SQLAlchemy-JSON-API Documentation

Release 0.4.6

Konsta Vesterinen

Jan 02, 2018

Contents

Contents:

CHAPTER 1

Installation

This part of the documentation covers the installation of SQLAlchemy-JSON-API.

1.1 Supported platforms

SQLAlchemy-JSON-API has been tested against the following Python platforms.

- cPython 2.7
- cPython 3.3
- cPython 3.4
- cPython 3.5

1.2 Installing an official release

You can install the most recent official SQLAlchemy-JSON-API version using pip:

```
pip install sqlalchemy-json-api
```

1.3 Installing the development version

To install the latest version of SQLAlchemy-JSON-API, you need first obtain a copy of the source. You can do that by cloning the git repository:

```
git clone git://github.com/kvesteri/sqlalchemy-json-api.git
```

Then you can install the source distribution using the `setup.py` script:

```
cd sqlalchemy-json-api
python setup.py install
```

1.4 Checking the installation

To check that SQLAlchemy-JSON-API has been properly installed, type `python` from your shell. Then at the Python prompt, try to import SQLAlchemy-JSON-API, and check the installed version:

```
>>> import sqlalchemy_json_api
>>> sqlalchemy_json_api.__version__
0.4.6
```

CHAPTER 2

Quickstart

Consider the following model definition.

```
import sqlalchemy as sa
from sqlalchemy.ext.declarative import declarative_base


Base = declarative_base()


class Article(Base):
    __tablename__ = 'article'
    id = sa.Column('_id', sa.Integer, primary_key=True)
    name = sa.Column(sa.String)
    content = sa.Column(sa.String)


class Comment(Base):
    __tablename__ = 'comment'
    id = sa.Column(sa.Integer, primary_key=True)
    content = sa.Column(sa.String)
    article_id = sa.Column(sa.Integer, sa.ForeignKey(Article.id))
    article = sa.orm.relationship(article_cls, backref='comments')
```

In order to use SQLAlchemy-JSON-API you need to first initialize a *QueryBuilder* by providing it a class mapping.

```
from sqlalchemy_json_api import QueryBuilder


query_builder = QueryBuilder({
    'articles': Article,
    'comments': Comment
})
```

Now we can start using it by selecting from the existing resources.

```
query = query_builder.select(Article)
result = session.execute(query).scalar()
# {
#     'data': [{
#         'id': '1',
#         'type': 'articles',
#         'attributes': {
#             'content': 'Some content',
#             'name': 'Some article',
#         },
#         'relationships': {
#             'comments': {
#                 'data': [
#                     {'id': '1', 'type': 'comments'},
#                     {'id': '2', 'type': 'comments'}
#                 ]
#             },
#         },
#     }]
# }
```

You can also make the query builder build queries that return the results as raw json by using the `as_text` parameter.

```
query = query_builder.select(Article, as_text=True)
result = session.execute(query).scalar()
# '{
#     "data": [{
#         "id": "1",
#         "type": "articles",
#         "attributes": {
#             "content": "Some content",
#             "name": "Some article",
#         },
#         "relationships": {
#             "comments": {
#                 "data": [
#                     {"id": "1", "type": "comments"},
#                     {"id": "2", "type": "comments"}
#                 ]
#             },
#         },
#     }]
# }'
```

CHAPTER 3

Selecting fields

By default SQLAlchemy-JSON-API selects all orm descriptors (except synonyms) for given model. This includes:

- Column properties
- Hybrid properties
- Relationship properties

Please notice that you can't include regular descriptors, only orm descriptors.

3.1 The id property

Each included model MUST have an `id` property. Usually this should be the primary key of your model. If your model doesn't have an `id` property you can add one by using for example SQLAlchemy hybrids.

```
from sqlalchemy.ext.hybrid import hybrid_property


class GroupInvitation(Base):
    group_id = sa.Column(
        sa.Integer,
        sa.ForeignKey(Group.id),
        primary_key=True
    )
    user_id = sa.Column(
        sa.Integer,
        sa.ForeignKey(User.id),
        primary_key=True
    )
    issued_at = sa.Column(sa.DateTime)

    @hybrid_property
    def id(self):
        return self.group_id + ':' + self.user_id
```

Note: SQLAlchemy-JSON-API always returns the id as a string. If the type of the id property is not a string SQLAlchemy-JSON-API tries to cast the given property to string.

3.2 Customizing field selection

You can customize this behaviour by providing the `fields` parameter to *QueryBuilder.select()*.

```
query_builder.select(Article, fields={'articles': ['name']})
result = session.execute(query).scalar()
# {
#     'data': [{
#         'id': '1',
#         'type': 'articles',
#         'attributes': {
#             'name': 'Some article',
#         },
#     }]
# }
```

If you only want to select id for given model you need to provide empty list for given model key.

```
query = query_builder.select(Article, fields={'articles': []})
result = session.execute(query).scalar()
# {
#     'data': [{
#         'id': '1',
#         'type': 'articles',
#     }]
# }
```

CHAPTER 4

Compound documents

You can create queries returning compound document responses by providing the `include` parameter to *`QueryBuilder.select()`*.

```
query = query_builder.select(
    Article,
    fields={'articles': ['name', 'comments']},
    include=['comments']
)
result = session.execute(query).scalar()
# {
#     'data': [{
#         'id': '1',
#         'type': 'articles',
#         'attributes': {
#             'content': 'Some content',
#             'name': 'Some article',
#         },
#         'relationships': {
#             'comments': {
#                 'data': [
#                     {'id': '1', 'type': 'comments'},
#                     {'id': '2', 'type': 'comments'}
#                 ]
#             },
#         },
#     }],
#     'included': [
#         {
#             'id': '1',
#             'type': 'comments',
#             'attributes': {
#                 'content': 'Some comment'
#             }
#         },
#         {
```

```
#             'id': '2',
#             'type': 'comments',
#             'attributes': {
#                 'content': 'Some other comment'
#             }
#         }
#     ]
# }
```

Note: SQLAlchemy-JSON-API always returns all included resources ordered by first type and then by id in ascending order. The consistent order of resources helps testing APIs.

CHAPTER 5

Sorting queries

You can apply an order by to builded query by providing it a `sort` parameter. This parameter should be a list of root resource attribute names.

Sort by name ascending

```
query = query_builder.select(
    Article,
    sort=['name']
)
```

Sort by name descending first and id ascending second

```
query = query_builder.select(
    Article,
    sort=['-name', 'id']
)
```

Note: SQLAlchemy-JSON-API does NOT support sorting by related resource attribute at the moment.

CHAPTER 6

Filtering queries

You can filter query results by providing the `from_obj` parameter for *`QueryBuilder.select()`*. This parameter can be any SQLAlchemy selectable construct.

```
base_query = session.query(Article).filter(Article.name == 'Some article')

query = query_builder.select(
    Article,
    fields={'articles': ['name']},
    from_obj=base_query
)
result = session.execute(query).scalar()
# {
#     'data': [{
#         'id': '1',
#         'type': 'articles',
#         'attributes': {
#             'name': 'Some article',
#         },
#     }]
# }
```

You can also limit the results by giving `limit` and `offset` parameters.

```
query = query_builder.select(
    Article,
    fields={'articles': ['name']},
    limit=5,
    offset=10
)
```

CHAPTER 7

Type based formatting

Sometimes you may want type based formatting, eg. forcing all datetimes in ISO standard format. You can easily achieve this by using `type_formatters` parameter for `QueryBuilder()`.

```
def isoformat(date):
    return sa.func.to_char(
        date,
        sa.text('\'YYYY-MM-DD"T"HH24:MI:SS.US"Z"\'')
    ).label(date.name)

query_builder.type_formatters = {
    sa.DateTime: isoformat
}

query = query_builder.select(
    Article,
    fields={'articles': ['name', 'created_at']},
    from_obj=base_query
)
result = session.execute(query).scalar()
# {
#     'data': [{
#         'id': '1',
#         'type': 'articles',
#         'attributes': {
#             'name': 'Some article',
#             'created_at': '2011-01-01T00:00:00.000000Z'
#         },
#     }]
# }
```

CHAPTER 8

API Documentation

This part of the documentation covers all the public classes and functions in SQLAlchemy-JSON-API.

class sqlalchemy_json_api.**QueryBuilder**(*model_mapping*, *base_url=None*, *type_formatters=None*, *sort_included=True*)

1. Simple example

```
query_builder = QueryBuilder({
    'articles': Article,
    'users': User,
    'comments': Comment
})
```

2. Example using type formatters:

```
def isoformat(date):
    return sa.func.to_char(
        date,
        sa.text(''YYYY-MM-DD"T"HH24:MI:SS.US"Z"'')
    ).label(date.name)

query_builder = QueryBuilder(
    {
        'articles': Article,
        'users': User,
        'comments': Comment
    },
    type_formatters={sa.DateTime: isoformat}
)
```

Parameters

- **model_mapping** – A mapping with keys representing JSON API resource identifier type names and values as SQLAlchemy models.

It is recommended to use lowercased pluralized and hyphenized names for resource identifier types. So for example model such as LeagueInvitiation should have an equivalent key of 'league-invitations'.

- **base_url** – Base url to be used for building JSON API compatible links objects. By default this is *None* indicating that no link objects will be built.
- **type_formatters** – A dictionary of type formatters
- **sort_included** – Whether or not to sort included objects by type and id.

select (*model*, ***kwargs*)

Builds a query for selecting multiple resource instances:

```
query = query_builder.select(
    Article,
    fields={'articles': ['name', 'author', 'comments']},
    include=['author', 'comments.author'],
    from_obj=session.query(Article).filter(
        Article.id.in_([1, 2, 3, 4])
    )
)
```

Results can be sorted:

```
# Sort by id in descending order
query = query_builder.select(
    Article,
    sort=['-id']
)

# Sort by name and id in ascending order
query = query_builder.select(
    Article,
    sort=['name', 'id']
)
```

Parameters

- **model** – The root model to build the select query from.
- **fields** – A mapping of fields. Keys representing model keys and values as lists of model descriptor names.
- **include** – List of dot-separated relationship paths.
- **sort** – List of attributes to apply as an order by for the root model.
- **limit** – Applies an SQL LIMIT to the generated query.
- **limit** – Applies an SQL OFFSET to the generated query.
- **links** – A dictionary of links to apply as top level links in the built query. Keys representing json keys and values as valid urls or dictionaries.
- **from_obj** – A SQLAlchemy selectable (for example a Query object) to select the query results from.
- **as_text** – Whether or not to build a query that returns the results as text (raw json).

select_one(*model*, *id*, ***kwargs*)

Builds a query for selecting single resource instance.

```
query = query_builder.select_one(
    Article,
    1,
    fields={'articles': ['name', 'author', 'comments']},
    include=['author', 'comments.author'],
)
```

Parameters

- **model** – The root model to build the select query from.
- **id** – The id of the resource to select.
- **fields** – A mapping of fields. Keys representing model keys and values as lists of model descriptor names.
- **include** – List of dot-separated relationship paths.
- **links** – A dictionary of links to apply as top level links in the built query. Keys representing json keys and values as valid urls or dictionaries.
- **from_obj** – A SQLAlchemy selectable (for example a Query object) to select the query results from.
- **as_text** – Whether or not to build a query that returns the results as text (raw json).

select_related(*obj*, *relationship_key*, ***kwargs*)

Builds a query for selecting related resource(s). This method can be used for building select queries for JSON requests such as:

```
GET articles/1/author
```

Usage:

```
article = session.query(Article).get(1)

query = query_builder.select_related(
    article,
    'category'
)
```

Parameters

- **obj** – The root object to select the related resources from.
- **fields** – A mapping of fields. Keys representing model keys and values as lists of model descriptor names.
- **include** – List of dot-separated relationship paths.
- **links** – A dictionary of links to apply as top level links in the built query. Keys representing json keys and values as valid urls or dictionaries.
- **sort** – List of attributes to apply as an order by for the root model.
- **from_obj** – A SQLAlchemy selectable (for example a Query object) to select the query results from.
- **as_text** – Whether or not to build a query that returns the results as text (raw json).

select_relationship(*obj*, *relationship_key*, ***kwargs*)

Builds a query for selecting relationship resource(s):

```
article = session.query(Article).get(1)

query = query_builder.select_related(
    article,
    'category'
)
```

Parameters

- **obj** – The root object to select the related resources from.
- **sort** – List of attributes to apply as an order by for the root model.
- **links** – A dictionary of links to apply as top level links in the built query. Keys representing json keys and values as valid urls or dictionaries.
- **from_obj** – A SQLAlchemy selectable (for example a Query object) to select the query results from.
- **as_text** – Whether or not to build a query that returns the results as text (raw json).

exception sqlalchemy_json_api.**IdPropertyNotFound**

exception sqlalchemy_json_api.**InvalidField**

exception sqlalchemy_json_api.**UnknownField**

exception sqlalchemy_json_api.**UnknownModel**

exception sqlalchemy_json_api.**UnknownFieldKey**

Python Module Index

S

Index