Flask-Restless Documentation

Jeffrey Finkelstein

Oct 14, 2023

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.

CHAPTER

ONE

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.

1.1 Downloading and installing Flask-Restless-NG

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-NG
```

Flask-Restless supports Python 3.6+

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

- Flask version 1.0 or greater
- SQLAlchemy version 1.3 or greater
- python-dateutil version strictly greater than 2.2
- Flask-SQLAlchemy, *only if* you want to define your models using Flask-SQLAlchemy

1.2 Quickstart

For the restless:

```
import flask
import flask_restless
import flask_sqlalchemy

```

(continues on next page)

(continued from previous page)

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


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'))


# Create the database tables.
db.create_all()

# Create the Flask-Restless API manager.
manager = flask_restless.APIManager(app, session=db.session)

# 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

1.3 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 named `id` of type `sqlalchemy.Integer` or type `sqlalchemy.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
```

(continues on next page)

(continued from previous page)

```
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, session=db.session)
```

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)
```

(continues on next page)

(continued from previous page)

```
>>> 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`.

1.3.1 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.

1.4 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.

1.4.1 Fetching resources and 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)

class Article(db.Model):
```

(continues on next page)

(continued from previous page)

```
    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, session=db.session)
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": [
    {
      "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",
```

(continues on next page)

(continued from previous page)

```
    "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": {
    "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": {
    "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": [
    {
      "id": "2",
      "links": {
        "self": "http://example.com/api/articles/2"
      },
      "relationships": {
        "author": {
```

(continues on next page)

(continued from previous page)

```
          "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": {
    "id": "2",
    "links": {
      "self": "http://example.com/api/articles/2"
    },
    "relationships": {
```

(continues on next page)

(continued from previous page)

```
      "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": [
    {
```

(continues on next page)

(continued from previous page)

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

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"
```

(continues on next page)

(continued from previous page)


```
        }
      }
    },
    "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

{
```

(continues on next page)

(continued from previous page)

```
  "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.

If `sort` parameter is not specified, data is sorted using the primary key

Clients can disable default sorting by using `sort=0`

Pagination

Pagination works as described in the JSON API specification, via the `page[number]` and `page[size]` query parameters. Pagination respects sorting 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 anything but a positive integer, the client will be able to specify arbitrarily large page sizes. If, further, `page_size` is set to anything but a positive integer, 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
```

(continues on next page)

(continued from previous page)

```
  }
}
```

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") or disjunction ("or") of other filter objects:

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

or

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

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
  }
}
```

1.4.2 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.ext.sqlalchemy import SQLAlchemy
from flask.ext.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, session=db.session)
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"
    },
```

(continues on next page)

(continued from previous page)

```
    "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-
```

(continues on next page)

(continued from previous page)

```
↪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.

1.4.3 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.ext.sqlalchemy import SQLAlchemy
from flask.ext.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, session=db.session)
manager.create_api(Person, method=['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.

1.4.4 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.ext.sqlalchemy import SQLAlchemy
from flask.ext.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, session=db.sesion)
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.

1.4.5 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.ext.sqlalchemy import SQLAlchemy
from flask.ext.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)
```

(continues on next page)

(continued from previous page)

```
    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, session=db.session)
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
    }
  ]
```

(continues on next page)

(continued from previous page)

```
}
```

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.

1.4.6 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.

1.4.7 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.

1.4.8 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.

1.4.9 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 certain types of errors, the SQLAlchemy session will be rolled back. Currently these errors are

- `DataError`,
- `IntegrityError`,
- `ProgrammingError`,
- `FlushError`.

1.4.10 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)
```

1.5 Customizing the ReSTful interface

1.5.1 HTTP methods

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

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

This creates an endpoint at `/api/person` which responds to GET, POST, and DELETE methods, but not to 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.

1.5.2 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`.

1.5.3 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!

1.5.4 Specifying one of many primary keys

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

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

If you do this, Flask-Restless will create URLs like `/api/user/myusername` instead of `/api/user/123`.

1.5.5 Enable bulk operations

Bulk operations via the JSON API Bulk extension are not yet supported.

1.5.6 Custom serialization

Flask-Restless-NG 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 a custom serialization class by providing it to `APIManager.create_api()` via the `serializer` keyword argument. Similarly, to provide a deserialization function that converts a Python dictionary representation to an instance of your model, use the `deserializer` 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!

Define your serialization functions like this:

```
from flask_restless import Serializer

class CustomSerializer(Serializer):
    def __init__(attributes, relationships):
        self.attributes = attributes
        self.relationships = relationships

    @property
    def attributes_columns(self):
        return set(self.attributes)

    @property
    def relationship_columns(self):
        return set(self.relationships)

    def serialize(self, instance, only=None):
        return {'id': instance.id, 'type': 'custom', 'attributes': {}}
```

`instance` is an instance of a SQLAlchemy model 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.

For deserialization, define your custom deserialization function like this:

```
from flask_restless import Deserializer

class DefaultDeserializer(Deserializer):
```

(continues on next page)

(continued from previous page)

```
    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. The function must return an instance of the model that has the requested fields.

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)

person_schema = PersonSchema()

class PersonSerializer(Serializer):
    @property
    def attributes_columns(self):
        return {'name'}

    @property
    def relationship_columns(self):
        return set()

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

class PersonDeserializer(Deserializer):
    def deserialize(self, document):
        return person_schema.load(data).data


person_serializer = PersonSerializer()
person_deserializer = PersonDeserializer()

manager = APIManager(app, session=session)
manager.create_api(Person, methods=['GET', 'POST'],
                   serializer=person_serializer,
                   deserializer=person_deserializer)
```

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-NG will use the `article_serializer` instance 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.

1.5.7 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-NG 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": [
```

(continues on next page)

(continued from previous page)

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

1.5.8 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
GET_RESOURCE	resource_id
GET_RELATION	resource_id, relation_name, filters, sort
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
GET_RESOURCE	result
GET_TO_MANY_RELATION	result, filters, sort
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
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, **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` keyword arguments, see *Filtering*. For more information about `sort` keyword argument, 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_RESOURCE=[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.ext.login import current_user
from mymodels import User
from mymodels import session
```

(continues on next page)

(continued from previous page)

```
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)
```

1.5.9 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".

Requiring authentication for some methods

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

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

def auth_func(*args, **kwargs):
    if not current_user.is_authenticated():
        raise ProcessingException(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.

CHAPTER

TWO

API REFERENCE

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

2.1 API

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

2.1.1 The API Manager class

class flask_restless.**APIManager**(*app=None*, *session=None*, *preprocessors=None*, *postprocessors=None*, *url_prefix='/api'*, *include_links: bool = False*)

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.

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

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

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

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*.

include_links controls whether to include link objects in resource objects https://jsonapi.org/format/#document-links

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.
api_manager = APIManager(session=mysession)
api_manager.create_api(User)
api_manager.create_api(Comment)

# Later, call `init_app` to register the blueprints for the
# APIs created earlier.
```

(continues on next page)

(continued from previous page)

```
app = Flask(__name__)
api_manager.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.
api_manager = APIManager(session=db.session)
api_manager.create_api(User)
api_manager.create_api(Comment)

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

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: str*, *model*, *methods=frozenset({'GET'})*, *url_prefix: Optional[str] = None*, *collection_name: Optional[str] = None*, *only=None*, *exclude=None*, *additional_attributes=None*, *validation_exceptions=None*, *page_size: int = 10*, *max_page_size: int = 100*, *preprocessors=None*, *postprocessors=None*, *primary_key: Optional[str] = None*, *serializer: Optional[*Serializer*] = None*, *deserializer: Optional[*Deserializer*] = None*, *includes=None*, *allow_to_many_replacement: bool = False*, *allow_delete_from_to_many_relationships: bool = False*, *allow_client_generated_ids: bool = 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 *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 allowlist 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 an blocklist 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 captured 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 and *deserializer* are custom serialization classes. 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 corre-

sponding 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*.

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 functionevaluation. This is `False` by default.

Warning: This is deprecated and going to be removed in the next major version

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

2.1.2 Serialization helpers

class flask_restless.**Serializer**

An object that, when called, returns a dictionary representation of a given instance of a SQLAlchemy model.

class flask_restless.**Deserializer**(*session*, *model*, *api_manager*)

An object that, when called, returns an instance of the SQLAlchemy model specified at instantiation time.

session is the SQLAlchemy session in which to look for any related resources.

model is the class of which instances will be created by the `__call__()` method.

This is a base class with no implementation.

class flask_restless.**SerializationException**(*instance*, *message=None*, *resource=None*, *resource_type=None*, *resource_id=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 `Serializer.__call__()` 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**(**args*, ***kw*)

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

Subclasses that wish to provide more detailed about the problem should set the `detail` attribute to be a string, either as a class-level attribute or as an instance attribute.

2.1.3 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

THREE

ADDITIONAL INFORMATION

Meta-information on Flask-Restless.

3.1 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

3.2 Copyright and license

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.

3.3 Changelog

3.4 Changes in Flask-Restless-NG

3.4.1 Unreleased

- Dropped savalidation support

3.4.2 Version 3.1.0 (2023-10-14):

- Added support for Flask 3.0
- Added support for Python 3.11
- Added support for Python 3.12
- Dropped support for Python 3.7

3.4.3 Version 3.0.0 (2023-03-19):

- Added support for SQLAlchemy 2.0
- Minimum required SQLAlchemy version: 1.4.18
- Minimum required Flask version: 2.2
- Drop Functions API support

3.4.4 Version 2.5.1 (2023-03-15):

- Restricted SQLAlchemy to <2.0. Support of 2.0 requires significant changes and will be a major release

3.4.5 Version 2.5.0 (2022-12-24):

- Added support for X-Forwarded- headers (pagination links now use the original host/proto) (#38)

3.4.6 Version 2.4.0 (2022-11-11):

- Clients can disable default sorting by using *sort=0* query parameter

3.4.7 Version 2.3.1:

- Fix for an incorrect error message
- Returns 500 instead 400 response code in case of serialization errors in POST requests
- Fix for pagination Flask-SQLAlchemy (now works also with 3.0+) (#37)

3.4.8 Version 2.3.0

- Allow sorting of nested fields

3.4.9 Version 2.2.9

- Do not erase *type* field from *attributes* (#31)

3.4.10 Version 2.2.8

- Make sure that POST response contains actual values from the DB

3.4.11 Version 2.2.7

- Fix Server Error for null relationship in POST (#29)
- Allow session rollback in PATCH_RESOURCE, PATCH_RELATIONSHIP, POST_RELATIONSHIP post-processors (#28)

3.4.12 Version 2.2.6

- Escape user input in error messages to prevent potential server-side cross-site scripting
- 'status' field in error objects is now a string, as required by JSON API standard https://jsonapi.org/format/#error-objects
- Returns '400' status if page number is provided but page size is 0
- Returns '400' status if unknown field was used for sorting
- Allow rolling back the current session in *POST_RESOURCE* postprocessors (#28)

3.4.13 Version 2.2.5

- Fix for #27 'relationship with secondary generates incorrect query'

3.4.14 Version 2.2.4

- Do not log exceptions for user related errors (bad query, etc)
- Update safe check for *selectinload* for `includes
- Update SQLAlchemy dependency to 1.3.6+

3.4.15 Version 2.2.3

- Add safe check for *selectinload* for *includes*

3.4.16 Version 2.2.2

- Fix an incorrect selectinload query for models with custom select queries

3.4.17 Version 2.2.1

- Minor improvements and fixes

3.4.18 Version 2.2.0

- Serialize To-One relationships using foreign key, instead of trying to fetch the whole

relationship object from the database

3.4.19 Version 2.1.1

- Only fetch primary keys from a database for relationships when no filtering is required

3.4.20 Version 2.1.0

- Re-added FunctionsAPI until the next major release to let users to implement an alternative #23

3.4.21 Version 2.0.3

- **Fix: #26 - selectinload is broken for models that have primary keys other than 'id'. Disabled until a new schema is**
  implemented
- Make 'primary_key' optional again #25 (by @tanj)

3.4.22 Version 2.0.2

- Fixed import for SQLAlchemy 1.3 #22

3.4.23 Version 2.0.0

Refactored fetching resource collections: - SQL query optimizations for 'include' and 'relationship' objects, using *selectinload*

(3x-5x performance improvement when tested on large datasets)

- New parameter 'include_links' which controls should relationship objects include links. They are not required by JSON API, and disabling them significantly improves performance
- New interfaces for Serializer and Deserializer classes.
- APIManager requires Serializer/Deserializer objects instead of functions for *serializer/deserializer* options

Deprecations: - 'single' parameter is no longer supported - makes code complicated, is not defined in JSON API specs and can be easily

replicated on a client side

- 'group' parameter is not longer supported - not defined in JSON API specifications, confusing and broken for PostrgeSQL
- JSONP callbacks are no longer supported - please reach out if you have a use case for them

3.4.24 Version 1.0.6

- Prevent redundant SQL queries during pagination and resource inclusion

3.4.25 Version 1.0.5

- #16 - Fix: including child of empty relationship (by @sharky98)

3.4.26 Version 1.0.4

- #15: Support SQLAlchemy 1.4.x

3.4.27 Version 1.0.2

- #1, #13: Fix for relationship updates not being committed (by @sharky98)
- #12: Fix for 500 when trying to include Null/None relationship
- Added TSQuery operator (by @augustin)

3.4.28 Version 1.0.1

- #4: *id* is an optional attribute as long as Model has a primary key
- #6: Fix for *flask_restless.views* not being included in the installed package.

3.4.29 Version 1.0.0

- Performance improvement: url_for() changed to build url locally instead of delegating it to Flask
- This is the last release that is backward compatible with the original Flask-Restless API.

3.4.30 Version 0.0.2

- New serializer (2-3x faster)
- Added lru_cache to helpers to reduce number of recursive calls (better performance)

3.4.31 Version 0.0.1

- Fixed 1.0+ compatibility
- Fix for hybrid_property

3.5 Original Flask-Restless

You can find the full changelog in the original repo

INDEX