flask-jsonapi Documentation

Release stable

Mar 17, 2022

Contents

JSONAPI 1.0 server implementation for Flask.

Flask-jsonapi is a server implementation of JSON API 1.0 specification for Flask. It allows for rapid creation of CRUD JSON API endpoints. Those endpoints can be used by JSON API clients, eg. JavaScript frontend applications written in Ember.js, React, Angular.js etc.

A compatible Python client can be found in the following package: jsonapi-request.

Flask-jsonapi depends on the following external libraries:

- Flask (micro web framework)
- marshmallow-jsonapi (serialization, deserialization, validation with JSON API format)
- SQLAlchemy (SQL ORM)

CHAPTER 1

Features

Flask-jsonapi implements the following parts of the JSON API specification:

- **fetching resources** [specification]
- **creating, updating and deleting resources** [specification]
- **inclusion of related resources** [specification]
- **filtering** - helps with adding filters to views (including helpers for SQLAlchemy), the format is compatible with recommendations
- **sorting** [specification]
- **pagination** [specification]
- **sparse fieldsets** [specification]
- **links** [specification] - resolved by marshmallow-jsonapi (docs)
- **error objects** [specification]
- **resource-level permissions** - view-level decorators support in ViewSets
- **object-level permissions** - interfaces added in [#43]

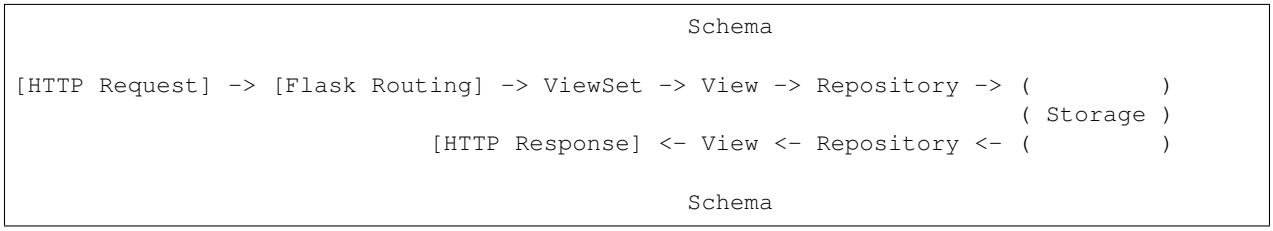
Not implemented yet:

- **fetching relationships** [specification] and **updating relationships** [specification] - #27

CHAPTER 2

Architecture

Flow through layers beginning with an HTTP Request and ending with an HTTP Response:



CHAPTER 3

Installation

To install (with SQLAlchemy support) run:

```
pip install Flask-jsonapi[sqlalchemy]
```

CHAPTER 4

Simple example

Let's create a working example of a minimal Flask application. It will expose a single resource `Article` as a REST endpoint with fetch/create/update/delete operations. For persistence layer, it will use an in-memory SQLite database with SQLAlchemy for storage.

4.1 Configuration

```
import flask
import sqlalchemy
from sqlalchemy.orm import scoped_session
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base
from marshmallow_jsonapi import Schema, fields

import flask_jsonapi
from flask_jsonapi.resource_repositories import sqlalchemy_repositories

db_engine = sqlalchemy.create_engine('sqlite:///')
session = scoped_session(sessionmaker(bind=db_engine))
Base = declarative_base()
Base.query = session.query_property()

class Article(Base):
    __tablename__ = 'articles'
    id = sqlalchemy.Column(sqlalchemy.Integer, primary_key=True)
    title = sqlalchemy.Column(sqlalchemy.String)

Base.metadata.create_all(db_engine)

class ArticleRepository(sqlalchemy_repositories.SqlAlchemyModelRepository):
    model = Article
    instance_name = 'articles'
    session = session
```

(continues on next page)

(continued from previous page)

```
class ArticleSchema(Schema):
    id = fields.Int()
    title = fields.Str()

    class Meta:
        type_ = 'articles'
        strict = True

class ArticleRepositoryViewSet(flask_jsonapi.resource_repository_views.
↪ResourceRepositoryViewSet):
    schema = ArticleSchema
    repository = ArticleRepository()

app = flask.Flask(__name__)
api = flask_jsonapi.Api(app)
api.repository(ArticleRepositoryViewSet(), 'articles', '/articles/')
app.run(host='127.0.0.1', port=5000)
```

4.2 Usage

Create a new `Article` with title "First article":

```
curl -H 'Content-Type: application/vnd.api+json' \
    -H 'Accept: application/vnd.api+json' \
    http://localhost:5000/articles/ \
    --data '{"data": {"attributes": {"title": "First article"}, "type": "articles"}}'␣
↪\
    2>/dev/null | python -m json.tool
```

Result:

```
{
    "data": {
        "type": "articles",
        "id": 1,
        "attributes": {
            "title": "First article"
        }
    },
    "jsonapi": {
        "version": "1.0"
    }
}
```

Get the list of `Articles`:

```
curl -H 'Accept: application/vnd.api+json' \
    http://localhost:5000/articles/ \
    2>/dev/null | python -m json.tool
```

Result:

```
{
    "data": [
        {
            "type": "articles",
            "id": 1,
            "attributes": {
                "title": "First article"
            }
        }
    ],
    "jsonapi": {
        "version": "1.0"
    },
    "meta": {
        "count": 1
    }
}
```

CHAPTER 5

Table of Contents

5.1 Tutorial

5.2 ViewSets

5.3 Views

5.4 Repositories

5.5 Including relationships

Flask-jsonapi supports inclusion of related resources. [specification]

Note:

By default when `include` parameter is not provided, marshmallow-jsonapi doesn't serialize resource linkage. You can enable it by passing `include_resource_linkage=True` and the resource `type_` argument to the desired fields in schema. [documentation]

5.5.1 Usage

Inclusion of related resource in compound document in could be requested as shown below:

```
/articles/1?include=comments
```

Inclusion of resources related to other resources can be achieved using a dot-separated path:

```
/articles/1?include=comments.author
```

Multiple related resources can be requested in a comma-separated list:

```
/articles/1?include=author,comments.author
```

5.6 Sparse Fieldsets

Flask-jsonapi supports `fields[TYPE]` parameter to restrict the server to return a set of fields for a given resource type. [specification]

Note:

By default when `include` parameter is not provided, marshmallow-jsonapi doesn't serialize resource linkage. You can enable it by passing `include_resource_linkage=True` and the resource `type_` argument to the desired fields in schema. [documentation]

5.6.1 Usage

Basic example:

```
/articles?fields[articles]=title,body
```

This example demonstrates combination on both `include` and `fields` parameters.

```
/articles?include=author&fields[articles]=title,body&fields[people]=name
```

Note:

For `include` parameter there are specific *attributes* of the resources are provided, but for `fields` parameter there are *types* provided.

5.7 Filtering

Flask-jsonapi supports resource filtering. [specification]

Note:

All results of examples are `filters` parameter passed to `read_many` method of `ResourceList` or `get_list` method of `ResourceRepositoryViewSet`.

5.7.1 Simple Usage

Let'd define a simple filter schema with one filter:

```
class ExampleFiltersSchema(filters_schema.FilterSchema):
    title = filters_schema.FilterField()
```

Now use it in `ResourceList` or `ResourceRepositoryViewSet`:

```
class ExampleListView(resources.ResourceList):
    schema = ExampleSchema()
    filter_schema = ExampleFiltersSchema()
```

Request:

```
/example-resource?filter[title]=something
```

Will result in:

```
{'title': 'something'}
```

5.7.2 List of filter values

To parse many values fo single filter use `ListFilterField`:

```
class ExampleFiltersSchema(filters_schema.FilterSchema):
    title = filters_schema.ListFilterField()
```

Request:

```
/example-resource?filter[title]=something,other
```

Will result in:

```
{'title': ['something', 'other']}
```

5.7.3 Parsing filter values

By default filter values are parsed as strings. To change it, pass a desired subclass of `marshmallow.fields.Field` as `type_` argument to field constructor:

[reference]

```
class ExampleFiltersSchema(filters_schema.FilterSchema):
    title = filters_schema.FilterField(type_=fields.Float)
```

Request:

```
/example-resource?filter[score]=4.5
```

Will result in:

```
{'score': 4.5}
```

5.7.4 Overriding filter attribute

You can override the key with witch the filter will be parsed with `attribute` parameter:

```
class ExampleFiltersSchema(filters_schema.FilterSchema):
    title = filters_schema.FilterField(attribute='renamed')
```

Request:

```
/example-resource?filter[title]=something
```

Will result in:

```
{'renamed': 'something}
```

5.7.5 Using Operators

The base JSONAPI specification is agnostic about filtering strategies supported by a server, Flask-JsonApi added a support for operators.

Note:

List of supported operators is available [here]

Note:

Filters with operators can be automatically applied to query using `sqlalchemy_repositories.SqlAlchemyModelRepository`. This is achieved using [sqlalchemy-django-query]

Defining a set of allowed operators:

```
class ExampleFiltersSchema(filters_schema.FilterSchema):
    title = filters_schema.FilterField(operators=['eq', 'ne'])
```

Request:

```
/example-resource?filter[title][ne]=something
```

Will result in:

```
{'title__ne': 'something}
```

Note:

You can also specify a default operator (when none are provided in query string) with `default_operator` parameter.

5.7.6 Marshmallow-Jsonapi Schema integration

Filters can be autogenerated using supplied schema.

Let's define a schema:

```
class ExampleSchema(marshmallow_jsonapi.Schema):
    id = fields.UUID(required=True)
    body = fields.Str()
    is_active = fields.Boolean()

    class Meta:
        type_ = 'example'
```

Let's define a filters schema that uses this schema:

```
class ExampleFiltersSchema(filters_schema.FilterSchema):
    class Meta:
        schema = ExampleSchema
        fields = ['id', 'body', 'is_active']
```

Now you can filter by the fields specified in `class Meta`:

Request:

```
/example-resource?filter[is-active]=True
```

Will result in:

```
{'is_active': True}
```

5.7.7 Relationship filtering

Relationships within resources can be also used in filtering.

Let's define two related filter schemas:

```
class FirstFiltersSchema(filters_schema.FilterSchema):
    attribute = filters_schema.FilterField()

class SecondFiltersSchema(filters_schema.FilterSchema):
    relationship = filters_schema.RelationshipFilterField(SecondFiltersSchema)
```

Request:

```
/second-resource?filter[relationship][attribute]=something
```

Will result in:

```
{'relationship__attribute': 'something}
```

Relationship filters can be automatically applied to query using `sqlalchemy_repositories.SqlAlchemyModelRepository`. This is achieved using [sqlalchemy-django-query]

5.8 Sorting

Flask-jsonapi supports sorting by one or more criteria . [specification]

Note:

To enable sorting by a "sort field" other than resource attributes, you need to specify this field in the schema and implement sorting method in the repository.

5.8.1 Usage

```
/articles?sort=-created,title
```

5.9 Pagination

Flask-jsonapi supports page-based pagination. [specification]

By default list endpoint results are not paginated.

5.9.1 Usage

```
/users?page[size]={page_size}&page[number]={page_number}
```

5.10 Extensions

Python Module Index

o

Index

O