restless Documentation

Release 2.1.1

Daniel Lindsley

Sep 27, 2017

Contents

A lightweight REST miniframework for Python.

Works great with Django, Flask, Pyramid & Tornado, but should be useful for many other Python web frameworks. Based on the lessons learned from Tastypie & other REST libraries.

CHAPTER 1

Features

- Small, fast codebase
- JSON output by default, but overridable
- RESTful
- Python 3.2+ (with shims to make broke-ass Python 2.6+ work)
- Flexible

CHAPTER 2

Anti-Features

(Things that will never be added...)

- Automatic ORM integration
- Authorization (per-object or not)
- Extensive filtering options
- XML output (though you can implement your own)
- Metaclasses
- Mixins
- HATEOAS

CHAPTER 3

Topics

Restless Tutorial

Restless is an alternative take on REST frameworks. While other frameworks attempt to be very complete, include special features or tie deeply to ORMs, Restless is a trip back to the basics.

It is fast, lightweight, and works with a small (but growing) number of different web frameworks. If you're interested in more of the backstory & reasoning behind Restless, please have a gander at the *Philosophy Behind Restless* documentation.

You can find some complete example implementation code in the repository.

Why Restless?

Restless tries to be RESTful by default, but flexible enough. The main `Resource` class has data methods (that you implement) for all the main RESTful actions. It also uses HTTP status codes as correctly as possible.

Restless is BYOD (bring your own data) and hence, works with almost any ORM/data source. If you can import a module to work with the data & can represent it as JSON, Restless can work with it.

Restless is small & easy to keep in your head. Common usages involve overridding just a few easily remembered method names. Total source code is a under a thousand lines of code.

Restless supports Python 3 **first**, but has backward-compatibility to work with Python 2.6+ code. Because the future is here.

Restless is JSON-only by default. Most everything can speak JSON, it's a *data* format (not a *document* format) & it's pleasant for both computers and humans to work with.

Restless is well-tested.

Installation

Installation is a relatively simple affair. For the most recent stable release, simply use pip to run:

```
$ pip install restless
```

Alternately, you can download the latest development source from Github:

```
$ git clone https://github.com/toastdriven/restless.git
$ cd restless
$ python setup.py install
```

Getting Started

Restless currently supports Django, Flask, Pyramid & **Tornado_**. For the purposes of most of this tutorial, we'll assume you're using Django. The process for developing & interacting with the API via Flask is nearly identical (& we'll be covering the differences at the end of this document).

There are only two steps to getting a Restless API up & functional. They are:

1. Implement a `restless.Resource` subclass
2. Hook up the resource to your URLs

Before beginning, you should be familiar with the common understanding of the behavior of the various REST methods.

About Resources

The main class in Restless is *`restless.resources.Resource`*. It provides all the dispatching/authentication/deserialization/serialization/response stuff so you don't have to.

Instead, you define/implement a handful of methods that strictly do data access/modification. Those methods are:

- `Resource.list` - *GET /*
- `Resource.detail` - *GET /identifier/*
- `Resource.create` - *POST /*
- `Resource.update` - *PUT /identifier/*
- `Resource.delete` - *DELETE /identifier/*

Restless also supports some less common combinations (due to their more complex & use-specific natures):

- `Resource.create_detail` - *POST /identifier/*
- `Resource.update_list` - *PUT /*
- `Resource.delete_list` - *DELETE /*

Restless includes modules for various web frameworks. To create a resource to work with Django, you'll need to subclass from *`restless.dj.DjangoResource`*. To create a resource to work with Flask, you'll need to subclass from `restless.fl.FlaskResource`.

`DjangoResource` is itself a subclass, inheriting from `restless.resource.Resource` & overrides a small number of methods to make things work smoothly.

The Existing Setup

We'll assume we're creating an API for our super-awesome blog platform. We have a `posts` application, which has a model setup like so...:

```python
# posts/models.py
from django.contrib.auth.models import User
from django.db import models


class Post(models.Model):
    user = models.ForeignKey(User, related_name='posts')
    title = models.CharField(max_length=128)
    slug = models.SlugField(blank=True)
    content = models.TextField(default='', blank=True)
    posted_on = models.DateTimeField(auto_now_add=True)
    updated_on = models.DateTimeField(auto_now=True)

    class Meta(object):
        ordering = ['-posted_on', 'title']

    def __str__(self):
        return self.title
```

This is just enough to get the ORM going & use some real data.

The rest of the app (views, URLs, admin, forms, etc.) really aren't important for the purposes of creating a basic Restless API, so we'll ignore them for now.

Creating A Resource

We'll start with defining the resource subclass. Where you put this code isn't particularly important (as long as other things can import the class you define), but a great convention is `<myapp>/api.py`. So in case of our tutorial app, we'll place this code in a new `posts/api.py` file.

We'll start with the most basic functional example.:

```python
# posts/api.py
from restless.dj import DjangoResource
from restless.preparers import FieldsPreparer

from posts.models import Post


class PostResource(DjangoResource):
    preparer = FieldsPreparer(fields={
        'id': 'id',
        'title': 'title',
        'author': 'user.username',
        'body': 'content',
        'posted_on': 'posted_on',
    })

    # GET /api/posts/ (but not hooked up yet)
    def list(self):
        return Post.objects.all()
    # GET /api/posts/<pk>/ (but not hooked up yet)
    def detail(self, pk):
        return Post.objects.get(id=pk)
```

As we've already covered, we're inheriting from `restless.dj.DjangoResource`. We're also importing our `Post` model, because serving data out of an API is kinda important.

The name of the class isn't particularly important, but it should be descriptive (and can play a role in hooking up URLs later on).

Fields

We define a `fields` attribute on the class. This dictionary provides a mapping between what the API will return & where the data is. This allows you to mask/rename fields, prevent some fields from being exposed or lookup information buried deep in the data model. The mapping is defined like...:

```
FieldsPreparer(fields={
    'the_fieldname_exposed_to_the_user': 'a_dotted_path_to_the_data',
})
```

This dotted path is what allows use to drill in. For instance, the `author` field above has a path of `user.username`. When serializing, this will cause Restless to look for an attribute (or a key on a dict) called `user`. From there, it will look further into the resulting object/dict looking for a `username` attribute/key, returning it as the final value.

Methods

We're also defining a `list` method & a `detail` method. Both can take optional postitional/keyword parameters if necessary.

These methods serve the **data** to present for their given endpoints. You don't need to manually construct any responses/status codes/etc., just provide what data should be present.

The `list` method serves the `GET` method on the collection. It should return a `list` (or similar iterable, like `QuerySet`) of data. In this case, we're simply returning all of our `Post` model instances.

The `detail` method also serves the `GET` method, but this time for single objects **ONLY**. Providing a `pk` in the URL allows this method to lookup the data that should be served.

Hooking Up The URLs

URLs to Restless resources get hooked up much like any other class-based view. However, Restless's *`restless.dj.DjangoResource`* comes with a special method called `urls`, which makes hooking up URLs more convenient.

You can hook the URLs for the resource up wherever you want. The recommended practice would be to create a URLconf just for the API portion of your site.:

```
# The ``settings.ROOT_URLCONF`` file
# myproject/urls.py
from django.conf.urls import url, include

# Add this!
from posts.api import PostResource

urlpatterns = [
    # The usual fare, then...
    # Add this!
    url(r'api/posts/', include(PostResource.urls())),
]
```

Note that unlike some other CBVs (admin specifically), the `urls` here is a **METHOD**, not an attribute/property. Those parens are important!

Manual URLconfs

You can also manually hook up URLs by specifying something like:

```
urlpatterns = [
    # ...

    # Identical to the above.
    url(r'api/posts/$', PostResource.as_list(), name='api_post_list'),
    url(r'api/posts/(?P<pk>\d+)/$', PostResource.as_detail(), name='api_post_detail'),
]
```

Testing the API

We've done enough to get the API (provided there's data in the DB) going, so let's make some requests!

Go to a terminal & run:

```
$ curl -X GET http://127.0.0.1:8000/api/posts/
```

You should get something like this back...:

```
{
    "objects": [
        {
            "id": 1,
            "title": "First Post!",
            "author": "daniel",
            "body": "This is the very first post on my shiny-new blog platform...",
            "posted_on": "2014-01-12T15:23:46",
        },
        {
            # More here...
        }
    ]
}
```

You can also go to the same URL in a browser (http://127.0.0.1:8000/api/posts/) & you should also get the JSON back.

You can then load up an individual object by changing the URL to include a `pk`.:

```
$ curl -X GET http://127.0.0.1:8000/api/posts/1/
```

You should get back...:

```
{
    "id": 1,
    "title": "First Post!",
    "author": "daniel",
    "body": "This is the very first post on my shiny-new blog platform...",
    "posted_on": "2014-01-12T15:23:46",
}
```

Note that the `objects` wrapper is no longer present & we get back the JSON for just that single object. Hooray!

Creating/Updating/Deleting Data

A read-only API is nice & all, but sometimes you want to be able to create data as well. So we'll implement some more methods.:

```
# posts/api.py
from restless.dj import DjangoResource
from restless.preparers import FieldsPreparer

from posts.models import Post


class PostResource(DjangoResource):
    preparer = FieldsPreparer(fields={
        'id': 'id',
        'title': 'title',
        'author': 'user.username',
        'body': 'content',
        'posted_on': 'posted_on',
    })

    # GET /api/posts/
    def list(self):
        return Post.objects.all()

    # GET /api/posts/<pk>/
    def detail(self, pk):
        return Post.objects.get(id=pk)

    # Add this!
    # POST /api/posts/
    def create(self):
        return Post.objects.create(
            title=self.data['title'],
            user=User.objects.get(username=self.data['author']),
            content=self.data['body']
        )

    # Add this!
    # PUT /api/posts/<pk>/
    def update(self, pk):
        try:
            post = Post.objects.get(id=pk)
        except Post.DoesNotExist:
            post = Post()

        post.title = self.data['title']
        post.user = User.objects.get(username=self.data['author'])
        post.content = self.data['body']
        post.save()
        return post

    # Add this!
    # DELETE /api/posts/<pk>/
    def delete(self, pk):
        Post.objects.get(id=pk).delete()
```

By adding the `create/update/delete` methods, we now have the ability to add new items, update existing ones & delete individual items. Most of this code is relatively straightforward ORM calls, but there are a few interesting new things going on here.

Note that the `create` & `update` methods are both using a special `self.data` variable. This is created by Restless during deserialization & is the **JSON** data the user sends as part of the request.

Warning: This data (within `self.data`) is mostly unsanitized (beyond standard JSON decoding) & could contain anything (not just the `fields` you define).

You know your data best & validation is **very** non-standard between frameworks, so this is a place where Restless punts.

Some people like cleaning the data with `Forms`, others prefer to hand-sanitize, some do model validation, etc. Do what works best for you.

You can refer to the *Extending Restless* documentation for recommended approaches.

Also note that `delete` is the first method with **no return value**. You can do the same thing on `create/update` if you like. When there's no meaningful data returned, Restless simply sends back a correct status code & an empty body.

Finally, there's no need to hook up more URLconfs. Restless delegates based on a list & a detail endpoint. All further dispatching is HTTP verb-based & handled by Restless.

Testing the API, Round 2

Now let's test out our new functionality! Go to a terminal & run:

```
$ curl -X POST -H "Content-Type: application/json" -d '{"title": "New library␣
↪released!", "author": "daniel", "body": "I just released a new thing!"}' http://127.
↪0.0.1:8000/api/posts/
```

You should get something like this back...:

```
{
    "error": "Unauthorized"
}
```

Wait, what?!! But we added our new methods & everything!

The reason you get unauthorized is that by default, **only GET** requests are allowed by Restless. It's the only sane/safe default to have & it's very easy to fix.

Error Handling

By default, Restless tries to serialize any exceptions that may be encountered. What gets serialized depends on two methods: `Resource.is_debug()` & `Resource.bubble_exceptions()`.

is_debug

Regardless of the error type, the exception's message will get serialized into the response under the "error" key. For example, if an IOError is raised during processing, you'll get a response like:

```
HTTP/1.0 500 INTERNAL SERVER ERROR
Content-Type: application/json
# Other headers...

{
    "error": "Whatever."
}
```

If Resource.is_debug() returns True (the default is False), Restless will also include a traceback. For example:

```
HTTP/1.0 500 INTERNAL SERVER ERROR
Content-Type: application/json
# Other headers...

{
    "error": "Whatever.",
    "traceback": "Traceback (most recent call last):\n # Typical traceback..."
}
```

Each framework-specific Resource subclass implements is_debug() in a way most appropriate for the framework. In the case of the DjangoResource, it returns settings.DEBUG, allowing your resources to stay consistent with the rest of your application.

bubble_exceptions

If Resource.bubble_exceptions() returns True (the default is False), any exception encountered will simply be re-raised & it's up to your setup to handle it. Typically, this behavior is undesirable except in development & with frameworks that can provide extra information/debugging on exceptions. Feel free to override it (return True) or implement application-specific logic if that meets your needs.

Authentication

We're going to override one more method in our resource subclass, this time adding the is_authenticated method.:

```python
# posts/api.py
from restless.dj import DjangoResource
from restless.preparers import FieldsPreparer

from posts.models import Post


class PostResource(DjangoResource):
    preparer = FieldsPreparer(fields={
        'id': 'id',
        'title': 'title',
        'author': 'user.username',
        'body': 'content',
        'posted_on': 'posted_on',
    }

    # Add this!
    def is_authenticated(self):
        # Open everything wide!
        # DANGEROUS, DO NOT DO IN PRODUCTION.
        return True

        # Alternatively, if the user is logged into the site...
        # return self.request.user.is_authenticated()

        # Alternatively, you could check an API key. (Need a model for this...)
        # from myapp.models import ApiKey
        # try:
        #     key = ApiKey.objects.get(key=self.request.GET.get('api_key'))
        #     return True
        # except ApiKey.DoesNotExist:
        #     return False

    def list(self):
        return Post.objects.all()

    def detail(self, pk):
        return Post.objects.get(id=pk)

    def create(self):
        return Post.objects.create(
            title=self.data['title'],
            user=User.objects.get(username=self.data['author']),
            content=self.data['body']
        )

    def update(self, pk):
        try:
            post = Post.objects.get(id=pk)
        except Post.DoesNotExist:
            post = Post()

        post.title = self.data['title']
        post.user = User.objects.get(username=self.data['author'])
        post.content = self.data['body']
        post.save()
        return post

    def delete(self, pk):
        Post.objects.get(id=pk).delete()
```

With that change in place, now our API should play nice...

Testing the API, Round 3

Back to the terminal & again run:

```
$ curl -X POST -H "Content-Type: application/json" -d '{"title": "New library␣
↪released!", "author": "daniel", "body": "I just released a new thing!"}' http://127.
↪0.0.1:8000/api/posts/
```

You should get something like this back...:

```
{
    "body": "I just released a new thing!",
    "title": "New library released!",
    "id": 3,
    "posted_on": "2014-01-13T10:02:55.926857+00:00",
    "author": "daniel"
}
```

Hooray! Now we can check for it in the list view (`GET` http://127.0.0.1:8000/api/posts/) or by a detail (`GET` http://127.0.0.1:8000/api/posts/3/).

We can also update it. Restless expects **complete** bodies (don't try to send partial updates, that's typically reserved for `PATCH`).:

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"title": "Another new library␣
↪released!", "author": "daniel", "body": "I just released a new piece of software!"}
↪' http://127.0.0.1:8000/api/posts/3/
```

And we can delete our new data if we decide we don't like it.:

```
$ curl -X DELETE http://127.0.0.1:8000/api/posts/3/
```

Conclusion

We've now got a basic, working RESTful API in a short amount of code! And the possibilities don't stop at the ORM. You could hook up:

- Redis
- the NoSQL flavor of the month
- text/log files
- wrap calls to other (nastier) APIs

You may also want to check the *Cookbook* for other interesting/useful possibilities & implementation patterns.

Bonus: Flask Support

Outside of the ORM, precious little of what we implemented above was Django-specific. If you used an ORM like Peewee or SQLAlchemy, you'd have very similar-looking code.

In actuality, there are just two changes to make the Restless-portion of the above work within Flask.

1. Change the inheritance
2. Change how the URL routes are hooked up.

Change The Inheritance

Restless ships with a `restless.fl.FlaskResource` class, akin to the `DjangoResource`. So the first change is dead simple.:

```
# Was: from restless.dj import DjangoResource
# Becomes:
from restless.fl import FlaskResource

# ...

# Was: class PostResource(DjangoResource):
# Becomes:
class PostResource(FlaskResource):
    # ...
```

Change How The URL Routes Are Hooked Up

Again, similar to the `DjangoResource`, the `FlaskResource` comes with a special method to make hooking up the routes easier.

Wherever your `PostResource` is defined, import your Flask `app`, then call the following:

```
PostResource.add_url_rules(app, rule_prefix='/api/posts/')
```

This will hook up the same two endpoints (list & detail, just like Django above) within the Flask app, doing similar dispatches.

You can also do this manually (but it's ugly/hurts).:

```
app.add_url_rule('/api/posts/', endpoint='api_posts_list', view_func=PostResource.as_
↪list(), methods=['GET', 'POST', 'PUT', 'DELETE'])
app.add_url_rule('/api/posts/<pk>/', endpoint='api_posts_detail', view_
↪func=PostResource.as_detail(), methods=['GET', 'POST', 'PUT', 'DELETE'])
```

Done!

Believe it or not, if your ORM could be made to look similar, that's all the further changes needed to get the same API (with the same end-user interactions) working on Flask! Huzzah!

Philosophy Behind Restless

Quite simply, I care about creating flexible & RESTFul APIs. In building Tastypie, I tried to create something extremely complete & comprehensive. The result was writing a lot of hook methods (for easy extensibility) & a lot of (perceived) bloat, as I tried to accommodate for everything people might want/need in a flexible/overridable manner.

But in reality, all I really ever personally want are the RESTful verbs, JSON serialization & the ability of override behavior.

This one is written for me, but maybe it's useful to you.

Pithy Statements

Keep it simple Complexity is the devil. It makes it harder to maintain, hard to be portable & hard for users to work with.

Python 3 is the default Why write for the past? We'll support it, but Python 2.X should be treated as a (well-supported) afterthought.

BSD Licensed Any other license is a bug.

Work with as many web frameworks as possible. Django is my main target, but I want portable code that I can use from other frameworks. Switching frameworks ought to be simply a change of inheritance.

RESTful by default REST is native to the web & works well. We should make it easy to *be* RESTful.

If I wanted RPC, I'd just write my own crazy methods that did whatever I wanted.

JSON-only Because XML sucks, bplist is Apple-specific & YAML is a security rats nest. Everything (I care about) speaks JSON, so let's keep it simple.

B.Y.O.D. (Bring Your Own Data) Don't integrate with a specific ORM. Don't mandate a specific access format. We expose 8-ish simple methods (that map cleanly to the REST verbs/endpoints). Data access/manipulation happens there & the user knows best, so they should implement it.

No HATEOAS I loved HATEOAS dearly, but it is complex & making it work with many frameworks is a windmill I don't care to tilt at. Most people never use the deep links anyhow.

No Authorization Authorization schemes vary so wildly & everyone wants something different. Give them the ability to write it without natively trying to support it.

Extending Restless

Restless is meant to handle many simpler cases well & have enough extensibility to handle more complex API tasks.

However, a specific goal of the project is to not expand the scope much & simply give you, the expert on your API, the freedom to build what you need.

We'll be covering:

- Custom endpoints
- Customizing data output
- Adding data validation
- Providing different serialization formats

Custom Endpoints

Sometimes you need to provide more than just the typical HTTP verbs. Restless allows you to hook up custom endpoints that can take advantage of much of the `Resource`.

Implementing these views requires a couple simple steps:

- Writing the method
- Adding to the `Resource.http_methods` mapping
- Adding to your URL routing

For instance, if you wanted to added a schema view (`/api/posts/schema/`) that responded to `GET` requests, you'd first write the method:

```
from restless.dj import DjangoResource
from restless.resources import skip_prepare


class PostResource(DjangoResource):
    # The usual methods, then...
    @skip_prepare
    def schema(self):
        # Return your schema information.
        # We're keeping it simple (basic field names & data types).
        return {
            'fields': {
                'id': 'integer',
                'title': 'string',
                'author': 'string',
                'body': 'string',
            },
        }
```

The next step is to update the `Resource.http_methods`. This can either be fully written out in your class or (as I prefer) a small extension to your `__init__`...:

```
from restless.dj import DjangoResource
from restless.resources import skip_prepare


class PostResource(DjangoResource):
    # We'll lightly extend the ``__init__``.
    def __init__(self, *args, **kwargs):
        super(PostResource, self).__init__(*args, **kwargs)

        # Add on a new top-level key, then define what HTTP methods it
        # listens on & what methods it calls for them.
        self.http_methods.update({
            'schema': {
                'GET': 'schema',
            }
        })

    # The usual methods, then...

    @skip_prepare
    def schema(self):
        return {
            'fields': {
                'id': 'integer',
                'title': 'string',
                'author': 'string',
                'body': 'string',
            },
        }
```

Finally, it's just a matter of hooking up the URLs as well. You can do this manually or (once again) by extending a built-in method.:

```
# Add the correct import here.
from django.conf.urls import url

from restless.dj import DjangoResource
from restless.resources import skip_prepare


class PostResource(DjangoResource):
    def __init__(self, *args, **kwargs):
        super(PostResource, self).__init__(*args, **kwargs)
        self.http_methods.update({
            'schema': {
                'GET': 'schema',
            }
        })

    # The usual methods, then...

    # Note: We're using the ``skip_prepare`` decorator here so that Restless
    # doesn't run ``prepare`` on the schema data.
    # If your custom view returns a typical ``object/dict`` (like the
    # ``detail`` method), you can omit this.
    @skip_prepare
    def schema(self):
        return {
            'fields': {
                'id': 'integer',
                'title': 'string',
                'author': 'string',
                'body': 'string',
            },
        }

    # Finally, extend the URLs.
    @classmethod
    def urls(cls, name_prefix=None):
        urlpatterns = super(PostResource, cls).urls(name_prefix=name_prefix)
        return [
            url(r'^schema/$', cls.as_view('schema'), name=cls.build_url_name('schema',
↪ name_prefix)),
        ] + urlpatterns
```

Note: This step varies from framework to framework around hooking up the URLs/routes. The code is specific to the *restless.dj.DjangoResource*, but the approach is the same regardless.

You should now be able to hit something like http://127.0.0.1/api/posts/schema/ in your browser & get a JSON schema view!

Customizing Data Output

There are four approaches to customizing your data ouput.

1. The built-in `Preparer/FieldsPreparer` (simple)
2. The included `SubPreparer/CollectionSubPreparer` (slightly more complex)
3. Overriding *restless.resources.Resource.prepare()* (happy medium)
4. Per-method data (flexible but most work)

Fields

Using `FieldsPreparer` is documented elsewhere (see the *Restless Tutorial*), but the basic gist is that you create a `FieldsPreparer` instance & assign it on your resource class. It takes a `fields` parameter, which should be a dictionary of fields to expose. Example:

```
class MyResource(Resource):
    preparer = FieldsPreparer(fields={
        # Expose the same name.
        "id": "id",
        # Rename a field.
        "author": "username",
        # Access deeper data.
        "type_id": "metadata.type.pk",
    })
```

This dictionary is a mapping, with keys representing the final name & the values acting as a lookup path.

If the lookup path **has no** periods (i.e. `name`) in it, it's considered to be an attribute/key on the item being processed. If that item looks like a `dict`, key access is attempted. If it looks like an `object`, attribute access is used. In either case, the found value is returned.

If the lookup path **has** periods (i.e. `entry.title`), it is split on the periods (like a Python import path) and recursively uses the previous value to look up the next value until a final value is found.

Subpreparers & Collections

Sometimes, your data isn't completely flat but is instead nested. This frequently occurs in conjunction with related data, such as a foreign key'd object or many-to-many scenario. In this case, you can lever "subpreparers". Restless ships with two of these, the `SubPreparer` & the `CollectionSubPreparer`.

The `SubPreparer` is useful for a single nested relation. You define a regular `Preparer/FieldsPreparer` (perhaps in a shareable location), then use the `SubPreparer` to pull it in & incorporate the nested data. For example:

```
# We commonly expose author information in our API as nested data.
# This definition can happen in its own module or wherever needed.
author_preparer = FieldsPreparer(fields={
    'id': 'pk',
    'username': 'username',
    'name': 'get_full_name',
})

# ...

# Then, in the main preparer, pull them in using `SubPreparer`.
preparer = FieldsPreparer(fields={
    'author': SubPreparer('user', author_preparer),
    # Other fields can come before/follow as normal.
    'content': 'post',
    'created': 'created_at',
})
```

This results in output like:

```
{
    "content": "Isn't my blog cool? I think so...",
    "created": "2017-05-22T10:34:48",
    "author": {
        "id": 5,
        "username": "joe",
        "name": "Joe Bob"
    }
}
```

The `CollectionSubPreparer` operates on the same principle (define a set of fields to be nested), but works with collections of things. These collections should be ordered & behave similar to iterables like `list``s & ``tuples`. As an example:

```
# Set up a preparer that handles the data for each thing in the broader
# collection.
# Again, this can be in its own module or just wherever it's needed.
comment_preparer = FieldsPreparer(fields={
    'comment': 'comment_text',
    'created': 'created',
})

# Use it with the ``CollectionSubPreparer`` to create a list
# of prepared sub items.
preparer = FieldsPreparer(fields={
    # A normal blog post field.
    'post': 'post_text',
    # All the comments on the post.
    'comments': CollectionSubPreparer('comments.all', comment_preparer),
})
```

Which would produce output like:

```
{
    "post": "Another day, another blog post.",
    "comments": [
        {
            "comment": "I hear you. Boring day here too.",
            "created": "2017-05-23T16:43:22"
        },
        {
            "comment": "SPAM SPAM SPAM",
            "created": "2017-05-24T21:21:21"
        }
    ]
}
```

Overriding `prepare`

For every item (`object` or `dict`) that gets serialized as output, it runs through a `prepare` method on your `Resource` subclass.

The default behavior checks to see if you have `fields` defined on your class & either just returns all the data (if there's no `fields`) or uses the `fields` to extract plain data.

However, you can use/abuse this method for your own nefarious purposes. For example, if you wanted to serve an API of users but sanitize the data, you could do something like:

```
from django.contrib.auth.models import User

from restless.dj import DjangoResource
```

```python
from restless.preparers import FieldsPreparer


class UserResource(DjangoResource):
    preparer = FieldsPreparer(fields={
        'id': 'id',
        'username': 'username',
        # We're including email here, but we'll sanitize it later.
        'email': 'email',
        'date_joined': 'date_joined',
    })

    def list(self):
        return User.objects.all()

    def detail(self, pk):
        return User.objects.get(pk=pk)

    def prepare(self, data):
        # ``data`` is the object/dict to be exposed.
        # We'll call ``super`` to prep the data, then we'll mask the email.
        prepped = super(UserResource, self).prepare(data)

        email = prepped['email']
        at_offset = email.index('@')
        prepped['email'] = email[:at_offset + 1] + "..."

        return prepped
```

This example is somewhat contrived, but you can perform any kind of transformation you want here, as long as you return a plain, serializable `dict`.

Per-Method Data

Because Restless can serve plain old Python objects (anything JSON serializable + `datetime` + `decimal`), the ultimate form of control is simply to load your data however you want, then return a simple/serializable form.

For example, Django's `models.Model` classes are not normally JSON-serializable. We also may want to expose related data in a nested form. Here's an example of doing something like that.:

```python
from restless.dj import DjangoResource

from posts.models import Post


class PostResource(DjangoResource):
    def detail(self, pk):
        # We do our rich lookup here.
        post = Post.objects.get(pk=pk).select_related('user')

        # Then we can simplify it & include related information.
        return {
            'title': post.title,
            'author': {
                'id': post.user.id,
                'username': post.user.username,
                'date_joined': post.user.date_joined,
                # We exclude things like ``password`` & ``email`` here
                # intentionally.
            },
            'body': post.content,
            # ...
        }
```

While this is more verbose, it gives you all the control.

If you have resources for your nested data, you can also re-use them to make the construction easier. For example:

```
from django.contrib.auth.models import User

from restless.dj import DjangoResource
from restless.preparers import FieldsPreparer

from posts.models import Post


class UserResource(DjangoResource):
    preparer = FieldsPreparer(fields={
        'id': 'id',
        'username': 'username',
        'date_joined': 'date_joined',
    })

    def detail(self, pk):
        return User.objects.get(pk=pk)


class PostResource(DjangoResource):
    def detail(self, pk):
        # We do our rich lookup here.
        post = Post.objects.get(pk=pk).select_related('user')

        # Instantiate the ``UserResource``
        ur = UserResource()

        # Then populate the data.
        return {
            'title': post.title,
            # We leverage the ``prepare`` method from above to build the
            # nested data we want.
            'author': ur.prepare(post.user),
            'body': post.content,
            # ...
        }
```

Data Validation

Validation can be a contentious issue. No one wants to risk data corruption or security holes in their services. However, there's no real standard or consensus on doing data validation even within the **individual** framework communities themselves, let alone *between* frameworks.

So unfortunately, Restless mostly ignores this issue, leaving you to do data validation the way you think is best.

The good news is that the data you'll need to validate is already in a convenient-to-work-with dictionary called

`Resource.data` (assigned immediately after deserialization takes place).

The recommended approach is to simply add on to your data methods themselves. For example, since Django `Form` objects are at least *bundled* with the framework, we'll use those as an example...:

```
from django.forms import ModelForm

from restless.dj import DjangoResource
from restless.exceptions import BadRequest


class UserForm(ModelForm):
    class Meta(object):
        model = User
        fields = ['username', 'first_name', 'last_name', 'email']


class UserResource(DjangoResource):
    preparer = FieldsPreparer(fields={
        "id": "id",
        "username": "username",
        "first_name": "first_name",
        "last_name": "last_name",
        "email": "email",
    })

    def create(self):
        # We can create a bound form from the get-go.
        form = UserForm(self.data)

        if not form.is_valid():
            raise BadRequest('Something is wrong.')

        # Continue as normal, using the form data instead.
        user = User.objects.create(
            username=form.cleaned_data['username'],
            first_name=form.cleaned_data['first_name'],
            last_name=form.cleaned_data['last_name'],
            email=form.cleaned_data['email'],
        )
        return user
```

If you're going to use this validation in other places, you're welcome to DRY up your code into a validation method. An example of this might look like...:

```
from django.forms import ModelForm

from restless.dj import DjangoResource
from restless.exceptions import BadRequest


class UserForm(ModelForm):
    class Meta(object):
        model = User
        fields = ['username', 'first_name', 'last_name', 'email']


class UserResource(DjangoResource):
    preparer = FieldsPreparer(fields={
        "id": "id",
        "username": "username",
        "first_name": "first_name",
        "last_name": "last_name",
        "email": "email",
    })

    def validate_user(self):
        form = UserForm(self.data)

        if not form.is_valid():
            raise BadRequest('Something is wrong.')

        return form.cleaned_data

    def create(self):
        cleaned = self.validate_user()
        user = User.objects.create(
            username=cleaned['username'],
            first_name=cleaned['first_name'],
            last_name=cleaned['last_name'],
            email=cleaned['email'],
        )
        return user

    def update(self, pk):
        cleaned = self.validate_user()
        user = User.objects.get(pk=pk)
        user.username = cleaned['username']
        user.first_name = cleaned['first_name']
        user.last_name = cleaned['last_name']
        user.email = cleaned['email']
        user.save()
        return user
```

Alternative Serialization

For some, Restless' JSON-only syntax might not be appealing. Fortunately, overriding this is not terribly difficult.

For the purposes of demonstration, we'll implement YAML in place of JSON. The process would be similar (but much more verbose) for XML (& brings a host of problems as well).

Start by creating a `Serializer` subclass for the YAML. We'll override a couple methods there. This code can live anywhere, as long as it is importable for your `Resource`.:

```
import yaml

from restless.serializers import Serializer


class YAMLSerializer(Serializer):
    def deserialize(self, body):
        # Do **NOT** use ``yaml.load`` here, as it can contain things like
        # *functions* & other dangers!
        return yaml.safe_load(body)
    def serialize(self, data):
        return yaml.dump(data)
```

Once that class has been created, it's just a matter of assigning an instance onto your `Resource`.:

```
# Old.
class MyResource(Resource):
    # This was present by default.
    serializer = JSONSerializer()

# New.
class MyResource(Resource):
    serializer = YAMLSerializer()
```

You can even do things like handle multiple serialization formats, say if the user provides a `?format=yaml` GET param...:

```
from restless.serializers import Serializer
from restless.utils import json, MoreTypesJSONEncoder

from django.template import Context, Template


class MultiSerializer(Serializer):
    def deserialize(self, body):
        # This is Django-specific, but all frameworks can handle GET
        # parameters...
        ct = request.GET.get('format', 'json')

        if ct == 'yaml':
            return yaml.safe_load(body)
        else:
            return json.load(body)

    def serialize(self, data):
        # Again, Django-specific.
        ct = request.GET.get('format', 'json')

        if ct == 'yaml':
            return yaml.dump(body)
        else:
            return json.dumps(body, cls=MoreTypesJSONEncoder)
```

Cookbook

This is a place for community-contributed patterns & ideas for extending Restless.

Authentication

If your framework has the concept of a logged-in user (like Django), you can do something like:

```
class MyResource(DjangoResource):
    def is_authenticated(self):
        return self.request.user.is_authenticated()
```

If you need a more fine graned authentication you could check your current endpoint and do something like that:

```python
class MyResource(DjangoResource):
    def is_authenticated(self):
        if self.endpoint in ('update', 'create'):
            return self.request.user.is_authenticated()
        else:
            return True
```

Contributing

Restless is open-source and, as such, grows (or shrinks) & improves in part due to the community. Below are some guidelines on how to help with the project.

Philosophy

- Restless is BSD-licensed. All contributed code must be either
  - the original work of the author, contributed under the BSD, or...
  - work taken from another project released under a BSD-compatible license.
- GPL'd (or similar) works are not eligible for inclusion.
- Restless's git master branch should always be stable, production-ready & passing all tests.
- Major releases (1.x.x) are commitments to backward-compatibility of the public APIs. Any documented API should ideally not change between major releases. The exclusion to this rule is in the event of a security issue.
- Minor releases (x.3.x) are for the addition of substantial features or major bugfixes.
- Patch releases (x.x.4) are for minor features or bugfixes.

Guidelines For Reporting An Issue/Feature

So you've found a bug or have a great idea for a feature. Here's the steps you should take to help get it added/fixed in Restless:

- First, check to see if there's an existing issue/pull request for the bug/feature. All issues are at https://github.com/toastdriven/restless/issues and pull reqs are at https://github.com/toastdriven/restless/pulls.
- If there isn't one there, please file an issue. The ideal report includes:
  - A description of the problem/suggestion.
  - How to recreate the bug.
  - If relevant, including the versions of your:
    - Python interpreter
    - Web framework
    - Restless
    - Optionally of the other dependencies involved
  - Ideally, creating a pull request with a (failing) test case demonstrating what's wrong. This makes it easy for us to reproduce & fix the problem. Instructions for running the tests are at *restless*

Guidelines For Contributing Code

If you're ready to take the plunge & contribute back some code/docs, the process should look like:

- Fork the project on GitHub into your own account.
- Clone your copy of Restless.
- Make a new branch in git & commit your changes there.
- Push your new branch up to GitHub.
- Again, ensure there isn't already an issue or pull request out there on it. If there is & you feel you have a better fix, please take note of the issue number & mention it in your pull request.
- Create a new pull request (based on your branch), including what the problem/feature is, versions of your software & referencing any related issues/pull requests.

In order to be merged into Restless, contributions must have the following:

- A solid patch that:
  - is clear.
  - works across all supported versions of Python.
  - follows the existing style of the code base (mostly PEP-8).
  - comments included as needed.
- A test case that demonstrates the previous flaw that now passes with the included patch.
- If it adds/changes a public API, it must also include documentation for those changes.
- Must be appropriately licensed (see "Philosophy").
- Adds yourself to the AUTHORS file.

If your contribution lacks any of these things, they will have to be added by a core contributor before being merged into Restless proper, which may take additional time.

Security

Restless takes security seriously. By default, it:

- does not access your filesystem in any way.
- only allows GET requests, demanding that the user think about who should be able to work with a given end-point.
- has `is_debug` as `False` by default.
- wraps JSON lists in an object to prevent exploits.

While no known vulnerabilities exist, all software has bugs & Restless is no exception.

If you believe you have found a security-related issue, please **DO NOT SUBMIT AN ISSUE/PULL REQUEST**. This would be a public disclosure & would allow for 0-day exploits.

Instead, please send an email to "daniel@toastdriven.com" & include the following information:

- A description of the problem/suggestion.
- How to recreate the bug.
- If relevant, including the versions of your:

  - Python interpreter
  - Web framework
  - Restless
  - Optionally of the other dependencies involved

Please bear in mind that I'm not a security expert/researcher, so a layman's description of the issue is very important.

Upon reproduction of the exploit, steps will be taken to fix the issue, release a new version & make users aware of the need to upgrade. Proper credit for the discovery of the issue will be granted via the AUTHORS file & other mentions.

CHAPTER 4

API Reference

Constants

restless.constants

A set of constants included with `restless`. Mostly nice status code mappings for use in exceptions or the `Resource.status_map`.

OK = `200`

CREATED = `201`

ACCEPTED = `202`

NO_CONTENT = `204`

UNAUTHORIZED = `401`

NOT_FOUND = `404`

METHOD_NOT_ALLOWED = `405`

APPLICATION_ERROR = `500`

METHOD_NOT_IMPLEMENTED = `501`

Data

restless.data

class `restless.data.`**`Data`**(*value*, *should_prepare=True*, *prepare_with=None*)

Exceptions

restless.exceptions

exception `restless.exceptions.`**`BadRequest`**(*msg=None*)

> **`msg`** = **'Bad request.'**
>
> **`status`** = **400**

exception `restless.exceptions.`**`Conflict`**(*msg=None*)

> **`msg`** = **'There was a conflict when processing the request.'**
>
> **`status`** = **409**

exception `restless.exceptions.`**`ExpectationFailed`**(*msg=None*)

> **`msg`** = **'Unable to satisfy requirements of Expect header.'**
>
> **`status`** = **417**

exception `restless.exceptions.`**`FailedDependency`**(*msg=None*)

> **`msg`** = **'Request failed due to a previous failed request.'**
>
> **`status`** = **424**

exception `restless.exceptions.`**`Forbidden`**(*msg=None*)

> **`msg`** = **'Permission denied.'**
>
> **`status`** = **403**

exception `restless.exceptions.`**`Gone`**(*msg=None*)

> **`msg`** = **'Resource removed permanently.'**
>
> **`status`** = **410**

exception `restless.exceptions.`**`HttpError`**(*msg=None*)

> The foundational HTTP-related error.
>
> All other HTTP errors in `restless` inherit from this one.
>
> Has a `status` attribute. If present, `restless` will use this as the `status_code` in the response.
>
> Has a `msg` attribute. Has a reasonable default message (override-able from the constructor).
>
> **`msg`** = **'Application Error'**
>
> **`status`** = **500**

exception `restless.exceptions.`**`IAmATeapot`**(*msg=None*)

> **`msg`** = **'This is a teapot; do not attempt to brew coffee with it.'**
>
> **`status`** = **418**

exception `restless.exceptions.`**`Locked`**(*msg=None*)

`msg` **= 'Resource is locked.'**

`status` **= 423**

exception `restless.exceptions.`**`MethodNotAllowed`**(*msg=None*)

`msg` **= 'The specified HTTP method is not allowed.'**

`status` **= 405**

exception `restless.exceptions.`**`MethodNotImplemented`**(*msg=None*)

`msg` **= 'The specified HTTP method is not implemented.'**

`status` **= 501**

exception `restless.exceptions.`**`NotAcceptable`**(*msg=None*)

`msg` **= "Unable to send content specified on the request's Accept header(s)."**

`status` **= 406**

exception `restless.exceptions.`**`NotFound`**(*msg=None*)

`msg` **= 'Resource not found.'**

`status` **= 404**

exception `restless.exceptions.`**`PreconditionFailed`**(*msg=None*)

`msg` **= 'Unable to satisfy one or more request preconditions.'**

`status` **= 412**

exception `restless.exceptions.`**`RestlessError`**

A common base exception from which all other exceptions in `restless` inherit from.

No special attributes or behaviors.

exception `restless.exceptions.`**`TooManyRequests`**(*msg=None*)

`msg` **= 'There was a conflict when processing the request.'**

`status` **= 429**

exception `restless.exceptions.`**`Unauthorized`**(*msg=None*)

`msg` **= 'Unauthorized.'**

`status` **= 401**

exception `restless.exceptions.`**`Unavailable`**(*msg=None*)

`msg` **= 'There was a conflict when processing the request.'**

`status` **= 503**

exception `restless.exceptions.`**`UnavailableForLegalReasons`**(*msg=None*)

> **`msg`** = **'Resource made unavailable by a legal decision.'**
>
> **`status`** = **451**

exception `restless.exceptions.`**`UnprocessableEntity`**(*msg=None*)

> **`msg`** = **'Request cannot be followed due to a semantic error.'**
>
> **`status`** = **422**

exception `restless.exceptions.`**`UnsupportedMediaType`**(*msg=None*)

> **`msg`** = **'Type of media provided on request is not supported.'**
>
> **`status`** = **415**

Preparers

restless.preparers

class `restless.preparers.`**`CollectionSubPreparer`**(*lookup*, *preparer*)

A preparation class designed to handle collections of data.

This is useful in the case where you have a 1-to-many or many-to-many relationship of data to expose as part of the parent data.

Example:

```
# First, set up a preparer that handles the data for each thing in
# the broader collection.
comment_preparer = FieldsPreparer(fields={
    'comment': 'comment_text',
    'created': 'created',
})
# Then use it with the ``CollectionSubPreparer`` to create a list
# of prepared sub items.
preparer = FieldsPreparer(fields={
    # A normal blog post field.
    'post': 'post_text',
    # All the comments on the post.
    'comments': CollectionSubPreparer('comments.all', comment_preparer),
})
```

`prepare`(*data*)

Handles passing each item in the collection data to the configured subpreparer.

Uses a loop and the `get_inner_data` method to provide the correct item of the data.

Returns a list of data as the response.

class `restless.preparers.`**`FieldsPreparer`**(*fields*)

A more complex preparation object, this will return a given set of fields.

This takes a `fields` parameter, which should be a dictionary of keys (fieldnames to expose to the user) & values (a dotted lookup path to the desired attribute/key on the object).

Example:

```
preparer = FieldsPreparer(fields={
    # ``user`` is the key the client will see.
    # ``author.pk`` is the dotted path lookup ``FieldsPreparer``
    # will traverse on the data to return a value.
    'user': 'author.pk',
})
```

lookup_data(*lookup*, *data*)

Given a lookup string, attempts to descend through nested data looking for the value.

Can work with either dictionary-alikes or objects (or any combination of those).

Lookups should be a string. If it is a dotted path, it will be split on `.` & it will traverse through to find the final value. If not, it will simply attempt to find either a key or attribute of that name & return it.

Example:

```
>>> data = {
...     'type': 'message',
...     'greeting': {
...         'en': 'hello',
...         'fr': 'bonjour',
...         'es': 'hola',
...     },
...     'person': Person(
...         name='daniel'
...     )
... }
>>> lookup_data('type', data)
'message'
>>> lookup_data('greeting.en', data)
'hello'
>>> lookup_data('person.name', data)
'daniel'
```

prepare(*data*)

Handles transforming the provided data into the fielded data that should be exposed to the end user.

Uses the `lookup_data` method to traverse dotted paths.

Returns a dictionary of data as the response.

class `restless.preparers.`**Preparer**

A plain preparation object which just passes through data.

It also is relevant as the protocol subclasses should implement to work with Restless.

prepare(*data*)

Handles actually transforming the data.

By default, this does nothing & simply returns the data passed to it.

class `restless.preparers.`**SubPreparer**(*lookup*, *preparer*)

A preparation class designed to be used within other preparers.

This is primary to enable deeply-nested structures, allowing you to compose/share definitions as well. Typical usage consists of creating a configured instance of a FieldsPreparer, then use a *SubPreparer* to pull it in.

Example:

```python
# First, define the nested fields you'd like to expose.
author_preparer = FieldsPreparer(fields={
    'id': 'pk',
    'username': 'username',
    'name': 'get_full_name',
})
# Then, in the main preparer, pull them in using `SubPreparer`.
preparer = FieldsPreparer(fields={
    'author': SubPreparer('user', author_preparer),
    # Other fields can come before/follow as normal.
    'content': 'post',
    'created': 'created_at',
})
```

`get_inner_data`(*data*)

Used internally so that the correct data is extracted out of the broader dataset, allowing the preparer being called to deal with just the expected subset.

`prepare`(*data*)

Handles passing the data to the configured preparer.

Uses the `get_inner_data` method to provide the correct subset of the data.

Returns a dictionary of data as the response.

Resources

restless.resources

class `restless.resources.`**`Resource`**(**args*, ***kwargs*)

Defines a RESTful resource.

Users are expected to subclass this object & implement a handful of methods:

- `list`
- `detail`
- `create` (requires authentication)
- `update` (requires authentication)
- `delete` (requires authentication)

Additionally, the user may choose to implement:

- `create_detail` (requires authentication)
- `update_list` (requires authentication)
- `delete_list` (requires authentication)

Users may also wish to define a `fields` attribute on the class. By providing a dictionary of output names mapped to a dotted lookup path, you can control the serialized output.

Users may also choose to override the `status_map` and/or `http_methods` on the class. These respectively control the HTTP status codes returned by the views and the way views are looked up (based on HTTP method & endpoint).

classmethod **`as_detail`**(**init_args*, ***init_kwargs*)

Used for hooking up the actual detail-style endpoints, this returns a wrapper function that creates a new instance of the resource class & calls the correct view method for it.

Parameters

- **`init_args`** – (Optional) Positional params to be persisted along for instantiating the class itself.
- **`init_kwargs`** – (Optional) Keyword params to be persisted along for instantiating the class itself.

Returns View function

classmethod **`as_list`**(**init_args*, ***init_kwargs*)

Used for hooking up the actual list-style endpoints, this returns a wrapper function that creates a new instance of the resource class & calls the correct view method for it.

Parameters

- **`init_args`** – (Optional) Positional params to be persisted along for instantiating the class itself.
- **`init_kwargs`** – (Optional) Keyword params to be persisted along for instantiating the class itself.

Returns View function

classmethod **`as_view`**(*view_type*, **init_args*, ***init_kwargs*)

Used for hooking up the all endpoints (including custom ones), this returns a wrapper function that creates a new instance of the resource class & calls the correct view method for it.

Parameters

- **`view_type`** (*`string`*) – Should be one of `list`, `detail` or `custom`.
- **`init_args`** – (Optional) Positional params to be persisted along for instantiating the class itself.
- **`init_kwargs`** – (Optional) Keyword params to be persisted along for instantiating the class itself.

Returns View function

`bubble_exceptions`()

Controls whether or not exceptions will be re-raised when encountered.

The default implementation returns `False`, which means errors should return a serialized response.

If you'd like exceptions to be re-raised, override this method & return `True`.

Returns Whether exceptions should be re-raised or not

Return type boolean

`build_error`(*err*)

When an exception is encountered, this generates a JSON error message for display to the user.

Parameters **`err`** (*`Exception`*) – The exception seen. The message is exposed to the user, so beware of sensitive data leaking.

Returns A response object

`build_response`(*data*, *status=200*)

Given some data, generates an HTTP response.

If you're integrating with a new web framework, you **MUST** override this method within your subclass.

Parameters

- **`data`** (*`string`*) – The body of the response to send
- **`status`** (*`integer`*) – (Optional) The status code to respond with. Default is `200`

Returns A response object

`create`(**args*, ***kwargs*)
Allows for creating data via a POST on a list-style endpoint.

MUST BE OVERRIDDEN BY THE USER - By default, this returns `MethodNotImplemented`.

Returns May return the created item or `None`

`create_detail`(**args*, ***kwargs*)
Creates a subcollection of data for a POST on a detail-style endpoint.

Uncommonly implemented due to the rarity of having nested collections.

MUST BE OVERRIDDEN BY THE USER - By default, this returns `MethodNotImplemented`.

Returns A collection of data

Return type *list* or iterable

`delete`(**args*, ***kwargs*)
Deletes data for a DELETE on a detail-style endpoint.

MUST BE OVERRIDDEN BY THE USER - By default, this returns `MethodNotImplemented`.

Returns `None`

`delete_list`(**args*, ***kwargs*)
Deletes *ALL* data in the collection for a DELETE on a list-style endpoint.

Uncommonly implemented due to potential of trashing large datasets. Implement with care.

MUST BE OVERRIDDEN BY THE USER - By default, this returns `MethodNotImplemented`.

Returns `None`

`deserialize`(*method*, *endpoint*, *body*)
A convenience method for deserializing the body of a request.

If called on a list-style endpoint, this calls `deserialize_list`. Otherwise, it will call `deserialize_detail`.

Parameters

- **`method`** (*`string`*) – The HTTP method of the current request
- **`endpoint`** (*`string`*) – The endpoint style (`list` or `detail`)
- **`body`** (*`string`*) – The body of the current request

Returns The deserialized data

Return type `list` or `dict`

`deserialize_detail`(*body*)
Given a string of text, deserializes a (presumed) object out of the body.

Parameters **`body`** (*`string`*) – The body of the current request

Returns The deserialized body or an empty `dict`

deserialize_list(*body*)
Given a string of text, deserializes a (presumed) list out of the body.

Parameters **body** (*string*) – The body of the current request

Returns The deserialized body or an empty `list`

detail(**args*, ***kwargs*)
Returns the data for a GET on a detail-style endpoint.

MUST BE OVERRIDDEN BY THE USER - By default, this returns `MethodNotImplemented`.

Returns An item

Return type object or dict

handle(*endpoint*, **args*, ***kwargs*)
A convenient dispatching method, this centralized some of the common flow of the views.

This wraps/calls the methods the user defines (`list/detail/create` etc.), allowing the user to ignore the authentication/deserialization/serialization/response & just focus on their data/interactions.

Parameters

- **endpoint** (*string*) – The style of URI call (typically either `list` or `detail`).
- **args** – (Optional) Any positional URI parameter data is passed along here. Somewhat framework/URL-specific.
- **kwargs** – (Optional) Any keyword/named URI parameter data is passed along here. Somewhat framework/URL-specific.

Returns A response object

handle_error(*err*)
When an exception is encountered, this generates a serialized error message to return the user.

Parameters **err** (*Exception*) – The exception seen. The message is exposed to the user, so beware of sensitive data leaking.

Returns A response object

http_methods = {'list': {'PUT': 'update_list', 'POST': 'create', 'DELETE': 'delete_list', 'GET': 'list'}, 'detail': {'PUT

is_authenticated()
A simple hook method for controlling whether a request is authenticated to continue.

By default, we only allow the safe `GET` methods. All others are denied.

Returns Whether the request is authenticated or not.

Return type boolean

is_debug()
Controls whether or not the resource is in a debug environment.

If so, tracebacks will be added to the serialized response.

The default implementation simply returns `False`, so if you're integrating with a new web framework, you'll need to override this method within your subclass.

Returns If the resource is in a debug environment

Return type boolean

list(**args*, ***kwargs*)

Returns the data for a GET on a list-style endpoint.

MUST BE OVERRIDDEN BY THE USER - By default, this returns `MethodNotImplemented`.

Returns A collection of data

Return type *list* or iterable

prepare(*data*)

Given an item (`object` or `dict`), this will potentially go through & reshape the output based on `self.prepare_with` object.

Parameters **data** (*object or dict*) – An item to prepare for serialization

Returns A potentially reshaped dict

Return type dict

preparer = <restless.preparers.Preparer object>

request_body()

Returns the body of the current request.

Useful for deserializing the content the user sent (typically JSON).

If you're integrating with a new web framework, you might need to override this method within your subclass.

Returns The body of the request

Return type string

request_method()

Returns the HTTP method for the current request.

If you're integrating with a new web framework, you might need to override this method within your subclass.

Returns The HTTP method in uppercase

Return type string

serialize(*method*, *endpoint*, *data*)

A convenience method for serializing data for a response.

If called on a list-style endpoint, this calls `serialize_list`. Otherwise, it will call `serialize_detail`.

Parameters

- **method** (*string*) – The HTTP method of the current request
- **endpoint** (*string*) – The endpoint style (`list` or `detail`)
- **data** (*string*) – The body for the response

Returns A serialized version of the data

Return type string

serialize_detail(*data*)

Given a single item (`object` or `dict`), serializes it.

Parameters **data** (*object or dict*) – The item to serialize

Returns The serialized body

Return type string

`serialize_list`(*data*)
Given a collection of data (`objects` or `dicts`), serializes them.

Parameters **`data`** (`list or iterable`) – The collection of items to serialize

Returns The serialized body

Return type string

`serializer` **= <restless.serializers.JSONSerializer object>**

`status_map` **= {'delete_list': 204, 'update_list': 202, 'create': 201, 'delete': 204, 'list': 200, 'detail': 200, 'create_detail':**

`update`(**args*, ***kwargs*)
Updates existing data for a PUT on a detail-style endpoint.

MUST BE OVERRIDDEN BY THE USER - By default, this returns `MethodNotImplemented`.

Returns May return the updated item or `None`

`update_list`(**args*, ***kwargs*)
Updates the entire collection for a PUT on a list-style endpoint.

Uncommonly implemented due to the complexity & (varying) busines-logic involved.

MUST BE OVERRIDDEN BY THE USER - By default, this returns `MethodNotImplemented`.

Returns A collection of data

Return type *list* or iterable

`wrap_list_response`(*data*)
Takes a list of data & wraps it in a dictionary (within the `objects` key).

For security in JSON responses, it's better to wrap the list results in an `object` (due to the way the `Array` constructor can be attacked in Javascript). See http://haacked.com/archive/2009/06/25/json-hijacking.aspx/ & similar for details.

Overridable to allow for modifying the key names, adding data (or just insecurely return a plain old list if that's your thing).

Parameters **`data`** (`list`) – A list of data about to be serialized

Returns A wrapping dict

Return type dict

`restless.resources.`**`skip_prepare`**(*func*)
A convenience decorator for indicating the raw data should not be prepared.

restless.dj

class `restless.dj.`**`DjangoResource`**(**args*, ***kwargs*)
A Django-specific `Resource` subclass.

Doesn't require any special configuration, but helps when working in a Django environment.

classmethod **`as_detail`**(**args*, ***kwargs*)

classmethod **`as_list`**(**args*, ***kwargs*)

`build_error`(*err*)

`build_response`(*data*, *status=200*)

classmethod **`build_url_name`**(*name*, *name_prefix=None*)

Given a `name` & an optional `name_prefix`, this generates a name for a URL.

Parameters

- **`name`** (*`string`*) – The name for the URL (ex. 'detail')
- **`name_prefix`** (*`string`*) – (Optional) A prefix for the URL's name (for resolving). The default is `None`, which will autocreate a prefix based on the class name. Ex: `BlogPostResource` -> `api_blog_post_list`

Returns The final name

Return type string

`is_debug`()

classmethod **`urls`**(*name_prefix=None*)

A convenience method for hooking up the URLs.

This automatically adds a list & a detail endpoint to your URLconf.

Parameters **`name_prefix`** (*`string`*) – (Optional) A prefix for the URL's name (for resolving). The default is `None`, which will autocreate a prefix based on the class name. Ex: `BlogPostResource` -> `api_blogpost_list`

Returns A list of `url` objects for `include(...)`

restless.fl

restless.pyr

restless.it

restless.tnd

Serializers

restless.serializers

class `restless.serializers.`**`JSONSerializer`**

`deserialize`(*body*)

The low-level deserialization.

Underpins `deserialize`, `deserialize_list` & `deserialize_detail`.

Has no built-in smarts, simply loads the JSON.

Parameters **`body`** (*`string`*) – The body of the current request

Returns The deserialized data

Return type `list` or `dict`

`serialize`(*data*)

The low-level serialization.

Underpins `serialize`, `serialize_list` & `serialize_detail`.

Has no built-in smarts, simply dumps the JSON.

Parameters **`data`** (*`string`*) – The body for the response

Returns A serialized version of the data

Return type string

class `restless.serializers.`**`Serializer`**

A base serialization class.

Defines the protocol expected of a serializer, but only raises `NotImplementedError`.

Either subclass this or provide an object with the same `deserialize/serialize` methods on it.

`deserialize`(*body*)

Handles deserializing data coming from the user.

Should return a plain Python data type (such as a dict or list) containing the data.

Parameters **`body`** (*`string`*) – The body of the current request

Returns The deserialized data

Return type `list` or `dict`

`serialize`(*data*)

Handles serializing data being sent to the user.

Should return a plain Python string containing the serialized data in the appropriate format.

Parameters **`data`** (`list` or `dict`) – The body for the response

Returns A serialized version of the data

Return type string

Utils

restless.utils

class `restless.utils.`**`MoreTypesJSONEncoder`**(*skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, encoding='utf-8', default=None*)

A JSON encoder that allows for more common Python data types.

In addition to the defaults handled by `json`, this also supports:

- `datetime.datetime`
- `datetime.date`
- `datetime.time`
- `decimal.Decimal`
- `uuid.UUID`

`default`(*data*)

`restless.utils.`**`format_traceback`**(*exc_info*)

CHAPTER 5

Release Notes

restless v2.0.3

date 2016-11-21

This release adds a change which was in restless v2.0.2 but got lost in the backporting process - sorry, everybody!

Features

- Changed all `Resource` subclasses so that a 204 No Content response sends `text/plain` on `Content-Type`. (SHA: 116da9f & SHA: b10be61)

restless v2.0.2

date 2016-11-14

This release makes some long-needed changes on error handling for `Resource` and its subclasses, plus support for both Django >= 1.9 and Tornado >= 4.0 and allowing alphanumeric PKs on all supported frameworks.

Features

- Allowed PKs with dashes and alphanumeric digits. (SHA: e52333b)
- Reworked test suite so that it uses `tox` for simultaneously testing on CPython and PyPy, both 2.x and 3.x (SHA: 2035e21, SHA: 9ca0e8c, SHA: 3915980 & SHA: a1d2d96)
- Reworked `Resource` so that it throws a `NotImplementedError` instead of returning an `HttpResponse` from Django. (SHA: 27859c8)
- Added several `HttpError` subclasses. (SHA: e2aff93)

- Changed `Resource` so that it allows any serializable object on the response body. (SHA: 1e3522b & SHA: b70a492)

Bugfixes

- Changed `JSONSerializer` to throw a `BadRequest` upon a serialization error. (SHA: 8471463)
- Updated `DjangoResource` to use lists instead of the deprecated `django.conf.urls.patterns` object. (SHA: f166e4d & SHA: f94c500)
- Fixed `FieldsPreparer` behavior when parsing objects with a custom `__getattr__`. (SHA: 665ef31)
- Applied Debian's fix to Tornado tests for version 4.0.0 onwards. (SHA: 372e00a)
- Skips tests for all unavailable frameworks. (SHA: 8b81b17)

restless v2.0.1

date 2014-08-20

This release has many bugfixes & introduces support for Tornado.

Features

- Tornado support. (SHA: 2f8abcb)
- Enabled testing for Python 3.4. (SHA: 67cd126)
- Added a `endpoint` variable onto `Resource`. (SHA: da162dd)
- Added coveralls for coverage checking. (SHA: ec42c8b)

Bugfixes

- Updated the tutorial around creating data. (SHA: 542914f)
- Removed an erroneous underscore in the Flask docs. (SHA: 691b388)
- Fixed `JSONSerializer` determining if `str` or `bytes`. (SHA: 5376ac2)
- Corrected an example in the "Extending" docs. (SHA: b39bca5)
- Fixed docs in the validation docs. (SHA: 691b388)

restless v2.0.0

date 2014-05-23

This release improves the way data preparation & serialization are handled. It introduces these as separate, composable objects (`Preparer` & `Serializer`) that are assigned onto a `Resource`.

Porting from 1.X.X to 2.0.0

Porting is relatively straightforward in both the preparation & serialization cases.

Preparation

If you were supplying `fields` on your `Resource`, such as:

```
# posts/api.py
from restless.dj import DjangoResource

from posts.models import Post


class PostResource(DjangoResource):
    fields = {
        'id': 'id',
        'title': 'title',
        'author': 'user.username',
        'body': 'content',
        'posted_on': 'posted_on',
    }
```

Porting is simply 1.adding an import & 2. changing the assignment.:

```
# posts/api.py
from restless.dj import DjangoResource
# 1. ADDED IMPORT
from restless.preparers import FieldsPreparer

from posts.models import Post


class PostResource(DjangoResource):
    # 2. CHANGED ASSIGNMENT
    preparer = FieldsPreparer{
        'id': 'id',
        'title': 'title',
        'author': 'user.username',
        'body': 'content',
        'posted_on': 'posted_on',
    }
```

Serialization

Serialization is even easier. If you performed no overridding, there's nothing to update. You simply get the new `JSONSerializer` object automatically.

If you were overriding either `raw_deserialize` or `raw_serialize`, you should create a new `Serializer` subclass & move the methods over to it, changing their signatures as you go. Then assign an instance of your new `Serializer` subclass onto your ‘‘Resource‘‘(s).

Unported YAML serialization:

```
import yaml

from restless import Resource


class MyResource(Resource):
    def raw_deserialize(self, body):
        return yaml.safe_load(body)

    def raw_serialize(self, data):
        return yaml.dump(data)
```

Ported serialization:

```
import yaml

from restless import Resource
from restless.serializers import Serializer


class YAMLSerializer(Serializer):
    def deserialize(self, body):
        return yaml.safe_load(body)

    def serialize(self, data):
        return yaml.dump(data)


class MyResource(Resource):
    serializer = YAMLSerializer()
```

Features

- Added syntax highlighting to docs. (SHA: d398fdb)
- Added a `BAD_REQUEST` constant & associated `BadRequest` error. (SHA: 93d73d6, SHA: 8d49b51 & SHA: a719c88)
- Moved to composition for data preparation & serialization. (SHA: 38aabb9)

restless v1.4.0

date 2014-02-20

This release improves the way errors are handled (serialized tracebacks in debug), making them more consistent. It also improves Django's support for `ObjectDoesNotExist/Http404` & switched to using `py.test` for testing.

Features

- Better not-found behavior in Django. (SHA: 7cd2cfc)
- Improved `Http404` behavior in Django. (SHA: 44b2e5f)
- Switched to `py.test`. (SHA: 30534a7)
- Better error handling support. (SHA: ae5a9cb)

Bugfixes

- Skips Itty's tests if it is not available. (SHA: b4e859b)

restless v1.3.0

date 2014-01-29

This release adds support for Itty! This only works under Python 2.X for now, due to itty itself.

Features

- Added support for Itty. (SHA: 5cc4acd)

restless v1.2.0

date 2014-01-15

BACKWARD-INCOMPATIBLE: This release alters the Pyramid `add_views` method signature slightly, to be more idiomatic. It changed from `endpoint_prefix` to become `routename_prefix`.

Given that the Pyramid support was first released yesterday & this is an optional argument, the hope is the impact of this change is low. This should be (barring any security fixes) the only backward-incompatible change before v2.0.0.

Bugfixes

- Altered the `PyramidResource.add_views` method signature, renaming the `endpoint_prefix` to `routename_prefix`. (SHA: 5a7edc8)

restless v1.1.0

date 2014-01-14

This release adds Pyramid support, easier-to-override serialization, more documentation & fixed Flask tests/`is_debug`.

Features

- Added support for Pyramid (`restless.pyr.PyramidResource`). Thanks to binarydud for the patch! (SHA: 27e343e)
- Added the `Resource.raw_deserialize` & `Resource.raw_serialize` methods to make changing the serialization format more DRY/easier. (SHA: 9d68aa5)
- Added more documentation on how to extend Restless. (SHA: 0be1346 & SHA: 730dde1)

Bugfixes

- Fixed the Flask tests to no longer be skipped. (SHA: 89d2bc7)
- Fixed `FlaskResource.is_debug` to now do the correct lookup. (SHA: 89d2bc7)

restless v1.0.0

date 2014-01-12

Initial production-ready release! Whoo hoo!

Includes:

- Full GET/CREATE/UPDATE/DELETE
- Django support
- Flask support
- Real, live docs
- OMG Tests Like Wow

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index