SQLAlchemy-JSONAPI Documentation

Release 1.0.0

Colton Provias

Jan 12, 2018

Contents

Contents:

CHAPTER 1

Quickstart

1.1 Installation

Installation of SQLAlchemy-JSONAPI can be done via pip:

```
pip install -U sqlalchemy_jsonapi
```

1.2 Attaching to Declarative Base

To initialize the serializer, you first have to attach it to an instance of SQLAlchemy's Declarative Base that is connected to your models:

```
from sqlalchemy_jsonapi import JSONAPI

class User(Base):
    __tablename__ = 'users'
    id = Column(UUIDType, primary_key=True)
    # ...

class Address(Base):
    __tablename__ = 'address'
    id = Column(UUIDType, primary_key=True)
    user_id = Column(UUIDType, ForeignKey('users.id'))
    # ...

serializer = JSONAPI(Base)
```

1.3 Serialization

Now that your serializer is initialized, you can quickly and easily serialize your models. Let's do a simple collection serialization:

```
@app.route('/api/users')
def users_list():
    response = serializer.get_collection(db.session, {}, 'users')
    return jsonify(response.data)
```

The third argument to *get_collection* where *users* is specified is the model type. This is auto-generated from the model name, but you can control this using *__jsonapi_type_override__*.

This is useful when you don't want hyphenated type names. For example, a model named *UserConfig* will have a generated type of *user-config*. You can change this declaratively on the model:

```
class UserConfig(Base):
    __tablename__ = 'userconfig'
    __jsonapi_type_override__ = 'userconfig'
```

1.4 Deserialization

Deserialization is also quick and easy:

```
@app.route('/api/users/<user_id>', methods=['PATCH'])
def update_user(user_id):
    json_data = request.get_json(force=True)
    response = serializer.patch_resource(db.session, json_data, 'users', user_id)
    return jsonify(response.data)
```

If you use Flask, this can be automated and simplified via the included Flask module.

CHAPTER 2

Preparing Your Models

2.1 Validation

SQLAlchemy-JSONAPI makes use of the SQLAlchemy validates decorator:

```
from sqlalchemy.orm import validates

class User(Base):
    email = Column(Unicode(255))

    @validates('email')
    def validate_email(self, key, email):
        """ Ultra-strong email validation. """
        assert '@' in email, 'Not an email'
        return email
```

Now raise your hand if you knew SQLAlchemy had that decorator. Well, now you know, and it's quite useful!

2.2 Attribute Descriptors

Sometimes, you may need to provide your own getters and setters to attributes:

```
from sqlalchemy_jsonapi import attr_descriptor, AttributeActions

class User(Base):
    id = Column(UUIDType)
    # ...

    @attr_descriptor(AttributeActions.GET, 'id')
    def id_getter(self):
        return str(self.id)
```

```
    @attr_descriptor(AttributeActions.SET, 'id')
    def id_setter(self, new_id):
        self.id = UUID(new_id)
```

Note: id is not able to be altered after initial setting in JSON API to keep it consistent.

2.3 Relationship Descriptors

Relationship's come in two flavors: to-one and to-many (or tranditional and LDS-flavored if you prefer those terms). To one descriptors have the actions GET and SET:

```
from sqlalchemy_jsonapi import relationship_descriptor, RelationshipActions

@relationship_descriptor(RelationshipActions.GET, 'significant_other')
def getter(self):
    # ...

@relationship_descriptor(RelationshipActions.SET, 'significant_other')
def setter(self, value):
    # ...
```

To-many have GET, APPEND, and DELETE:

```
@relationship_descriptor(RelationshipActions.GET, 'angry_exes')
def getter(self):
    # ...

@relationship_descriptor(RelationshipActions.APPEND, 'angry_exes')
def appender(self):
    # ...

@relationship_descriptor(RelationshipActions.DELETE, 'angry_exes')
def remover(self):
    # ...
```

2.4 Permission Testing

Permissions are a complex challenge in relational databases. While the solution provided right now is extremely simple, it is almost guaranteed to evolve and change drastically as this library gets used more in production. Thus it is advisable that on every major version number increment, you should check this section for changes to permissions.

Anyway, there are currently four permissions that are checked: GET, CREATE, EDIT, and DELETE. Permission tests can be applied module-wide or to specific fields:

```
@permission_test(Permissions.VIEW)
def can_view(self):
    return self.is_published

@permission_test(Permissions.EDIT, 'slug')
def can_edit_slug(self):
    return False
```

CHAPTER 3

Serializer

CHAPTER 4

Flask

To those who use Flask, setting up SQLAlchemy-JSONAPI can be extremely complex and frustrating. Let's look at an example:

```
from sqlalchemy_jsonapi import FlaskJSONAPI

app = Flask(__name__)
db = SQLAlchemy(app)
api = FlaskJSONAPI(app, db)
```

And after all that work, you should now have a full working API.

4.1 Signals

As Flask makes use of signals via Blinker, it would be appropriate to make use of them in the Flask module for SQLALchemy-JSONAPI. If a signal receiver returns a value, it can alter the final response.

4.1.1 on_request

Triggered before serialization:

```
@api.on_request.connect
def process_api_request(sender, method, endpoint, data, req_args):
    # Handle the request
```

4.1.2 on_success

Triggered after successful serialization:

```
@api.on_success.connect
def process_api_success(sender, method, endpoint, data, req_args, response):
    # Handle the response dictionary
```

4.1.3 on_error

Triggered after failed handling:

```
@api.on_error.connect
def process_api_error(sender, method, endpoint, data, req_args, error):
    # Handle the error
```

4.1.4 on_response

Triggered after rendering of response:

```
@api.on_response.connect
def process_api_response(sender, method, endpoint, data, req_args, rendered_response):
    # Handle the rendered response
```

4.2 Wrapping the Handlers

While signals provide some control, sometimes you want to wrap or override the handler for the particular endpoint and method. This can be done through a specialized decorator that allows you to specify in what cases you want the handler wrapped:

```
@api.wrap_handler(['users'], [Methods.GET], [Endpoints.COLLECTION, Endpoints.
↪RESOURCE])
def log_it(next, *args, **kwargs):
    logging.info('In a wrapped handler!')
    return next(*args, **kwargs)
```

Handlers are placed into a list and run in order of placement within the list. That means you can perform several layers of checks and override as needed.

4.3 API

CHAPTER 5

Errors

CHAPTER 6

Indices and tables

- genindex
- modindex
- search