RESTAlchemy Documentation

Release 0.1

Daniel Kraus

Jan 20, 2019

Contents:

CHAPTER 1

Front Matter

A REST framework build on top of Pyramid and SQLAlchemy.

If you use the SQLAlchemy ORM for your models, you already specify the type of the column, if it's nullable, a default value and other properties.

So instead of duplicating what you already specified in your SQLAlchemy models and writing validation for your REST API again either from scratch or with another utility like *marshmallow*, RESTAlchemy inspects your models and derives the API from this.

Of course, sometimes you want to return something else for some attribute or restrict access to a resource etc. For that, you have many special RESTAlchemy attributes or functions that you can use on your model.

You don't have to jump between your view implementation, the model and validation code. Everything is right there at your model. (Of course, you can still split everything up if that's what you prefer).

1.1 Goals

- DRY

  Most things you need for your REST API is already specified in your sql model. What type, what values are allowed, is it nullable or required, default values, relationships, etc.

  There is no need to write extra validation or rendering code.

- Simple to use and extend yet very powerful and flexible

  For those attributes that you want to handle differently, you can easily do that by adding extra methods (or variables) to your models without much boilerplate.

  It's really easy to only return certain attributes on some conditions or write your own create/change/query functions.

- Provide a simple and easy to use REST Interface

  With a very quick look at a few API calls, the user should be able to figure out how to query stuff without much thinking and the URL schema and query parameters should feel "natural".

You can easily query a resource with some relationships expanded without having to do multiple calls or stitching your model back together like in most other REST frameworks.

1.2 Project Info

The python *restalchemy* package is hosted on github.

Releases and project status are available on Pypi.

The most recent published version of this documentation is at http://restalchemy.readthedocs.org/en/latest/index.html.

1.3 Support

For questions and general discussion, join our mailing list.

For feature requests or bug reports open a GitHub issue.

Check the website for updates.

1.4 Installation

1.4.1 Existing app

If you already have a Pyramid app and want to extend your app with REST functionality, simply

```
$ pip install restalchemy
```

And add RESTAlchemy to your `pyramid.config.Configurator` object with *config.include("restalchemy")*.

You have to tell RESTAlchemy how to find your models, for that you need to set a function with `config.set_set_model_function()` that takes a string (which is the model name as part of the URL) and returns your model class or *None*. You can check this function from the cookiecutter template for inspiration.

So at minimum you have to add this to your config:

```
config.include("restalchemy")
config.set_get_model_function("yourapp.utils.get_model")
```

See *Configuration* for more infos.

1.4.2 New app

If you want to start a fresh project, it's best to use cookiecutter:

```
$ cookiecutter gh:restalchemy/cookiecutter-restalchemy
```

and follow the steps show from this command.

CHAPTER 2

Quickstart

The quickest way to get started with a new project is to use the RESTAlchemy **cookiecutter_**

```
cookiecutter gh:restalchemy/cookiecutter-restalchemy
```

and follow the steps shown from this command.

You can then add your SQLAlchemy models in the *models* folder and import them in *models/__init__.py*.

RESTAlchemy will automatically provide endpoints to query your models (HTTP GET) with filter parameters, relationship expansion, etc and endpoints for model creation (POST), update (PUT), deletion (DELETE).

The template has only a single *User* model by default that you can query like:

```
curl localhost:6543/v1/users
```

Which would return a list of users in your system:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8

{
    "success": true,
    "timestamp": "2019-01-17T15:13:49.234368+00:00",
    "sort": null,
    "offset": 0,
    "limit": 100,
    "filter": [],
    "count": 1,
    "previous": null,
    "next": null,
    "resource": "users",
    "users": [{
        "created_at": "2019-01-16T17:12:33+00:00",
        "email": "user@example.com",
        "id": 1,
```

(continues on next page)

(continued from previous page)

```
        "name": "User",
        "updated_at": null
    }]
}
```

CHAPTER 3

Tutorial

Coming soon. . .

CHAPTER 4

REST API

4.1 Overview

RESTAlchemy provides a very intuitive and simple but still very powerful REST API.

It's designed for 95% of usages of JSON REST APIs, which is to give the user easy access to the data storage of your application from a browser or other apps.

See *comparisons* to see how RESTAlchemy calls look compared to other popular frameworks.

4.2 API Endpoints

RESTAlchemy uses the common REST API scheme for URLs where you can access the collection of your resource (on Python side, this would be a list of your SQLAlchemy models) under */{resource}*. You can access one particular resource (in Python that would be a single instance of your model) with */{resource}/{ID}* and you can access an attribute from this resource (which can again be a resource or a collection or resources) with */{resource}/{ID}/{attribute}*.

It's also convention to include the API version in the URL, so the first path segment must always be the API version. E.g. */v1/*.

Quick overview of URL endpoints (need to be prefixed with version number):

HTTP GET to query:

- */{resource}*: Return a list of resources.
- */{resource}/{ID}*: Return a resource
- */{resource}/{ID}/{attribute}*: Return attribute of a resource

HTTP POST to create:

- */{resource}*: Create a new resource
- */{resource}/{ID}/{attribute}*: Only possible if *{attribute}* is list of resources. Then it will append a newly created resource.

HTTP PUT to update:

- */{resource}/{ID}*: Update existing resource
- */{resource}/{ID}/{attribute}*: Update attribute of existing resource

HTTP DELETE to delete:

- */{resource}/{ID}*: Delete resource
- */{resource}/{ID}/{attribute}*: Only possible if *{attribute}* is a relation to another resources. Then it will delete the resource and reference to it.

4.3 Query parameters

- `limit`: Limit the number of entries to return. (default: 100; default maximum limit is 1000)
- `offset`: Number of entries to skip. (default: 0)
- `sort`: attribute to sort by in descending order. You can optionally specify the sort order by appending .asc or .desc to the attribute (default: id.asc). You can also sort by sub-attributes, e.g. GET `/v3/creatives?sort=category.name.asc` or by quickstats, e.g. GET `/v3/campaigns?quickstats&sort=quickstats.lifetime.clicks.asc`. If you specify a relationship model without specifying an attribute of it you sort by count. E.g. GET `/v3/publishers?sort=sites.desc&limit=5` returns the 5 publishers that have the most sites.
- `depth`: Specifies how attributes that are relationships to another model or a list of other models is returned. There are 3 options:
  - depth=0: don't return any relationship attributes at all
  - depth=1: return a list of attribute IDs (default)
  - depth=2: return the expanded attribute objects
- `attributes`: comma separated list of attributes you want to have included in your result. You can also exclude certain attributes bei prepending '!' to the attribute name. (not set as defaults and all attributes are returned) e.g. get sites but only return id and names without anything else: HTTP GET to `/v3/sites?attributes=id,name` or get advertiser with id 3 but without campaigns and creatives attribute: `/v3/advertiser/3?attributes=!creatives,!campaigns`
- `expand`: comma separated list of attributes to expand (instead of only showing the ID(s)). Useful if you don't want to set *depth* to a higher value because you only want one or a few attributes expanded or you set depth to 2 already and want to expand one attribute a level deeper. (not set as default) E.g. get all sites and also expand the domains `/v3/sites?expand=domain`
- `attribute filter`: every attribute other then the above (limit, offset, sort, depth, attributes, expand) is used as a filter for the result set. The URL parameter in general looks like `attribute_to_filter=filter_string` If `attribute_to_filter` is starting with *!* the filter is negated. `filter_string` can be a comma separated list of multiple values or contain * as wildcard for matches in strings. (no filter set as default) E.g. find all .mx TLDs `/v3/domains?hostname=*.mx`

4.4 Special Endpoints

If you use the authentication module from RESTAlchemy, you get a special */login* endpoint to receive an auth token. What JSON you exactly have to post to this API is depending on the implementer. See authentiaction for more details.

4.5 Custom endpoints or query parameters

RESTAlchemy gives you the full power of Pyramid, so it's easy to overwrite the default URL routes or query parameters or add your own.

See *Configuration* for a way to change routes/query parameters or create new ones.

4.6 Examples

4.6.1 Usage with curl

To login and receive the authentication token:

```
$ curl -X POST -d '{"email": "test@example.com", "password": "test"}' -H "Content-
↪Type: application/json" localhost:6543/v1/login
```

A sample response would be:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Date: Tue, 28 Oct 2014 11:37:25 GMT

{
    "auth_token": "your-long-token-here",
    "message": "test (user id 1) logged in",
    "success": true,
    "user": {
        "created_at": "2014-09-09T15:34:56",
        "email": "test@example.com",
        "id": 1,
        "name": "test",
        "updated_at": "2014-10-14T19:01:11"
    }
}
```

Now you can pass the auth token in the header of your next request(s) to access more resources. To do so add an 'Authorization' Header with 'Bearer ' + `auth_token` as value.

E.g. get all sites:

```
$ curl -H 'Authorization: Bearer your-long-token-here' localhost:6543/v1/todos
```

Would result in a json response that lists all TODOs available for the test@example.com user:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8

{
    "filter": [],
    "limit": null,
    "offset": null,
    "todos": [
        {
            "id": 5,
            "todo": 'todo five',
```

(continues on next page)

(continued from previous page)

```
            "description": "todo description",
            "created_at": "2014-10-29T17:36:42",
            "updated_at": "2014-10-29T17:38:25"
        },
        {
            "id": 9,
            "todo": 'todo nine',
            "description": "todo description",
            "created_at": "2014-10-29T18:36:42",
            "updated_at": "2014-10-29T18:38:25"
        },
        /* {... more todos */ ...}
    ],
    "sort": "id.asc",
    "success": true,
    "timestamp": "2015-04-03T00:14:35.072516"
}
```

To create a new entry you have to POST with the necessary data you want to set. E.g. creating a new *todo*:

```
$ curl -H 'Authorization: Bearer your-long-token-here' -X POST -d '{"todo": "test todo
↪", "description": "test description"}' -H "Content-Type: application/json"␣
↪localhost:6543/v1/todo
```

Would create a new todo and the response would look like:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8

{
    "todo": {
        "id": 23,
        "todo": "test todo",
        "description": "test description"
        "created_at": "2014-10-28T21:56:44",
    },
    "status": "OK"
}
```

4.6.2 API Client Libraries

- Python: https://github.com/restalchemy/restalchemy-client-python
- Emacs: https://github.com/restalchemy/restalchemy-client-emacs
- TODO JavaScript: https://github.com/restalchemy/restalchemy-client-javascript
- TODO Go: https://github.com/restalchemy/restalchemy-client-go

4.7 Comparisons

Let's look at some common queries and there outputs with RESTAlchemy, Eve, JSON API and Django Rest Framework

TODO

CHAPTER 5

Configuration

5.1 CORS

TODO

5.2 Routes and views

TODO

5.3 Renderer

TODO

5.4 Response

TODO

5.5 Validators

TODO

CHAPTER 6

Roadmap

- Implement query caching
- Return and respect cache control headers (ETag; Last-Modified)
- Automatically create OpenAPI (swagger) specification

  This can then be used to quickly create pretty documentation.
- More RESTAlchemy clients
- Improve documentation

CHAPTER 7

API

CHAPTER 8

restalchemy changelog

8.1 0.1 (unreleased)

- First release

CHAPTER 9

Indices and tables

- genindex
- modindex
- search