Django REST Framework JSON API Documentation

Release 2.0.0-alpha.1

Jerel Unruh

Jan 25, 2018

Contents

Contents:

CHAPTER 1

Getting Started

Note: this package is named Django REST Framework JSON API to follow the naming convention of other Django REST Framework packages. Since that's quite a bit to say or type this package will be referred to as DJA elsewhere in these docs.

By default, Django REST Framework produces a response like:

```
{
    "count": 20,
    "next": "http://example.com/api/1.0/identities/?page=3",
    "previous": "http://example.com/api/1.0/identities/?page=1",
    "results": [{
        "id": 3,
        "username": "john",
        "full_name": "John Coltrane"
    }]
}
```

However, for the same `identity` model in JSON API format the response should look like the following:

```
{
    "links": {
        "first": "http://example.com/api/1.0/identities",
        "last": "http://example.com/api/1.0/identities?page=5",
        "next": "http://example.com/api/1.0/identities?page=3",
        "prev": "http://example.com/api/1.0/identities",
    },
    "data": [{
        "type": "identities",
        "id": 3,
        "attributes": {
            "username": "john",
            "full-name": "John Coltrane"
        }
    }],
    "meta": {
```

```
        "pagination": {
          "page": "2",
          "pages": "5",
          "count": "20"
        }
    }
}
```

1.1 Requirements

1. Python (2.7, 3.4, 3.5, 3.6)
2. Django (1.11, 2.0)
3. Django REST Framework (3.6, 3.7)

1.2 Installation

From PyPI

```
pip install djangorestframework-jsonapi
```

From Source

```
git clone https://github.com/django-json-api/django-rest-framework-json-api.git
cd django-rest-framework-json-api && pip install -e .
```

1.3 Running the example app

```
git clone https://github.com/django-json-api/django-rest-framework-json-api.git
cd django-rest-framework-json-api
pip install -e .
pip install -r example/requirements.txt
django-admin.py runserver
```

Browse to http://localhost:8000

1.4 Running Tests

```
python runtests.py
```

CHAPTER 2

Usage

The DJA package implements a custom renderer, parser, exception handler, and pagination. To get started enable the pieces in `settings.py` that you want to use.

Many features of the JSON:API format standard have been implemented using Mixin classes in `serializers.py`. The easiest way to make use of those features is to import ModelSerializer variants from `rest_framework_json_api` instead of the usual `rest_framework`

2.1 Configuration

We suggest that you copy the settings block below and modify it if necessary.

```
REST_FRAMEWORK = {
    'PAGE_SIZE': 10,
    'EXCEPTION_HANDLER': 'rest_framework_json_api.exceptions.exception_handler',
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework_json_api.pagination.PageNumberPagination',
    'DEFAULT_PARSER_CLASSES': (
        'rest_framework_json_api.parsers.JSONParser',
        'rest_framework.parsers.FormParser',
        'rest_framework.parsers.MultiPartParser'
    ),
    'DEFAULT_RENDERER_CLASSES': (
        'rest_framework_json_api.renderers.JSONRenderer',
        # If you're performance testing, you will want to use the browseable API
        # without forms, as the forms can generate their own queries.
        # If performance testing, enable:
        # 'example.utils.BrowsableAPIRendererWithoutForms',
        # Otherwise, to play around with the browseable API, enable:
        'rest_framework.renderers.BrowsableAPIRenderer'
    ),
    'DEFAULT_METADATA_CLASS': 'rest_framework_json_api.metadata.JSONAPIMetadata',
}
```

If `PAGE_SIZE` is set the renderer will return a `meta` object with record count and a `links` object with the next, previous, first, and last links. Pages can be selected with the `page` GET parameter. The query parameter used to retrieve the page can be customized by subclassing `PageNumberPagination` and overriding the `page_query_param`. Page size can be controlled per request via the `PAGINATE_BY_PARAM` query parameter (`page_size` by default).

2.1.1 Performance Testing

If you are trying to see if your viewsets are configured properly to optimize performance, it is preferable to use `example.utils.BrowsableAPIRendererWithoutForms` instead of the default `BrowsableAPIRenderer` to remove queries introduced by the forms themselves.

2.2 Serializers

It is recommended to import the base serializer classes from this package rather than from vanilla DRF. For example,

```
from rest_framework_json_api import serializers

class MyModelSerializer(serializers.ModelSerializer):
    # ...
```

2.3 Setting the resource_name

You may manually set the `resource_name` property on views, serializers, or models to specify the `type` key in the json output. In the case of setting the `resource_name` property for models you must include the property inside a `JSONAPIMeta` class on the model. It is automatically set for you as the plural of the view or model name except on resources that do not subclass `rest_framework.viewsets.ModelViewSet`:

Example - `resource_name` on View:

```
class Me(generics.GenericAPIView):
    """
    Current user's identity endpoint.

    GET /me
    """
    resource_name = 'users'
    serializer_class = identity_serializers.IdentitySerializer
    allowed_methods = ['GET']
    permission_classes = (permissions.IsAuthenticated, )
```

If you set the `resource_name` property on the object to `False` the data will be returned without modification.

Example - `resource_name` on Model:

```
class Me(models.Model):
    """
    A simple model
    """
    name = models.CharField(max_length=100)

    class JSONAPIMeta:
        resource_name = "users"
```

If you set the `resource_name` on a combination of model, serializer, or view in the same hierarchy, the name will be resolved as following: view > serializer > model. (Ex: A view `resource_name` will always override a `resource_name` specified on a serializer or model). Setting the `resource_name` on the view should be used sparingly as serializers and models are shared between multiple endpoints. Setting the `resource_name` on views may result in a different `type` being set depending on which endpoint the resource is fetched from.

2.4 Inflecting object and relation keys

This package includes the ability (off by default) to automatically convert json requests and responses from the python/rest_framework's preferred underscore to a format of your choice. To hook this up include the following setting in your project settings:

```
JSON_API_FORMAT_KEYS = 'dasherize'
```

Possible values:

- dasherize
- camelize (first letter is lowercase)
- capitalize (camelize but with first letter uppercase)
- underscore

Note: due to the way the inflector works `address_1` can camelize to `address1` on output but it cannot convert `address1` back to `address_1` on POST or PUT. Keep this in mind when naming fields with numbers in them.

Example - Without format conversion:

```
{
    "data": [{
        "type": "identities",
        "id": 3,
        "attributes": {
            "username": "john",
            "first_name": "John",
            "last_name": "Coltrane",
            "full_name": "John Coltrane"
        },
    }],
    "meta": {
        "pagination": {
          "count": 20
        }
    }
}
```

Example - With format conversion set to `dasherize`:

```
{
    "data": [{
        "type": "identities",
        "id": 3,
        "attributes": {
            "username": "john",
            "first-name": "John",
            "last-name": "Coltrane",
            "full-name": "John Coltrane"
```

```
        },
    }],
    "meta": {
        "pagination": {
          "count": 20
        }
    }
}
```

2.4.1 Types

A similar option to JSON_API_FORMAT_KEYS can be set for the types:

```
JSON_API_FORMAT_TYPES = 'dasherize'
```

Example without format conversion:

```
{
    "data": [{
        "type": "blog_identity",
        "id": 3,
        "attributes": {
                ...
        },
        "relationships": {
            "home_town": {
                "data": [{
                    "type": "home_town",
                    "id": 3
                }]
            }
        }
    }]
}
```

When set to dasherize:

```
{
    "data": [{
        "type": "blog-identity",
        "id": 3,
        "attributes": {
                ...
        },
        "relationships": {
            "home_town": {
                "data": [{
                    "type": "home-town",
                    "id": 3
                }]
            }
        }
    }]
}
```

It is also possible to pluralize the types like so:

```
JSON_API_PLURALIZE_TYPES = True
```

Example without pluralization:

```
{
    "data": [{
        "type": "identity",
        "id": 3,
        "attributes": {
                ...
        },
        "relationships": {
            "home_towns": {
                "data": [{
                    "type": "home_town",
                    "id": 3
                }]
            }
        }
    }]
}
```

When set to pluralize:

```
{
    "data": [{
        "type": "identities",
        "id": 3,
        "attributes": {
                ...
        },
        "relationships": {
            "home_towns": {
                "data": [{
                    "type": "home_towns",
                    "id": 3
                }]
            }
        }
    }]
}
```

2.5 Related fields

Because of the additional structure needed to represent relationships in JSON API, this package provides the `ResourceRelatedField` for serializers, which works similarly to `PrimaryKeyRelatedField`. By default, `rest_framework_json_api.serializers.ModelSerializer` will use this for related fields automatically. It can be instantiated explicitly as in the following example:

```
from rest_framework_json_api import serializers
from rest_framework_json_api.relations import ResourceRelatedField

from myapp.models import Order, LineItem, Customer
```

```
class OrderSerializer(serializers.ModelSerializer):
    class Meta:
        model = Order

    line_items = ResourceRelatedField(
        queryset=LineItem.objects,
        many=True  # necessary for M2M fields & reverse FK fields
    )

    customer = ResourceRelatedField(
        queryset=Customer.objects  # queryset argument is required
    )                              # except when read_only=True
```

In the JSON API spec, relationship objects contain links to related objects. To make this work on a serializer we need to tell the `ResourceRelatedField` about the corresponding view. Use the `HyperlinkedModelSerializer` and instantiate the `ResourceRelatedField` with the relevant keyword arguments:

```
from rest_framework_json_api import serializers
from rest_framework_json_api.relations import ResourceRelatedField

from myapp.models import Order, LineItem, Customer


class OrderSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Order

    line_items = ResourceRelatedField(
        queryset=LineItem.objects,
        many=True,
        related_link_view_name='order-lineitems-list',
        related_link_url_kwarg='order_pk',
        self_link_view_name='order_relationships'
    )

    customer = ResourceRelatedField(
        queryset=Customer.objects,
        related_link_view_name='order-customer-detail',
        related_link_url_kwarg='order_pk',
        self_link_view_name='order-relationships'
    )
```

- `related_link_view_name` is the name of the route for the related view.
- `related_link_url_kwarg` is the keyword argument that will be passed to the view that identifies the 'parent' object, so that the results can be filtered to show only those objects related to the 'parent'.
- `self_link_view_name` is the name of the route for the `RelationshipView` (see below).

In this example, `reverse('order-lineitems-list', kwargs={'order_pk': 3}` should resolve to something like `/orders/3/lineitems`, and that route should instantiate a view or viewset for `LineItem` objects that accepts a keyword argument `order_pk`. The drf-nested-routers package is useful for defining such nested routes in your urlconf.

The corresponding viewset for the `line-items-list` route in the above example might look like the following. Note that in the typical use case this would be the same viewset used for the `/lineitems` endpoints; when accessed through the nested route `/orders/<order_pk>/lineitems` the queryset is filtered using the `order_pk` keyword argument to include only the lineitems related to the specified order.

```
from rest_framework import viewsets

from myapp.models import LineItem
from myapp.serializers import LineItemSerializer


class LineItemViewSet(viewsets.ModelViewSet):
    queryset = LineItem.objects
    serializer_class = LineItemSerializer

    def get_queryset(self):
        queryset = self.queryset

        # if this viewset is accessed via the 'order-lineitems-list' route,
        # it wll have been passed the `order_pk` kwarg and the queryset
        # needs to be filtered accordingly; if it was accessed via the
        # unnested '/lineitems' route, the queryset should include all LineItems
        if 'order_pk' in self.kwargs:
            order_pk = self.kwargs['order_pk']
            queryset = queryset.filter(order__pk=order_pk)

        return queryset
```

2.6 RelationshipView

`rest_framework_json_api.views.RelationshipView` is used to build relationship views (see the JSON API spec). The `self` link on a relationship object should point to the corresponding relationship view.

The relationship view is fairly simple because it only serializes Resource Identifier Objects rather than full resource objects. In most cases the following is sufficient:

```
from rest_framework_json_api.views import RelationshipView

from myapp.models import Order


class OrderRelationshipView(RelationshipView):
    queryset = Order.objects
```

The urlconf would need to contain a route like the following:

```
url(
    regex=r'^orders/(?P<pk>[^/.]+)/relationships/(?P<related_field>[^/.]+)$',
    view=OrderRelationshipView.as_view(),
    name='order-relationships'
)
```

The `related_field` kwarg specifies which relationship to use, so if we are interested in the relationship represented by the related model field `Order.line_items` on the Order with pk 3, the url would be `/order/3/relationships/line_items`. On `HyperlinkedModelSerializer`, the `ResourceRelatedField` will construct the url based on the provided `self_link_view_name` keyword argument, which should match the `name=` provided in the urlconf, and will use the name of the field for the `related_field` kwarg. Also we can override `related_field` in the url. Let's say we want the url to be: `/order/3/relationships/order_items` - all we need to do is just add `field_name_mapping` dict to the class:

```
field_name_mapping = {
        'line_items': 'order_items'
    }
```

2.7 Working with polymorphic resources

Polymorphic resources allow you to use specialized subclasses without requiring special endpoints to expose the specialized versions. For example, if you had a `Project` that could be either an `ArtProject` or a `ResearchProject`, you can have both kinds at the same URL.

DJA tests its polymorphic support against django-polymorphic. The polymorphic feature should also work with other popular libraries like django-polymodels or django-typed-models.

2.7.1 Writing polymorphic resources

A polymorphic endpoint can be set up if associated with a polymorphic serializer. A polymorphic serializer takes care of (de)serializing the correct instances types and can be defined like this:

```
class ProjectSerializer(serializers.PolymorphicModelSerializer):
    polymorphic_serializers = [ArtProjectSerializer, ResearchProjectSerializer]

    class Meta:
        model = models.Project
```

It must inherit from `serializers.PolymorphicModelSerializer` and define the `polymorphic_serializers` list. This attribute defines the accepted resource types.

Polymorphic relations can also be handled with `relations.PolymorphicResourceRelatedField` like this:

```
class CompanySerializer(serializers.ModelSerializer):
    current_project = relations.PolymorphicResourceRelatedField(
        ProjectSerializer, queryset=models.Project.objects.all())
    future_projects = relations.PolymorphicResourceRelatedField(
        ProjectSerializer, queryset=models.Project.objects.all(), many=True)

    class Meta:
        model = models.Company
```

They must be explicitly declared with the `polymorphic_serializer` (first positional argument) correctly defined. It must be a subclass of `serializers.PolymorphicModelSerializer`.

2.8 Meta

You may add metadata to the rendered json in two different ways: `meta_fields` and `get_root_meta`.

On any `rest_framework_json_api.serializers.ModelSerializer` you may add a `meta_fields` property to the `Meta` class. This behaves in the same manner as the default `fields` property and will cause `SerializerMethodFields` or model values to be added to the `meta` object within the same `data` as the serializer.

To add metadata to the top level `meta` object add:

```
def get_root_meta(self, resource, many):
    if many:
      # Dealing with a list request
      return {
          'size': len(resource)
      }
    else:
      # Dealing with a detail request
      return {
        'foo': 'bar'
      }
```

to the serializer. It must return a dict and will be merged with the existing top level `meta`.

To access metadata in incoming requests, the `JSONParser` will add the metadata under a top level `_meta` key in the parsed data dictionary. For instance, to access meta data from a `serializer` object, you may use `serializer.initial_data.get("_meta")`. To customize the `_meta` key, see here.

2.9 Links

Adding `url` to `fields` on a serializer will add a `self` link to the `links` key.

Related links will be created automatically when using the Relationship View.

2.10 Included

JSON API can include additional resources in a single network request. The specification refers to this feature as Compound Documents. Compound Documents can reduce the number of network requests which can lead to a better performing web application. To accomplish this, the specification permits a top level `included` key. The list of content within this key are the extra resources that are related to the primary resource.

To make a Compound Document, you need to modify your `ModelSerializer`. The two required additions are `included_resources` and `included_serializers`.

For example, suppose you are making an app to go on quests, and you would like to fetch your chosen knight along with the quest. You could accomplish that with:

```
class KnightSerializer(serializers.ModelSerializer):
    class Meta:
        model = Knight
        fields = ('id', 'name', 'strength', 'dexterity', 'charisma')


class QuestSerializer(serializers.ModelSerializer):
    included_serializers = {
        'knight': KnightSerializer,
    }

    class Meta:
        model = Quest
        fields = ('id', 'title', 'reward', 'knight')

    class JSONAPIMeta:
        included_resources = ['knight']
```

`included_resources` informs DJA of **what** you would like to include. `included_serializers` tells DJA **how** you want to include it.

2.10.1 Performance improvements

Be aware that using included resources without any form of prefetching **WILL HURT PERFORMANCE** as it will introduce m*(n+1) queries.

A viewset helper was designed to allow for greater flexibility and it is automatically available when subclassing `views.ModelViewSet`

```
 # When MyViewSet is called with ?include=author it will dynamically prefetch author␣
↪and author.bio
 class MyViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    prefetch_for_includes = {
    '__all__': [],
    'author': ['author', 'author__bio']
    'category.section': ['category']
}
```

The special keyword `__all__` can be used to specify a prefetch which should be done regardless of the include, similar to making the prefetch yourself on the QuerySet.

Using the helper to prefetch, rather than attempting to minimise queries via select_related might give you better performance depending on the characteristics of your data and database.

For example:

If you have a single model, e.g. Book, which has four relations e.g. Author, Publisher, CopyrightHolder, Category.

To display 25 books and related models, you would need to either do:

a) 1 query via selected_related, e.g. SELECT * FROM books LEFT JOIN author LEFT JOIN publisher LEFT JOIN CopyrightHolder LEFT JOIN Category

b) 4 small queries via prefetch_related.

If you have 1M books, 50k authors, 10k categories, 10k copyrightholders in the select_related scenario, you've just created a in-memory table with 1e18 rows which will likely exhaust any available memory and slow your database to crawl.

The prefetch_related case will issue 4 queries, but they will be small and fast queries.

CHAPTER 3

API

3.1 mixins

3.1.1 MultipleIDMixin

Add this mixin to a view to override `get_queryset` to automatically filter records by `ids[]=1&ids[]=2` in URL query params.

3.2 rest_framework_json_api.renderers.JSONRenderer

The `JSONRenderer` exposes a number of methods that you may override if you need highly custom rendering control.

3.2.1 extract_attributes

```
extract_attributes(fields, resource)
```

Builds the `attributes` object of the JSON API resource object.

3.2.2 extract_relationships

```
extract_relationships(fields, resource, resource_instance)
```

Builds the `relationships` top level object based on related serializers.

3.2.3 extract_included

```
extract_included(fields, resource, resource_instance, included_resources)
```

Adds related data to the top level `included` key when the request includes `?include=example,example_field2`

3.2.4 extract_meta

`extract_meta(serializer, resource)`

Gathers the data from serializer fields specified in `meta_fields` and adds it to the `meta` object.

3.2.5 extract_root_meta

`extract_root_meta(serializer, resource)`

Calls a `get_root_meta` function on a serializer, if it exists.

3.2.6 build_json_resource_obj

`build_json_resource_obj(fields, resource, resource_instance, resource_name)`

Builds the resource object (type, id, attributes) and extracts relationships.

3.3 rest_framework_json_api.parsers.JSONParser

Similar to `JSONRenderer`, the `JSONParser` you may override the following methods if you need highly custom parsing control.

3.3.1 parse_metadata

`parse_metadata(result)`

Returns a dictionary which will be merged into parsed data of the request. By default, it reads the `meta` content in the request body and returns it in a dictionary with a `_meta` top level key.

CHAPTER 4

Indices and tables

- genindex
- search