pg-jsonapi

Release 0.1.0.dev0

Oct 28, 2019

Table of Contents

pg-jsonapi is an asynchronous Python library for building JSON API v1.0 compliant calls using a very simple declarative syntax.

Only `PostgreSQL` is supported. `PostgreSQL` integration is powered by the asyncpgsa library. SQLAlchemy is required for describing database objects. Under the hood, the marshmallow library is used for object serialization. No previous knowledge of `marshmallow` is needed.

The user defines models that map to `SQLAlchemy` tables. Each model represents a single JSONAPI resource. Each resource has a type. A set of fields can be defined for each resource. A field can be a simple attribute mapping directly to a database column, or derived from multiple columns. The user may also define aggregate fields (ex. counts, max values, etc.). Relationship fields can be used to define relationships between models.

The library supports the fetching of resource data, inclusion of related resources, sparse fieldsets, sorting, pagination, and filtering.

CHAPTER 1

Quick Start

As an example, we create a resource model and use it to implement two basic API calls.

First we use SQLAlchemy to describe the database tables:

```
import datetime as dt
import sqlalchemy as sa

metadata = sa.MetaData()

PASSWORD_HASH_LENGTH = 128

users_t = sa.Table(
    'users', metadata,
    sa.Column('id', sa.Integer, primary_key=True),
    sa.Column('email', sa.Text, unique=True, nullable=False),
    sa.Column('created_on', sa.DateTime, nullable=False,
              default=dt.datetime.utcnow),
    sa.Column('password', sa.String(PASSWORD_HASH_LENGTH),
              nullable=False))

user_names_t = sa.Table(
    'user_names', metadata,
    sa.Column('user_id', sa.Integer, sa.ForeignKey('users.id'),
              primary_key=True, autoincrement=False),
    sa.Column('title', sa.Text),
    sa.Column('first', sa.Text, nullable=False),
    sa.Column('middle', sa.Text),
    sa.Column('last', sa.Text, nullable=False),
    sa.Column('suffix', sa.Text),
    sa.Column('nickname', sa.Text))
```

Then we define the model:

```
from jsonapi.model import Model
from jsonapi.fields import Derived
```

(continues on next page)

(continued from previous page)

```
class UserModel(Model):
    from_ = users_t, user_names_t
    fields = ('email', 'first', 'last', 'created_on'
              Derived('name', lambda rec: rec.first + ' ' + rec.last))
```

Note: The `id` and `type` fields are predefined. Attempting to do define them explicitly will raise an exception.

The primary key of the mapped table is automatically assigned to the `id` field, regardless of what the database column is called. The `type` field is determined by the value of the `Model.type_` attribute (see *Defining Models* for more details).

Note: Composite primary keys are not allowed in the mapped tables.

Note: You can define fields for a subset of the available database columns. In the example above, we chose not to expose the `password` column, for example.

Now we are ready to implement the API calls. We use the `Quart` web framework for demonstration purposes:

```
import asyncio
import uvloop
from quart import Quart, jsonify, request
from asyncpgsa import pg
from jsonapi.model import MIME_TYPE
from jsonapi.tests.model import UserModel

asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())

app = Quart('jsonapi-test')
app.config['JSONIFY_MIMETYPE'] = MIME_TYPE

@app.before_first_request
async def init():
    await pg.init(database='jsonapi',
                  user='jsonapi',
                  password='jsonapi',
                  min_size=5, max_size=10)

@app.route('/users/')
async def users():
    return jsonify(await UserModel().get_collection(request.args))

@app.route('/users/<int:user_id>')
async def user(user_id):
    return jsonify(await UserModel().get_object(request.args, user_id))

if __name__ == "__main__":
    app.run(host="localhost", port=8080, loop=asyncio.get_event_loop())
```

1.1 Example 1. Fetching a Single Object

```
GET http://localhost/users/1
    ?fields[user]=email,name
```

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

{
  "data": {
    "attributes": {
      "email": "dianagraham@fisher.com",
      "name": "Robert Camacho"
    },
    "id": "1",
    "type": "user"
  }
}
```

1.2 Example 2. Fetching a Collection of Objects

```
GET http://localhost/users/
    ?fields[user]=created-on,name,email
    &sort=-created-on
    &page[size]=10
```

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

{
  "data": [
    {
      "attributes": {
        "createdOn": "2019-10-03T16:27:01Z",
        "email": "dana58@wall.org",
        "name": "Tristan Nguyen"
      },
      "id": "888",
      "type": "user"
    },
    {
      "attributes": {
        "createdOn": "2019-10-03T11:18:34Z",
        "email": "gilbertjacob@yahoo.com",
        "name": "Christian Bennett"
      },
      "id": "270",
      "type": "user"
    },
    ...
  ],
  "meta": {
    "total": 1000
```

(continues on next page)

(continued from previous page)

```
    }
}
```

1.3 Next Steps

In the following sections we will guide you through the different features available.

CHAPTER 2

Defining Models

In this section we explore resource model definition in more detail.

To define a resource model, first declare a class that inherits from *Model* and set the `from_` attribute to an SQLAlchemy `Table` object:

```
from jsonapi.model import Model
from jsonapi.tests.db import users_t

UserModel(Model):
    from_ = users_t
```

By default, the primary key column is mapped to the reserved `id` field.

Note: Mapped tables must have single column primary keys. Composite primary key columns are not supported.

No additional attributes are required.

By default, the `type` of the resource is generated automatically from the model class name. If you wish to override it, you can set it using the *type_* attribute:

```
UserModel(Model):
    type_ = 'users'
    from_ = users_t
```

The *fields* attribute can be used to define the set of attributes and relationships for the resource. To define attributes that map directly to table columns of the same name, simply list their names as string literals:

```
class UserModel(Model):
    type_ = 'user'
    from_ = users_t
    fields = 'email', 'created_on'
```

The fields `email` and `created_on` will be created and mapped to the database columns: `users_t.c.email` and `users_t.c.created_on`, respectively.

2.1 Custom Fields

The datatype of the field is determined automatically from the datatype of the database column it maps to. To override datatype auto-detection, you can explicitly set the datatype using the *Field* class:

```
from jsonapi.fields import Field
from jsonapi.datatypes import Date

class UserModel(Model):
    from_ = users_t
    fields = ('email', 'status', Field('created_on', field_type=Date))
```

The type of `created_on` field, which maps to a timestamp database column, is set to `Date` instead of `DateTime`.

To define a field that maps to a number of columns (a column expression):

```
class UserModel(Model):
        from_ = users_t, user_names_t
        fields = 'email', Field('name', lambda c: c.first + ' ' + c.last)
```

The second argument takes a function that accepts an SQLAlchemy `ImmutableColumnCollection` representing the join result of the mapped tables listed in the `from_` attribute. The function must return a valid column expression (including function calls):

```
Field('age', lambda c: func.extract('year', func.age(c.birthday)), Integer))
```

If you wish to map a field to a database column of a different name:

```
class UserModel(Model):
    from_ = users_t
    fields = 'status', Field('email_address', lambda c: c.email)
```

You can also pass an SQLAlchemy `Column` explicitly. This is useful when mapping to multiple tables that share the same column names (see *Multiple Tables*):

```
class UserModel(Model):
    from_ = users_t
    Field('email-address', users_t.c.email)
```

2.2 Multiple Tables

A resource model can be mapped to more than one database table:

```
from jsonapi.tests.db import user_names_t

class UserModel(Model):
    from_ = users_t, user_names_t
```

In the following example, we define three fields. One maps to `users_t.c.email` column, and two map to `user_names_t.c.first` and `user_names_t.c.last` columns:

```
class UserModel(Model):
    type_ = 'user'
    from_ = users_t, user_names_t
    fields = 'email', 'first', 'last'
```

If a left outer join or an explicit on clause is needed, a `FromItem` object can be passed in place of the table object. For example:

```
from jsonapi import FromItem

UserModel(Model):
    from_ = users_t, FromItem(user_names_t, left=True)
```

2.3 Relationships

The *Relationship* class can be used to define relationships between resource models:

```
from jsonapi import ONE_TO_MANY, MANY_TO_ONE
from jsonapi.tests.db import articles_t

class UserModel(Model):
    from_ = users_t, user_names_t
    fields = ('email',
              Derived('name', lambda c: c.first + ' ' + c.last)
              Relationship('articles', 'ArticleModel',
                           ONE_TO_MANY, articles_t.c.author_id))

class ArticleModel(Model):
    from_ = articles_t
    fields = ('title', 'body', 'created_on',
              Relationship('author', 'UserModel',
                           MANY_TO_ONE, articles_t.c.author_id))
```

In this example, a `user` can author multiple `article` s, and an `article` is authored by one `user`. This relationship is represented by two *Relationship* objects, each defined in it's respective model.

The first argument to *Relationship* is the name of the field. The second arguments is the target model class name. The third argument is cardinality of the relationship.

The fourth and possibly fifth arguments depend on the cardinality of the relationship. Both arguments, if provided must be SQLAlchemy `Column` objects that are part of a foreign key and are not a primary key of any model.

- For `ONE_TO_ONE` relationships, no argument is required in most cases. The only exception is when the foreign key of the relationship lives in a standalone one-to-one join table.
- For `ONE_TO_MANY` and `MANY_TO_ONE` relationships, one argument is required.
- For `MANY_TO_MANY` relationships, two arguments are required and the second must be the one referencing the primary key of the related resource model

2.4 Aggregate Fields

The `Aggregate` class can be used to define fields that map to aggregate functions:

```
class UserModel(Model):
    from_ = users_t

    fields = ('email', 'name', 'created_on',
              Relationship('articles', 'ArticleModel',
```

(continues on next page)

(continued from previous page)

```
                        ONE_TO_MANY, articles_t.c.author_id),
                Aggregate('article_count', 'articles', sa.func.count))
```

Note: The second argument to `Aggregate` must match a name of a `Relationship` field defined in the same model, as shown above.

CHAPTER 3

Fetching Data

For a given model instance, three methods are available for fetching resource data. All three methods expect a dictionary of options as the first argument. These options represent the request query string parameters.

3.1 Single Objects

To fetch a single resource object, call the `Model.get_object()` method, supplying the object id as the second argument:

```
>>> await UserModel().get_object({}, 1)
{
    'data': {
        'id': '1',
        'type': 'user',
        'attributes': {
            'email': 'dianagraham@fisher.com',
            'first': 'Robert',
            'last': 'Camacho',
            'createdOn': '2019-05-18T11:49:43Z',
            'status': 'active',
            'name': 'Robert Camacho'}
    }
}
```

The generated SQL query may look like this:

```
SELECT users.id AS id,
       users.email AS email,
       user_names.first AS first,
       user_names.last AS last,
       users.created_on AS created_on,
       users.status AS status,
       user_names.first || ' ' || public.user_names.last AS name
```

(continues on next page)

(continued from previous page)

```
FROM public.users
JOIN public.user_names ON public.users.id = public.user_names.user_id
WHERE public.users.id = 1
```

If the resource object does not exist, a `NotFound` exception is raised:

```
>>> await UserModel().get_object({}, 1001)
Traceback (most recent call last):
  ...
jsonapi.exc.NotFound: [UserModel] object not found: user(1001)
```

3.2 Collections

To fetch a collection of objects, call the `Model.get_collection()` method:

```
>>> await UserModel().get_collection({})
{'data': [
    {
        'id': '1',
        'type': 'user',
        'attributes': {
            'email': 'dianagraham@fisher.com',
            'first': 'Robert',
            'last': 'Camacho',
            'createdOn': '2019-05-18T11:49:43Z',
            'status': 'active',
            'name': 'Robert Camacho'
        }
    },
    ...
]}
```

The generated SQL query may look like this:

```
SELECT users.id AS id,
       users.email AS email,
       user_names.first AS first,
       user_names.last AS last,
       users.created_on AS created_on,
       users.status AS status,
       user_names.first || ' ' || user_names.last AS name
FROM users
JOIN user_names ON users.id = user_names.user_id
```

3.3 Related Objects

To fetch related resource objects, call the `Model.get_related()` method, supplying the object id and the relation name as the second and third arguments:

```
>>> await UserModel().get_related({}, 276, 'bio')
{
```

(continues on next page)

(continued from previous page)

```
    'data': {
        'id': '276',
        'type': 'user-bio',
        'attributes': {
            'summary': '...',
            'birthday': '2000-04-15',
            'age': 19
        }
    }
}
```

The generated SQL query may look like this:

```
SELECT _bio__user_bios_t.user_id AS id,
       _bio__user_bios_t.summary AS summary,
       _bio__user_bios_t.birthday AS birthday,
       EXTRACT(year FROM age(_bio__user_bios_t.birthday)) AS age
FROM user_bios AS _bio__user_bios_t
WHERE _bio__user_bios_t.user_id = :user_id_1
```

Note: When fetching related (or included) objects, table aliases are used to allow the referencing of relationships involving the same model definitions (for example: self-reference relationships). This is an implementation detail and users of the library do not need to worry about it.

3.4 Sparse Fieldsets

By default, only non-aggregate attribute fields are included in the response. Aggregate fields must be requested explicitly and relationship fields are included when related resources are requested using the `include` request parameter (see *Inclusion of Related Resources*).

To include specific attributes in the response, pass a comma separated list of field names using the appropriate `fields[TYPE]` option, where `TYPE` is the `type` of the resource model:

```
>>> await UserModel().get_object({
>>>     'fields[user]': 'name,email,created-on'
>>> }, 1)
{
    'data': {
        'id': '1',
        'type': 'user',
        'attributes': {
            'email': 'dianagraham@fisher.com',
            'createdOn': '2019-05-18T11:49:43Z',
            'name': 'Robert Camacho'}
    }
}
```

You can pass an option for different resource types included in the response. For an example, see *this*.

3.5 Inclusion of Related Resources

To include related resources pass a comma separated list of relationship field names using the `include` option:

```
>>> await UserModel().get_object({
>>>     'include': 'bio',
>>>     'fields[user]': 'name,email'}, 1)
{
    'data': {
        'id': '1',
        'type': 'user',
        'attributes': {
            'email': 'dianagraham@fisher.com',
            'name': 'Robert Camacho'},
        'relationships': {
            'bio': None
        }
    }
}
```

This user has no bio record, as indicated by the `None` value of the `bio` relationship. In the case of a non-empty `bio` relationship, the value will be set to a resource identifier object and a corresponding resource object will be listed in the `included` section of the response document:

```
>>> await UserModel().get_object({
>>>     'include': 'bio',
>>>     'fields[user]': 'name,email',
>>>     'fields[user-bio]': 'birthday,age'}, 276)
{
    'data': {
        'id': '276',
        'type': 'user',
        'attributes': {
            'email': 'dbarnes@yahoo.com',
            'name': 'Bryce Price'
        },
        'relationships': {
            'bio': {
                'id': '276',
                'type': 'user-bio'
            }
        }
    },
    'included': [
        {
            'id': '276',
            'type': 'user-bio',
            'attributes': {
                'birthday': '2000-04-15',
                'age': 19
            }
        }
    ]
}
```

You can use dot notation to include nested relationships:

```
>>> await UserModel().get_object({
>>>     'include': 'articles.comments.replies,'
>>>     'articles.keywords,articles.author,articles.publisher,'
>>>     'bio'})
```

There is no limit on how many relationships can be included or nested.

3.6 Sorting

To sort a collection of resource objects, pass on the names of fields to order by as a comma separated list using the `sort` request parameter. You can use "+" or "-" prefix to indicate the sorting direction for each filed: ascending or descending, respectively. No prefix implies ascending order.

For example, the following returns a collection of `user` objects sorted by the `created-on` field in descending order:

```
>>> await UserModel().get_collection({'sort': '-created-on'})
```

To sort the collection of followers of a specific `user` by last name first, and then by first name:

```
>>> await UserModel().get_related({
>>>     'sort': 'last,first'
>>> }, 1, 'followers')
```

Collections can be sorted by aggregate fields:

```
>>> await UserModel().get_collection({'sort': '-follower-count'})
```

Sorting by a relationship field, will sort by the related resource `id` field. The following calls have identical effect:

```
>>> await ArticleModel().get_collection({'sort': 'author'})
>>> await ArticleModel().get_collection({'sort': 'author.id'})
```

You can use dot notation to specify a different attribute. To sort articles by the author name:

```
>>> await ArticleModel().get_collection({'sort': 'author.name'})
```

3.7 Pagination

To limit the number of objects returned in the `data` section of the response document, you can pass on the number as the value of the `page[size]` option:

```
>>> await UserModel().get_collection({
>>>     'page[size]': 10,
>>>     'sort': '-created-on'
>>> })
```

The above call will return the 10 most recent `user` accounts.

To return the next batch, you can set the `page[number]` option to the appropriate value (defaults to 1):

```
>>> await UserModel().get_collection({
>>>     'page[size]': 10,
>>>     'page[number]': 2,
```

(continues on next page)

(continued from previous page)

```
>>>     'sort': '-created-on'
>>> })
```

Note: If `page[number]` parameter is set without providing `page[size]`, an exception will be raised.

When pagination options are set, the `total` number of objects is provided in the `meta` section of the response document:

```
>>> await UserModel().get_collection({'page[size]': 10})
{
    'data': [...],
    'meta': {
        'total': 1000
    }
}
```

3.8 Filtering

The `filter[SPEC]` option can be used to filter a collection of objects (or related objects).

In the simplest form, `SPEC` can be the name of any field in the model. This filter would include all objects where the field has a value equal to that supplied by the filter. The value is parsed based on the datatype of the field.

For example, the following returns a list of active user accounts:

```
>>> await UserModel().get_collection({
>>>     'filter[status]': 'active'
>>> })
```

By default, the filter will use the equality operator. To specify a different operator, you can append a colin and the operator symbol to the name of the field. For example, to return all accounts created since September of 2019:

```
>>> await UserModel().get_collection({
>>>     'filter[created-on:gt]': '2019-09'
>>> })
```

The supported operators (and their symbols) and value formats depends of the field's datatype. For more details see :module:`jsonapi.datatypes`.

You can combine multiple filters, which will be AND-ed together:

```
>>> await UserModel().get_collection({
>>>     'filter[status:eq]': 'active',
>>>     'filter[created-on:gt]': '2019-09'
>>> })
```

Some datatypes accept comma-separated values. To fetch a specific list of users:

```
>>> await UserModel().get_collection({
>>>     'filter[id]': '1,2,3,6,8,9,10,11,12'
>>> })
```

Ranges are also supported. The following is equivalent to the example above:

```
>>> await UserModel().get_collection({'filter[id]': '<=3,6,>=8,12'})
>>> await UserModel().get_collection({'filter[id]': '<4,6,>7,<13'})
```

Ranges are also supported by the date and time datatypes. To return all accounts created in September of 2019:

```
>>> await UserModel().get_collection({
>>>     'filter[created-on]': '>2019-09,<2019-10'
>>> })
```

When filtering by a relationship fields, the objects are filtered by the `id` field related resource model. The following calls are equivalent:

```
>>> await ArticleModel().get_collection({'filter[author]': '1,2,3'})
>>> await ArticleModel().get_collection({'filter[author.id]': '1,2,3'})
```

You can also filter by an attribute of a related resource model:

```
>>> await ArticleModel().get_collection({'filter[author.status]': 'active'})
```

The reserved literals `none`, `null`, or `na` can be used to filter empty relationships. The values are case-insensitive. The following calls return articles with and without a publisher, respectively:

```
>>> await ArticleModel().get_collection({'filter[publisher:ne]': 'none'})
>>> await ArticleModel().get_collection({'filter[publisher:eq]': 'none'})
```

CHAPTER 4

Protecting Models

By default, models are not protected and all their objects are accessible (i.e. visible).

Protected models control access to their objects. A protected model checks access for each object to be included in the response. If access is not granted, the object is either silently excluded from the response or an appropriate exception is raised, depending on the context.

A model can be protected by setting the *access* attribute of the model to an SQL function that accepts two arguments: the `id` of the resource object and the current (logged-in) user id, and returns a boolean.

Here is an example SQL function to protect our `article` resource model:

```
CREATE FUNCTION check_article_access(
        p_article_id integer,
        p_user_id integer) RETURNS boolean
    LANGUAGE plpgsql AS
$$
BEGIN

    -- always return true for superusers
    PERFORM * FROM users WHERE id = p_user_id AND is_superuser;
    IF FOUND THEN
        RETURN TRUE;
    END IF;

    -- return true if the user is the author of the article
    PERFORM *
    FROM articles
    WHERE id = p_article_id
      AND author_id = p_user_id;
    IF found THEN
        RETURN TRUE;
    END IF;

    -- check if the user has read access
    PERFORM *
```

(continues on next page)

(continued from previous page)

```
    FROM article_read_access
    WHERE article_id = p_article_id
      AND user_id = p_user_id;
    RETURN found;
END;
$$;
```

In addition, the *user* attribute of the model must evaluate to an object representing the user in whose behalf the request is made, i.e. the current (logged-in) user:

```
class User:

    def __init__(user_id, ...):
        self.id = user_id
        ...
```

In a WSGI application, the variable holding this object must be thread-safe. For this purpose you may want to use `LocalProxy` from the `werkzeug` library:

```
from werkzeug.local import LocalProxy
from quart import g

current_user = LocalProxy(lambda: g.get('user'))
```

Note: The authentication layer should be responsible for ensuring the value of this variable is set correctly (this is outside the scope of this article).

As an example, to protect the `article` resource model, we redefine it as follows:

```
import sqlalchemy as sa
from auth import current_user

class ArticleModel(Model):
    from_ = articles_t
    fields = ('title', 'body', 'created_on', ...)
    access = sa.func.check_article_access
    user = current_user
```

If the current_user variable evaluates to `None`, access is not granted for all objects of this type, otherwise access is granted if the supplied function returns `TRUE`.

When fetching a single object, or a related object in a to-one relationship a `Forbidden` exception is raised if no user is logged in or the current user does not have access to the object in question:

```
>>> await ArticleModel().get_object({}, 1)
Traceback (most recent call last):
  ...
jsonapi.exc.Forbidden: [ArticleModel] access denied for: article(1)
>>> await ArticleModel().get_related({}, 1, 'author')
Traceback (most recent call last):
  ...
jsonapi.exc.Forbidden: [ArticleModel] access denied for: article(1)
```

When fetching a collection or related objects in a to-many relationship, objects to which access is not granted are silently excluded from the response.

CHAPTER 5

Searching

A model can be made searchable by setting the *search* attribute of the model to an SQLAlchemy `Table` object representing a fulltext search index table. The table must consist of a primary key column that references the model's primary key, and an indexed `TSVECTOR` column.

For an example, to make our `UserModel` searchable we define a text search table:

```
from sqlalchemy.dialects.postgresql import TSVECTOR

users_ts = sa.Table(
    'users_ts', metadata,
    sa.Column('user_id', sa.Integer,
              sa.ForeignKey('users.id'),
              primary_key=True),
    sa.Column('tsvector', TSVECTOR,
              index=True, nullable=False))
```

And then we redefine the model:

```
class UserModel(Model):
    from_ = users_t, user_names_t
    fields = (...)
    search = users_ts
```

As an example, the following SQL can be used to populate the index table:

```
INSERT INTO users_ts (user_id, tsvector)
SELECT users.id,
       setweight(to_tsvector(users.email), 'A') ||
       setweight(to_tsvector(user_names.last), 'B') ||
       setweight(to_tsvector(user_names.first), 'B')
FROM users
         JOIN user_names ON users.id = user_names.user_id
ORDER BY users.id;
```

5.1 A Single Model

To search a single resource model, simply pass the fulltext search term as the `search` argument of *`Model.get_collection()`* or *`Model.get_related()`* for to-many relationships:

```
>>> await UserModel().get_collection({}, search='John')
>>> await UserModel().get_related(1, 'followers', search='John')
```

The result of a search query is always sorted by the search result ranking, and any `sort` option provided will be ignored.

Filtering and searching are not compatible, and cannot be used simultaneously. Doing so will raise an exception:

```
>>> await UserModel().get_collection({'filter[id]': '1,2,3'}, search='John')
Traceback (most recent call last):
...
jsonapi.exc.APIError: [UserModel] cannot filter and search at the same time
```

5.2 Multiple Models

Use the *`search()`* function to search multiple resource models at once and return a heterogeneous collection of objects:

```
>>> from jsonapi.model import search
>>> search({}, 'John', UserModel, ArticleModel)
```

The first argument is a dictionary of options representing the request parameters. The second argument is PostgreSQL full text search query string.

Any additional arguments are expected to be model classes or instances. At least two unique models are expected.

Pagination is supported, while filtering and sorting are not. To include related resources for a model type, provide an appropriate `include[TYPE]` option. A simple `include` option will be ignored:

```
>>> search({'include[user]': 'bio',
>>>         'include[article]': 'keywords,author.bio,publisher.bio',
>>>         'fields[user]': 'name,email',
>>>         'fields[user-bio]': 'birthday,age',
>>>         'fields[article]': 'title'},
>>>         'John', UserModel, ArticleModel)
```

CHAPTER 6

API Reference

6.1 Models

`jsonapi.model.`**`MIME_TYPE = 'application/vnd.api+json'`**
The mime type to be used as the value of the Content-Type header

`jsonapi.model.`**`SEARCH_PAGE_SIZE = 50`**
The default value for the "page[size]" option when searching

`class` `jsonapi.model.`**`Model`**
A model defines a JSON API resource.

`type_`
Unique resource type (str)

See *Defining Models* for more details.

`from_`
A variable length list of tables, table aliases, or `jsonapi.db.FromItem`s.

See *Defining Models* for more details.

`fields`
A sequence of fields or field names.

See *Defining Models* for more details.

`access`
An SQL function providing object-level access protection.

See *Protecting Models* for more details.

`user`
A thread-safe object representing a logged-in user.

See *Protecting Models* for more details.

`search`
A full-text index table.

See *Searching* for more details.

get_object (*args*, *object_id*)
Fetch a resource object.

```
>>> from jsonapi.tests.model import UserModel
>>> await UserModel().get_object({}, 1)
{
    'data': {
        'id': '1',
        'type': 'user',
        'attributes': {
            'email': 'dianagraham@fisher.com',
            'first': 'Robert',
            'last': 'Camacho'
        }
    }
}
>>> await UserModel().get_object({}, email='dianagraham@fisher.com')
>>> await UserModel().get_object({}, first='Robert', last: 'Camacho'})
```

Parameters

- **args** (*dict*) – a dictionary representing the request query string
- **object_id** (*int* | *str*) – the resource object id

Returns JSON API response document

See *Fetching Data: Single Object* for more details.

get_collection (*args*, *search=None*)
Fetch a collection of resources.

```
>>> from jsonapi.tests.model import UserModel
>>> await UserModel().get_collection({})
{'data':
    [
        {'id': '1',
         'type': 'user',
         'attributes': {
             'email': 'dianagraham@fisher.com',
             'first': 'Robert',
             'last': 'Camacho',
             'createdOn': '2019-05-18T11:49:43Z',
             'status': 'active',
             'name': 'Robert Camacho'}
         },
        ...
    ]
}
```

Parameters

- **args** (*dict*) – a dictionary representing the request query string
- **search** (*str*) – an optional search term

Returns JSON API response document

See *Fetching Data: Collections* for more details.

get_related(*args*, *object_id*, *relationship_name*, *search=None*)
Fetch a collection of related resources.

```
>>> from jsonapi.tests.model import ArticleModel
>>> await ArticleModel().get_related({
>>>     'include': 'articles.comments,articles.keywords',
>>>     'fields[article]': 'title,body',
>>>     'fields[comments]': 'body'
>>> }, 1, 'author')
```

Parameters

- **args** (*dict*) – a dictionary representing the request query string
- **object_id** (*int | str*) – the resource object id
- **relationship_name** (*str*) – relationship name
- **search** (*str*) – an optional search term

Returns JSON API response document

See *Fetching Data: Related Objects* for more details.

jsonapi.model.**search**(*args*, *term*, **models*)
Search multiple models.

Returns a heterogeneous list of objects, sorted by search result rank.

```
>>> from jsonapi.model import search
>>> search({'include[user]': 'bio',
>>>         'include[article]': 'keywords,author.bio,publisher.bio',
>>>         'fields[user]': 'name,email',
>>>         'fields[user-bio]': 'birthday,age',
>>>         'fields[article]': 'title'},
>>>         'John', UserModel, ArticleModel)
```

Parameters

- **args** (*dict*) – a dictionary representing the request query string
- **term** (*str*) – a PostgreSQL full text search query string (e.x. `'foo:* & !bar'`)
- **models** (*Model*) – variable length list of model classes or instances

Returns JSON API response document

See *Searching* for more details.

6.2 From Items

class jsonapi.db.table.**FromItem**(*table*, ***kwargs*)
Represent a single item in the FROM list of a SELECT query.

Each *FromItem* item represents a table (or an alias to a table). A *FromItem* object may include an `onclause` used when joining with other *FromItem* objects. If the `left` flag is set, an outer left join is performed.

```
>>> from jsonapi.tests.db import users_t, user_names_t
>>> a = FromItem(user_names_t)
>>> b = FromItem(user_names_t, users_t.c.id == user_names_t.c.id)
>>> c = FromItem(user_names_t, users_t.c.id == user_names_t.c.id, left=True)
>>> d = FromItem(user_names_t, left=True)
>>> b
<FromItem(user_names)>
>>> b.onclause
<sqlalchemy.sql.elements.BinaryExpression object at ...>
>>> b.left
False
>>> b.name
user_names
>>> print(b)
user_names
```

__init__(*table*, ***kwargs*)

Parameters

- **table** – an SQLAlchemy Table or Alias object
- **onclause** – an onclause join expression (optional)
- **left** – if set perform outer left join (optional)

6.3 Fields

6.3.1 Basic Fields

class jsonapi.fields.**Field**(*name*, *col=None*, *data_type=None*)

Basic field type, which maps to a database table column or a column expression.

```
>>> from jsonapi.datatypes import Date
>>> from jsonapi.tests.db import users_t
>>>
>>> Field('email')
>>> Field('email-address', users_t.c.email)
>>> Field('name', lambda c: c.first + ' ' + c.last)
>>> Field('created-on', data_type=Date)
```

__init__(*name*, *col=None*, *data_type=None*)

Parameters

- **name** (*str*) – a unique field name
- **func** (*lambda*) – a lambda function that accepts a ColumnCollection (optional)
- **data_type** (*DataType*) – defaults to String (optional)

6.3.2 Aggregate Fields

class jsonapi.fields.**Aggregate**(*name*, *expr*, *func*, *field_type=Integer*)

Aggregate.**__init__**(*name*, *expr*, *func*, *field_type=Integer*)

Parameters

- **name** (*str*) – field name
- **rel_name** – relationship name
- **func** – SQLAlchemy aggregate function (ex. func.count)
- **data_type** (*DataType*) – one of the supported data types (optional)

6.3.3 Relationships

class jsonapi.fields.**Relationship**(*name*, *model_name*, *cardinality*, **refs*, ***kwargs*)

Represents a relationship field.

```
>>> from jsonapi.model import ONE_TO_MANY
>>> from jsonapi.tests.db import articles_t
>>>
>>> Relationship('articles', 'ArticleModel', ONE_TO_MANY,
>>>              articles_t.c.author_id)
```

__init__(*name*, *model_name*, *cardinality*, **refs*, ***kwargs*)

Parameters

- **name** (*str*) – relationship name
- **model_name** (*str*) – related model name
- **cardinality** (Cardinality) – relationship cardinality
- **refs** – a variable length list of foreign key columns

Cardinality

class jsonapi.db.table.**Cardinality**

The cardinality of a relationship between two models.

- ONE_TO_ONE
- MANY_TO_ONE
- ONE_TO_MANY
- MANY_TO_MANY

Python Module Index

j

Index

Symbols

A

C

F

G

J

M

R

S

T

U