flask-rest-jsonapi Documentation

Release 0.1

miLibris

Jun 04, 2019

Contents

1 Main concepts **3**

2 Features **5**

3 User's Guide **7**

3.1 Installation . . . 7
3.2 Quickstart . . . 7
3.2.1 First example . . . 8
3.2.2 Classical CRUD operations . . . 12
3.2.3 Relationships . . . 15
3.3 Logical data abstraction . . . 21
3.4 Resource Manager . . . 23
3.4.1 Required attributes . . . 23
3.4.2 Optional attributes . . . 24
3.4.3 ResourceList . . . 25
3.4.4 ResourceDetail . . . 25
3.4.5 ResourceRelationship . . . 26
3.5 Data layer . . . 26
3.5.1 SQLAlchemy . . . 28
3.5.2 Custom data layer . . . 28
3.6 Routing . . . 28
3.7 Filtering . . . 29
3.7.1 SQLAlchemy . . . 29
3.7.2 Simple filters . . . 31
3.8 Include related objects . . . 32
3.9 Sparse fieldsets . . . 35
3.10 Pagination . . . 35
3.10.1 Size . . . 35
3.10.2 Number . . . 36
3.10.3 Size + Number . . . 36
3.10.4 Disable pagination . . . 36
3.11 Sorting . . . 36
3.11.1 Multiple sort . . . 36
3.11.2 Descending sort . . . 37
3.11.3 Multiple sort + Descending sort . . . 37
3.12 Errors . . . 37
3.13 Api . . . 38

3.14 Permission 39
3.15 OAuth 40
3.16 Configuration 41

4 API Reference **43**

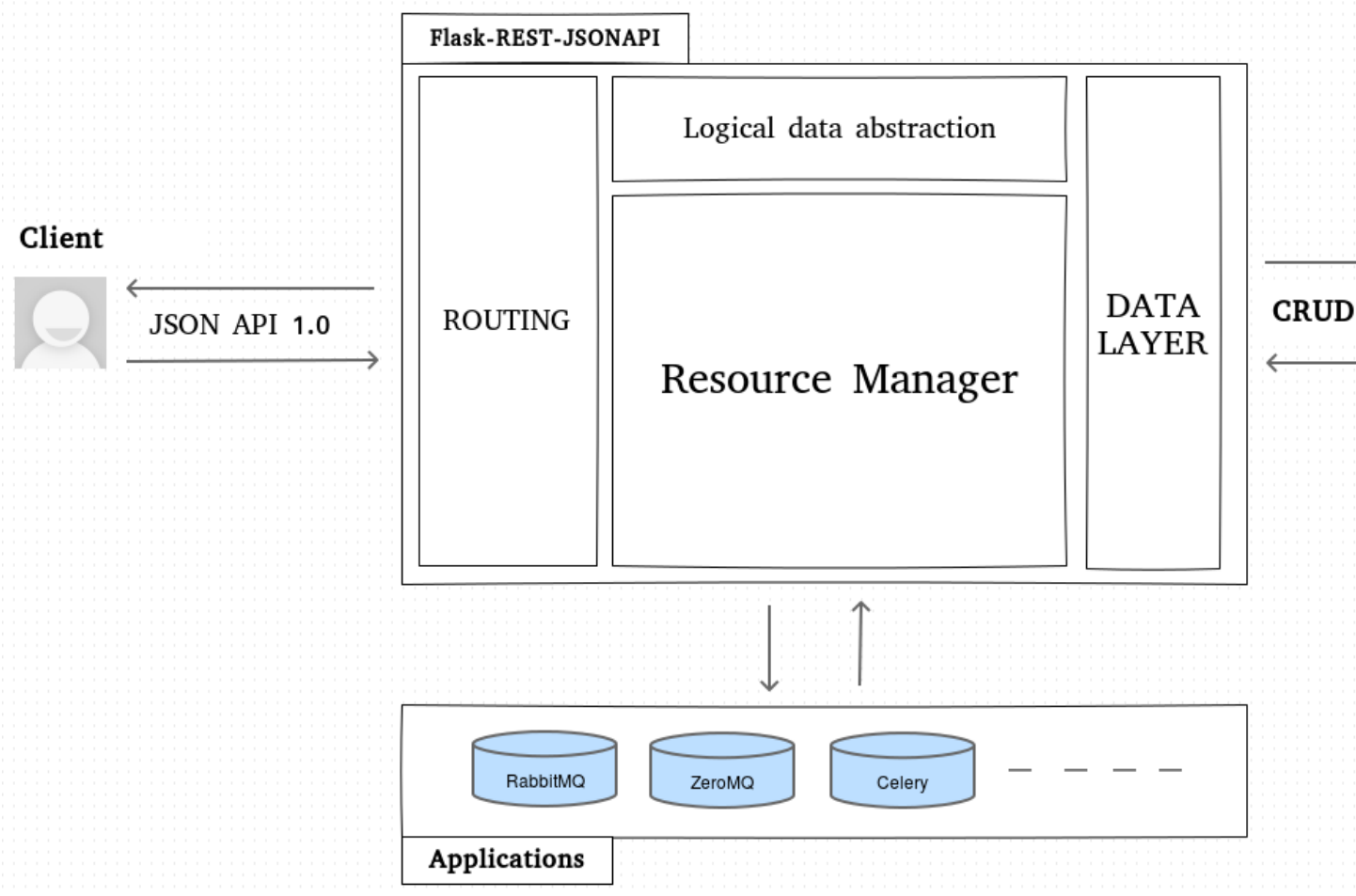
Python Module Index **45**

Index **47**

Flask-REST-JSONAPI is an extension for Flask that adds support for quickly building REST APIs with huge flexibility around the JSONAPI 1.0 specification. It is designed to fit the complexity of real life environments so Flask-REST-JSONAPI helps you to create a logical abstraction of your data called "resource" and can interface any kind of ORMs or data storage through data layer concept.

CHAPTER 1

Main concepts



* JSON API 1.0 specification: it is a very popular specification about client server interactions for REST JSON API. It helps you work in a team because it is very precise and sharable. Thanks to this specification your API offers lots of features like a strong structure of request and response, filtering, pagination, sparse fieldsets, including related objects, great error formatting etc.

* **Logical data abstraction**: you usually need to expose resources to clients that don't fit your data table architecture. For example sometimes you don't want to expose all attributes of a table, compute additional attributes or create a resource that uses data from multiple data storages. Flask-REST-JSONAPI helps you create a logical abstraction of your data with Marshmallow / marshmallow-jsonapi so you can expose your data through a very flexible way.

* **Data layer**: the data layer is a CRUD interface between your resource manager and your data. Thanks to it you can use any data storage or ORMs. There is an already full featured data layer that uses the SQLAlchemy ORM but you can create and use your own custom data layer to use data from your data storage(s). You can even create a data layer that uses multiple data storages and ORMs, send notifications or make any custom work during CRUD operations.

CHAPTER 2

Features

Flask-REST-JSONAPI has lots of features:

- Relationship management
- Powerful filtering
- Include related objects
- Sparse fieldsets
- Pagination
- Sorting
- Permission management
- OAuth support

CHAPTER 3

User's Guide

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

3.1 Installation

Install Flask-REST-JSONAPI with `pip`

```
pip install flask-rest-jsonapi
```

The development version can be downloaded from its page at GitHub.

```
git clone https://github.com/miLibris/flask-rest-jsonapi.git
cd flask-rest-jsonapi
mkvirtualenv venv
python setup.py install
```

Note: If you don't have virtualenv please install it before

$ pip install virtualenv

Flask-RESTful requires Python version 2.7, 3.4 or 3.5.

3.2 Quickstart

It's time to write your first REST API. This guide assumes you have a working understanding of Flask, and that you have already installed both Flask and Flask-REST-JSONAPI. If not, then follow the steps in the *Installation* section.

In this section you will learn basic usage of Flask-REST-JSONAPI around a small tutorial that use the SQLAlchemy data layer. This tutorial show you an example of a person and his computers.

3.2.1 First example

An example of Flask-REST-JSONAPI API looks like this:

```
# -*- coding: utf-8 -*-

from flask import Flask
from flask_rest_jsonapi import Api, ResourceDetail, ResourceList, ResourceRelationship
from flask_rest_jsonapi.exceptions import ObjectNotFound
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.orm.exc import NoResultFound
from marshmallow_jsonapi.flask import Schema, Relationship
from marshmallow_jsonapi import fields

# Create the Flask application
app = Flask(__name__)
app.config['DEBUG'] = True


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


# Create data storage
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    email = db.Column(db.String)
    birth_date = db.Column(db.Date)
    password = db.Column(db.String)


class Computer(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    serial = db.Column(db.String)
    person_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    person = db.relationship('Person', backref=db.backref('computers'))

db.create_all()


# Create logical data abstraction (same as data storage for this first example)
class PersonSchema(Schema):
    class Meta:
        type_ = 'person'
        self_view = 'person_detail'
        self_view_kwargs = {'id': '<id>'}
        self_view_many = 'person_list'

    id = fields.Integer(as_string=True, dump_only=True)
    name = fields.Str(required=True, load_only=True)
    email = fields.Email(load_only=True)
    birth_date = fields.Date()
    display_name = fields.Function(lambda obj: "{} <{}>".format(obj.name.upper(), obj.
↪email))
    computers = Relationship(self_view='person_computers',
                             self_view_kwargs={'id': '<id>'},
```

(continues on next page)

(continued from previous page)

```
                              related_view='computer_list',
                              related_view_kwargs={'id': '<id>'},
                              many=True,
                              schema='ComputerSchema',
                              type_='computer')


class ComputerSchema(Schema):
    class Meta:
        type_ = 'computer'
        self_view = 'computer_detail'
        self_view_kwargs = {'id': '<id>'}

    id = fields.Integer(as_string=True, dump_only=True)
    serial = fields.Str(required=True)
    owner = Relationship(attribute='person',
                         self_view='computer_person',
                         self_view_kwargs={'id': '<id>'},
                         related_view='person_detail',
                         related_view_kwargs={'computer_id': '<id>'},
                         schema='PersonSchema',
                         type_='person')


# Create resource managers
class PersonList(ResourceList):
    schema = PersonSchema
    data_layer = {'session': db.session,
                  'model': Person}


class PersonDetail(ResourceDetail):
    def before_get_object(self, view_kwargs):
        if view_kwargs.get('computer_id') is not None:
            try:
                computer = self.session.query(Computer).filter_by(id=view_kwargs[
↪'computer_id']).one()
            except NoResultFound:
                raise ObjectNotFound({'parameter': 'computer_id'},
                                     "Computer: {} not found".format(view_kwargs[
↪'computer_id']))
            else:
                if computer.person is not None:
                    view_kwargs['id'] = computer.person.id
                else:
                    view_kwargs['id'] = None

    schema = PersonSchema
    data_layer = {'session': db.session,
                  'model': Person,
                  'methods': {'before_get_object': before_get_object}}


class PersonRelationship(ResourceRelationship):
    schema = PersonSchema
    data_layer = {'session': db.session,
                  'model': Person}
```

(continues on next page)

(continued from previous page)

```
class ComputerList(ResourceList):
    def query(self, view_kwargs):
        query_ = self.session.query(Computer)
        if view_kwargs.get('id') is not None:
            try:
                self.session.query(Person).filter_by(id=view_kwargs['id']).one()
            except NoResultFound:
                raise ObjectNotFound({'parameter': 'id'}, "Person: {} not found".
↪format(view_kwargs['id']))
            else:
                query_ = query_.join(Person).filter(Person.id == view_kwargs['id'])
        return query_

    def before_create_object(self, data, view_kwargs):
        if view_kwargs.get('id') is not None:
            person = self.session.query(Person).filter_by(id=view_kwargs['id']).one()
            data['person_id'] = person.id

    schema = ComputerSchema
    data_layer = {'session': db.session,
                  'model': Computer,
                  'methods': {'query': query,
                              'before_create_object': before_create_object}}


class ComputerDetail(ResourceDetail):
    schema = ComputerSchema
    data_layer = {'session': db.session,
                  'model': Computer}


class ComputerRelationship(ResourceRelationship):
    schema = ComputerSchema
    data_layer = {'session': db.session,
                  'model': Computer}


# Create endpoints
api = Api(app)
api.route(PersonList, 'person_list', '/persons')
api.route(PersonDetail, 'person_detail', '/persons/<int:id>', '/computers/
↪<int:computer_id>/owner')
api.route(PersonRelationship, 'person_computers', '/persons/<int:id>/relationships/
↪computers')
api.route(ComputerList, 'computer_list', '/computers', '/persons/<int:id>/computers')
api.route(ComputerDetail, 'computer_detail', '/computers/<int:id>')
api.route(ComputerRelationship, 'computer_person', '/computers/<int:id>/relationships/
↪owner')

if __name__ == '__main__':
    # Start application
    app.run(debug=True)
```

This example provides this api:

url	method	endpoint	action
/persons	GET	person_list	Retrieve a collection of persons
/persons	POST	person_list	Create a person
/persons/<int:id>	GET	person_detail	Retrieve details of a person
/persons/<int:id>	PATCH	person_detail	Update a person
/persons/<int:id>	DELETE	person_detail	Delete a person
/persons/<int:id>/computers	GET	computer_list	Retrieve a collection computers related to a person
/persons/<int:id>/computers	POST	computer_list	Create a computer related to a person
/persons/<int:id>/relationship/computers	GET	person_computers	Retrieve relationships between a person and computers
/persons/<int:id>/relationship/computers	POST	person_computers	Create relationships between a person and computers
/persons/<int:id>/relationship/computers	PATCH	person_computers	Update relationships between a person and computers
/persons/<int:id>/relationship/computers	DELETE	person_computers	Delete relationships between a person and computers
/computers	GET	computer_list	Retrieve a collection of computers
/computers	POST	computer_list	Create a computer
/computers/<int:id>	GET	computer_detail	Retrieve details of a computer
/computers/<int:id>	PATCH	computer_detail	Update a computer
/computers/<int:id>	DELETE	computer_detail	Delete a computer
/computers/<int:id>/owner	GET	person_detail	Retrieve details of the owner of a computer
/computers/<int:id>/owner	PATCH	person_detail	Update the owner of a computer
/computers/<int:id>/owner	DELETE	person_detail	Delete the owner of a computer
/computers/<int:id>/relationship/owner	GET	person_computers	Retrieve relationships between a person and computers
/computers/<int:id>/relationship/owner	POST	person_computers	Create relationships between a person and computers
/computers/<int:id>/relationship/owner	PATCH	person_computers	Update relationships between a person and computers
/computers/<int:id>/relationship/owner	DELETE	person_computers	Delete relationships between a person and computers

Warning: In this example, I use Flask-SQLAlchemy so you have to install it before to run the example.

$ pip install flask_sqlalchemy

Save this as api.py and run it using your Python interpreter. Note that we've enabled Flask debugging mode to provide code reloading and better error messages.

```
$ python api.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

Warning: Debug mode should never be used in a production environment!

3.2.2 Classical CRUD operations

Create object

Request:

```
POST /computers HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "computer",
    "attributes": {
      "serial": "Amstrad"
    }
  }
}
```

Response:

```
HTTP/1.1 201 Created
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "computer",
    "id": "1",
    "attributes": {
      "serial": "Amstrad"
    },
    "relationships": {
      "owner": {
        "links": {
          "related": "/computers/1/owner",
          "self": "/computers/1/relationships/owner"
        }
      }
    },
    "links": {
      "self": "/computers/1"
    }
  },
  "links": {
    "self": "/computers/1"
  },
  "jsonapi": {
    "version": "1.0"
  }
}
```

List objects

Request:

```
GET /computers HTTP/1.1
Accept: application/vnd.api+json
```

Response:

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

{
  "data": [
    {
      "type": "computer",
      "id": "1",
      "attributes": {
        "serial": "Amstrad"
      },
      "relationships": {
        "owner": {
          "links": {
            "related": "/computers/1/owner",
            "self": "/computers/1/relationships/owner"
          }
        }
      },
      "links": {
        "self": "/computers/1"
      }
    }
  ],
  "meta": {
    "count": 1
  },
  "links": {
    "self": "/computers"
  },
  "jsonapi": {
    "version": "1.0"
  },
}
```

Update object

Request:

```
PATCH /computers/1 HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "computer",
    "id": "1",
    "attributes": {
      "serial": "Amstrad 2"
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Response:

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

{
  "data": {
    "type": "computer",
    "id": "1",
    "attributes": {
      "serial": "Amstrad 2"
    },
    "relationships": {
      "owner": {
        "links": {
          "related": "/computers/1/owner",
          "self": "/computers/1/relationships/owner"
        }
      }
    },
    "links": {
      "self": "/computers/1"
    }
  },
  "links": {
    "self": "/computers/1"
  },
  "jsonapi": {
    "version": "1.0"
  }
}
```

Delete object

Request:

```
DELETE /computers/1 HTTP/1.1
Accept: application/vnd.api+json
```

Response:

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

{
  "meta": {
    "message": "Object successfully deleted"
  },
  "jsonapi": {
    "version": "1.0"
  }
}
```

3.2.3 Relationships

Now let's use relationships tools. First, create 3 computers named Halo, Nestor and Comodor like in previous example.

Done ?
Ok. So let's continue this tutorial.

We assume that Halo has id: 2, Nestor id: 3 and Comodor has id: 4.

Create object with related object(s)

Request:

```
POST /persons?include=computers HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "person",
    "attributes": {
      "name": "John",
      "email": "john@gmail.com",
      "birth_date": "1990-12-18"
    },
    "relationships": {
      "computers": {
        "data": [
          {
            "type": "computer",
            "id": "1"
          }
        ]
      }
    }
  }
}
```

Response:

```
HTTP/1.1 201 Created
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "person",
    "id": "1",
    "attributes": {
      "display_name": "JOHN <john@gmail.com>",
      "birth_date": "1990-12-18"
    },
    "links": {
      "self": "/persons/1"
    },
```

(continues on next page)

(continued from previous page)


```
    "relationships": {
      "computers": {
        "data": [
          {
            "id": "1",
            "type": "computer"
          }
        ],
        "links": {
          "related": "/persons/1/computers",
          "self": "/persons/1/relationships/computers"
        }
      }
    },
  },
  "included": [
    {
      "type": "computer",
      "id": "1",
      "attributes": {
        "serial": "Amstrad"
      },
      "links": {
        "self": "/computers/1"
      },
      "relationships": {
        "owner": {
          "links": {
            "related": "/computers/1/owner",
            "self": "/computers/1/relationships/owner"
          }
        }
      }
    }
  ],
  "jsonapi": {
    "version": "1.0"
  },
  "links": {
    "self": "/persons/1"
  }
}
```


You can see that I have added the querystring parameter "include" to the url

```
POST /persons?include=computers HTTP/1.1
```

Thanks to this parameter, related computers details are included to the result. If you want to learn more: *Include related objects*

Update object and his relationships

Now John sell his Amstrad and buy a new computer named Nestor (id: 3). So we want to link this new computer to John. John have also made a mistake in his birth_date so let's update this 2 things in the same time.

Request:

```
PATCH /persons/1?include=computers HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "person",
    "id": "1",
    "attributes": {
      "birth_date": "1990-10-18"
    },
    "relationships": {
      "computers": {
        "data": [
          {
            "type": "computer",
            "id": "3"
          }
        ]
      }
    }
  }
}
```

Response:

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

{
  "data": {
    "type": "person",
    "id": "1",
    "attributes": {
      "display_name": "JOHN <john@gmail.com>",
      "birth_date": "1990-10-18",
    },
    "links": {
      "self": "/persons/1"
    },
    "relationships": {
      "computers": {
        "data": [
          {
            "id": "3",
            "type": "computer"
          }
        ],
        "links": {
          "related": "/persons/1/computers",
          "self": "/persons/1/relationships/computers"
        }
      }
    },
  },
  "included": [
    {
```

(continues on next page)

(continued from previous page)

```
      "type": "computer",
      "id": "3",
      "attributes": {
        "serial": "Nestor"
      },
      "relationships": {
        "owner": {
          "links": {
            "related": "/computers/3/owner",
            "self": "/computers/3/relationships/owner"
          }
        }
      },
      "links": {
        "self": "/computers/3"
      }
    }
  ],
  "links": {
    "self": "/persons/1"
  },
  "jsonapi": {
    "version": "1.0"
  }
}
```

Create relationship

Now John buy a new computer named Comodor so let's link it to John.

Request:

```
POST /persons/1/relationships/computers HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": [
    {
      "type": "computer",
      "id": "4"
    }
  ]
}
```

Response:

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

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

(continues on next page)

(continued from previous page)


```
      "display_name": "JOHN <john@gmail.com>",
      "birth_date": "1990-10-18"
    },
    "relationships": {
      "computers": {
        "data": [
          {
            "id": "3",
            "type": "computer"
          },
          {
            "id": "4",
            "type": "computer"
          }
        ],
        "links": {
          "related": "/persons/1/computers",
          "self": "/persons/1/relationships/computers"
        }
      }
    },
    "links": {
      "self": "/persons/1"
    }
  },
  "included": [
    {
      "type": "computer",
      "id": "3",
      "attributes": {
        "serial": "Nestor"
      },
      "relationships": {
        "owner": {
          "links": {
            "related": "/computers/3/owner",
            "self": "/computers/3/relationships/owner"
          }
        }
      },
      "links": {
        "self": "/computers/3"
      }
    },
    {
      "type": "computer",
      "id": "4",
      "attributes": {
        "serial": "Comodor"
      },
      "relationships": {
        "owner": {
          "links": {
            "related": "/computers/4/owner",
            "self": "/computers/4/relationships/owner"
          }
        }
```


(continues on next page)

(continued from previous page)

```
      },
      "links": {
        "self": "/computers/4"
      }
    }
  ],
  "links": {
    "self": "/persons/1/relationships/computers"
  },
  "jsonapi": {
    "version": "1.0"
  }
}
```

Delete relationship

Now John sell his old Nestor computer so let's unlink it from John.

Request:

```
DELETE /persons/1/relationships/computers HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": [
    {
      "type": "computer",
      "id": "3"
    }
  ]
}
```

Response:

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

{
  "data": {
    "type": "person",
    "id": "1",
    "attributes": {
      "display_name": "JOHN <john@gmail.com>",
      "birth_date": "1990-10-18"
    },
    "relationships": {
      "computers": {
        "data": [
          {
            "id": "4",
            "type": "computer"
          }
        ],
        "links": {
```

(continues on next page)

(continued from previous page)

```
          "related": "/persons/1/computers",
          "self": "/persons/1/relationships/computers"
        }
      }
    },
    "links": {
      "self": "/persons/1"
    }
  },
  "included": [
    {
      "type": "computer",
      "id": "4",
      "attributes": {
        "serial": "Comodor"
      },
      "relationships": {
        "owner": {
          "links": {
            "related": "/computers/4/owner",
            "self": "/computers/4/relationships/owner"
          }
        }
      },
      "links": {
        "self": "/computers/4"
      }
    }
  ],
  "links": {
      "self": "/persons/1/relationships/computers"
  },
  "jsonapi": {
      "version": "1.0"
  }
}
```

If you want to see more examples go to JSON API 1.0 specification

3.3 Logical data abstraction

The first thing to do in Flask-REST-JSONAPI is to create a logical data abstraction. This part of the api discribes schemas of resources exposed by the api that is not the exact mapping of the data architecture. The declaration of schemas is made by Marshmallow / marshmallow-jsonapi. Marshmallow is a very popular serialization / deserialization library that offers a lot of features to abstract your data architecture. Moreover there is an other library called marshmallow-jsonapi that fit the JSONAPI 1.0 specification and provides Flask integration.

Example:

In this example, let's assume that we have 2 legacy models Person and Computer, and we want to create an abstraction over them.

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()
```

(continues on next page)

(continued from previous page)

```
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    email = db.Column(db.String)
    birth_date = db.Column(db.String)
    password = db.Column(db.String)


class Computer(db.Model):
    computer_id = db.Column(db.Integer, primary_key=True)
    serial = db.Column(db.String)
    person_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    person = db.relationship('Person', backref=db.backref('computers'))
```

Now let's create the logical abstraction to illustrate this concept.

```
from marshmallow_jsonapi.flask import Schema, Relationship
from marshmallow_jsonapi import fields

class PersonSchema(Schema):
    class Meta:
        type_ = 'person'
        self_view = 'person_detail'
        self_view_kwargs = {'id': '<id>'}
        self_view_many = 'person_list'

    id = fields.Integer(as_string=True, dump_only=True)
    name = fields.Str(required=True, load_only=True)
    email = fields.Email(load_only=True)
    birth_date = fields.Date()
    display_name = fields.Function(lambda obj: "{} <{}>".format(obj.name.upper(), obj.
↪email))
    computers = Relationship(self_view='person_computers',
                             self_view_kwargs={'id': '<id>'},
                             related_view='computer_list',
                             related_view_kwargs={'id': '<id>'},
                             many=True,
                             schema='ComputerSchema',
                             type_='computer',
                             id_field='computer_id')


class ComputerSchema(Schema):
    class Meta:
        type_ = 'computer'
        self_view = 'computer_detail'
        self_view_kwargs = {'id': '<id>'}

    id = fields.Str(as_string=True, dump_only=True, attribute='computer_id')
    serial = fields.Str(required=True)
    owner = Relationship(attribute='person',
                         self_view='computer_person',
                         self_view_kwargs={'id': '<id>'},
                         related_view='person_detail',
                         related_view_kwargs={'computer_id': '<id>'},
                         schema='PersonSchema',
```

(continues on next page)

(continued from previous page)

```
                        type_='person')
```

You can see several differences between models and schemas exposed by the api.

First, take a look at the Person compared to PersonSchema:

- we can see that Person has an attribute named "password" and we don't want to expose it through the api so it is not set in PersonSchema
- PersonSchema has an attribute named "display_name" that is the result of concatenation of name and email
- In the computers Relationship() defined on PersonSchema we have set the id_field to "computer_id" as that is the primary key on the Computer(db.model). Without setting id_field the relationship looks for a field called "id".

Second, take a look at the Computer compared to ComputerSchema:

- we can see that the attribute computer_id is exposed as id for consistency of the api
- we can see that the person relationship between Computer and Person is exposed in ComputerSchema as owner because it is more explicit

As a result you can see that you can expose your data in a very flexible way to create the api of your choice over your data architecture.

3.4 Resource Manager

Resource manager is the link between your logical data abstraction, your data layer and optionally other software. It is the place where logic management of your resource is located.

Flask-REST-JSONAPI provides 3 kinds of resource manager with default methods implementation according to the JSONAPI 1.0 specification:

- **ResourceList**: provides get and post methods to retrieve a collection of objects or create one.
- **ResourceDetail**: provides get, patch and delete methods to retrieve details of an object, update an object and delete an object
- **ResourceRelationship**: provides get, post, patch and delete methods to get relationships, create relationships, update relationships and delete relationships between objects.

You can rewrite each default methods implementation to make custom work. If you rewrite all default methods implementation of a resource manager or if you rewrite a method and disable access to others, you don't have to set any attribute of your resource manager.

3.4.1 Required attributes

If you want to use one of the resource manager default method implementation you have to set 2 required attributes in your resource manager: schema and data_layer.

schema the logical data abstraction used by the resource manager. It must be a class inherited from marshmallow_jsonapi.schema.Schema.

data_layer data layer information used to initialize your data layer (If you want to learn more: *Data layer*)

Example:

```
from flask_rest_jsonapi import ResourceList
from your_project.schemas import PersonSchema
from your_project.models import Person
from your_project.extensions import db

class PersonList(ResourceList):
    schema = PersonSchema
    data_layer = {'session': db.session,
                  'model': Person}
```

3.4.2 Optional attributes

All resource managers are inherited from flask.views.MethodView so you can provides optional attributes to your resource manager:

methods a list of methods this resource manager can handle. If you don't specify any method, all methods are handled.

decorators a tuple of decorators plugged to all methods that the resource manager can handle

You can provide default schema kwargs for each resource manager methods with these optional attributes:

- **get_schema_kwargs**: a dict of default schema kwargs in get method
- **post_schema_kwargs**: a dict of default schema kwargs in post method
- **patch_schema_kwargs**: a dict of default schema kwargs in patch method
- **delete_schema_kwargs**: a dict of default schema kwargs in delete method

Each method of a resource manager gets a pre and post process methods that takes view args and kwargs as parameters for the pre process methods, and the result of the method as parameter for the post process method. Thanks to this you can make custom work before and after the method process. Available methods to override are:

before_get pre process method of the get method

after_get post process method of the get method

before_post pre process method of the post method

after_post post process method of the post method

before_patch pre process method of the patch method

after_patch post process method of the patch method

before_delete pre process method of the delete method

after_delete post process method of the delete method

Example:

```
from flask_rest_jsonapi import ResourceDetail
from your_project.schemas import PersonSchema
from your_project.models import Person
from your_project.security import login_required
from your_project.extensions import db

class PersonList(ResourceDetail):
    schema = PersonSchema
    data_layer = {'session': db.session,
```

(continues on next page)

(continued from previous page)

```
                  'model': Person}
    methods = ['GET', 'PATCH']
    decorators = (login_required, )
    get_schema_kwargs = {'only': ('name', )}

    def before_patch(*args, **kwargs):
        """Make custom work here. View args and kwargs are provided as parameter
        """

    def after_patch(result):
        """Make custom work here. Add something to the result of the view.
        """
```

3.4.3 ResourceList

ResourceList manager has its own optional attributes:

view_kwargs if you set this flag to True view kwargs will be used to compute the list url. If you have a list url pattern with parameter like that: /persons/<int:id>/computers you have to set this flag to True

Example:

```
from flask_rest_jsonapi import ResourceList
from your_project.schemas import PersonSchema
from your_project.models import Person
from your_project.extensions import db

class PersonList(ResourceList):
    schema = PersonSchema
    data_layer = {'session': db.session,
                  'model': Person}
```

This minimal ResourceList configuration provides GET and POST interface to retrieve a collection of objects and create an object with all powerful features like pagination, sorting, sparse fieldsets, filtering and including related objects.

If your schema has relationship field(s) you can create an object and link related object(s) to it in the same time. For an example see *Quickstart*.

3.4.4 ResourceDetail

Example:

```
from flask_rest_jsonapi import ResourceDetail
from your_project.schemas import PersonSchema
from your_project.models import Person
from your_project.extensions import db

class PersonDetail(ResourceDetail):
    schema = PersonSchema
    data_layer = {'session': db.session,
                  'model': Person}
```

This minimal ResourceDetail configuration provides GET, PATCH and DELETE interface to retrieve details of objects, update an objects and delete an object with all powerful features like sparse fieldsets and including related objects.

If your schema has relationship field(s) you can update an object and also update his link(s) to related object(s) in the same time. For an example see *Quickstart*.

3.4.5 ResourceRelationship

Example:

```
from flask_rest_jsonapi import ResourceRelationship
from your_project.schemas import PersonSchema
from your_project.models import Person
from your_project.extensions import db

class PersonRelationship(ResourceRelationship):
    schema = PersonSchema
    data_layer = {'session': db.session,
                  'model': Person}
```

This minimal ResourceRelationship configuration provides GET, POST, PATCH and DELETE interface to retrieve relationship(s), create relationship(s), update relationship(s) and delete relationship(s) between objects with all powerful features like sparse fieldsets and including related objects.

3.5 Data layer

The data layer is a CRUD interface between resource manager and data. It is a very flexible system to use any ORM or data storage. You can even create a data layer that use multiple ORMs and data storage to manage your own objects. The data layer implements a CRUD interface for objects and relationships. It also manage pagination, filtering and sorting.

Flask-REST-JSONAPI has a full featured data layer that use the popular ORM SQLAlchemy.

Note: The default data layer used by a resource manager is the SQLAlchemy one. So if you want to use it, you don't have to specify the class of the data layer in the resource manager

---

To configure the data layer you have to set its required parameters in the resource manager.

Example:

```
from flask_rest_jsonapi import ResourceList
from your_project.schemas import PersonSchema
from your_project.models import Person

class PersonList(ResourceList):
    schema = PersonSchema
    data_layer = {'session': db.session,
                  'model': Person}
```

You can also plug additional methods to your data layer in the resource manager. There are two kinds of additional methods:

- query: the "query" additional method takes view_kwargs as parameter and return an alternative query to retrieve the collection of objects in the GET method of the ResourceList manager.
- pre / post process methods: all CRUD and relationship(s) operations have a pre / post process methods. Thanks to it you can make additional work before and after each operations of the data layer. Parameters of each pre / post process methods are available in the flask_rest_jsonapi.data_layers.base.Base base class.

Example:

```
from sqlalchemy.orm.exc import NoResultFound
from flask_rest_jsonapi import ResourceList
from flask_rest_jsonapi.exceptions import ObjectNotFound
from your_project.models import Computer, Person

class ComputerList(ResourceList):
    def query(self, view_kwargs):
        query_ = self.session.query(Computer)
        if view_kwargs.get('id') is not None:
            try:
                self.session.query(Person).filter_by(id=view_kwargs['id']).one()
            except NoResultFound:
                raise ObjectNotFound({'parameter': 'id'}, "Person: {} not found".
↪format(view_kwargs['id']))
            else:
                query_ = query_.join(Person).filter(Person.id == view_kwargs['id'])
        return query_

    def before_create_object(self, data, view_kwargs):
        if view_kwargs.get('id') is not None:
            person = self.session.query(Person).filter_by(id=view_kwargs['id']).one()
            data['person_id'] = person.id

    schema = ComputerSchema
    data_layer = {'session': db.session,
                  'model': Computer,
                  'methods': {'query': query,
                              'before_create_object': before_create_object}}
```

Note: You don't have to declare additional data layer methods in the resource manager. You can declare them in a dedicated module or in the declaration of the model.

Example:

```
from sqlalchemy.orm.exc import NoResultFound
from flask_rest_jsonapi import ResourceList
from flask_rest_jsonapi.exceptions import ObjectNotFound
from your_project.models import Computer, Person
from your_project.additional_methods.computer import before_create_object

class ComputerList(ResourceList):
    schema = ComputerSchema
    data_layer = {'session': db.session,
                  'model': Computer,
                  'methods': {'query': Computer.query,
                              'before_create_object': before_create_object}}
```

3.5.1 SQLAlchemy

Required parameters:

session the session used by the data layer

model the model used by the data layer

Optional parameters:

id_field the field used as identifier field instead of the primary key of the model

url_field the name of the parameter in the route to get value to filter with. Instead "id" is used.

By default SQLAlchemy eagerload related data specified in include querystring parameter. If you want to disable this feature you must add eagerload_includes: False to data layer parameters.

3.5.2 Custom data layer

Like I said previously you can create and use your own data layer. A custom data layer must inherit from flask_rest_jsonapi.data_layers.base.Base. You can see the full scope of possibilities of a data layer in this base class.

Usage example:

```
from flask_rest_jsonapi import ResourceList
from your_project.schemas import PersonSchema
from your_project.data_layers import MyCustomDataLayer

class PersonList(ResourceList):
    schema = PersonSchema
    data_layer = {'class': MyCustomDataLayer,
                  'param_1': value_1,
                  'param_2': value_2}
```

Note: All items except "class" in the data_layer dict of the resource manager will be plugged as instance attributes of the data layer. It is easier to use in the data layer.

3.6 Routing

The routing system is very simple and fits this pattern

```
api.route(<Resource manager>, <endpoint name>, <url_1>, <url_2>, ...)
```

Example:

```
# all required imports are not displayed in this example
from flask_rest_jsonapi import Api

api = Api()
api.route(PersonList, 'person_list', '/persons')
api.route(PersonDetail, 'person_detail', '/persons/<int:id>', '/computers/
↪<int:computer_id>/owner')
api.route(PersonRelationship, 'person_computers', '/persons/<int:id>/relationships/
↪computers')
```

(continues on next page)

(continued from previous page)

```
api.route(ComputerList, 'computer_list', '/computers', '/persons/<int:id>/computers')
api.route(ComputerDetail, 'computer_detail', '/computers/<int:id>')
api.route(ComputerRelationship, 'computer_person', '/computers/<int:id>/relationships/
↪owner')
```

3.7 Filtering

Flask-REST-JSONAPI as a very flexible filtering system. The filtering system is completely related to the data layer used by the ResourceList manager. I will explain the filtering interface for SQLAlchemy data layer but you can use the same interface to your filtering implementation of your custom data layer. The only requirement is that you have to use the "filter" querystring parameter to make filtering according to the JSONAPI 1.0 specification.

Note: Examples are not urlencoded for a better readability

3.7.1 SQLAlchemy

The filtering system of SQLAlchemy data layer has exactly the same interface as the filtering system of Flask-Restless. So this is a first example:

```
GET /persons?filter=[{"name":"name","op":"eq","val":"John"}] HTTP/1.1
Accept: application/vnd.api+json
```

In this example we want to retrieve persons which name is John. So we can see that the filtering interface completely fit the filtering interface of SQLAlchemy: a list a filter information.

name the name of the field you want to filter on

op the operation you want to use (all sqlalchemy operations are available)

val the value that you want to compare. You can replace this by "field" if you want to compare against the value of an other field

Example with field:

```
GET /persons?filter=[{"name":"name","op":"eq","field":"birth_date"}] HTTP/1.1
Accept: application/vnd.api+json
```

In this example, we want to retrieve persons that name is equal to his birth_date. I know, this example is absurd but it is just to explain the syntax of this kind of filter.

If you want to filter through relationships you can do that:

```
GET /persons?filter=[
  {
    "name": "computers",
    "op": "any",
    "val": {
      "name": "serial",
      "op": "ilike",
      "val": "%Amstrad%"
    }
  }
```

(continues on next page)

(continued from previous page)

```
] HTTP/1.1
Accept: application/vnd.api+json
```

Note: When you filter on relationships use "any" operator for "to many" relationships and "has" operator for "to one" relationships.

There is a shortcut to achieve the same filter:

```
GET /persons?filter=[{"name":"computers__serial","op":"ilike","val":"%Amstrad%"}]␣
↪HTTP/1.1
Accept: application/vnd.api+json
```

You can also use boolean combination of operations:

```
GET /persons?filter=[
  {
    "name":"computers__serial",
    "op":"ilike",
    "val":"%Amstrad%"
  },
  {
    "or": {
      [
        {
          "not": {
            "name": "name",
            "op": "eq",
            "val":"John"
          }
        },
        {
          "and": [
            {
              "name": "name",
              "op": "like",
              "val": "%Jim%"
            },
            {
              "name": "birth_date",
              "op": "gt",
              "val": "1990-01-01"
            }
          ]
        }
      ]
    }
  }
] HTTP/1.1
Accept: application/vnd.api+json
```

Common available operators:

- any: used to filter on to many relationships
- between: used to filter a field between two values

- endswith: check if field ends with a string
- eq: check if field is equal to something
- ge: check if field is greater than or equal to something
- gt: check if field is greater than to something
- has: used to filter on to one relationships
- ilike: check if field contains a string (case insensitive)
- in_: check if field is in a list of values
- is_: check if field is a value
- isnot: check if field is not a value
- like: check if field contains a string
- le: check if field is less than or equal to something
- lt: check if field is less than to something
- match: check if field match against a string or pattern
- ne: check if field is not equal to something
- notilike: check if field does not contains a string (case insensitive)
- notin_: check if field is not in a list of values
- notlike: check if field does not contains a string
- startswith: check if field starts with a string

Note: Availables operators depend on field type in your model

3.7.2 Simple filters

Simple filter adds support for a simplified form of filters and supports only *eq* operator. Each simple filter transforms to original filter and appends to list of filters.

For example

```
GET /persons?filter[name]=John HTTP/1.1
Accept: application/vnd.api+json
```

equals to:

```
GET /persons?filter[name]=[{"name":"name","op":"eq","val":"John"}] HTTP/1.1
Accept: application/vnd.api+json
```

You can also use more than one simple filter in request:

```
GET /persons?filter[name]=John&filter[gender]=male HTTP/1.1
Accept: application/vnd.api+json
```

which equals to:

```
GET /persons?filter[name]=[{"name":"name","op":"eq","val":"John"}, {"name":"gender",
↪"op":"eq","val":"male"}] HTTP/1.1
```

3.8 Include related objects

You can include related object(s) details to responses with the querystring parameter named "include". You can use "include" parameter on any kind of route (classical CRUD route or relationships route) and any kind of http methods as long as method return data.

This features will add an additional key in result named "included"

Example:

Request:

```
GET /persons/1?include=computers HTTP/1.1
Accept: application/vnd.api+json
```

Response:

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

{
  "data": {
    "type": "person",
    "id": "1",
    "attributes": {
      "display_name": "JEAN <jean@gmail.com>",
      "birth_date": "1990-10-10"
    },
    "relationships": {
      "computers": {
        "data": [
          {
            "type": "computer",
            "id": "1"
          }
        ],
        "links": {
          "related": "/persons/1/computers",
          "self": "/persons/1/relationships/computers"
        }
      }
    },
    "links": {
      "self": "/persons/1"
    }
  },
  "included": [
    {
      "type": "computer",
      "id": "1",
      "attributes": {
        "serial": "Amstrad"
      },
      "relationships": {
        "owner": {
          "links": {
            "related": "/computers/1/owner",
            "self": "/computers/1/relationships/owner"
```

(continues on next page)

(continued from previous page)

```
            }
          }
        },
        "links": {
          "self": "/computers/1"
        }
      }
    ],
    "links": {
      "self": "/persons/1"
    },
    "jsonapi": {
      "version": "1.0"
    }
}
```

You can even follow relationships with include

Example:

Request:

```
GET /persons/1?include=computers.owner HTTP/1.1
Accept: application/vnd.api+json
```

Response:

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

{
  "data": {
    "type": "person",
    "id": "1",
    "attributes": {
      "display_name": "JEAN <jean@gmail.com>",
      "birth_date": "1990-10-10"
    },
    "relationships": {
      "computers": {
        "data": [
          {
            "type": "computer",
            "id": "1"
          }
        ],
        "links": {
          "related": "/persons/1/computers",
          "self": "/persons/1/relationships/computers"
        }
      }
    },
    "links": {
      "self": "/persons/1"
    }
  },
  "included": [
```

(continues on next page)

(continued from previous page)


```
    {
      "type": "computer",
      "id": "1",
      "attributes": {
        "serial": "Amstrad"
      },
      "relationships": {
        "owner": {
          "data": {
            "type": "person",
            "id": "1"
          },
          "links": {
            "related": "/computers/1/owner",
            "self": "/computers/1/relationships/owner"
          }
        }
      },
      "links": {
        "self": "/computers/1"
      }
    },
    {
      "type": "person",
      "id": "1",
      "attributes": {
        "display_name": "JEAN <jean@gmail.com>",
        "birth_date": "1990-10-10"
      },
      "relationships": {
        "computers": {
          "links": {
            "related": "/persons/1/computers",
            "self": "/persons/1/relationships/computers"
          }
        }
      },
      "links": {
        "self": "/persons/1"
      }
    }
  ],
  "links": {
    "self": "/persons/1"
  },
  "jsonapi": {
    "version": "1.0"
  }
}
```


I know it is an absurd example because it will include details of related person computers and details of the person that is already in the response. But it is just for example.

3.9 Sparse fieldsets

You can restrict the fields returned by api with the querystring parameter called "fields". It is very useful for performance purpose because fields not returned are not resolved by api. You can use "fields" parameter on any kind of route (classical CRUD route or relationships route) and any kind of http methods as long as method return data.

Note: Examples are not urlencoded for a better readability

The syntax of a fields is like that

```
?fields[<resource_type>]=<list of fields to return>
```

Example:

```
GET /persons?fields[person]=display_name HTTP/1.1
Accept: application/vnd.api+json
```

In this example person's display_name is the only field returned by the api. No relationships links are returned so the response is very fast because api doesn't have to compute relationships link and it is a very costly work.

You can manage returned fields for the entire response even for included objects

Example:

If you don't want to compute relationships links for included computers of a person you can do something like that

```
GET /persons/1?include=computers&fields[computer]=serial HTTP/1.1
Accept: application/vnd.api+json
```

And of course you can combine both like that:

Example:

```
GET /persons/1?include=computers&fields[computer]=serial&fields[person]=name,
↪computers HTTP/1.1
Accept: application/vnd.api+json
```

Warning: If you want to use both "fields" and "include" don't forget to specify the name of the relationship in fields; if you don't the include wont work.

3.10 Pagination

When you use the default implementation of get method on a ResourceList your results will be paginated by default. Default pagination size is 30 but you can manage it from querystring parameter named "page".

Note: Examples are not urlencoded for a better readability

3.10.1 Size

You can control page size like that:

```
GET /persons?page[size]=10 HTTP/1.1
Accept: application/vnd.api+json
```

3.10.2 Number

You can control page number like that:

```
GET /persons?page[number]=2 HTTP/1.1
Accept: application/vnd.api+json
```

3.10.3 Size + Number

Of course, you can control both like that:

```
GET /persons?page[size]=10&page[number]=2 HTTP/1.1
Accept: application/vnd.api+json
```

3.10.4 Disable pagination

You can disable pagination with size = 0

```
GET /persons?page[size]=0 HTTP/1.1
Accept: application/vnd.api+json
```

3.11 Sorting

You can sort results with querystring parameter named "sort"

Note: Examples are not urlencoded for a better readability

Example:

```
GET /persons?sort=name HTTP/1.1
Accept: application/vnd.api+json
```

3.11.1 Multiple sort

You can sort on multiple fields like that:

```
GET /persons?sort=name,birth_date HTTP/1.1
Accept: application/vnd.api+json
```

3.11.2 Descending sort

You can make desc sort with the character "-" like that:

```
GET /persons?sort=-name HTTP/1.1
Accept: application/vnd.api+json
```

3.11.3 Multiple sort + Descending sort

Of course, you can combine both like that:

```
GET /persons?sort=-name,birth_date HTTP/1.1
Accept: application/vnd.api+json
```

3.12 Errors

JSONAPI 1.0 specification recommand to return errors like that:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/vnd.api+json

{
  "errors": [
    {
      "status": "422",
      "source": {
        "pointer":"/data/attributes/first-name"
      },
      "title":  "Invalid Attribute",
      "detail": "First name must contain at least three characters."
    }
  ],
  "jsonapi": {
    "version": "1.0"
  }
}
```

The "source" field gives information about the error if it is located in data provided or in querystring parameter.

The previous example displays error located in data provided instead of this next example displays error located in querystring parameter "include":

```
HTTP/1.1 400 Bad Request
Content-Type: application/vnd.api+json

{
  "errors": [
    {
      "status": "400",
      "source": {
        "parameter": "include"
      },
      "title":  "BadRequest",
      "detail": "Include parameter is invalid"
```

(continues on next page)

(continued from previous page)

```
    }
  ],
  "jsonapi": {
    "version": "1.0"
  }
}
```

Flask-REST-JSONAPI provides two kind of helpers to achieve error displaying:

* **the errors module**: you can import jsonapi_errors from the errors module to create the structure of a list of errors according to JSONAPI 1.0 specification

* **the exceptions module**: you can import lot of exceptions from this module that helps you to raise exceptions that will be well formatted according to JSONAPI 1.0 specification

When you create custom code for your api I recommand to use exceptions from exceptions module of Flask-REST-JSONAPI to raise errors because JsonApiException based exceptions are catched and rendered according to JSONAPI 1.0 specification.

Example:

```
# all required imports are not displayed in this example
from flask_rest_jsonapi.exceptions import ObjectNotFound

class ComputerList(ResourceList):
    def query(self, view_kwargs):
        query_ = self.session.query(Computer)
        if view_kwargs.get('id') is not None:
            try:
                self.session.query(Person).filter_by(id=view_kwargs['id']).one()
            except NoResultFound:
                raise ObjectNotFound({'parameter': 'id'}, "Person: {} not found".
↪format(view_kwargs['id']))
            else:
                query_ = query_.join(Person).filter(Person.id == view_kwargs['id'])
        return query_
```

3.13 Api

You can provide global decorators as tuple to the Api.

Example:

```
from flask_rest_jsonapi import Api
from your_project.security import login_required

api = Api(decorators=(login_required,))
```

3.14 Permission

Flask-REST-JSONAPI provides a very agnostic permission system.

Example:

```
from flask import Flask
from flask_rest_jsonapi import Api
from your_project.permission import permission_manager

app = Flask(__name__)

api = Api()
api.init_app(app)
api.permission_manager(permission_manager)
```

In this previous example, the API will check permission before each method call with the permission_manager function.

The permission manager must be a function that looks like this:

```
def permission_manager(view, view_args, view_kwargs, *args, **kwargs):
    """The function use to check permissions

    :param callable view: the view
    :param list view_args: view args
    :param dict view_kwargs: view kwargs
    :param list args: decorator args
    :param dict kwargs: decorator kwargs
    """
```

Note: Flask-REST-JSONAPI use a decorator to check permission for each method named has_permission. You can provide args and kwargs to this decorators so you can retrieve this args and kwargs in the permission_manager. The default usage of the permission system does not provides any args or kwargs to the decorator.

If permission is denied I recommand to raise exception like that:

```
raise JsonApiException(<error_source>,
                       <error_details>,
                       title='Permission denied',
                       status='403')
```

You can disable the permission system or make custom permission checking management of a resource like that:

```
from flask_rest_jsonapi import ResourceList
from your_project.extensions import api

class PersonList(ResourceList):
    disable_permission = True

    @api.has_permission('custom_arg', custom_kwargs='custom_kwargs')
    def get(*args, **kwargs):
        return 'Hello world !'
```

Warning: If you want to use both permission system and oauth support to retrieve information like user from oauth (request.oauth.user) in the permission system you have to initialize permission system before to initialize oauth support because of decorators cascading.

Example:

```
from flask import Flask
from flask_rest_jsonapi import Api
from flask_oauthlib.provider import OAuth2Provider
from your_project.permission import permission_manager

app = Flask(__name__)
oauth2 = OAuth2Provider()

api = Api()
api.init_app(app)
api.permission_manager(permission_manager) # initialize permission system first
api.oauth_manager(oauth2) # initialize oauth support second
```

3.15 OAuth

Flask-REST-JSONAPI support OAuth via Flask-OAuthlib

Example:

```
from flask import Flask
from flask_rest_jsonapi import Api
from flask_oauthlib.provider import OAuth2Provider

app = Flask(__name__)
oauth2 = OAuth2Provider()

api = Api()
api.init_app(app)
api.oauth_manager(oauth2)
```

In this example Flask-REST-JSONAPI will protect all your resource methods with this decorator

```
oauth2.require_oauth(<scope>)
```

The pattern of the scope is like that

```
<action>_<resource_type>
```

Where action is:

- list: for the get method of a ResourceList
- create: for the post method of a ResourceList
- get: for the get method of a ResourceDetail
- update: for the patch method of a ResourceDetail
- delete: for the delete method of a ResourceDetail

Example

```
list_person
```

If you want to customize the scope you can provide a function that computes your custom scope. The function have to looks like that:

```
def get_scope(resource, method):
        """Compute the name of the scope for oauth

        :param Resource resource: the resource manager
        :param str method: an http method
        :return str: the name of the scope
        """
        return 'custom_scope'
```

Usage example:

```
from flask import Flask
from flask_rest_jsonapi import Api
from flask_oauthlib.provider import OAuth2Provider

app = Flask(__name__)
oauth2 = OAuth2Provider()

api = Api()
api.init_app(app)
api.oauth_manager(oauth2)
api.scope_setter(get_scope)
```

Note: You can name the custom scope computation method as you want but you have to set the 2 required parameters: resource and method like in this previous example.

If you want to disable OAuth or make custom methods protection for a resource you can add this option to the resource manager.

Example:

```
from flask_rest_jsonapi import ResourceList
from your_project.extensions import oauth2

class PersonList(ResourceList):
    disable_oauth = True

    @oauth2.require_oauth('custom_scope')
    def get(*args, **kwargs):
        return 'Hello world !'
```

3.16 Configuration

You have access to 5 configration keys:

- PAGE_SIZE: the number of items in a page (default is 30)
- MAX_PAGE_SIZE: the maximum page size. If you specify a page size greater than this value you will receive 400 Bad Request response.

- MAX_INCLUDE_DEPTH: the maximum length of an include through schema relationships
- ALLOW_DISABLE_PAGINATION: if you want to disallow to disable pagination you can set this configuration key to False
- CATCH_EXCEPTIONS: if you want flask_rest_jsonapi to catch all exceptions and return as JsonApiException (default is True)

CHAPTER 4

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

- genindex
- modindex

Python Module Index

f

flask_rest_jsonapi, ??

Index

F

flask_rest_jsonapi (*module*), 1