Dynamic REST Documentation

Release 1.3.15

ant@altschool.com, ryo@altschool.com

February 13, 2016

Contents

CHAPTER 1

Dynamic REST Tutorial

Contents

1.1 Introduction

This document will take you through the basics (and not-so-basics) of developing Dynamic REST APIs. Dynamic REST (or "DREST") is an extension to Django REST Framework ("DRF") that makes it easier to implement uniform REST APIs with powerful features like field inclusion/exclusion, flexible filtering, pagination, and efficient "sideloading" of related/nested data.

1.1.1 Setup

Before starting the tutorial, you'll need to set up Dynamic REST in your dev environment. Refer to the README for details, but it should look something like:

```
$ git clone git@github.com:AltSchool/dynamic-rest.git
$ cd dynamic_rest
$ make fixtures
$ make server
```

This should start up Django at http://localhost:9002.

1.2 Minimal API

We'll first implement a very basic API that exposes a minimal view of a basic model.

1.2.1 Model

We will be creating an API for the following model. The model has conveniently already been created for you, and is listed here for reference (the actual code is in tests/models.py):

```
class Event(models.Model):
    name = models.TextField()
    status = models.TextField()
    location = models.ForeignKey('Location', null=True, blank=True)
    users = models.ManyToManyField('User')
```

1.2.2 Minimal serializer

All objects returned by APIs must have a serializer. The serializer is effectively your contract with the API user – it's where you declare which fields are available, what the field data types are, which fields are deferred, and so on.

Add the following to tests/serializers.py (note that Event needs to be imported at the top):

```
from tests.models import Event

class EventSerializer(DynamicModelSerializer):
    class Meta:
        model = Event
        name = 'event'
        fields = ('id', 'name', 'status')
```

The example above is a complete serializer. In this particular example, the Meta class contains all the information the framework needs. It knows what the underlying model is, which fields to include in the output, and what to label the returned data. (If you're wondering why we don't have location and users, don't worry, we'll get to those in a bit.)

1.2.3 Minimal ViewSet

Next we'll add a viewset, which implements the API (in tests/viewsets.py).:

```
from tests.serializers import EventSerializer
from tests.models import Event

class EventViewSet(DynamicModelViewSet):
    """
    Events API.
    """
    queryset = Event.objects.all()
    serializer_class = EventSerializer
```

This implements a simple API, which provides full CRUD capabilities, as well as list functionality with filtering, field inclusion, sideloading, and pagination.

1.2.4 Map to URL

Update the URLs file (in tests/urls.py):

```
router.register(r'events', viewsets.EventViewSet)
```

Note that, as a convention, the URL resource name should be the pluralization of the name declared in the serializer Meta class (for irregular pluralizations, a plural_name attribute can be set in the Meta class).

1.2.5 Try it out

Try out a few URLs (assumes you're running locally on port 9002):

- **http://localhost:9002/events/**
  - You should get a pretty HTML-ified view of the API. This view is generated by Django REST Framework, and is only returned if text/html is in the Accept header (i.e. in a non-XHR browser request).
  - Click on the OPTIONS button at the top-right, to get information about the fields available in the response.
- **http://localhost:9002/events/?page=2&per_page=1**
  - Pagination works out of the box.
- **http://localhost:9002/events/?filter{name}=Event+3**
  - Filtering works out of the box too.

1.2.6 Serializer with relations

We'll extend the previous example by adding a couple of relational fields to the serializer:

```
class EventSerializer(DynamicModelSerializer):
    location = DynamicRelationField('LocationSerializer', deferred=False)
    users = DynamicRelationField(
        'UserSerializer', many=True, deferred=True)

    class Meta:
        model = Event
        name = 'event'
        fields = ('id', 'name', 'status', 'location', 'users')
```

Here, we've added the two relational fields. A few notes:

- For relational fields, map a field name to a DynamicRelationField object, and then pass in the serializer to use when that field is being serialized.
- If using a serializer that hasn't been defined yet, you can use the serializer name (or full import path) as a string.
- We set deferred=True on users, which means that field will not be returned unless specifically requested. For many-relations, this could yield better performance since that saves a DB query.
- Relational fields can be a Foreign-Key (one to many), a Many-to-Many, or Many-to-One (a.k.a inverse of a foreign key).

Now try some queries:

- **http://localhost:9002/events/**
  - Now you should see that location is included, though note that only the ID is returned.
  - Users still don't show up, because we set deferred=True
- **http://localhost:9002/events/?filter{location.name}=1**
  - You can filter by relations. Here, we filter for events where the location's name is "1"
- **http://localhost:9002/events/?include[]=users**
  - Now you get users, but, again, only IDs by default
- **http://localhost:9002/events/?include[]=users.***
  - Now we get users sideloaded
- **http://localhost:9002/events/?include[]=users.*&filter{users|location}=1**
  - Now we get users, but only users whose location is 1 (the '|' operator indicates we want to filter the related objects themselves (i.e. users), not the root object (events)).
- Object creation through POST request also works. At the bottom of the page, click on the "Raw data" tab, and paste in the following JSON (or your own) and hit the POST button:

```
{
    "event": {
        "name": "new event!",
        "status": "current",
        "location": 2,
        "users": [1,2]
    }
}
```

- **http://localhost:9002/events/5/?include[]=users**
  - Single resource GET works, as do PUT, PATCH and DELETE commands.
  - Note that you get the same field inclusion/deferral behavior for single resource retrieval as you do in list queries.
- **http://localhost:9002/events/5/users/**
  - DREST also auto-generates an endpoint to return just the related data (useful as link objects).

1.2.7 Note on optimizations

DREST has reasonable optimization strategies built in, which frees up the API developer from having to understand and employ Django optimization strategies. Some optimizations currently implemented include:

- Prefetching of sideloaded fields - For example, when we sideloaded users above, DREST internally constructed a Prefetch query so Django only performed 2 queries: one for events, one for users.
- Automatic prefetching - If we were to turn deferred off on users, DREST will automatically prefetch users (otherwise Django will issue a separate query per event object).
- Field selection - DREST will only request fields that are necessary from the DB, which could reduce data transfer between Django and the DB.
- Prefetch filtering for sideloads - When we filtered sideloads, the filtering criteria was converted into a Django query and attached to the prefetch request so that it could be converted into the appropriate SQL query.

1.3 Advanced Topics

As you saw, simple APIs can be implemented with very little code. Obviously, life is more complicated than that...

1.3.1 Serializer field aliasing

Sometimes we want serializer fields to be named something other than the underlying model (or Django-ism like `*_set`). We can do this by using the DRF source field attribute. Try modifying the EventSerializer thusly:

```
class EventSerializer(DynamicModelSerializer):
    location = DynamicRelationField('LocationSerializer', deferred=False)
    participants = DynamicRelationField(
            'UserSerializer', source='users', many=True, deferred=True)

    class Meta:
        model = Event
        name = 'event'
        fields = ('id', 'name', 'status', 'location', 'participants
')
```

Everything still works as expected:

- http://localhost:9002/events/?include[]=participants&filter{participants|location}=1

1.3.2 Query parameter injection

Sometimes we want to modify DREST's default behaviors. Perhaps we want default filters applied. Or we want some relations to be sideloaded by default. One easy way to do this is through query parameter injection. Try adding the following method to the EventViewSet:

```
class EventViewSet(DynamicModelViewSet):
    # ...

    def list(self, request, *args, **kwargs):
        # sideload location by default
        request.query_params.add('include[]', 'location.')

        # filter for status=current by default
        status = request.query_params.get('filter{status}')
        if not status:
            request.query_params.add('filter{status}','current')

        return super(EventViewSet, self).list(request, *args, **kwargs)
```

Checkout the default results: http://localhost:9002/events/

1.3.3 Queryset override

By default, DREST/DRF will query the model declared in the viewset's serializer, which is to say, all objects in that model are in-scope and query-able. If you want to change that, you can override the `get_queryset()` method in your viewset. One use-case might be to dynamically apply filters that can't/shouldn't be overridden by `filter{}` params. In a viewset, you might do something like:

```
def get_queryset(self, *args, **kwargs):
    is_admin = user_is_admin(self.user)
    if is_admin:
        return Foo.objects.all()
    else:
        return Foo.objects.filter(creator=self.user)
```

In this hypothetical example, this would constrain the scope of query-able objects for non-admin users to only those objects created by them.

(Note: `Foo.objects.all()` does not actually return any objects. It returns a QuerySet which only gets evaluated when its contents are requested, and until a QuerySet is evaluated, it is possible to keep chaining more filters. Internally, DREST/DRF takes the QuerySet returned by get_queryset and modifies it, before it is eventually evaluated.)

1.3.4 Setting serializer context

The DRF way of setting serializer context works as well (serializer context is accessible within the serializer as self.context).:

```
class FooViewSet(DynamicModelViewSet):
    # ....
    def get_serializer_context(self):
        context = super(FooViewSet, self).get_serializer_context()
        foo = self.request.query_params.get('foo')
        # modify context
        return context
```

1.3.5 DREST-ifying Existing ViewSet

When migrating existing APIs, it might be possible to "layer" on DREST into an existing ViewSet by using WithDynamicViewSetMixin. Note that getting the old class to play nice might require some shenanigans (see super below):

```
from dynamic_rest.viewsets import WithDynamicViewSetMixin

class NewViewSet(WithDynamicViewSetMixin, OldViewSet):
    # ...
    def list(self, request, *args, **kwargs):
        # ...

        # Skip parent's list() method
        return super(OldViewSet, self).list(request, *args, **kwargs)
```

1.3.6 DREST-ifying Existing Serializer

As with ViewSets, there's a mixin to DREST-ify an existing serializer. Same shenanigans warning applies as above:

```
class NewFooSerializer(WithDynamicModelSerializerMixin, OldFooSerializer):
    # Must override Meta class with DREST attributes
    class Meta:
        name = 'foo'
        model = Foo
```

1.3.7 Customizing Serializers

Occasionally, it is useful or necessary to customize serializers themselves. One simple way to customize how objects get serialized in DRF, is to override the `to_representation()` method:

```
class FooSerializer(DynamicModelSerializer):
    # ...
    def to_representation(self, instance):
        # modify instance here
        # ...

        # pass through default serializer:
        data = super(FooSerializer, self).to_representation(instance)

        # modify data (dict) here
        # ...
        return data
```

1.3.8 Computed Fields

Historically, we've implemented computed fields using SerializerMethodField, which led to a proliferation of one-off methods with ad hoc implementations. SerializerMethodFields are also problematic because they may not play nice with standard features (like inclusion/sideloading), and don't have declared data types. In DREST, we introduced a DynamicComputedField base-class, to encourage developers to define and implement (or use) reusable computed fields.:

```
from dynamic_rest.fields import DynamicComputedField

class HasPermsField(DynamicComputedField):
    def __init__(self, required_perms, **kwargs):
        self.required_perms = required_perms
        kwargs['field_type'] = bool
        super(HasPermsField, self).__init__(**kwargs)

    def get_attribute(self, instance):
        # Override to get field value
        perm_checker = self.context['permission_checker']
        user = self.context['user']
        return perm_checker.has_perms(user, instance, self.required_perms)

    def to_representation(self, value):
        # Override if we need to convert complex data-type to a
        # primitive data type that's serializable.
        return bool(value)

# in serializer:
class DocumentSerializer(...):
    can_write = HasPermsField('w')
    can_destroy = HasPermsField('d')
```

1.3.9 Ephemeral Serializers

Sometimes, the output objects don't map cleanly to any existing model. One approach is to return adhoc JSON, but that makes it difficult or cumbersome to support features like dynamic sideloading/inclusion or pagination (which in turn leads to inconsistent and unpredictable implementations). DREST attempts to address these issues by providing limited support for serializers that are not backed by models.

One use-case for ephemeral serializers is when we want to represent data that is context-sensitive. Consider the earlier query:

http://localhost:9002/events/?include[]=users.&filter{users|location}=1

Note that Event objects returned by this API call only contain users whose location is 1. However, there is nothing in the object indicating that its user set is incomplete, so if that object is cached, there's no way to know by looking at the object whether the user-set should be considered complete or not.

An alternative representation of the data might look something like this:

```
class EventLocationUsersSerializer(DynamicEphemeralSerializer):
    class Meta:
        name = 'event-location-users'

    id = CharField()
    user_location = DynamicRelationField('LocationSerializer')
    users = DynamicRelationField('UserSerializer', many=True)
    event = DynamicRelationField('EventSerializer')

    def to_representation(self, event):
        location = self.context['location']

        # Construct dict representing data we want.
        data = {}
        data['id'] = data['pk'] = "%s--%s" % (event.id, location.id)
        data['user_location'] = location
        data['users'] = list(event.users.all())
        data['event'] = event

        # Construct EphemeralObject instance, and let DREST serialize it.
        event_location = EphemeralObject(data)
        return super(EventLocationSerializer, self).to_representation(
            event_location
        )
```

This serializer will take an Event object with its users set pre-filtered, and emit an object with a unique ID and context that makes it safe for caching and re-use. If hooked up correctly to a viewset, the resulting API would have support for DREST features like field inclusion/sideloading, auto-generated OPTIONS response, and pagination.

1.3.10 Embedded Fields

The DynamicRelationField's embed option will ensure that the related objects are always included, and also returned nested in the parent object. This is useful for cases where a nested response is desired for legacy reasons, and/or when the related objects should always be returned with the parent objects, and expecting the caller to always include[] those fields is burdensome.:

```
class BlogPostSerializer(DynamicModelSerializer):
    # ...

    author = DynamicRelationField(UserSerializer, embed=True)
```

1.3.11 Default Filtering of Related Objects

DynamicRelationFields can have a default queryset/filter. While clients can apply filtering on related objects (and viewsets can do the same through query injection), sometimes a default filter needs to be applied in all cases, and you don't want to leave it to the client to know that. An example might be if we wanted to supply a `Location.upcoming_events` field, where we want to filter `Location.events` for active events and spare the client of having to do `filter{events|status}=current`.

To solve this problem, a default queryset can be defined in the DynamicRelationField:

```
class LocationSerializer(DynamicModelSerializer):
    # ...

    upcoming_events = DynamicRelationField(
        EventSerializer,
        source='events',
        many=True,
        queryset=Event.objects.filter(status='current')   # <--
    )
```

Notes:

- The default filter only applies to read operations, so it will not affect the write-paths.
- When creating/updating an object with relations, this default queryset is ignored in the response, so related objects that don't match the filter may be returned.

1.3.12 Fun Times with Default Querysets

Default QuerySets on DynamicRelationField can also be used to do almost* anything QuerySets can do. The following examples are also valid:

```
class BlogPostSerializer(DynamicModelSerializer):

    # default sort applied
    comments = DynamicRelationField(
        CommentSerializer,
        many=True,
        queryset=Comment.objects.order_by('posted_at')
    )

    # most recent comment
    recent_comment = DynamicRelationField(
        CommentSerializer,
        source='comments',
        queryset=Comment.objects.order_by('posted_at').first()
    )
```

Almost anything? Yes, some things you shouldn't do:

- Anything that would cause the queryset to be evaluated. For instance, this will actually run the queryset when the class is loaded, which is NOT what you want:

```
    queryset=Foo.objects.filter(foo='bar')[:10]
```

- **Some queryset operations will conflict with DREST's internal query optimization. These include (but may not be limited t**
  - only() - DREST also uses only()

  - values_list() - Will probably confuse DREST because the data returned won't match what it's expecting.

1.3.13 Dynamic Default Querysets

In the previous examples above, the default querysets are constructed when the module loads. For more dynamic filters that can be constructed at run-time, the queryset attribute can be set to a function (or a lambda):

```
class BlogPostSerializer(DynamicModelSerializer):

    def get_recent_comment_queryset(field, *args, **kwargs):
        # Return queryset to filter comments made in last 3 hours
        recent = datetime.now() - timedelta(hours=3)
        return Comment.objects.filter(
            posted_at__gte=recent).order_by('posted_at')

    # default sort applied
    comments = DynamicRelationField(
        CommentSerializer,
        many=True,
        queryset=get_recent_comment_queryset
    )
```

Notes: The function mapped to a queryset should accept one parameter, which is the field (i.e. a DynamicRelationField instance) and return a QuerySet instance. It is also possible to access the parent serializer as field.parent and the child serializer as field.serializer (e.g. in the example above, field.parent refers to a BlogPostSerializer instance, while field.serializer would be a CommentSerializer instance).

1.3.14 DynamicMethodField and Prefetch Hinting

DREST will try pretty hard to optimize queries, specifically by only fetching fields that are required, and by using Django's prefetch features. In most cases, DREST will automatically do the right thing, but sometimes it doesn't have all the information it needs to pull the right data. Specific examples include:

- **Serializer method fields** - DREST doesn't know what kind of shenanigans you're up to in that serializer method field, and so it won't be able to infer what data you need.
- **Computed properties in models** - Basically the same problem as serializer method fields.

To address this issue, DREST fields like DynamicField and DynamicMethodField support a `requires` attribute that allows you to specify model fields that are required. DREST will then incorporate that information in its optimization strategy:

```
class UserSerializer(DynamicModelSerializer):
    preferred_full_name = DynamicMethodField(
        requires=[
            'profile.preferred_first_name',
            'profile.preferred_last_name'
        ]
    )

    def get_preferred_full_name(self, user):
        return '%s %s' % (
            user.profile.preferred_first_name,
            user.profile.preferred_last_name
        )
```

CHAPTER 2

dynamic_rest package

2.1 Submodules

2.2 dynamic_rest.bases module

This module contains base classes for DREST.

class dynamic_rest.bases.**DynamicSerializerBase**
Bases: object

Base class for all DREST serializers.

2.3 dynamic_rest.conf module

2.4 dynamic_rest.constants module

2.5 dynamic_rest.datastructures module

This module contains custom data-structures.

class dynamic_rest.datastructures.**TreeMap**
Bases: dict

Tree structure implemented with nested dictionaries.

get_paths()
Get all paths from the root to the leaves.

For example, given a chain like *{'a':{'b':{'c':None}}}*, this method would return *[['a', 'b', 'c']]*.

Returns A list of lists of paths.

insert(*parts*, *leaf_value*, *update=False*)
Add a list of nodes into the tree.

The list will be converted into a TreeMap (chain) and then merged with the current TreeMap.

For example, this method would insert *['a','b','c']* as *{'a':{'b':{'c':{}}}}*.

Parameters

- **parts** – List of nodes representing a chain.
- **leaf_value** – Value to insert into the leaf of the chain.
- **update** – Whether or not to update the leaf with the given value or to replace the value.

Returns self

2.6 dynamic_rest.fields module

2.7 dynamic_rest.filters module

2.8 dynamic_rest.links module

This module contains utilities to support API links.

dynamic_rest.links.**merge_link_object**(*serializer*, *data*, *instance*)
Add a 'links' attribute to the data that maps field names to URLs.

NOTE: This is the format that Ember Data supports, but alternative implementations are possible to support other formats.

2.9 dynamic_rest.meta module

Module containing Django meta helpers.

dynamic_rest.meta.**get_model_field**(*model*, *field_name*)
Return a field given a model and field name.

Parameters

- **model** – a Django model
- **field_name** – the name of a field

Returns A Django field if *field_name* is a valid field for *model*, None otherwise.

dynamic_rest.meta.**is_field_remote**(*model*, *field_name*)
Check whether a given model field is a remote field.

A remote field is the inverse of a one-to-many or a many-to-many relationship.

Parameters

- **model** – a Django model
- **field_name** – the name of a field

Returns True if *field_name* is a remote field, False otherwise.

dynamic_rest.meta.**is_model_field**(*model*, *field_name*)
Check whether a given field exists on a model.

Parameters

- **model** – a Django model
- **field_name** – the name of a field

Returns True if *field_name* exists on *model*, False otherwise.

2.10 dynamic_rest.metadata module

2.11 dynamic_rest.pagination module

2.12 dynamic_rest.patches module

This module contains patches for Django issues.

These patches are meant to be short-lived and are extracted from Django code changes.

`dynamic_rest.patches.`**`patch_prefetch_one_level`**`()`

This patch address Django bug https://code.djangoproject.com/ticket/24873, which was merged into Django master in commit 025c6553771a09b80563baedb5b8300a8b01312f into django.db.models.query.

The code that follows is identical to the code in the above commit, with all comments stripped out.

2.13 dynamic_rest.processors module

2.14 dynamic_rest.related module

This module provides backwards compatibility for RelatedObject.

2.15 dynamic_rest.renderers module

2.16 dynamic_rest.routers module

2.17 dynamic_rest.serializers module

2.18 dynamic_rest.tagged module

2.19 dynamic_rest.viewsets module

2.20 Module contents

Dynamic REST (or DREST) is an extension of Django REST Framework.

DREST offers the following features on top of the standard DRF kit:

- Linked/embedded/sideloaded relationships
- Field inclusions/exlusions
- Field-based filtering/sorting

- Directory panel for the browsable API
- Optimizations

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

d

D

G

I

M

P

T