Restosaur Documentation

Release 0.6.7

Marcin Nowak

Mar 30, 2017

Contents

A nimble RESTful library for any Python web framework.

CHAPTER 1

About

What Restosaur is?

Restosaur is a RESTful foundation library for Python, which aims to be a transparent layer for building REST-like or RESTful services, adaptable to any web framework.

The key concepts are:

- make library layer as transparent as possible,
- operate on object of any type as a model,
- focus on resources and their representations,
- base on content types as a client-server contract machinery,
- do not force an API developer to use any specific structures, patterns, solutions,
- maximize flexibility of constructing control flow and data factories,
- provide clean interface, sane defaults and handy toolkit,
- make library independent from any web framework and provide adapters to common web frameworks,
- follow *explicit is better than implicit* rule (see PEP20) – no magic inside.

Note: Up to v0.8 Restosaur is a Django-only project for historical reasons. Please follow *Roadmap* for details.

What Restosaur is not?

Restosaur is not a framework. There are no batteries included. There are no paginators, CRUD mappers, authenticators, permission system nor autogenerated HTML representation (which is a bad concept at all).

Every REST-like or RESTful API may be very specific. As an API author you shold have possibility to do anything you want and use any tool you need.

If you're using a web framework, you have decent tools already. Restosaur just helps you to adapt your data and business logic to a RESTful service.

CHAPTER 2

Quickstart

Note: Up to v0.8 Restosaur is a Django-only project for historical reasons, and this Quickstart guide is based on a Django project. Please follow *Roadmap* for details.

Installation

Restosaur is hosted on PyPI. Use `pip`, `easy_install` or any similar software to install it:

```
pip install restosaur
```

Prepare project

To start work with Restosaur is good to create core module for your API, for example:

```
touch <myproject>/webapi.py
```

And fill the `webapi.py` file with:

```
import restosaur

# import handy shortcuts here
from django.shortcuts import get_object_or_404  # NOQA

api = restosaur.API()
```

Configure Django

To setup Restosaur with Django, follow these steps:

- Add `restosaur` to `INSTALLED_APPS` in your settings module
- Include your API url patterns in main `urls.py` module

Example of the `urls.py` module:

```
from django.conf.urls import url
from webapi import api

urlpatterns = [...]  # Your other patterns here
urlpatterns += api.urlpatterns()
```

If your project is based only on Restosaur, just write:

```
urlpatterns = api.urlpatterns()
```

Note: For Django <1.7 you must call autodiscover explicitely. The good place is your `urls.py` module:

```
from django.conf.urls import url
from webapi import api

import restosaur
restosaur.autodiscover()  # Before api.urlpatterns() call

# ... rest of urls.py file...
```

Build your API

Let's assume you're creating an another Blog project and the app name is called `blog`. In your `blog/models.py` you have Post model defined as:

```
class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
```

Initializing API module

First, create a `restapi.py` module, which will be autodiscovered by default:

```
touch blog/restapi.py
```

Then import your `api` object, handy shortcuts and your models:

```
from webapi import api, get_object_or_404
from .models import Post
```

Creating resources

Now create a resources - first for list of Posts and second for Post detail.

Note: There is no difference between collection and detail - both are just resources but their controller logic and representation differs.

To do that use `resource()` factory and provide URL template fragment as an argument:

```
post_list = api.resource('posts')
post_detail = api.resource('posts/:pk')
```

Registering HTTP services

Now you have two variables - instances of `Resource` class. Resource is a container for HTTP method callbacks, and they can be registered using decorators. You have to choose from:: `.get()`, `.post()`, `.put()`, `.patch()`, `.delete()` and `.head()`.

Let's create a callback (a controller/view) for Posts list, which will be accessible by HTTP GET method. The callback takes at least one argument - a `context`, which is similar to request object. The callback must return a Response object:

```
@post_list.get()
def post_list_view(context):
    return context.Response(Post.objects.all())  # 200 OK response
```

Response takes at least one argument - a data object. It may be anything. The data object will be passed "as is" to representation factories. In the example above we're passing a Post's queryset object.

Representations

Now there is time to register a representation factory. The return value must be serializable by content type serializer. In our case we will use plain Python `dict` which will be passed internally to `JsonSerializer` (the default).

The representation factory callbacks takes two positionl arguments:

- a data object returned from controller / view in a Response,
- a context.

Let's register a representation factory for Posts list:

```
@post_list.representation()
def posts_list_as_dict(posts, context):
    return {
        'posts': [post_as_dict(post, context) for post in posts]
    }
```

As you can see Posts list representation factory uses a `post_as_dict()` method. There is no magic, so you must implement it:

```
def post_as_dict(post, context):
    return {
            'id': post.pk,
            'title': post.title,
            'content': post.content,
            }
```

Note: Representation factories takes two positional arguments: data object and the context. There is a good practice to define helper functions in that way. The context contains request state, which may be used for checking permissions, for example, and provides tool for creating links between resources. You may consider making context optional:

```
def post_as_dict(post, context=None):
   # ...
```

Reusing respresentation factories

Now let's create a Post's detail controller and bind to HTTP GET method of a `post_detail` resource:

```
@post_detail.get()
def post_detail_view(context, pk):
    return context.Response(get_object_or_404(Post, pk=pk))
```

The implementation is very similar to Posts list controller. We're returning a data object, which is a Post model instance in our case. There is a second argument defined in our callback (it's name is taken from URI template, a `:pk` var). And we're raising `Http404` exception when Post is not found.

Note: Restosaur catches `Http404` exception raised by `get_object_or_404` and converts it to `NotFoundResponse` internally. This is quite handy shortcut.

We can now create a representation for Post detail. But please note that we have one already! This is a `post_as_dict` function. So in that case we need just to register it as a representation:

```
@post_detail.representation()
def post_as_dict(post, context):
    # ...
```

Linking to resources

REST services are about representations and relations between them, so linking them together is fundamental. The links can be cathegorized as internal and external. Internal links are handled by Restosaur, but external links may be just URIs passed as a strings.

Let's complete the Post's representation by adding a URIs of every object. We'll use `context.url_for()` method to generate them:

```
context.url_for(post_detail, pk=post.pk)
```

You need just to add this call to `post_as_dict` factory:

```
@post_detail.representation()
def post_as_dict(post, context):
    return {
            'id': post.pk,
            'title': post.title,
            'content': post.content,
            # create link (URI) to this object
            'href': context.url_for(post_detail, pk=post.pk),
            }
```

Note: Linking resources by passing URI without HTTP method nor additional description is insufficiet to build really RESTful service. Restosaur allows you to do anything you want, so you may create own link factories and use them in yours representaion factories. For example:

```
def json_link(uri, method='GET', **extra):
    data = dict(extra)
    data.update({
        'uri': uri,
        'method': method,
        })
    return data


@post_detail.representation()
def post_as_dict(post, context):
    return {
        'id': post.pk,
        'title': post.title,
        'content': post.content,
        'link': json_link(context.url_for(post_detail, pk=post.pk)),
        }
```

Just place `json_link` helper in your core `webapi.py` module and import it when needed:

```
from webapi import api, get_object_or_404, json_link
```

Complete example of the module

```
from webapi import api, get_object_or_404
from .models import Post

# register resources

post_list = api.resource('posts')
post_detail = api.resource('posts/:pk')


# register methods callbacks

@post_list.get()
def post_list_view(context):
    return context.Response(Post.objects.all())  # 200 OK response


@post_detail.get()
def post_detail_view(context, pk):
    return context.Response(get_object_or_404(Post, pk=pk))
```

```
# register representation factories

@post_detail.representation()
def post_as_dict(post, context):
    return {
            'id': post.pk,
            'title': post.title,
            'content': post.content,
            # create link (URI) to this object
            'href': context.url_for(post_detail, pk=post.pk),
            }


@post_list.representation()
def posts_list_as_dict(posts, context):
    return {
            'posts': [post_as_dict(post, context) for post in posts]
        }
```

Test your API

- Start your Django project by calling:

  ```
  python manage.py runserver
  ```

- Add some posts by admin interface or directly in database
- And browse your posts via http://localhost:8000/posts

Making resources private

You may want to make some of your resources private, especially when your controllers require a logged `user` instance.

To achieve that you'll need to use a `login_required` decorator and wrap your controllers/views with it. Add to your main `webapi.py` module:

```
from restosaur.decorators import login_required
```

import decorator in your `blog/restapi.py` at the top of the module:

```
from webapi import login_required
```

and wrap your controllers with it:

```
@post_list.get()  # must be outermost decorator
@login_required
def post_list_view(context):
    # ...


@post_detail.get()  # must be outermost decorator
@login_required
```

```
def post_detail_view(context, pk):
    # ...
```

Note: The `login_required` decorator will be moved to `restosaur.contrib.django.decorators` module in the future (from v0.8). After upgrading you will need to change just one import in your core `webapi.py` module.

Accessing the request object

The original request object will be always delivered as a `context.request` property. In our casue it will be an original Django `WSGIRequest` instance.

Context properties

Restosaur's context delivers unified request data. You can access query parameters, the payload, uploaded files and headers.

`context.parameters` An URI query parameters wrapped with `QueryDict` dict-like instance.

`context.body` Deserialized request payload

`context.raw` Original request payload

`context.files` Uploaded files dictionary (depends on framework adapter)

`context.headers` Dictionary that contain normalized HTTP headers

`context.request` Original HTTP request object, dependent on your web framework used

Response factories

Context object delivers factories for common response types:

- `context.Response()` – `200 OK` response
- `context.Created()` – `201 Created` response
- `context.NoContent()` – `204 No Content` response
- `context.SeeOther()` – `303 See Other` response
- `context.NotModified()` – `304 Not Modified` response
- `context.BadRequest()` – `400 BadRequest` response
- `context.Unauthorized()` – `401 Forbidden` response
- `context.Forbidden()` – `403 Forbidden` response
- `context.NotFound()` – `404 Not Foud` response

Other statuses can be set by `context.Response()`, for example:

```
return context.Response(data, status=402)
```

Extending the context

Restosaur has initial support for middlewares. You can use them to extend the context object as you need.

Middlewares are simple classes similar to Django's middlewares. You can define set of middlewares in your `restosaur.API` instance.

For example, let's add an `user` property to our context. To do that extend your `webapi.py` core module with:

```
class UserContextMiddleware(object):
    def process_request(self, request, context):
        context.user = request.user
```

and change your API object initialization to:

```
api = restosaur.API(middlewares=[UserContextMiddleware()])
```

Now you'll be able to access the `user` via `context.user` property. In our case it will be a Django `User` or `AnonymousUser` class instance.

Note: The main advantage over Django middlewares is that the middlewares can be set for every `API` object independely. Your web application server may handle different middlewares depending on your requirements. This is very important for request-response processing speed.

Permissions

Your API services may be accessible only for:

- a specified group of users – controller/view level permissions,
- the data might be limited – object level permissions,
- and a representations may be limited – content level permissions.

Restosaur allows you to use any method and does not force you to do it in a specified way.

Controller/view level permissions

You may decorate any controller/view with a decorator which checks user's permissions.

Restosaur provides `staff_member_required` decorator as an example of Django's decorator of same name. You need to import it into `webapi.py` module:

```
from restosaur.decorators import staff_member_required
```

import it to your `blog/restapi.py` module:

```
from webapi import staff_member_required
```

and just wrap your callbacks with it:

```
@post_list.get()  # must be outermost decorator
@login_required
@staff_member_required
```

```
def post_list_view(context):
    # ...
```

Note: The `staff_member_required` decorator will be moved to `restosaur.contrib.django.decorators` module in the future (from v0.8). After upgrading you will need to change just one import in your core `webapi.py` module.

Object level permissions

In that case you should wrap your data generation within views/controllers with a desired filter.

Let's say that some users should not access a Posts with titles starting with a "X" letter. Create filter somewhere, i.e. a `blog/perms.py` file:

```
def posts_for_user(user, posts):
    if not user.has_perm('can_view_posts_starting_with_X_letter'):
        posts = posts.exclude(title__startswith='X')
    return posts
```

Then wrap your Posts queryset with that filter:

```
from . import perms


def get_user_posts(user):
    '''Returns a Posts queryest available for a specified user'''
    return perms.posts_for_user(user, Posts.objects.all())


def post_list_view(context):
    return context.Response(get_user_posts(context.user))


def post_detail_view(context, pk):
    posts = get_user_posts(context.user)
    return context.Response(get_object_or_404(posts, pk=pk))
```

Limiting representation data

Let's assume that non-admin users can't view Posts content.

To do that you can extend your `blog/perms.py` with a helper function:

```
def can_view_post_content(user):
    return user.is_superuser
```

Now modify your representation factory:

```
def post_as_dict(post, context):
    data = {
            'id': post.pk,
            'title': post.title,
            }
    if perms.can_view_post_content(context.user):
        data.update({
            'content': post.content,
            })
    return data
```

CHAPTER 3

Development status

Restosaur is currently in alpha stage, but it is pretty stable. API may be changed a little in newer versions, especially from v0.7.

Restosaur v0.6 is used in some production environments and works without any problems, although it is lacking some functionality.

The unit tests does not cover 100% of the code, currently.

As always – use it at your own risk.

CHAPTER 4

Integrations

Restosaur v0.6 works only with Django. The supported versions are:

- Django 1.6
- Django 1.7
- Django 1.8
- Django 1.9
- Django 1.10 (beta1)

Plain WSGI interface will be supported starting from v0.8. Built-in support for other web frameworks will be added in v1.0. It is possible that web framework adapters will be provided as a separate Python packages.

CHAPTER 5

Roadmap

0.6 (alpha, current)

- Django-only
- A prototype of a final interface
- Basic represenation support and content negotiaion

0.7 (beta)

- stable representations and services API
- removed obsolete code
- better test coverage
- enhanced content negotiation and requests dispatching
- added final middleware support

0.8 (beta)

First web framework independent release:

- added wsgi interface
- moved Django adapter to contrib module
- moved Django helpers to contrib module

0.9 (beta)

- Python 3.x support

1.0

- stable API
- ~100% test coverage
- adapters for common web frameworks
- Python 2 and Python 3 support
- complete documentation

CHAPTER 6

Contribution

The repository

Restosaur is hosted on GitHib – https://github.com/marcinn/restosaur

If you would like to contribute, please create issues on GitHub, make a fork and send pull requests.

Coding standards

- Please follow PEP8 guidelines

CHAPTER 7

Indices and tables

- genindex
- modindex
- search