django-conduit Documentation

Release 0.0.1

Alec Koumjian

Apr 24, 2017

Contents

Easy and powerful REST APIs for Django.

CHAPTER 1

Why Use Django-Conduit?

- Easy to read, easy to debug, easy to extend
- Smart and efficient *Related Resources*

See the full list of features.

CHAPTER 2

Table of Contents

Filtering and Ordering

During a get list view, it is useful to be able to filter or rearrange the results. Django-Conduit provides a few helpful properties and hooks to to filter your resources.

Server Side Filters to Limit Access

The `default_filters` dict on a ModelResource's Meta class will apply the listed queryset filters before fetching results. The keys in `default_filters` ought to be a valid Queryset filter method for the specified model. Here is an example that only returns Foo objects that have a name starting with the the word 'lamp':

```
class FooResource(ModelResource):
        class Meta(ModelResource.Meta):
                default_filters = {
                        'name__startswith': 'lamp'
                }
```

The default filters will eventually be applied to the queryset during the `apply_filters` method, resulting in something like this:

```
filtered_instances = Foo.objects.filter(name__startswith='lamp')
```

Client Side Filtering with Get Params

API consumers often need to be able to filter against certain resource fields using GET parameters. Filtering is enabled by specifying the `allowed_filters` array. The array takes a series of Queryset filter keywords:

```
class FooResource(ModelResource):
        class Meta(ModelResource.Meta):
                allowed_filters = [
```

```
                        'name__icontains',
                        'created__lte',
                        'created__gte',
                        'bar__name'
                ]
```

In the above example, API consumers will be allowed to get Foo objects by searching for strings in the Foo name, or by finding Foos created before or after a given datetime.

Note: Each Queryset filter has to be specified using the entire filter name. While verbose, this allows custom or related field parameters such as `bar__name` to be easily specified.

Ordering Results

If you want to specify the default order for objected, returned, you can simply specify the `order_by` string using the `default_ordering` Meta field:

```
class FooResource(ModelResource):
        class Meta(ModelResource.Meta):
                default_ordering='-created'
```

The value of `default_ordering` should be the same one you would use when performing order_by on a queryset. The above example will result in the following operation:

```
Foo.objects.order_by('-created')
```

To allow API consumers to order the results, the `allowed_ordering` field is an array of valid ordering keys:

```
class FooResource(ModelResource):
        class Meta(ModelResource.Meta):
                allowed_ordering = [
                        'created',
                        '-created'
                ]
```

Note how the forward and reverse string both have to be specified. This is to provide precise control over client ordering values.

How Filters & Ordering are Applied

Filtering and ordering happens inside two steps in the default conduit pipeline. The first happens inside `process_filters`. To determine order, first the method looks for an order_by GET parameter. If none are specified, it defaults to the `default_ordering` attribute. If the order_by parameter is not a valid value, the client receives a 400.

The filters start with the `default_filters` dictionary. This dictionary is then updated from filters specified in the GET parameters, provided they are specified in `allowed_filters`.

After the order_by and filters are determined, their values are sent forward in the kwargs dictionary where they are picked up again in `pre_get_list`. This is the method that first applies the `kwargs['order_by']` value, and then applies the values inside `kwargs['filters']`. It stores the ordered and filtered queryset inside of `kwargs['objs']`. The objects are then subject to authorization limits and paginated inside `get_list` before the final set of objects is determined.

Related Resources & Objects

`django-conduit` treats related ForeignKey, GenericForeignKey, and ManyToMany objects in an intuitive and efficient manner. You can use related resources to treat them similarly to Django's ORM, or you can default to their simple behavior as pointers to primary keys.

Related Resource Fields

Conduit lets you use other ModelResources for your related object fields. You can use a related resource by referencing it in the Fields metaclass. The below FooResource example using two related resource fields:

```
class FooResource(ModelResource):
    class Meta(ModelResource.Meta):
        model = Foo
    class Fields:
        bar = ForeignKeyField(attribute='bar', resource_cls='api.views.BarResource')
        bazzes = ManyToManyField(attribute='bazzes', resource_cls='api.views.
↪BazResource', embed=True)

class BarResource(ModelResource):
    class Meta(ModelResource.Meta):
        model = Bar


class BazResource(ModelResource):
    class Meta(ModelResource.Meta):
        model = Baz
```

Using a related resource lets you embed the entire resource data inside of the parent resource. One of the resources above is set to embed=True, while the other is not and will default to the `resource_uri`. An example of the above FooResource would look like this:

```
{
    "bar": "/api/v1/bar/23/",
    "name": "stuffs",
    "id": 1,
    "bazzes": [
        {
            "resource_uri": "/api/v1/baz/1/",
            "id": 1,
            "name": "Baz 1"
        },
        {
            "resource_uri": "/api/v1/baz/7/",
            "id": 7,
            "name": "Baz 7"
        }
    ],
    "resource_uri": "/api/v1/foo/1/"
}
```

Updating Related Resources

The **real** power of using related resources is that they follow the rules of the resource they point to. Using our previous example, let's say you update one of the Baz objects in place and then send a PUT to our parent resource at `/api/v1/foo/1/`:

```
{
        ...
    "bazzes": [
        {
            "resource_uri": "/api/v1/baz/1/",
            "id": 1,
            "name": "MODIFIED BAZ NAME"
        },
        {
            "resource_uri": "/api/v1/baz/7/",
            "id": 7,
            "name": "Baz 7"
        }
    ],
    ...
}
```

The Baz object with id == 1 will now have the name "MODIFIED BAZ NAME" unless the BazResource determines the request is not authorized (using the methods described in *Access & Authorization<access_authorization>*) or if the data doesn't validate, etc.

If you include data for a related resource without a primary key, it will created the related object for you and add it to the parent resource object. For example, if you send a PUT to our /api/v1/foo/1/ resource with the following data:

```
{
        ...
    "bazzes": [
        {
            "resource_uri": "/api/v1/baz/1/",
            "id": 1,
            "name": "MODIFIED BAZ NAME"
        },
        {
            "resource_uri": "/api/v1/baz/7/",
            "id": 7,
            "name": "Baz 7"
        },
        {
            "name": "New Baz"
        }
    ],
    ...
}
```

The related BazResource will attempt to create a new Baz as if you had sent a POST to `/api/v1/baz/`. Then it will add the new Baz object to Foo's ManyToMany field. In the return response, the object will be filled in with its new id and resource_uri.

Similarly if you PUT to `/api/v1/foo/1/` and omit one of the existing Baz objects, it will remove it from the ManyToMany field. It will NOT delete the Baz object, however:

```
{
        ...
    "bazzes": [
        {
            "resource_uri": "/api/v1/baz/1/",
            "id": 1,
            "name": "MODIFIED BAZ NAME"
        }
    ],
    ...
}
```

The above request will remove all but the Baz 1 object from Foo's bazzes field.

GenericForeignKeyField

When a model should relate to multiple types of models, Django provides the GenericForeignKey field and ContentTypes framework, allowing a model to relate to multiple models based on a ContentType and Id.

If a model is using Django's `GenericForeignKey`, the `GenericForeignKeyField` provided by Conduit can be used to setup a resource.

Here's an example of a model using a GenericForeignKey:

```
from django.contrib.contenttypes import generic
from django.contrib.contenttypes.models import ContentType
from django.db import models

class Item(models.Model):
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    content_object = generic.GenericFreignKey('content_object', 'object_id')
```

A `ModelResource` using Conduit's `GenericForeignKeyField` would look like this:

```
from conduit.api import ModelResource
from conduit.api.fields import GenericForeignKeyField

from myapp.models import Item

class ItemResource(ModelResource):
    class Meta(ModelResource.Meta):
        model = Item

    class Fields:
        content_object = GenericForeignKeyField(
            attribute='content_object',
            resource_map={
                'Bar': 'api.views.BarResource',
                'Foo': 'api.views.FooResource',
            }
        )
```

The `resource_map` attribute enables the resource to lookup a related resource based on it's `Model` defined by the *ContentType*. In this example we use the *Foo* and *Bar* models and resources explained above.

Here is an example of what the api would return for a GET request:

```
{
    "object_id": 1,
    "content_object": "/api/v1/bar/1/",
    "id": 1,
    "content_type": 8,
    "resource_uri": "/api/v1/item/1/"
}
```

If `embed=True` is set, then the full related resource will be included using the same behavior for a `ForeignKeyField` or `ManyToManyField`.

Customizing Related Resource Fields

The default ForeignKeyField and ManyToManyField that ship with Conduit can easily be subclassed and customized. The fields work very similarly to ModelResources, except instead of a single Meta.conduit pipeline, they have two pipelines. One if for updating from request data, and the other is for fetching the existing resource.

A subclassed FK field which adds a custom additional step to the pipeline would look like this:

```
class CustomForeignKeyField(ForeignKeyField):
    dehydrate_conduit = (
        'objs_to_bundles',
        ## Adds a custom step when grabbing and object
        ## and turning it to json data
        'myapp.resources.CustomResource.custom_method'
        'add_resource_uri',
    )

    save_conduit = (
        'check_allowed_methods',
        'get_object_from_kwargs',
        'hydrate_request_data',
        ## Adds a custom step when preparing data
        ## for updating / creating new object
        'myapp.resources.CustomResource.custom_method'
        'initialize_new_object',
        'save_fk_objs',
        'auth_put_detail',
        'auth_post_detail',
        'form_validate',
        'put_detail',
        'post_list',
        'save_m2m_objs',
    )
```

Default Behavior

By default, conduit will serialize your model's related object fields by their raw value. A ForeignKey field will produce the primary key of your related object. A ManyToMany field will produce a list of primary keys.

An example resource Foo has one FK and one M2M field:

```
class Foo(models.Model):
        name = models.CharField(max_length=255)
```

```
        bar = models.ForeignKey(Bar)
        bazzes = models.ManyToManyField(Baz)
```

Will produce a detail response looking like this:

```
{
        "name": "My Foo",
        "bar": 45,
        "bazzes": [5, 87, 200],
        "resource_uri": "/api/v1/foo/1/"
}
```

When updating a ForeignKey field, conduit will set the model's [field]_id to the integer you send it. Be careful not to set it to a nonexistent related model, since there are not constraint checks done when saved to the database.

Similarly, when updated a ManyToMany field and give it a nonexistent primary key, the add will silently fail and the invalid primary key will not enter the ManyToMany list.

Important: Updating raw primary keys will not produce errors for invalid keys.

Access, Authorization & Permissions

Django-Conduit provides several 'out of the box' ways to control access to your resources.

Note: See the *Filtering & Ordering'* guide to limit retrievable objected based on static values.

Allowed Methods

One quick way to prevent create or update access to a resource is to limit the allowed http methods:

```
class FooResource(ModelResource):
        class Meta(ModelResource.Meta):
                model = Foo
                allowed_methods = ['get']
```

The above example will prevent sending put, post, or delete requests to the Foo Resource. Currently only 'get', 'put', 'post', and 'delete' are valid values.

Authorization Hooks

For granular permissions on individual objects, Django-Conduit provides a list of ready made hooks for you to implement your permission checks. The hook methods follow a naming pattern of `auth_[method]_[list/detail]`. The method being the http method, and list/detail whether it is an action on a list url (`api/v1/foo`) or a detail url (`api/v1/foo/1`)

It is entirely up to you how you want to handle permissions, but here are a couple suggestions.

1. Filter objects a user can retrieve based on ownership. Ownership is determined by the user specified in a owner field on the model:

```
@match(match=['get', 'list'])
def auth_get_list(self, request, *args, **kwargs):
    objs = kwargs['objs']
    objs = objs.filter(owner=request.user)
    return (request, args, kwargs)
```

2. Disable update access if a user does not own an object:

```
@match(match=['put', 'detail'])
def auth_put_detail(self, request, *args, **kwargs):
    # single obj is still found in 'objs' kwarg
    obj = kwargs['objs'][0]
    if request.user != obj.owner:
            # HttpInterrupt will immediately end processing
            # the request. Get it by:
            # from conduit.exceptions import HttpInterrupt
        response = HttpResponse('', status=403)
        raise HttpInterrupt(response)
    return (request, args, kwargs)
```

Note: It is important that you include the `@match` wrapper so that the check is only executed on the right requests. More about Method Subscriptions

Here is a full list of the authorization hook methods:

```
'auth_get_detail',
'auth_get_list',
'auth_put_detail',
'auth_put_list',
'auth_post_detail',
'auth_post_list',
'auth_delete_detail',
'auth_delete_list'
```

Some of these already have checks, such as `auth_put_list`, which will automatically raise an HttpInterrupt. This is because sending a PUT to a list endpoint is not a valid request.

Forms & Validation

With django-conduit you can use Django's ModelForms to easily validate your resources. You can specify a form to be used by assigning the `form_class` on the resource's Meta class:

```
from example.forms import FooForm

class FooResource(ModelResource):
    class Meta(ModelResource.Meta):
        form_class = FooForm
```

The form validation happens during POST and PUT requests. Any data sent in the request that does not correspond to the model's field names will be discarded for validation.

If errors are found during validation, they are serialized into JSON and immediately return a 400 Http response. If an error occurs while validating a related field, the JSON error is specified as having occured within that related field.

Conduit Overview

What is a Conduit?

Conduits are views that send requests through a simple list of functions to produce a response. This process is often called a pipeline (hence the name conduit). Here is an example:

```
conduit = (
    'deserialize_json',
    'run_form_validation',
    'response'
)
```

Each of the items in the `conduit` tuple reference a method. Each method is called in succession. This is very similar to how Django's `MIDDLEWARE_CLASSES` work. A conduit pipeline is specified in a `Conduit` view like this:

```
class FormView(Conduit):
    """
    Simple view for processing form input
    """
    form_class = MyForm

    class Meta:
        conduit = (
            'deserialized_json_data',
            'validate_form',
            'process_data',
            'response'
        )
```

Conduit Methods

All functions in a conduit pipeline take the same four parameters as input.

1. **self** The Conduit view instance
2. **request** The Django request object
3. ***args** Capture variable number of arguments
4. ****kwargs** Capture variable number of keyword arguments

The methods also return these same values, though they may be modified in place. The only response that is different is the last, which must return a response, most likely an `HttpResponse`.

Warning: The last method in a conduit must return a response, such as HttpResponse

Inheriting & Extending

To inherit the conduit tuple from another `Conduit` view, your metaclass must do the inheriting. We can use a different form with the above view by inheriting its methods and conduit, while overriding its form_class:

```
class OtherFormView(FormView):
    """
    Process a different form
    """
    form_class = OtherForm

    class Meta(FormView.Meta):
        pass
```

If you want to add or remove a step from another conduit, you must specify the new pipeline in its entirety. Here is a simple but not recommended example that extends our view from above by adding a `publish_to_redis` method:

```
class PublishFormView(FormView):
    """
    Process a form and publish event to redis
    """
    form_class = OtherForm

    class Meta:
        conduit = (
            'deserialized_json_data',
            'validate_form',
            'process_data',
            'publish_to_redis',
            'response'
        )
```

In this example, we didn't inherit the meta class since we were overriding conduit anyway.

Warning: Class inheritance is NOT the recommended way to customize your Conduit views.

While inheriting views, including multiple inheritance, is very familiar to Django developers, there is another more flexible way to extend your Conduit views. The methods in the conduit can reference any namespaced function, as long as they take the correct 4 input parameters.

Using namespaced methods, the recommended way to create the above view would look like this:

```
class PublishFormView(Conduit):
    """
    Process a form and publish event to redis
    """
    form_class = OtherForm

    class Meta:
        conduit = (
            'myapp.views.FormView.deserialized_json_data',
            'myapp.views.FormView.validate_form',
            'myapp.views.FormView.process_data',
            'publish_to_redis',
            'myapp.views.FormView.response'
        )
```

The advantage here over multiple inheritance is that the source of the methods is made explicit. This makes debugging much easier if a little inconvenient.

About

Django-Conduit is meant to accomodate the most common API patterns such as dealing with related resources and permissions access. It was designed to be very easy to read, debug, and extend, and ultimately make your code more readable as well.

The conduit based views are aimed to respect and take advantage of the request-response cycle. This includes ideas such as transaction support, minimizing database queries, but most importantly a pipeline of events which can be easily intercepted, extended, and introspected.

We believe Django-Conduit is the fastest way to develop a REST API that works like you would expect or want it to.

Why not Django-REST-Framework?

DRF was built around Django's Class Based Views and as a result produces highly cryptic yet verbose view classes. Interacting with CBVs often involves using special class methods that hopefully hook into the right part of the request-response cycle. Class based inheritance makes a very frustrating experience when developing complex views.

Why not Django-Tastypie?

Django-Conduit is heavily inspired by Tastypie. It uses a similar declarative syntax for resources, as well as a similar syntax for related fields and metaclasses. Conduit was partly built to simplify and streamline a lot of Tastypie's internal logic. While built rather differently, Conduit aims to emulate some of Tastypie's best features.

CHAPTER 3

Getting Started

Django-Conduit will automatically create your starting api based on your existing models.

1. Install via PyPI: `pip install django-conduit`
2. Add the following to your INSTALLED_APPS:

```
INSTALLED_APPS = (
    ...
    'conduit',
    # 'api',
)
```

3. Generate your API app by running the following:

```
./manage.py create_api [name_of_your_app] --folder=api
```

4. Uncomment 'api' in your INSTALLED_APPS
5. Point your main URLconf (normally project_name/urls.py) to your new 'api' app:

```
urlpatterns = patterns('',
    ...
    url(r'^api/', include('api.urls')),
    ...
)
```

6. Visit `localhost:8000/api/v1/[model_name]` to fetch one of your new resources!

All your new resources will be defined in api/views.py, and they will be registered with your Api object in api/urls.py.

Topics

- *Access & Authorization*
- Custom Fields
- *Forms & Validation*
- *Conduit Views*
- ModelResource
- Customizing Resources‘

CHAPTER 4

Indices and tables

- genindex
- modindex
- search