filters.
    filters.append(filt)

preprocessors = {'GET_COLLECTION': [preprocessor]}
manager.create_api(Person, preprocessors=preprocessors)
```

#### When does the session get committed?

For requests to create a resource, update a resource, or delete a resource, the session is flushed *before* the postprocessor is executed and committed *after*. Therefore, if a postprocessor raises a `ProcessingException`, then the session has *not* been committed, so your code can then decide to, for example, roll back the session or commit it.

#### Requiring authentication for some methods

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

```
from flask import Flask
from flask_restless import APIManager
from flask_restless import ProcessingException
```

```
from flask_login import current_user
from mymodels import User

def auth_func(*args, **kwargs):
    if not current_user.is_authenticated():
        raise ProcessingException(detail='Not authenticated', status=401)

app = Flask(__name__)
api_manager = APIManager(app)
# Set `auth_func` to be a preprocessor for any type of endpoint you want to
# be guarded by authentication.
preprocessors = {'GET_RESOURCE': [auth_func], ...}
api_manager.create_api(User, preprocessors=preprocessors)
```

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

## 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 PATCH.

If you allow GET requests, you will have access to endpoints of the following forms.

**`GET /api/person`**

**`GET /api/person/1`**

**`GET /api/person/1/comments`**

**`GET /api/person/1/relationships/comments`**

**`GET /api/person/1/comments/2`**

The first four are described explicitly in the JSON API specification. The last is particular to Flask-Restless; it allows you to access a particular related resource via a relationship on another resource.

If you allow DELETE requests, you will have access to endpoints of the form

**`DELETE /api/person/1`**

If you allow POST requests, you will have access to endpoints of the form

**`POST /api/person`**

Finally, if you allow PATCH requests, you will have access to endpoints of the following forms.

**`PATCH /api/person/1`**

**`POST /api/person/1/relationships/comments`**

**`PATCH /api/person/1/relationships/comments`**

**`DELETE /api/person/1/relationships/comments`**

The last three allow the client to interact with the relationships of a particular resource. The last two must be enabled explicitly by setting the `allow_to_many_replacement` and `allow_delete_from_to_many_relationships`, respectively, to `True` when creating an API using the `APIManager.create_api()` method.

## API prefix

To create an API at a prefix other than the default `/api`, 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`.

## Collection name

By default, the name of the collection that 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 `__table__.name` 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`.

---

**Note:** According to the JSON API specification,

> Note: This spec is agnostic about inflection rules, so the value of type can be either plural or singular. However, the same value should be used consistently throughout an implementation.

It's up to you to make sure your collection names are either all plural or all singular!

---

### 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/123`.

### 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.

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],
                      methods=['PATCH', 'POST'])
```

---

**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

{
  "errors": [
    {
      "status": 400,
      "title": "Validation error",
      "detail": "age: must be an integer"
    }
  ]
}
```

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

## Bulk operations

Bulk operations are not supported, though they may be in the future.

## Custom serialization and deserialization

You can provide a custom serializer using the `serializer_class` keyword argument and a custom deserializer using the `deserializer_class` keyword argument. For a full description of how to use these arguments, see *Custom serialization*.

### Request preprocessors and postprocessors

You can have custom code executed before or after Flask-Restless handles the incoming request by using the `preprocessors` and `postprocessors` keyword arguments, respectively. For a full description of how to use these arguments, see *Request preprocessors and postprocessors*.

## Common SQLAlchemy setups

Flask-Restless automatically handles SQLAlchemy models defined with association proxies and polymorphism.

### Association proxies

Flask-Restless handles many-to-many relationships transparently through association proxies. It exposes the remote table in the `relationships` element of a resource in the JSON document and hides the association table or association object.

#### Proxying association objects

*For more information on using association proxies with association objects, see the 'Simplifying Assocation Objects'_ section of the SQLAlchemy documentation.*

When proxying a to-many relationship via an association object, the related resources will appear in the `relationships` element of the resource object in addition to the association object. For example, in the following setup, each article has a to-many relationship to tags via the `ArticleTag` object:

```
from sqlalchemy import Column, Integer, Unicode, ForeignKey
from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Article(Base):
    __tablename__ = 'article'
    id = Column(Integer, primary_key=True)
    articletags = relationship('ArticleTag',
                               cascade='all, delete-orphan')
    tags = association_proxy('articletags', 'tag',
                             creator=lambda tag: ArticleTag(tag=tag))

class ArticleTag(Base):
    __tablename__ = 'articletag'
    article_id = Column(Integer, ForeignKey('article.id'),
```

```
                        primary_key=True)
    tag_id = Column(Integer, ForeignKey('tag.id'), primary_key=True)
    tag = relationship('Tag')

class Tag(Base):
    __tablename__ = 'tag'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode)
```

Resource objects of type `'article'` will have both an `articletags` relationship as well as a `tags` relationship that proxies directly to the `Tag` resource through the `ArticleTag` table.

```
{
  "data": {
    "id": "1",
    "type": "article",
    "relationships": {
      "articletags": {
        "data": [
          {
            "id": "1",
            "type": "articletag"
          },
          {
            "id": "2",
            "type": "articletag"
          }
        ]
      },
      "tags": {
        "data": [
          {
            "id": "1",
            "type": "tag"
          },
          {
            "id": "2",
            "type": "tag"
          }
        ]
      }
    }
  }
}
```

If you wish to exclude the association object relationship, use the `exclude` keyword argument when creating the API for the `Article` model:

```
manager.create_api(Article, exclude=['articletags'])
```

### Proxying association tables

*For more information on using association proxies with association objects, see the 'Simplifying Scalar Collections'_ section of the SQLAlchemy documentation.*

When proxying an attribute of a to-many relationship via an association table, the attribute will appear in the `attributes` element of the resource object and the to-many relationship will appear in the `relationships` element of the resource object but the association table will not appear. For example, in the following setup, each article has an association proxy `tag_names` which is a list of the `name` attribute of each related tag:

```
from sqlalchemy import Column, Integer, Unicode, ForeignKey
from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Article(Base):
    __tablename__ = 'article'
    id = Column(Integer, primary_key=True)
    tags = relationship('Tag', secondary=lambda: articletags_table)
    tag_names = association_proxy('tags', 'name',
                                  creator=lambda s: Tag(name=s))

class Tag(Base):
    __tablename__ = 'tag'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode)

articletags_table = \
    Table('articletags', Base.metadata,
          Column('article_id', Integer, ForeignKey('article.id'),
                 primary_key=True),
          Column('tag_id', Integer, ForeignKey('tag.id'),
                 primary_key=True))
```

Resource objects of type `'article'` will have a `tag_names` attribute that is a list of tag names in addition to a `tags` relationship. The intermediate `articletags` table does not appear as a relationship in the resource object:

```
{
  "data": {
    "id": "1",
    "type": "article",
    "attributes": {
      "tag_names": [
```

```
        "foo",
        "bar"
      ]
    },
    "relationships": {
      "tags": {
        "data": [
          {
            "id": "1",
            "type": "tag"
          },
          {
            "id": "2",
            "type": "tag"
          }
        ],
      }
    }
  }
}
```

## Polymorphic models

Flask-Restless automatically handles polymorphic models defined using either single table or joined table inheritance. We have made some design choices we believe are reasonable. Requests to create, update, or delete a resource must specify a `type` that matches the collection name of the endpoint. This means you cannot request to create a resource of the subclass type at the endpoint for the superclass type, for example. On the other hand, requests to fetch a collection of objects that have a subclass will yield a response that includes all resources of the superclass and all resources of any subclass.

For example, consider a setup where there are employees and some employees are managers:

```
from sqlalchemy import Column, Integer, Enum
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    type = Column(Enum('employee', 'manager'), nullable=False)
    __mapper_args__ = {
        'polymorphic_on': type,
        'polymorphic_identity': 'employee'
    }

class Manager(Employee):
```

```
    __mapper_args__ = {
        'polymorphic_identity': 'manager'
    }
```

### Collection name

When creating an API for these models, Flask-Restless chooses the polymorphic identity as the collection name:

```
>>> from flask.ext.restless import collection_name
>>>
>>> manager.create_api(Employee)
>>> manager.create_api(Manager)
>>> collection_name(Employee)
'employee'
>>> collection_name(Manager)
'manager'
```

### Creating and updating resources

Creating a resource require the `type` element of the resource object in the request to match the collection name of the endpoint:

```
>>> from flask import json
>>> import requests
>>>
>>> headers = {
...     'Accept': 'application/vnd.api+json',
...     'Content-Type': 'application/vnd.api+json'
... }
>>> resource = {'data': {'type': 'employee'}}
>>> data = json.dumps(resource)
>>> response = requests.post('https://example.com/api/employee', data=data,
...                          headers=headers)
>>> response.status_code
201
>>> resource = {'data': {'type': 'manager'}}
>>> data = json.dumps(resource)
>>> response = requests.post('https://example.com/api/manager', data=data,
...                          headers=headers)
>>> response.status_code
201
```

If the `type` does not match the collection name for the endpoint, the server responds with a 409 Conflict:

```
>>> resource = {'data': {'type': 'manager'}}
>>> data = json.dumps(resource)
```

```
>>> response = requests.post('https://example.com/api/employee', data=data,
...                          headers=headers)
>>> response.status_code
409
```

The same rules apply for updating resources.

### Fetching resources

Assume the database contains an employee with ID 1 and a manager with ID 2. You can only fetch each individual resource at the endpoint for the exact type of that resource:

```
>>> response = requests.get('https://example.com/api/employee/1')
>>> response.status_code
200
>>> response = requests.get('https://example.com/api/manager/2')
>>> response.status_code
200
```

You cannot access individual resources of the subclass at the endpoint for the superclass:

```
>>> response = requests.get('https://example.com/api/employee/2')
>>> response.status_code
404
>>> response = requests.get('https://example.com/api/manager/1')
>>> response.status_code
404
```

Fetching from the superclass endpoint yields a response that includes resources of the superclass and resources of the subclass:

```
>>> response = requests.get('https://example.com/api/employee')
>>> document = json.loads(response.data)
>>> resources = document['data']
>>> employee, manager = resources
>>> employee['type']
'employee'
>>> employee['id']
'1'
>>> manager['type']
'manager'
>>> manager['id']
'2'
```

Deleting resources

Assume the database contains an employee with ID 1 and a manager with ID 2. You can only delete from the endpoint that matches the exact type of the resource:

```
>>> response = requests.delete('https://example.com/api/employee/2')
>>> response.status_code
404
>>> response = requests.delete('https://example.com/api/manager/1')
>>> response.status_code
404
>>> response = requests.delete('https://example.com/api/employee/1')
>>> response.status_code
204
>>> response = requests.delete('https://example.com/api/manager/2')
>>> response.status_code
204
```

CHAPTER **2**

# API reference

A technical description of the classes, functions, and idioms of Flask-Restless.

## API

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

### The API Manager class

**class** flask_restless.**APIManager**(*app=None*, *session=None*, *flask_sqlalchemy_db=None*, *preprocessors=None*, *postprocessors=None*, *url_prefix=None*)

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

The `Flask` object can either be specified in the constructor, or after instantiation time by calling the `init_app()` method.

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

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

*flask_sqlalchemy_db* is the `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_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_restless import APIManager
from flask_sqlalchemy import SQLAlchemy

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

*url_prefix* is the URL prefix at which each API created by this instance will be accessible. For example, if this is set to `'foo'`, then this method creates endpoints of the form `/foo/<collection_name>` when `create_api()` is called. If the *url_prefix* is set in the `create_api()`, the URL prefix set in the constructor will be ignored for that endpoint.

*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*.

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

Creates and possibly registers a ReSTful API blueprint for the given SQLAlchemy model.

If a Flask application was provided in the constructor of this class, the created blueprint is immediately registered on that application. Otherwise, the blueprint is stored for later registration when the `init_app()` method is invoked. In that case, the blueprint will be registered each time the `init_app()` method is invoked.

The keyword arguments for this method are exactly the same as those for `create_api_blueprint()`, and are passed directly to that method. However, unlike that method, this method accepts only a single positional argument, *model*, the SQLAlchemy model for which to create the API. A UUID will be

automatically generated for the blueprint name.

For example, if you only wish to create APIs on a single Flask application:

```
app = Flask(__name__)
session = ...  # create the SQLAlchemy session
manager = APIManager(app=app, session=session)
manager.create_api(User)
```

If you want to create APIs before having access to a Flask application, you can call this method before calling `init_app()`:

```
session = ...  # create the SQLAlchemy session
manager = APIManager(session=session)
manager.create_api(User)

# later...
app = Flask(__name__)
manager.init_app(app)
```

If you want to create an API and register it on multiple Flask applications, you can call this method once and `init_app()` multiple times with different *app* arguments:

```
session = ...  # create the SQLAlchemy session
manager = APIManager(session=session)
manager.create_api(User)

# later...
app1 = Flask('application1')
app2 = Flask('application2')
manager.init_app(app1)
manager.init_app(app2)
```

**create_api_blueprint**(*name*, *model*, *methods=frozenset({'GET'})*, *url_prefix=None*, *collection_name=None*, *allow_functions=False*, *only=None*, *exclude=None*, *additional_attributes=None*, *validation_exceptions=None*, *page_size=10*, *max_page_size=100*, *preprocessors=None*, *postprocessors=None*, *primary_key=None*, *serializer_class=None*, *deserializer_class=None*, *includes=None*, *allow_to_many_replacement=False*, *allow_delete_from_to_many_relationships=False*, *allow_client_generated_ids=False*)

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.__table__.name`. (If any black magic was performed on `model.__table__`, 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 that handles the endpoints for the model. The returned `Blueprint` has *not* been registered with the `Flask` application object specified in the constructor of this class, so you will need to register it yourself to make it available on the application. If you don't need access to the `Blueprint` object, use `create_api_blueprint()` instead, which handles registration automatically.

*name* is the name of the blueprint that will be created.

*model* is the SQLAlchemy model class for which a ReSTful interface will be created.

*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* is a list of strings specifying the HTTP methods that will be made available on the ReSTful API for the specified model.

- If `'GET'` is in the list, GET requests will be allowed at endpoints for collections of resources, resources, to-many and to-one relations of resources, and particular members of a to-many relation. Furthermore, relationship information will be accessible. For more information, see *Fetching resources and relationships*.
- If `'POST'` is in the list, POST requests will be allowed at endpoints for collections of resources. For more information, see *Creating resources*.
- If `'DELETE'` is in the list, DELETE requests will be allowed at endpoints for individual resources. For more information, see *Deleting resources*.
- If `'PATCH'` is in the list, PATCH requests will be allowed at endpoints for individual resources. Replacing a to-many relationship when issuing a request to update a resource can be enabled by setting `allow_to_many_replacement` to `True`.

  Furthermore, to-one relationships can be updated at the relationship endpoints under an individual resource via PATCH requests. This also allows you to add to a to-many relationship via the POST method, delete from a to-many relationship via the DELETE method (if `allow_delete_from_to_many_relationships` is set to `True`), and replace a to-many relationship via the PATCH method (if `allow_to_many_replacement` is set to `True`). For more information, see *Updating resources* and *Updating relationships*.

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

*url_prefix* is the URL prefix at which this API will be accessible. For example, if this is set to `'/foo'`, then this method creates endpoints of the form `/foo/<collection_name>`. If not set, the default URL prefix specified in the constructor of this class will be used. If that was not set either, the default `'/api'` will be used.

*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. For example, if this is set to `'foo'`, then this method creates endpoints of the form `/api/foo`, `/api/foo/<id>`, etc.

If *allow_functions* is `True`, then GET requests to `/api/eval/<collection_name>` 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 is `False` by default.

> **Warning:** If `allow_functions` is `True`, you must not create an API for a model whose name is `'eval'`.

If *only* is not `None`, it must be a list of columns and/or relationships of the specified *model*, given either as strings or as the attributes themselves. If it is a list, only these fields will appear in the resource object representation of an instance of *model*. In other words, *only* is a whitelist of fields. The `id` and `type` elements of the resource object will always be present regardless of the value of this argument. If *only* contains a string that does not name a column in *model*, it will be ignored.

If *additional_attributes* is a list of strings, these attributes of the model will appear in the JSON representation of an instance of the model. This is useful if your model has an attribute that is not a SQLAlchemy column but you want it to be exposed. If any of the attributes does not exist on the model, a `AttributeError` is raised.

If *exclude* is not `None`, it must be a list of columns and/or relationships of the specified *model*, given either as strings or as the attributes themselves. If it is a list, all fields **except** these will appear in the resource object representation of an instance of *model*. In other words, *exclude* is a blacklist of fields. The `id` and `type` elements of the resource object will always be present regardless of the value of this argument. If *exclude* contains a string that does not name a column in *model*, it will be ignored.

If either *only* or *exclude* is not `None`, exactly one of them must be specified; if both are not `None`, then this function will raise a `IllegalArgumentError`.

See *Specifying which fields appear in responses* for more information on specifying which fields will be included in the resource object representation.

*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 the format described by the JSON API specification. For more information on how to use validation, see *Capturing validation errors*.

*page_size* must be a positive integer that represents the default page size for responses that consist of a collection of resources. Requests made by clients may override this default by specifying `page_size` as a query parameter. *max_page_size* must be a positive integer that represents the maximum page size that a client can request. Even if a client specifies that greater than *max_page_size* should be returned, at most *max_page_size* results will be returned. For more information, see *Pagination*.

*serializer_class* and *deserializer_class* are custom serializer and deserializer classes. The former must be a subclass of `DefaultSerializer` and the latter a subclass of `DefaultDeserializer`. For more information on using these, see *Custom serialization*.

*preprocessors* is a dictionary mapping strings to lists of functions. Each key represents a type of endpoint (for example, `'GET_RESOURCE'` or `'GET_COLLECTION'`). 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. For more information, see *Specifying one of many primary keys*.

*includes* must be a list of strings specifying which related resources will be included in a compound document by default when fetching a resource object representation of an instance of *model*. Each element of *includes* is the name of a field of *model* (that is, either an attribute or a relationship). For more information, see *Inclusion of related resources*.

If *allow_to_many_replacement* is `True` and this API allows PATCH requests, the server will allow two types of requests. First, it allows the client to replace the entire collection of resources in a to-many relationship when updating an individual instance of the model. Second, it allows the client to replace the entire to-many relationship when making a PATCH request to a to-many relationship endpoint. This is `False` by default. For more information, see *Updating resources* and *Updating relationships*.

If *allow_delete_from_to_many_relationships* is `True` and this API allows PATCH requests, the server will allow the client to delete resources from any to-many relationship of the model. This is `False` by default. For more information, see *Updating relationships*.

If *allow_client_generated_ids* is `True` and this API allows POST requests, the server will allow the client to specify the ID for the resource to create. JSON API recommends that this be a UUID. This is `False` by default. For more information, see *Creating resources*.

**init_app**(*app*)

Registers any created APIs on the given Flask application.

This function should only be called if no Flask application was provided in the *app* keyword argument to the constructor of this class.

When this function is invoked, any blueprint created by a previous invocation of `create_api()` will be registered on *app* by calling the `register_blueprint()` method.

To use this method with pure SQLAlchemy, for example:

```
from flask import Flask
from flask_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()

# Here create model classes, for example User, Comment, etc.
...

# Create the API manager and create the APIs.
apimanager = APIManager(session=mysession)
apimanager.create_api(User)
apimanager.create_api(Comment)

# Later, call `init_app` to register the blueprints for the
# APIs created earlier.
app = Flask(__name__)
apimanager.init_app(app)
```

and with models defined with Flask-SQLAlchemy:

```
from flask import Flask
from flask_restless import APIManager
from flask_sqlalchemy import SQLAlchemy

db = SQLALchemy(app)

# Here create model classes, for example User, Comment, etc.
...

# Create the API manager and create the APIs.
apimanager = APIManager(flask_sqlalchemy_db=db)
```

```
apimanager.create_api(User)
apimanager.create_api(Comment)

# Later, call `init_app` to register the blueprints for the
# APIs created earlier.
app = Flask(__name__)
apimanager.init_app(app)
```

**class** flask_restless.**IllegalArgumentError**
This exception is raised when a calling function has provided illegal arguments to a function or method.

## Search helper functions

flask_restless.**register_operator**(*name*, *op*)
Register an operator so the system can create expressions involving it.

*name* is a string naming the operator and *op* is a function that takes up to two arguments as input. If the name provided is one of the built-in operators (see *Operators*), it will override the default behavior of that operator. For example, calling

```
register_operator('gt', myfunc)
```

will cause `myfunc()` to be invoked in the SQLAlchemy expression created for this operator instead of the default "greater than" operator.

## Global helper functions

flask_restless.**collection_name**(*model*, *_apimanager=None*)
Returns the collection name for the specified model, as specified by the `collection_name` keyword argument to `APIManager.create_api()` when it was previously invoked on the model.

*model* is a SQLAlchemy model class. This should be a model on which `APIManager.create_api_blueprint()` (or `APIManager.create_api()`) has been invoked previously. If no API has been created for it, this function raises a *ValueError*.

If *_apimanager* is not `None`, it must be an instance of `APIManager`. Restrict our search for endpoints exposing *model* to only endpoints created by the specified `APIManager` instance.

For example, suppose you have a model class `Person` and have created the appropriate Flask application and SQLAlchemy session:

```
>>> from mymodels import Person
>>> manager = APIManager(app, session=session)
>>> manager.create_api(Person, collection_name='people')
```

```
>>> collection_name(Person)
'people'
```

This function is the inverse of `model_for()`:

```
>>> manager.collection_name(manager.model_for('people'))
'people'
>>> manager.model_for(manager.collection_name(Person))
<class 'mymodels.Person'>
```

`flask_restless.`**`model_for`**(*collection_name*, *_apimanager=None*)

Returns the model corresponding to the given collection name, as specified by the `collection_name` keyword argument to `APIManager.create_api()` when it was previously invoked on the model.

*collection_name* is a string corresponding to the "type" of a model. This should be a model on which `APIManager.create_api_blueprint()` (or `APIManager.create_api()`) has been invoked previously. If no API has been created for it, this function raises a *ValueError*.

If *_apimanager* is not `None`, it must be an instance of `APIManager`. Restrict our search for endpoints exposing *model* to only endpoints created by the specified `APIManager` instance.

For example, suppose you have a model class `Person` and have created the appropriate Flask application and SQLAlchemy session:

```
>>> from mymodels import Person
>>> manager = APIManager(app, session=session)
>>> manager.create_api(Person, collection_name='people')
>>> model_for('people')
<class 'mymodels.Person'>
```

This function is the inverse of `collection_name()`:

```
>>> manager.collection_name(manager.model_for('people'))
'people'
>>> manager.model_for(manager.collection_name(Person))
<class 'mymodels.Person'>
```

`flask_restless.`**`serializer_for`**(*model*, *_apimanager=None*)

Returns the callable serializer object for the specified model, as specified by the *serializer* keyword argument to `APIManager.create_api()` when it was previously invoked on the model.

*model* is a SQLAlchemy model class. This should be a model on which `APIManager.create_api_blueprint()` (or `APIManager.create_api()`) has been invoked previously. If no API has been created for it, this function raises a *ValueError*.

If *_apimanager* is not `None`, it must be an instance of `APIManager`. Restrict our search for endpoints exposing *model* to only endpoints created by the specified

APIManager instance.

For example, suppose you have a model class Person and have created the appropriate Flask application and SQLAlchemy session:

```
>>> from mymodels import Person
>>> def my_serializer(model, *args, **kw):
...     # return something cool here...
...     return {}
...
>>> manager = APIManager(app, session=session)
>>> manager.create_api(Person, serializer=my_serializer)
>>> serializer_for(Person)
<function my_serializer at 0x...>
```

flask_restless.**primary_key_for**(*model*, *_apimanager=None*)

Returns the primary key to be used for the given model or model instance, as specified by the primary_key keyword argument to APIManager.create_api() when it was previously invoked on the model.

*primary_key* is a string corresponding to the primary key identifier to be used by flask-restless for a model. If no primary key has been set at the flask-restless level (by using the primary_key keyword argument when calling APIManager.create_api_blueprint(), the model's primary key will be returned. If no API has been created for the model, this function raises a *ValueError*.

If *_apimanager* is not None, it must be an instance of APIManager. Restrict our search for endpoints exposing *model* to only endpoints created by the specified APIManager instance.

For example, suppose you have a model class Person and have created the appropriate Flask application and SQLAlchemy session:

```
>>> from mymodels import Person
>>> manager = APIManager(app, session=session)
>>> manager.create_api(Person, primary_key='name')
>>> primary_key_for(Person)
'name'
>>> my_person = Person(name="Bob")
>>> primary_key_for(my_person)
'name'
```

This is in contrast to the typical default:

```
>>> manager = APIManager(app, session=session)
>>> manager.create_api(Person)
>>> primary_key_for(Person)
'id'
```

flask_restless.**url_for**(*model*, *instid=None*, *relationname=None*, *relationinstid=None*, *_apimanager=None*, ***kw*)

Returns the URL for the specified model, similar to flask.url_for().

*model* is a SQLAlchemy model class. This should be a model on which `APIManager.create_api_blueprint()` (or `APIManager.create_api()`) has been invoked previously. If no API has been created for it, this function raises a *ValueError*.

If *_apimanager* is not `None`, it must be an instance of `APIManager`. Restrict our search for endpoints exposing *model* to only endpoints created by the specified `APIManager` instance.

The *resource_id*, *relation_name*, and *relationresource_id* keyword arguments allow you to get the URL for a more specific sub-resource.

For example, suppose you have a model class `Person` and have created the appropriate Flask application and SQLAlchemy session:

```
>>> manager = APIManager(app, session=session)
>>> manager.create_api(Person, collection_name='people')
>>> url_for(Person, resource_id=3)
'http://example.com/api/people/3'
>>> url_for(Person, resource_id=3, relation_name=computers)
'http://example.com/api/people/3/computers'
>>> url_for(Person, resource_id=3, relation_name=computers,
...         related_resource_id=9)
'http://example.com/api/people/3/computers/9'
```

If a *resource_id* and a *relation_name* are provided, and you wish to determine the relationship endpoint URL instead of the related resource URL, set the *relationship* keyword argument to `True`:

```
>>> url_for(Person, resource_id=3, relation_name=computers,
...         relationship=True)
'http://example.com/api/people/3/relatonships/computers'
```

The remaining keyword arguments, *kw*, are passed directly on to `flask.url_for()`.

Since this function creates absolute URLs to resources linked to the given instance, it must be called within a Flask request context.

## Serialization and deserialization

**class** `flask_restless.`**`DefaultSerializer`**(*only=None*, *exclude=None*, *additional_attributes=None*, ***kw*)

A default implementation of a JSON API serializer for SQLAlchemy models.

The `serialize()` method of this class returns a complete JSON API document as a dictionary containing the resource object representation of the given instance of a SQLAlchemy model as its primary data. Similarly, the `serialize_many()` method returns a JSON API document containing a a list of resource objects as its primary data.

If *only* is a list, only these fields and relationships will in the returned dictionary. The only exception is that the keys `'id'` and `'type'` will always appear, regardless of whether they appear in *only*. These settings take higher priority than the *only* list provided to the `serialize()` or `serialize_many()` methods: if an attribute or relationship appears in the *only* argument to those method but not here in the constructor, it will not appear in the returned dictionary.

If *exclude* is a list, these fields and relationships will **not** appear in the returned dictionary.

If *additional_attributes* is a list, these attributes of the instance to be serialized will appear in the returned dictionary. This is useful if your model has an attribute that is not a SQLAlchemy column but you want it to be exposed.

You **must not** specify both *only* and *exclude* lists; if you do, the behavior of this function is undefined.

You **must not** specify a field in both *exclude* and in *additional_attributes*; if you do, the behavior of this function is undefined.

**serialize**(*instance*, *only=None*)

Returns a complete JSON API document as a dictionary containing the resource object representation of the given instance of a SQLAlchemy model as its primary data.

The returned dictionary is suitable as an argument to `flask.json.jsonify()`. Specifically, date and time objects (`datetime.date`, `datetime.time`, `datetime.datetime`, and `datetime.timedelta`) as well as `uuid.UUID` objects are converted to string representations, so no special JSON encoder behavior is required.

If *only* is a list, only the fields and relationships whose names appear as strings in *only* will appear in the resulting dictionary. This filter is applied *after* the default fields specified in the *only* keyword argument to the constructor of this class, so only fields that appear in both *only* keyword arguments will appear in the returned dictionary. The only exception is that the keys `'id'` and `'type'` will always appear, regardless of whether they appear in *only*.

Since this method creates absolute URLs to resources linked to the given instance, it must be called within a Flask request context.

**serialize_many**(*instances*, *only=None*)

Serializes each instance using its model-specific serializer.

This method works for heterogeneous collections of instances (that is, collections in which each instance is of a different type).

The *only* keyword argument must be a dictionary mapping resource type name to list of fields representing a sparse fieldset. The values in this dictionary must be valid values for the *only* keyword argument in the `DefaultSerializer.serialize()` method.

**class** flask_restless.**DefaultDeserializer**(*session*, *model*, *allow_client_generated_ids=False*, ***kw*)

A default implementation of a deserializer for SQLAlchemy models.

When called, this object returns an instance of a SQLAlchemy model with fields and relations specified by the provided dictionary.

**deserialize**(*document*)

Creates and returns a new instance of the SQLAlchemy model specified in the constructor whose attributes are given in the JSON API document.

*document* must be a dictionary representation of a JSON API document containing a single resource as primary data, as specified in the JSON API specification. For more information, see the Resource Objects section of the JSON API specification.

*Implementation note:* everything in the document other than the `data` element is ignored.

**class** flask_restless.**SerializationException**(*instance*, *message=None*, *resource=None*, **args*, ***kw*)

Raised when there is a problem serializing an instance of a SQLAlchemy model to a dictionary representation.

*instance* is the (problematic) instance on which `DefaultSerializer.serialize()` was invoked.

*message* is an optional string describing the problem in more detail.

*resource* is an optional partially-constructed serialized representation of `instance`.

Each of these keyword arguments is stored in a corresponding instance attribute so client code can access them.

**class** flask_restless.**DeserializationException**(*status=400*, *detail=None*, **args*, ***kw*)

Raised when there is a problem deserializing a Python dictionary to an instance of a SQLAlchemy model.

*status* is an integer representing the HTTP status code that corresponds to this error. If not specified, it is set to 400, representing 400 Bad Request.

*detail* is a string describing the problem in more detail. If provided, this will be incorporated in the return value of `message()`.

Each of the keyword arguments *status* and *detail* are assigned directly to instance-level attributes `status` and `detail`.

**detail = None**

A string describing the problem in more detail.

**message**()

Returns a more detailed description of the problem as a string.

**status = None**
The HTTP status code corresponding to this error.

flask_restless.**simple_serialize**(*self*, *instance*, *only=None*)
Provides basic, uncustomized serialization functionality as provided by the `DefaultSerializer.serialize()` method.

This function is suitable for calling on its own, no other instantiation or customization necessary.

flask_restless.**simple_serialize_many**(*self*, *instances*, *only=None*)
Provides basic, uncustomized serialization functionality as provided by the `DefaultSerializer.serialize_many()` method.

This function is suitable for calling on its own, no other instantiation or customization necessary.

**class** flask_restless.**MultipleExceptions**(*exceptions*, **args*, ***kw*)
Raised when there are multiple problems in serialization or deserialization.

*exceptions* is a non-empty sequence of other exceptions that have been raised in the code.

You may wish to raise this exception when implementing the `DefaultSerializer.serialize_many()` method, for example, if there are multiple exceptions

## Pre- and postprocessor helpers

**class** flask_restless.**ProcessingException**(*id_=None*, *links=None*, *status=400*, *code=None*, *title=None*, *detail=None*, *source=None*, *meta=None*, **args*, ***kw*)
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.

The keyword arguments `id_`, `href` `status`, `code`, `title`, `detail`, `links`, `paths` correspond to the elements of the JSON API error object; the values of these keyword arguments will appear in the error object returned to the client.

Any additional positional or keyword arguments are supplied directly to the superclass, `werkzeug.exceptions.HTTPException`.

CHAPTER 3

# Additional information

Meta-information on Flask-Restless.

## 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
- Django REST framework
- Restless

## Copyright and license

- 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 4.0 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 4.0 license.

# Changelog

Here you can see the full list of changes between each Flask-Restless release. Version 1.0.0 saw a major overhaul of Flask-Restless to make it compliant with JSON API, so changes from prior versions may not be relevant to more recent versions.

Numbers following a pound sign (#) refer to GitHub issues.

## Version 1.0.0b2-dev

*This is a beta release; these changes will appear in the 1.0.0 release.*

Not yet released.

- Eliminates all documentation build warnings for bad references.
- Changes serialization/deserialization to class-based implementation instead of a function-based implementation. This also adds support for serialization of heterogeneous collections.
- Removes mimerender as a dependency.
- #7: allows filtering before function evaluation.
- #49: deserializers now expect a complete JSON API document.
- #200: be smarter about determining the `collection_name` for polymorphic models defined with single-table inheritance.
- #253: don't assign to callable attributes of models.
- #268: adds top-level endpoint that exposes API schema.
- #479: removes duplicate (and sometimes conflicting) Content-Type header in responses.

- #481#488: added negation (`not`) operator for search.
- #492: support JSON API recommended "simple" filtering.
- #508: flush the session before postprocessors, and commit after.
- #534: when updating a resource, give a clearer error message if the resource ID in the JSON document is not a JSON string.
- #536: adds support for single-table inheritance.
- #540: correctly support models that don't have a column named "id".
- #545: refactors implementation of `DefaultDeserializer` so that it is easier for subclasses to override different subprocedures.
- #546: adds support for joined table inheritance.
- #548: requests can now use the `Accept: */*` header.
- #559: fixes bug that stripped attributes with JSON API reserved names (like "type") when deserializing resources.
- #583: fixes failing tests when using simplejson.
- #590: allows user to specify a custom operator for filters.
- #599: fixes *unicode* bug using `urlparse.urljoin()` with the future library in resource serialization.
- #625: adds schema metadata to root endpoint.
- #626: allows the client to request case-insensitive sorting.

## Version 1.0.0b1

*This is a beta release; these changes will appear in the 1.0.0 release.*

Released on April 2, 2016.

- #255: adds support for filtering by PostgreSQL network operators.
- #257: ensures additional attributes specified by the user actually exist on the model.
- #363 (partial solution): don't use `COUNT` on requests that don't require pagination.
- #404: **Major overhaul of Flask-Restless to support JSON API**.
- Increases minimum version requirement for `python-dateutil` to be strictly greater than 2.2 to avoid parsing bug.
- #331#415: documents the importance of URL encoding when using the `like` operator to filter results.
- #376: add a `not_like` operator for filter objects.
- #431: adds a `url_prefix` keyword argument to the `APIManager` constructor, so one can specify a URL prefix once for all created APIs.

- #449: roll back the session on any SQLAlchemy error, not just a few.
- #432#462: alias relation names when sorting by multiple attributes on a relationship.
- #436#453: use `__table__.name` instead of `__tablename__` to infer the collection name for the SQLAlchemy model.
- #440#475: uses the serialization function provided at the time of invoking `APIManager.create_api()` to serialize each resource correctly, depending on its type.
- #474: include license files in built wheel for distribution.
- #501: allows empty string for *url_prefix* keyword argument to `APIManager.create_api()`.
- #476: use the primary key provided at the time of invoking `APIManager.create_api()` to build resource urls in responses.

## Older versions

---

**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.

---

Version 0.17.0

Released on February 17, 2015.

- Corrects bug to allow delayed initialization of multiple Flask applications.
- #167: allows custom serialization/deserialization functions.
- #198: allows arbitrary Boolean expressions in search query filters.
- #226: allows creating APIs before initializing the Flask application object.
- #274: adds the `url_for()` function for computing URLs from models.
- #379: improves datetime parsing in search requests.
- #398: fixes bug where DELETE_SINGLE processors were not actually used.
- #400: disallows excluding a primary key on a POST request.

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.

Version 0.15.1

Released on January 2, 2015.

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

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*)

Version 0.14.2

Released on September 2, 2014.

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

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.

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 `flask_restless.views.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.

Version 0.13.1

Released on April 21, 2014.

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

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.types.Time` fields in JSON responses.

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.

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.

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.

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.

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 :issue:'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#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.

Version 0.9.3

Released on February 4, 2013.

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

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.

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.

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.

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.

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.

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.

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 `APIManager.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.

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 `APIManager.create_api()`.
- Fixed bug where we weren't allowing PUT requests in `APIManager.create_api()`.
- Added `collection_name` keyword argument to `APIManager.create_api()` to allow user provided names in URLs.
- Added `allow_patch_many` keyword argument to `APIManager.create_api()` to allow enabling or disabling the PATCH many functionality.
- Disable the PATCH many functionality by default.

Version 0.3

Released on March 4, 2012.

- Initial release in Flask extension format.

# Index