Restosaur Documentation

Release 0.7-dev

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.

Restosaur helps you to adapt your data and business logic to a powerful RESTful service.

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.

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

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.

Linking to resources

We'll use `context.link()` method to generate URL for a Post instance detail view:

```
context.link(post_detail, post)
```

Note: This will generate a URL for the `post_detail` resource, which has defined an URL template as `posts/:pk`. The `:pk` variable will be read from `post` instance.

The only rule is that Restosaur expects `pk` to be an object's property or a key/index.

This is an equivalent of:

```
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.link(post_detail, post),
            }
```

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.link(post_detail, post)),
        }
```

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
```

Linking to models

Restosaur gives a possibility to register link views for your models. This approach is a next layer of encapsulation and DRY improvement.

The low-level `context.url_for()` method requires a resource and path's specific arguments to generate the URL. There is no encapsulation at all, and DRY is broken.

The `context.link()` shortcut encapsulates URL generation by passing resource and model instance as arguments. You don't need to repeat URL arguments.

And finally `context.link_model()` shortcut encapsulates URL generation by referencing directly to the model instance or class. You don't need to provide resource nor argument at all. This level of resource linking encapsulation provides best DRY principles.

The `context.link_model()` requires model view registration. This can be done several ways:

- using a resource class decorator shourcut – `resource.model(ModelClass)`
- using an API instance – `api.register_view(ModelClass, resource)`
- using a class decorator on the model class – `@api.view(resource)`

Example of using a resource shourtcut:

```
@post_detail.model(Post)
class Post(models.Model):
    pass
```

Example of using an API instance:

```
api.register_view(Post, post_detail)
```

Example of using an API class decorator:

```
@api.view(post_detail)
class Post(models.Model):
    pass
```

Note: Buiding complex API you may split it into many modules. In that cases there is a high risk of circular imports problem.

Linking shortcuts are designed to avoid import problems and selecting a way of registering view for the model is highly dependent on specific case.

To avoid circular import problems you may also pass dotted resource path instead of resource instance:

```
api.register_view(Post, 'blog.restapi.post_detail')

# or using a decorator:

@api.view('blog.restapi.post_detail')
class Post(models.Model):
    pass
```

Note: Model can be an object of any type, not only Django's `django.db.Model`. There is no limitation.

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.link(post_detail, post),
            }


@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.contrib.django.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):
    # ...
```

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 shortcut factories for common response types:

- `context.OK()` – `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 indepedely. 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.contrib.django.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):
    # ...
```

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

The Guide

Glossary

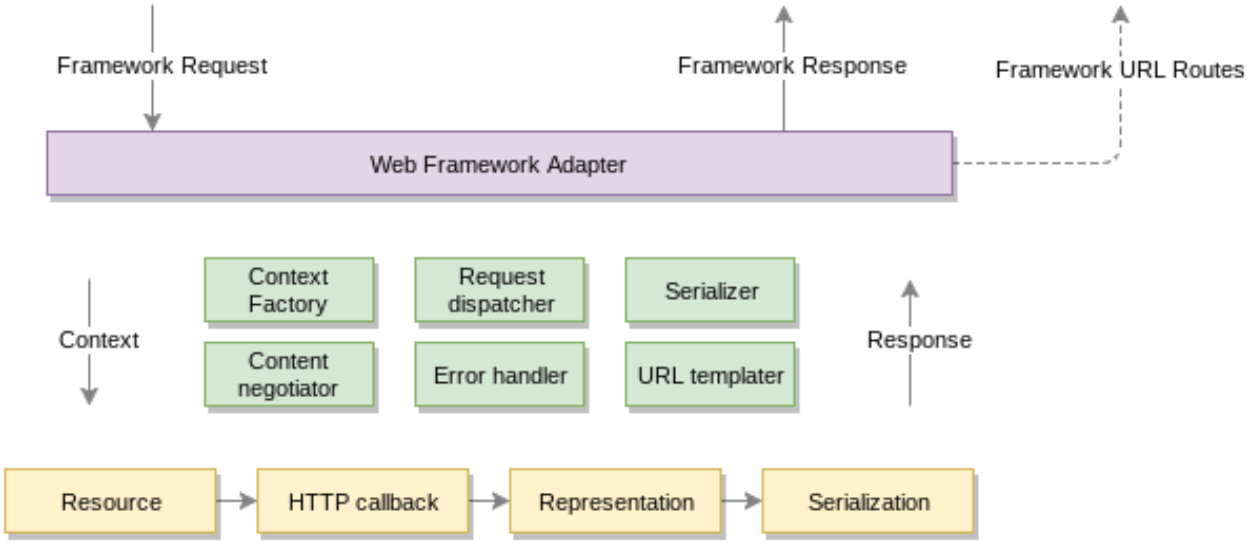
Benefits of using Restosaur

Restosaur's internals

Architecture

Restosaur is a library which provides tools for implementing RESTful server in a Python programming language. It also provides an Integration Layer which connects Restosaur's internals to existing web frameworks through a Web Framework Adapters.

The process of adaptation transforms original HTTP requests into internal Context objects, and transforms internal Response objects into web framework HTTP responses. Web Framework Adapter connetcs URLs and HTTP callbacks to the framework's routing system (just once at the startup stage).



The Context

A `Context` is a class which represents a HTTP request data. It is similar to request objects of web frameworks, but may mutate during dispatching. In such cases it may not be identical to a HTTP request. That's why it is named differently.

A Context object holds an original request object, reference to the API instance, matched resource object, normalized headers, normalized HTTP method name, GET, POST and FILES parameters, and of course the payload of the request (raw and deserialized).

The context instance is passed to every object during dispatch process, so every part of a pipeline can access not only for request data, but also for proper API and resource instances.

Context class defines handy response factories (shortcuts).

Resources

Resource is an object, which manages one and the only one unique URI, and allows clients to read and modify it's state.

A resource instance can handle GET, POST, PUT, PATCH, DELETE and OPTIONS methods. By default there are no callbacks registered. If no callback is registered for a requested method, the `405 Method Not Allowed` response will be automatically generated.

A method callbacks are similar to controllers or views, known well from MVC or MVT paradigms, and they must always return a Response instance.

Responses

Responses are objects which holds information required to generate final HTTP response. They holds a HTTP response code, context instance, response HTTP headers, response content type and optional resource state object.

Restosaur defines common types of responses:

Class	Context shortcut	Code
responses.OKResponse	ctx.OK()	200
responses.CreatedResponse	ctx.Created()	201
responses.NoContentResponse	ctx.NoContent()	204
responses.SeeOtherResponse	ctx.SeeOther(uri)	303
responses.NotModifiedResponse	ctx.NotModified()	304
responses.BadRequestResponse	ctx.BadRequest()	400
responses.UnauthorizedResponse	ctx.Unauthorized()	401
responses.ForbiddenResponse	ctx.Forbidden()	403
responses.NotFoundResponse	ctx.NotFound()	404
responses.MethodNotAllowedResponse	ctx.MethodNotAllowed()	405
responses.NotAcceptableResponse	ctx.NotAcceptable()	406
responses.UnsupportedMediaTypeResponse	ctx.UnsupportedMediaType()	415
responses.InternalErrorResponse	–	500
responses.NotImplementedResponse	–	501

API objects

Web Framework Adapters

Data flow

Note: The facts worth remembering:

- HTTP callbacks are responsible for reading and changing a resource state.
- Representation factories are responsible for transforming the state into transport structure.

Let's assume you have a `Car` class and a `car_detail` resource, which is responsible for managing a single car identitfied by `plate_number`. A `GET` method callback of the resource is responsible for reading a `Car` state from the database. The resource has a registered representation for an `application/json` type and a `Car` class, which is responsible for translating `Car` instance into pure Python's dict.

When client requests a details of a car, in a `application/json` format, a `GET` callback is called. After handling a HTTP method Restosaur begins a content negotiation procedure. The `application/json` callback is found as the best matching and `Car` translates into a dict. Then a dict object is simply serialized for a streaming.

Preparing the environment

Initializing the API instance

Restosaur is designed to support multi-apps project, so at least one API instance is required to start working with the library.

There is no special place nor style to do it, but a common practice is to create a decicated module or package with your's API internals. The typical name is just `webapi.py`. The only one requirement is that your module/package must be importable.

The `restosaur` package contains common objects imported to it's namespace. This is a top level interface of a package. In most cases you'll need just one import, or just few flat-style imports directly from `restosaur`. A rare exceptions are related to a contrib packages.

Note: It is very handy to make your `webapi` module/package a top-level interface of the API. Just import there common objects from a `restosaur` namespace and define or import custom helpers.

The common rule of thumb is to avoid direct imports from a `restosaur` namespace in a rest of your code.

Let's make an API instance by importing adequate class. For Django it will be `restosaur.contrib.django.API`:

```
from restosaur.contrib.django import API

webapi = API('api')
```

The `webapi` object holds now API instance, which is responsible mostly for managing the API resources, serializers registry, models and their views registry, and configured middlewares.

The interface of an API initializer is:

```
restosaur.api.API.__init__(
    path=None, middlewares=None, context_class=None,
    default_charset=None, debug=False)
```

path An URL prefix of your API instance (relative).

Leave it empty if you want to bind the API instance as a root resource.

middlewares iterable of instantiated middlewares

context_class a class used by internal context factory

default_charset the default charset of the text/html output (default: `utf-8`)

debug a flag that turns on debug mode

Note: API instance holds no resources by default. If you want to make an API root view, you must create a resource explicitely:

```
root = webapi.resource('/')
```

Creating a resources

To create a resource you'll need to use a factory method:

```
webapi.resource(path, *args, **kwargs)
```

This is a factory of the `restosaur.resource.Resource` class. Resource class holds callbacks for HTTP methods and representations registry. It describes how to read and manage the resource's state, and how to build the state representation understandable for clients.

path required resource's path, relative to the API's path.

Let's create two resources, one for a list (collection) and second for a details view:

```
car_list = webapi.resource('cars')
car_detail = webapi.resource('cars/:plate_number')
```

As a result a two endpoints will be available as a URL patterns (Django example):

- /api/cars
- /api/cars/(?P<plate_number>w+)

Note that no trailing slash is generated. If you need to be compatible with Django's APPEND_SLASH setting, you must:

- append trailing slashes explicitely in resource paths,
- instantiate `restosaur.contrib.django.API` with `append_slash` set to `True`.

```
API.view(resource, view_name=None)(model_class)
```

A decorator which binds a model_class to the resource as a named view. If view_name is not specified, the resource is linked to the model as a default one.

An example:

```
car_detail = webapi.resource('cars/:plate_number')

@webapi.view(car_detail)
class Car(object):
    def __init__(self, plate_number, name):
        self.plate_number = plate_number
        self.name = name
```

The above example creates a default link between `car_detail` resource and `Car` class.

This will allow you to generate an URLs to the Car resources in a DRY approach. Anytime you'll need to change a resource's URI, the linking to the model will not require a change:

```
@car_detail.representation('application/json')
def car_json(car, ctx):
    return {
        'plate_number': car.plate_number,
        'name': car.name,
        '@id': ctx.link_model(car),
    }
```

CHAPTER 4

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 5

Integrations

Restosaur v0.7 works only with Django. The supported versions are:

- Django 1.8
- Django 1.9
- Django 1.10

Support for Django 1.6 and 1.7 is officially dropped, but you can try to use it using Python 2.7, 3.3 or 3.4.

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 6

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
- remove obsolete code
- better test coverage
- enhance content negotiation and requests dispatching
- add final middleware support
- Python 3.x support

0.8 (beta)

First web framework independent release:

- add wsgi interface
- move Django adapter to contrib module
- move Django helpers to contrib module

1.0

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

CHAPTER 7

Contribution

Development

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

Documentation

The documentation is hosted in the same repository. It is built with Sphinx.

Translation

The documentation requires a correction and translations. Help in translation and working with English version is very appreciated.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search