# CONTENTS

1 Getting Started ........................................... 3
   1.1 Requirements .................................................. 4
   1.2 Installation ....................................................... 4
   1.3 Running the example app ........................................ 5
   1.4 Running Tests .................................................... 5

2 Usage .............................................................. 7
   2.1 Configuration ..................................................... 7
   2.2 Pagination ......................................................... 8
   2.3 Filter Backends .................................................. 9
   2.4 Exception handling ............................................. 11
   2.5 Performance Testing .......................................... 12
   2.6 Serializers ....................................................... 12
   2.7 Setting the resource_name ..................................... 12
   2.8 Inflecting object and relation keys .......................... 13
   2.9 Related fields ................................................... 16
   2.10 RelationshipView ............................................... 20
   2.11 Working with polymorphic resources ........................ 21
   2.12 Meta ............................................................ 21
   2.13 Links ............................................................ 22
   2.14 Included ......................................................... 22
   2.15 Generating an OpenAPI Specification (OAS) 3.0 schema document ................................................. 24

3 API Reference ................................................... 27
   3.1 rest_framework_json_api package .............................. 27

4 Contributing ..................................................... 45
   4.1 Setup development environment ................................ 45
   4.2 For maintainers .................................................. 46

5 Indices and tables .............................................. 47

Python Module Index ........................................... 49

Index ............................................................... 51
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:

```json
{
   "count": 20,
   "next": "https://example.com/api/1.0/identities/?page=3",
   "previous": "https://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:

```json
{
   "links": {
      "first": "https://example.com/api/1.0/identities",
      "last": "https://example.com/api/1.0/identities?page=5",
      "next": "https://example.com/api/1.0/identities?page=3",
      "prev": "https://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"
      }
   }
}
```

(continues on next page)
1.1 Requirements

1. Python (3.6, 3.7, 3.8, 3.9, 3.10)
2. Django (2.2, 3.0, 3.1, 3.2, 4.0)
3. Django REST framework (3.12)

We highly recommend and only officially support the latest patch release of each Python, Django and REST framework series.

Generally Python and Django series are supported till the official end of life. For Django REST framework the last two series are supported.

1.2 Installation

Install using pip...

```bash
pip install djangorestframework-jsonapi
# for optional package integrations
pip install djangorestframework-jsonapi[\'django-filter\']
pip install djangorestframework-jsonapi[\'django-polymorphic\']
pip install djangorestframework-jsonapi[\'openapi\']
```

or from source...

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

and add rest_framework_json_api to your INSTALLED_APPS setting below rest_framework.

```python
INSTALLED_APPS = [
    ...
    'rest_framework',
    'rest_framework_json_api',
    ...
]
```
1.3 Running the example app

```bash
git clone https://github.com/django-json-api/django-rest-framework-json-api.git
cd django-rest-framework-json-api
python3 -m venv env
source env/bin/activate
pip install -Ur requirements.txt
django-admin migrate --settings=example.settings
django-admin loaddata drf_example --settings=example.settings
django-admin runserver --settings=example.settings
```

Browse to

- `http://localhost:8000` for the list of available collections (in a non-JSON:API format!),
- `http://localhost:8000/swagger-ui/` for a Swagger user interface to the dynamic schema view, or

1.4 Running Tests

```bash
pip install tox
tox
```
CHAPTER TWO

USAGE

The DJA package implements a custom renderer, parser, exception handler, query filter backends, 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.JsonApiPageNumberPagination',
    '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_json_api.renderers.BrowsableAPIRenderer'
    ),
    'DEFAULT_METADATA_CLASS': 'rest_framework_json_api.metadata.JSONAPIMetadata',
    'DEFAULT_SCHEMA_CLASS': 'rest_framework_json_api.schemas.openapi.AutoSchema',
    'DEFAULT_FILTER_BACKENDS': (
        'rest_framework_json_api.filters.QueryParameterValidationFilter',
        'rest_framework_json_api.filters.OrderingFilter',
        'rest_framework_json_api.django_filters.DjangoFilterBackend',
        'rest_framework.filters.SearchFilter',
    ),
    'SEARCH_PARAM': 'filter[search]',
}(continues on next page)
2.2 Pagination

DJA pagination is based on DRF pagination. When pagination is enabled, the renderer will return a meta object with record count and a links object with the next, previous, first, and last links.

Optional query parameters can also be provided to customize the page size or offset limit.

2.2.1 Configuring the Pagination Style

Pagination style can be set on a particular viewset with the pagination_class attribute or by default for all viewsets by setting REST_FRAMEWORK['DEFAULT_PAGINATION_CLASS'] and by setting REST_FRAMEWORK['PAGE_SIZE'].

You can configure fixed values for the page size or limit – or allow the client to choose the size or limit via query parameters.

Two pagination classes are available:

- **JsonApiPageNumberPagination** breaks a response up into pages that start at a given page number with a given size (number of items per page). It can be configured with the following attributes:
  - page_query_param (default page[number])
  - page_size_query_param (default page[size]) Set this to None if you don’t want to allow the client to specify the size.
  - page_size (default REST_FRAMEWORK['PAGE_SIZE']) default number of items per page unless overridden by page_size_query_param.
  - max_page_size (default 100) enforces an upper bound on the page_size_query_param. Set it to None if you don’t want to enforce an upper bound.

- **JsonApiLimitOffsetPagination** breaks a response up into pages that start from an item’s offset in the viewset for a given number of items (the limit). It can be configured with the following attributes:
  - offset_query_param (default page[offset]).
  - limit_query_param (default page[limit]).
  - default_limit (default REST_FRAMEWORK['PAGE_SIZE']) is the default number of items per page unless overridden by limit_query_param.
  - max_limit (default 100) enforces an upper bound on the limit. Set it to None if you don’t want to enforce an upper bound.
Examples

These examples show how to configure the parameters to use non-standard names and different limits:

```python
from rest_framework_json_api.pagination import JsonApiPageNumberPagination, JsonApiLimitOffsetPagination

class MyPagePagination(JsonApiPageNumberPagination):
    page_query_param = 'page_number'
    page_size_query_param = 'page_length'
    page_size = 3
    max_page_size = 1000

class MyLimitPagination(JsonApiLimitOffsetPagination):
    offset_query_param = 'offset'
    limit_query_param = 'limit'
    default_limit = 3
    max_limit = None
```

2.3 Filter Backends

Following are descriptions of JSON:API-specific filter backends and documentation on suggested usage for a standard DRF keyword-search filter backend that makes it consistent with JSON:API.

2.3.1 QueryParameterValidationFilter

QueryParameterValidationFilter validates query parameters to be one of the defined JSON:API query parameters (sort, include, filter, fields, page) and returns a 400 Bad Request if a non-matching query parameter is used. This can help the client identify misspelled query parameters, for example.

If you want to change the list of valid query parameters, override the .query_regex attribute:

```python
import re
from rest_framework_json_api.filters import QueryParameterValidationFilter

class MyQPValidator(QueryParameterValidationFilter):
    query_regex = re.compile(r'^\^(sort|include|page|page_size)$|^\^(filter|fields|page)(\[[\w.\-]+\])?$')
```

For example:

```python
import re
from rest_framework_json_api.filters import QueryParameterValidationFilter

class MyQPValidator(QueryParameterValidationFilter):
    query_regex = re.compile(r'^\^(sort|include|page|page_size)$|^\^(filter|fields|page)(\[[\w.\-]+\])?$')
```

If you don’t care if non-JSON:API query parameters are allowed (and potentially silently ignored), simply don’t use this filter backend.
2.3.2 OrderingFilter

OrderingFilter implements the JSON:API sort and uses DRF’s ordering filter.

Per the JSON:API specification, “If the server does not support sorting as specified in the query parameter sort, it MUST return 400 Bad Request.” For example, for ?sort=abc,foo,def where foo is a valid field name and the other two are not valid:

```json
{
   "errors": [
      {
         "detail": "invalid sort parameters: abc,def",
         "source": {
            "pointer": "/data"
         },
         "status": "400"
      }
   ]
}
```

If you want to silently ignore bad sort fields, just use `rest_framework.filters.OrderingFilter` and set `ordering_param` to `sort`.

2.3.3 DjangoFilterBackend

DjangoFilterBackend implements a Django ORM-style JSON:API filter using the `django-filter` package.

This filter is not part of the JSON:API standard per-se, other than the requirement to use the `filter` keyword: It is an optional implementation of a style of filtering in which each filter is an ORM expression as implemented by `DjangoFilterBackend` and seems to be in alignment with an interpretation of the JSON:API recommendations, including relationship chaining.

Filters can be:

- A resource field equality test: ?filter[qty]=123
- Apply other field lookup operators: ?filter[name.icontains]=bar or ?filter[name.isnull]=true
- Membership in a list of values: ?filter[name.in]=abc,123,zzz (name in ['abc','123','zzz'])
- Filters can be combined for intersection (AND): ?filter[qty]=123&filter[name.in]=abc,123,zzz&filter[...] or ?filter[authors.id]=1&filter[authors.id]=2
- A related resource path can be used: ?filter[inventory.item.partNum]=123456 (where inventory.item is the relationship path)

The filter returns a 400 Bad Request error for invalid filter query parameters as in this example for GET http://127.0.0.1:8000/nopage-entries?filter[bad]=1:

```json
{
   "errors": [
      {
         "detail": "invalid filter[bad]",
         "source": {
            "pointer": "/data"
         },
         "status": "400"
      }
   ]
}
```

(continues on next page)
As this feature depends on django-filter you need to run

```
pip install djangorestframework-jsonapi['django-filter']
```

### 2.3.4 SearchFilter

To comply with JSON:API query parameter naming standards, DRF's SearchFilter should be configured to use a `filter[_something_]` query parameter. This can be done by default by adding the SearchFilter to `REST_FRAMEWORK['DEFAULT_FILTER_BACKENDS']` and setting `REST_FRAMEWORK['SEARCH_PARAM']` or adding the `.search_param` attribute to a custom class derived from SearchFilter. If you do this and also use `DjangoFilterBackend`, make sure you set the same values for both classes.

### 2.3.5 Configuring Filter Backends

You can configure the filter backends either by setting the `REST_FRAMEWORK['DEFAULT_FILTER_BACKENDS']` as shown in the example settings or individually add them as `.filter_backends` View attributes:

```python
from rest_framework_json_api import filters
from rest_framework_json_api import django_filters
from rest_framework import SearchFilter
from models import MyModel

class MyViewSet(ModelViewSet):
    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer
    filter_backends = (filters.QueryParameterValidationFilter, filters.OrderingFilter,
                       django_filters.DjangoFilterBackend, SearchFilter)

    filterset_fields = {
        'id': ('exact', 'lt', 'gt', 'gte', 'lte', 'in'),
        'description': ('icontains', 'iexact', 'contains'),
        'tagline': ('icontains', 'iexact', 'contains'),
    }
    search_fields = ('id', 'description', 'tagline',)
```

### 2.4 Exception handling

For the `exception_handler` class, if the optional `JSON_API_UNIFORM_EXCEPTIONS` is set to True, all exceptions will respond with the JSON:API error format.

When `JSON_API_UNIFORM_EXCEPTIONS` is False (the default), non-JSON:API views will respond with the normal DRF error format.

2.4. Exception handling
2.5 Performance Testing

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

2.6 Serializers

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

```python
from rest_framework_json_api import serializers

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

2.7 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:

```python
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:

```python
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.

## 2.8 Inflecting object and relation keys

This package includes the ability (off by default) to automatically convert JSON:API field names of 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_FIELD_NAMES = '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 PATCH. Keep this in mind when naming fields with numbers in them.

Example - Without format conversion:

```json
{
    "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`:

```json
{
    "data": [{
        "type": "identities",
        "id": "3",
        "attributes": {
            "username": "john",
            "first-name": "John",
            "last-name": "Coltrane",
            "full-name": "John Coltrane"
        }
    },
    "meta": {
        "pagination": {
            "count": 20
        }
    }
}
```
2.8.1 Types

A similar option to \texttt{JSON\_API\_FORMAT\_FIELD\_NAMES} can be set for the types:

\texttt{JSON\_API\_FORMAT\_TYPES = 'dasherize'}

Example without format conversion:

\begin{verbatim}
{
    "data": [{
        "type": "blog_identity",
        "id": "3",
        "attributes": {
            ...
        },
        "relationships": {
            "home_town": {
                "data": [{
                    "type": "home_town",
                    "id": 3
                }]
            }
        }
    }]
}
\end{verbatim}

When set to dasherize:

\begin{verbatim}
{
    "data": [{
        "type": "blog-identity",
        "id": "3",
        "attributes": {
            ...
        },
        "relationships": {
            "home_town": {
                "data": [{
                    "type": "home-town",
                    "id": 3
                }]
            }
        }
    }]
}
\end{verbatim}
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"
                        }
                    ]
                }
            }
        }
    ]
}
```
2.8.2 Related URL segments

Serializer properties in relationship and related resource URLs may be infected using the JSON_API_FORMAT_RELATED_LINKS setting.

```
JSON_API_FORMAT_RELATED_LINKS = 'dasherize'
```

For example, with a serializer property created_by and with 'dasherize' formatting:

```
{
   "data": {
      "type": "comments",
      "id": "1",
      "attributes": {
         "text": "Comments are fun!"
      },
      "links": {
         "self": "/comments/1"
      },
      "relationships": {
         "created_by": {
            "links": {
               "self": "/comments/1/relationships/created-by",
               "related": "/comments/1/created-by"
            }
         }
      }
   },
   "links": {
      "self": "/comments/1"
   }
}
```

The relationship name is formatted by the JSON_API_FORMAT_FIELD_NAMES setting, but the URL segments are formatted by the JSON_API_FORMAT_RELATED_LINKS setting.

2.9 Related fields

2.9.1 ResourceRelatedField

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):
```

(continues on next page)
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:

```python
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, # necessary for M2M fields & reverse FK fields
    )

    customer = ResourceRelatedField(
        queryset=Customer.objects, # queryset argument is required
        related_link_view_name='order-customer-detail', # except when read_only=True
        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.

```python
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 = super().get_queryset()

        # if this viewset is accessed via the 'order-lineitems-list' route,
        # it will 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
        order_pk = self.kwargs.get('order_pk')
        if order_pk is not None:
            queryset = queryset.filter(order__pk=order_pk)

        return queryset
```

### 2.9.2 HyperlinkedRelatedField

`relations.HyperlinkedRelatedField` has same functionality as `ResourceRelatedField` but does not render data. Use this in case you only need links of relationships and want to lower payload and increase performance.

### 2.9.3 SerializerMethodResourceRelatedField

`relations.SerializerMethodResourceRelatedField` combines behaviour of DRF `SerializerMethodField` and `ResourceRelatedField`, so it accepts `method_name` together with `model` and links-related arguments. Data is rendered in `ResourceRelatedField` manner.

```python
from rest_framework_json_api import serializers
from rest_framework_json_api.relations import SerializerMethodResourceRelatedField

from myapp.models import Order, LineItem

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

    line_items = SerializerMethodResourceRelatedField(
        model=LineItem,
        many=True,
        method_name='get_big_line_items'
    )
```

(continues on next page)
small_line_items = SerializerMethodResourceRelatedField(
    model=LineItem,
    many=True,
    # default to method_name='get_small_line_items'
)

def get_big_line_items(self, instance):
    return LineItem.objects.filter(order=instance).filter(amount__gt=1000)

def get_small_line_items(self, instance):
    return LineItem.objects.filter(order=instance).filter(amount__lte=1000)

or using related_link_* with HyperlinkedModelSerializer

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

        line_items = SerializerMethodResourceRelatedField(
            model=LineItem,
            many=True,
            method_name='get_big_line_items',
            related_link_view_name='order-lineitems-list',
            related_link_url_kwarg='order_pk',
        )

        def get_big_line_items(self, instance):
            return LineItem.objects.filter(order=instance).filter(amount__gt=1000)

2.9.4 Related urls

There is a nice way to handle “related” urls like /orders/3/lineitems/ or /orders/3/customer/. All you need is just add to urls.py:

```
url(r'^orders/(?P<pk>[/\w]+)/$',
    OrderViewSet.as_view({'get': 'retrieve'}),
    name='order-detail'),
url(r'^orders/(?P<pk>[/\w]+)/(?P<related_field>[-\w]+)/$',
    OrderViewSet.as_view({'get': 'retrieve_related'}),
    name='order-related'),
```

Make sure that RelatedField declaration has related_link_url_kwarg='pk' or simply skipped (will be set by default):

```
line_items = ResourceRelatedField(
    queryset=LineItem.objects,
    many=True,
    related_link_view_name='order-related',
    related_link_url_kwarg='pk',
    self_link_view_name='order-relationships'
)
```
customer = ResourceRelatedField(
    queryset=Customer.objects,
    related_link_view_name='order-related',
    self_link_view_name='order-relationships'
)

And, the most important part - declare serializer for each related entity:

class OrderSerializer(serializers.HyperlinkedModelSerializer):
    ...
    related_serializers = {
        'customer': 'example.serializers.CustomerSerializer',
        'line_items': 'example.serializers.LineItemSerializer'
    }

Or, if you already have included_serializers declared and your related_serializers look the same, just skip it:

class OrderSerializer(serializers.HyperlinkedModelSerializer):
    ...
    included_serializers = {
        'customer': 'example.serializers.CustomerSerializer',
        'line_items': 'example.serializers.LineItemSerializer'
    }

2.10 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:

```python
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:

```python
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 /orders/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:

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

## 2.11 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.

As this feature depends on django-polymorphic you need to run

```bash
pip install djangorestframework-jsonapi["django-polymorphic"]
```

### 2.11.1 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:

```python
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:

```python
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.
2.12 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:

```python
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.

2.13 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.

2.14 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. `included_serializers` is required to inform DJA of what and how you would like to include. `included_resources` tells DJA what you want to include by default.

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:

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

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

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

class JSONAPIMeta:
    included_resources = ['knight']

2.14.1 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 therefore designed to automatically preload data when possible. Such is automatically available when subclassing ModelViewSet or ReadOnlyModelViewSet.

It also allows to define custom select_related and prefetch_related for each requested include when needed in special cases:

```python
from rest_framework_json_api import views
# When MyViewSet is called with ?include=author it will dynamically prefetch author and author.bio
class MyViewSet(views.ModelViewSet):
    queryset = Book.objects.all()
    select_for_includes = {
        'author': ['author__bio'],
    }
    prefetch_for_includes = {
        '__all__': [],
        'all_authors': [Prefetch('all_authors', queryset=Author.objects.select_related('bio'))],
        'category.section': ['category']
    }
```

An additional convenience DJA class exists for read-only views, just as it does in DRF.

```python
from rest_framework_json_api import views

class MyReadOnlyViewSet(views.ReadOnlyModelViewSet):
    # ...
```

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.

## 2.15 Generating an OpenAPI Specification (OAS) 3.0 schema document

DRF has a OAS schema functionality to generate an OAS 3.0 schema as a YAML or JSON file.

DJA extends DRF’s schema support to generate an OAS schema in the JSON:API format.

### 2.15.1 AutoSchema Settings

In order to produce an OAS schema that properly represents the JSON:API structure you have to either add a schema attribute to each view class or set the REST_FRAMEWORK[‘DEFAULT_SCHEMA_CLASS’] to DJA’s version of AutoSchema.

**View-based**

```python
from rest_framework_json_api.schemas.openapi import AutoSchema
class MyViewSet(ModelViewSet):
    schema = AutoSchema
    ...
```

**Default schema class**

```python
REST_FRAMEWORK = {
    # ...
    'DEFAULT_SCHEMA_CLASS': 'rest_framework_json_api.schemas.openapi.AutoSchema',
}
```
2.15.2 Adding additional OAS schema content

You can extend the OAS schema document by subclassing `SchemaGenerator` and extending `get_schema`. Here’s an example that adds OAS `info` and `servers` objects.

```python
from rest_framework_json_api.schemas.openapi import SchemaGenerator as JSONAPISchemaGenerator

class MySchemaGenerator(JSONAPISchemaGenerator):
    """
    Describe my OAS schema info in detail (overriding what DRF put in) and list the
    ...servers where it can be found.
    """
    def get_schema(self, request, public):
        schema = super().get_schema(request, public)
        schema['info'] = {
            'version': '1.0',
            'title': 'my demo API',
            'description': 'A demonstration of [OAS 3.0](https://www.openapis.org)',
            'contact': {
                'name': 'my name'
            },
            'license': {
                'name': 'BSD 2 clause',
                'url': 'https://github.com/django-json-api/django-rest-framework-json-
                ...api/blob/master/LICENSE',
            }
        }
        schema['servers'] = [
            {'url': 'http://localhost/v1', 'description': 'local docker'},
            {'url': 'http://localhost:8000/v1', 'description': 'local dev'},
            {'url': 'https://api.example.com/v1', 'description': 'demo server'},
            {'url': '{serverURL}', 'description': 'provide your server URL',
                'variables': {'serverURL': {'default': 'http://localhost:8000/v1'}}
            ]
        return schema
```

2.15.3 Generate a Static Schema on Command Line

See DRF documentation for `generateschema` To generate an OAS schema document, use something like:

```
$ django-admin generateschema --settings=example.settings \
    --generator_class myapp.views.MySchemaGenerator >myschema.
    ...yaml
```

You can then use any number of OAS tools such as `swagger-ui-watcher` to render the schema:

```
$ swagger-ui-watcher myschema.yaml
```

Note: Swagger-ui-watcher will complain that “DELETE operations cannot have a requestBody” but it will still work. This error in the OAS specification will be fixed when OAS 3.1.0 is published.
(swagger-ui will work silently.)

### 2.15.4 Generate a Dynamic Schema in a View

See DRF documentation for a Dynamic Schema.

```python
from rest_framework.schemas import get_schema_view

urlpatterns = [
    ...
    path('openapi', get_schema_view(
        title="Example API",
        description="API for all things ...",
        version="1.0.0",
        generator_class=MySchemaGenerator,
    ), name='openapi-schema'),
    path('swagger-ui/', TemplateView.as_view(
        template_name='swagger-ui.html',
        extra_context={'schema_url': 'openapi-schema'}
    ), name='swagger-ui'),
    ...
]
```
This API reference is autogenerated from the Python docstrings – which need to be improved!

### 3.1 rest_framework_json_api package

#### 3.1.1 Subpackages

**rest_framework_json_api.django_filters package**

**Submodules**

**rest_framework_json_api.django_filters.backends module**

```python
class rest_framework_json_api.django_filters.backends.DjangoFilterBackend
    Bases: django_filters.rest_framework.backends.DjangoFilterBackend

    A Django-style ORM filter implementation, using `django-filter`.
    This is not part of the JSON:API standard per-se, other than the requirement to use the `filter` keyword: This is an optional implementation of style of filtering in which each filter is an ORM expression as implemented by DjangoFilterBackend and seems to be in alignment with an interpretation of https://jsonapi.org/recommendations/#filtering, including relationship chaining. It also returns a 400 error for invalid filters.

    Filters can be:
    - A resource field equality test:
      `?filter[qty]=123`
    - Apply other https://docs.djangoproject.com/en/stable/ref/models/querysets/#field-lookups operators:
      `?filter[name.icontains]=bar` or `?filter[name.isnull]=true...`
    - Membership in a list of values:
      `?filter[name.in]=abc,123,zzz` (name in ['abc','123','zzz'])
    - Filters can be combined for intersection (AND):
      `?filter[qty]=123&filter[name.in]=abc,123,zzz&filter[...]`
    - A related resource path can be used:
      `?filter[inventory.item.partNum]=123456` (where `inventory.item` is the relationship path)
```
If you are also using `rest_framework.filters.SearchFilter` you’ll want to customize the name of the query parameter for searching to make sure it doesn’t conflict with a field name defined in the filterset. The recommended value is: `search_param='filter[search]'` but just make sure it's `filter[something]` to comply with the JSON:API spec requirement to use the filter keyword. The default is “search” unless overridden but it’s used here just to make sure we don’t complain about it being an invalid filter.

```python
search_param = 'filter[search]'
```

```python
filter_regex = re.compile('^filter(?P<ldelim>[\[]?)(?P<assoc>\[\w\.\-\]*)(?P<rdelim>[\]?$)\$')
```

```python
get_filterset(request, queryset, view)
```

Sometimes there’s no `filterset_class` defined yet the client still requests a filter. Make sure they see an error too. This means we have to `get_filterset_kwargs()` even if there’s no `filterset_class`.

```python
def get_filterset_kwargs(request, queryset, view):
    # Turns filter[<field>]=<value> into <field>=<value> which is what DjangoFilterBackend expects
    Raises ValidationError – for bad filter syntax
```

```python
def get_schema_operation_parameters(view):
    # Convert backend filter name to JSON:API-style filter[name]. For filters that are relationship paths, rewrite ORM-style __ to our preferred .. For example: blog__name__contains becomes filter[blog.name.contains].
    This is basically the reverse of `get_filterset_kwargs` above.
```

### rest_framework_json_api.schemas package

**Submodules**

### rest_framework_json_api.schemas.openapi module

**class** `rest_framework_json_api.schemas.openapi.AutoSchema`

```python
Bases: rest_framework.schemas.openapi.AutoSchema
```

Extend DRF’s `openapi.AutoSchema` for JSON:API serialization.

```python
content_types = ['application/vnd.api+json']
```

```python
get_operation(path, method)
```

JSON:API adds some standard fields to the API response that are not in upstream DRF: - some that only apply to GET/HEAD methods. - collections - special handling for POST, PATCH, DELETE

```python
get_operation_id(path, method)
```

The upstream DRF version creates non-unique operationIDs, because the same view is used for the main path as well as such as related and relationships. This concatenates the (mapped) method name and path as the spec allows most any

```python
def get_request_body(path, method)
    A request body is required by JSON:API for POST, PATCH, and DELETE methods.
```

```python
map_serializer(serializer)
```

Custom map_serializer that serializes the schema using the JSON:API spec. Non-attributes like related and identity fields, are move to ‘relationships’ and ‘links’.
class rest_framework_json_api.schemas.openapi.SchemaGenerator(title=None, url=None, description=None, patterns=None, urlconf=None, version=None)

    Bases: rest_framework.schemas.openapi.SchemaGenerator

    Extend DRF’s SchemaGenerator to implement JSON:API flavored generateschema command.

    get_schema(request=None, public=False)
        Generate a JSON:API OpenAPI schema. Overrides upstream DRF’s get_schema.
Django REST framework JSON:API Documentation, Release 4.3.0

jsonapi_components = {'parameters': {'fields': {'description': '[sparse fieldsets](https://jsonapi.org/format/#fetching-sparse-fieldsets).\nUse fields\[\<typename\>\]=field1,field2,...,fieldN', 'explode': True, 'in': 'query', 'name': 'fields', 'required': False, 'schema': {'type': 'object', 'style': 'deepObject'}, 'include': {'description': '[list of included related resources](https://jsonapi.org/format/#fetching-includes)', 'in': 'query', 'name': 'include', 'required': False, 'schema': {'type': 'object', 'style': 'form'}, 'sort': {'description': '[list of fields to sort by](https://jsonapi.org/format/#fetching-sorting)', 'in': 'query', 'name': 'sort', 'required': False, 'schema': {'type': 'string', 'style': 'form'}, 'schemas': {'ResourceIdentifierObject': {'oneOf': [{'$ref': '#/components/schemas/relationshipToOne'}, {'$ref': '#/components/schemas/relationshipToMany'}]}}, 'id': {'description': 'A single, unique resource.', 'type': 'string'}, 'jsonapi': {'$ref': '#/components/schemas/jsonapi'}, '.links': {'$ref': '#/components/schemas/links'}, 'meta': {'$ref': '#/components/schemas/meta'}, 'relationships': {'required': ['*errors'], 'type': 'object'}, 'errors': {'items': {'$ref': '#/components/schemas/errors'}, 'type': 'array'}, 'uniqueItems': True, 'failure': {'properties': {'errors': {'$ref': '#/components/schemas/errors'}, 'jsonapi': {'$ref': '#/components/schemas/jsonapi'}, 'links': {'$ref': '#/components/schemas/links'}, 'meta': {'$ref': '#/components/schemas/meta'}, 'id': {'description': 'Each resource object’s type and id pair MUST [identify](https://jsonapi.org/format/#document-resource-object-identification) a single, unique resource.', 'type': 'string'}, 'jsonapi': {'$ref': '#/components/schemas/jsonapi'}, 'additionalProperties': False, 'description': 'The server’s implementation', 'properties': {'meta': {'$ref': '#/components/schemas/meta'}, 'version': {'type': 'string'}, 'type': 'object'}, 'link': {'oneOf': [{'description': 'a string containing the link’s URL', 'type': 'string', 'format': 'uri-reference'}, {'type': 'object', 'required': ['$href'], 'properties': {'$href': {'description': 'a string containing the link’s URL', 'type': 'string', 'format': 'uri-reference'}, 'meta': {'$ref': '#/components/schemas/meta'}}]}, 'linkage': {'description': 'the type and id', 'properties': {'id': {'$ref': '#/components/schemas/id'}, 'meta': {'$ref': '#/components/schemas/meta'}}, 'type': 'object'}, 'required': ['type', 'id'], 'additionalProperties': False, 'description': 'optional references to other resource objects', 'properties': {'related': {'$ref': '#/components/schemas/linkage'}, 'self': {'$ref': '#/components/schemas/linkage'}, 'type': 'object'}, 'relationshipToMany': {'description': 'An array of objects each containing the type and id for to-many relationships', 'items': {'$ref': '#/components/schemas/resource'}}}, 'pagination': {'properties': {'first': {'$ref': '#/components/schemas/pageref'}, 'last': {'$ref': '#/components/schemas/pageref'}, 'next': {'$ref': '#/components/schemas/pageref'}, 'prev': {'$ref': '#/components/schemas/pageref'}, 'type': 'object'}, 'relationshipLinks': {'additionalProperties': True, 'description': 'optional references to other resource objects', 'properties': {'related': {'$ref': '#/components/schemas/linkage'}, 'self': {'$ref': '#/components/schemas/linkage'}, 'type': 'object'}, 'relationshipToOne': {'anyOf': [{'$ref': '#/components/schemas/nulltype'}, {'$ref': '#/components/schemas/linkage'}], 'description': 'reference to other resource in a to-one relationship'}, 'reltomany': {'description': 'a multiple

30 Chapter 3 API Reference
3.1.2 Submodules

rest_framework_json_api.exceptions module

rest_framework_json_api.exceptions.rendered_with_json_api(view)
rest_framework_json_api.exceptions.exception_handler(exc, context)

exception rest_framework_json_api.exceptions.Conflict(detail=None, code=None)
   Bases: rest_framework.exceptions.APIException

   status_code = 409
   default_detail = 'Conflict.'

rest_framework_json_api.filters module

class rest_framework_json_api.filters.OrderingFilter
   Bases: rest_framework.filters.OrderingFilter

   A backend filter that implements https://jsonapi.org/format/#fetching-sorting and raises a 400 error if any sort field is invalid.

   If you prefer not to report 400 errors for invalid sort fields, just use rest_framework.filters.OrderingFilter with ordering_param = “sort”

   Also supports undo of field name formatting (See JSON:API-compliant query parameter name).

   ordering_param = 'sort'

   remove_invalid_fields(queryset, fields, view, request)
       Extend rest_framework.filters.OrderingFilter.remove_invalid_fields() to validate that all provided sort fields exist (as contrasted with the super’s behavior which is to silently remove invalid fields).

       Raises ValidationError – if a sort field is invalid.

class rest_framework_json_api.filters.QueryParameterValidationFilter
   Bases: rest_framework.filters.BaseFilterBackend

   A backend filter that performs strict validation of query parameters for JSON:API spec conformance and raises a 400 error if non-conforming usage is found.

   If you want to add some additional non-standard query parameters, override query_regex adding the new parameters. Make sure to comply with the rules at https://jsonapi.org/format/#query-parameters.

   query_regex = re.compile('^(sort|include)$|^(?P<type>filter|fields|page)(\[[\w.\-]+\])?$')
       compiled regex that matches the allowed https://jsonapi.org/format/#query-parameters: sort and include stand alone; filter, fields, and page have [[]’s

   validate_query_params(request)
       Validate that query params are in the list of valid query keywords in query_regex

       Raises ValidationError – if not.

   filter_queryset(request, queryset, view)
       Overrides BaseFilterBackend.filter_queryset() by first validating the query params with validate_query_params()
rest_framework_json_api.metadata module

```python
class rest_framework_json_api.metadata.JSONAPIMetadata
    Bases: rest_framework.metadata.SimpleMetadata

This is the JSON:API metadata implementation. It returns an ad-hoc set of information about the view. There are not any formalized standards for OPTIONS responses for us to base this on.

    type_lookup = <rest_framework.utils.field_mapping.ClassLookupDict object>
    relation_type_lookup = <rest_framework.utils.field_mapping.ClassLookupDict object>
    determine_metadata(request, view)
    get_serializer_info(serializer)
    get_field_info(field)
```

rest_framework_json_api.pagination module

Pagination fields

```python
class rest_framework_json_api.pagination.JsonApiPageNumberPagination
    Bases: rest_framework.pagination.PageNumberPagination

A JSON:API compatible pagination format.

    page_query_param = 'page[number]'  
    page_size_query_param = 'page[size]'  
    max_page_size = 100
    build_link(index)
    get_paginated_response(data)
```

```python
class rest_framework_json_api.pagination.JsonApiLimitOffsetPagination
    Bases: rest_framework.pagination.LimitOffsetPagination

A limit/offset based style. For example:

http://api.example.org/accounts/?page[limit]=100
http://api.example.org/accounts/?page[offset]=400&page[limit]=100

    limit_query_param = 'page[limit]'  
    offset_query_param = 'page[offset]'  
    max_limit = 100
    get_last_link()  
    get_first_link()  
    get_paginated_response(data)
```
rest_framework_json_api.parsers module

Parsers

class rest_framework_json_api.parsers.JSONParser
    Bases: rest_framework.parsers.JSONParser

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

    A JSON:API client will send a payload that looks like this:

    ```
    {
        "data": {
            "type": "identities",
            "id": 1,
            "attributes": {
                "first_name": "John",
                "last_name": "Coltrane"
            }
        }
    }
    ```

    We extract the attributes so that DRF serializers can work as normal.

    media_type = 'application/vnd.api+json'

    renderer_class
        alias of rest_framework_json_api.renderers.JSONRenderer

    static parse_attributes(data)

    static parse_relationships(data)

    static 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.

    parse(stream, media_type=None, parser_context=None)
        Parses the incoming bytestream as JSON and returns the resulting data

rest_framework_json_api.relations module

class rest_framework_json_api.relations.SkipDataMixin(*args, **kwargs)
    Bases: object

    This workaround skips “data” rendering for relationships in order to save some sql queries and improve performance

    get_attribute(instance)

    to_representation(*args)

class rest_framework_json_api.relations.ManyRelatedFieldWithNoData(*args, **kwargs)
    Bases: rest_framework_json_api.relations.SkipDataMixin, rest_framework.relations.ManyRelatedField

    3.1. rest_framework_json_api package
class rest_framework_json_api.relations.HyperlinkedMixin(self_link_view_name=None, related_link_view_name=None, **kwargs):

    Bases: object

    self_link_view_name = None
    related_link_view_name = None
    related_link_lookup_field = 'pk'

get_url(name, view_name, kwargs, request)
    Given a name, view name and kwargs, return the URL that hyperlinks to the object.

    May raise a NoReverseMatch if the view_name and lookup_field attributes are not configured to correctly match the URL conf.

get_links(obj=None, lookup_field='pk')

class rest_framework_json_api.relations.HyperlinkedRelatedField(*args, **kwargs)

    Bases: rest_framework_json_api.relations.HyperlinkedMixin, rest_framework_json_api.relations.SkipDataMixin, rest_framework.relations.RelatedField

    classmethod many_init(*args, **kwargs)
        This method handles creating a parent ManyRelatedField instance when the many=True keyword argument is passed.

        Typically you won’t need to override this method.

        Note that we’re over-cautious in passing most arguments to both parent and child classes in order to try to cover the general case. If you’re overriding this method you’ll probably want something much simpler, eg:

        @classmethod
        def many_init(cls, *args, **kwargs):
            kwgars['child'] = cls()
            return CustomManyRelatedField(*args, **kwargs)

    class rest_framework_json_api.relations.ResourceRelatedField(*args, **kwargs)

        Bases: rest_framework_json_api.relations.HyperlinkedMixin, rest_framework.relations.PrimaryKeyRelatedField

        self_link_view_name = None
        related_link_view_name = None
        related_link_lookup_field = 'pk'

        default_error_messages = {'does_not_exist': 'Invalid pk \'{pk_value}’ – object does not exist.', 'incorrect_relation_type': 'Incorrect relation type. Expected {relation_type}, received {received_type}.'}, 'incorrect_type': 'Incorrect type. Expected resource identifier object, received {data_type}.'}, 'missing_id': 'Invalid resource identifier object: missing 'id' attribute', 'missing_type': 'Invalid resource identifier object: missing 'type' attribute', 'no_match': 'Invalid hyperlink - No URL match.'}, 'required': 'This field is required.'}

        use_pk_only_optimization()

        conflict(key, **kwargs)
            A helper method that simply raises a validation error.

        to_internal_value(data)

        to_representation(value)
get_resource_type_from_included_serializer()

Check to see if this resource has a different resource_name when included and return that name, or None

get_parent_serializer()

is_serializer(candidate)

getchoice(cutoff=None)

class rest_framework_json_api.relations.PolymorphicResourceRelatedField(*args, **kwargs)

Bases: rest_framework_json_api.relations.ResourceRelatedField

Inform DRF that the relation must be considered polymorphic. Takes a polymorphic_serializer as the first positional argument to retrieve then validate the accepted types set.

default_error_messages = {'does_not_exist': 'Invalid pk "{pk_value}" - object does not exist.', 'incorrect_relation_type': 'Incorrect relation type. Expected one of [{relation_type}], received {received_type}.', 'incorrect_type': 'Incorrect type. Expected resource identifier object, received {data_type}.', 'missing_id': "Invalid resource identifier object: missing 'id' attribute", 'missing_type': "Invalid resource identifier object: missing 'type' attribute", 'no_match': 'Invalid hyperlink - No URL match.', 'required': 'This field is required.'}

use_pk_only_optimization()

to_internal_value(data)

class rest_framework_json_api.relations.SerializerMethodFieldBase(*args, **kwargs)

Bases: rest_framework.fields.Field

bind(field_name, parent)

getchoattribute(instance)

class rest_framework_json_api.relations.ManySerializerMethodResourceRelatedField(*args, **kwargs)

Bases: rest_framework_json_api.relations.SerializerMethodFieldBase,
rest_framework_json_api.relations.ResourceRelatedField

to_representation(value)

class rest_framework_json_api.relations.SerializerMethodResourceRelatedField(*args, **kwargs)

Bases: rest_framework_json_api.relations.SerializerMethodFieldBase,
rest_framework_json_api.relations.ResourceRelatedField

Allows us to use serializer method RelatedFields with return querysets

many_kwargs = ['read_only', 'write_only', 'required', 'default', 'initial', 'source', 'label', 'help_text', 'style', 'error_messages', 'allow_empty', 'html_cutoff', 'html_cutoff_text', 'self_link_view_name', 'related_link_view_name', 'related_link_lookup_field', 'related_link_url_kwarg', 'method_name', 'model']

many_clas

alias of rest_framework_json_api.relations.ManySerializerMethodResourceRelatedField

classmethod many_init(*args, **kwargs)

class rest_framework_json_api.relations.ManySerializerMethodHyperlinkedRelatedField(*args, **kwargs)

Bases: rest_framework_json_api.relations.SkipDataMixin, rest_framework_json_api.relations.ManySerializerMethodResourceRelatedField

3.1. rest_framework_json_api package
class rest_framework_json_api.relations.SerializerMethodHyperlinkedRelatedField(*args, **kwargs):
    Bases: rest_framework_json_api.relations.SkipDataMixin, rest_framework_json_api.relations.SerializerMethodResourceRelatedField

    many_cls
    alias of rest_framework_json_api.relations.ManySerializerMethodHyperlinkedRelatedField

rest_framework_json_api.renderers module

Renderers
class rest_framework_json_api.renderers.JSONRenderer
    Bases: rest_framework.renderers.JSONRenderer

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

Render a JSON response per the JSON:API spec:

```json
{
    "data": [
        {
            "type": "companies",
            "id": "1",
            "attributes": {
                "name": "Mozilla",
                "slug": "mozilla",
                "date-created": "2014-03-13 16:33:37"
            }
        }
    ]
}
```

media_type = 'application/vnd.api+json'
format = 'vnd.api+json'
classmethod extract_attributes(fields, resource)
    Builds the attributes object of the JSON:API resource object.
classmethod extract_relationships(fields, resource, resource_instance)
    Builds the relationships top level object based on related serializers.
classmethod extract_relation_instance(field, resource_instance)
    Determines what instance represents given relation and extracts it.
    Relation instance is determined exactly same way as it determined in parent serializer
classmethod extract_included(fields, resource, resource_instance, included_resources, included_cache)
    Adds related data to the top level included key when the request includes ?include=example,example_field2
classmethod extract_meta(serializer, resource)
    Gathers the data from serializer fields specified in meta_fields and adds it to the meta object.
classmethod extract_root_meta(serializer, resource)
    Calls a get_root_meta function on a serializer, if it exists.
classmethod build_json_resource_obj(fields, resource, resource_instance, resource_name, serializer, force_type_resolution=False)

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

render_relationship_view(data, accepted_media_type=None, renderer_context=None)

render_errors(data, accepted_media_type=None, renderer_context=None)

render(data, accepted_media_type=None, renderer_context=None)

class rest_framework_json_api.renderersBrowsableAPIRenderer
Bases: rest_framework.renderers.BrowsableAPIRenderer

template = 'rest_framework_json_api/api.html'
includes_template = 'rest_framework_json_api/includes.html'

class rest_framework_json_api.serializers.ResourceIdentifierObjectSerializer(*args, **kwargs)
Bases: rest_framework.serializers.BaseSerializer

default_error_messages = {
    'does_not_exist': 'Invalid pk "{pk_value}" - object does not exist.',
    'incorrect_model_type': 'Incorrect model type. Expected {model_type}, received {received_type}.',
    'incorrect_type': 'Incorrect type. Expected pk value, received {data_type}.'}

model_class = None

to_representation(instance)

to_internal_value(data)

class rest_framework_json_api.serializers.SparseFieldsetsMixin(*args, **kwargs)
Bases: object

A serializer mixin that adds support for sparse fieldsets through fields query parameter.

Specification: https://jsonapi.org/format/#fetching-sparse-fieldsets

class rest_framework_json_api.serializers.IncludedResourcesValidationMixin(*args, **kwargs)
Bases: object

A serializer mixin that adds validation of include query parameter to support compound documents.


class rest_framework_json_api.serializers.ReservedFieldNamesMixin
Bases: object

Ensures that reserved field names are not used and an error raised instead.

get_fields()

class rest_framework_json_api.serializers.LazySerializersDict(parent, serializers)
Bases: collections.abc.Mapping

A dictionary of serializers which lazily import dotted class path and self.
class rest_framework_json_api.serializers.SerializerMetaclass:
  Bases: rest_framework.serializers.SerializerMetaclass

class rest_framework_json_api.serializers.Serializer(*args, **kwargs):
  Bases: 
    rest_framework_json_api.serializers.IncludedResourcesValidationMixin, 
    rest_framework_json_api.serializers.SparseFieldsetsMixin, 
    rest_framework_json_api.serializers.ReservedFieldNamesMixin, 
    rest_framework.serializers.Serializer

A Serializer is a model-less serializer class with additional support for JSON:API spec features.

As in JSON:API specification a type is always required you need to make sure that you define resource_name in your Meta class when deriving from this class.

Included Mixins:
- A mixin class to enable sparse fieldsets is included
- A mixin class to enable validation of included resources is included

class rest_framework_json_api.serializers.HyperlinkedModelSerializer(*args, **kwargs):
  Bases: 
    rest_framework_json_api.serializers.IncludedResourcesValidationMixin, 
    rest_framework_json_api.serializers.SparseFieldsetsMixin, 
    rest_framework_json_api.serializers.ReservedFieldNamesMixin, 
    rest_framework.serializers.HyperlinkedModelSerializer

A type of ModelSerializer that uses hyperlinked relationships instead of primary key relationships. Specifically:
- A 'url' field is included instead of the 'id' field.
- Relationships to other instances are hyperlinks, instead of primary keys.

Included Mixins:
- A mixin class to enable sparse fieldsets is included
- A mixin class to enable validation of included resources is included

class rest_framework_json_api.serializers.ModelSerializer(*args, **kwargs):
  Bases: 
    rest_framework_json_api.serializers.IncludedResourcesValidationMixin, 
    rest_framework_json_api.serializers.SparseFieldsetsMixin, 
    rest_framework_json_api.serializers.ReservedFieldNamesMixin, 
    rest_framework.serializers.ModelSerializer

A ModelSerializer is just a regular Serializer, except that:
- A set of default fields are automatically populated.
- A set of default validators are automatically populated.
- Default .create() and .update() implementations are provided.

The process of automatically determining a set of serializer fields based on the model fields is reasonably complex, but you almost certainly don’t need to dig into the implementation.

If the ModelSerializer class doesn’t generate the set of fields that you need you should either declare the extra/differing fields explicitly on the serializer class, or simply use a Serializer class.

Included Mixins:
- A mixin class to enable sparse fieldsets is included
- A mixin class to enable validation of included resources is included

serializer_related_field
  alias of rest_framework_json_api.relations.ResourceRelatedField
get_field_names(declared_fields, info)
We override the parent to omit explicity defined meta fields (such as SerializerMethodFields) from the list of declared fields.

class rest_framework_json_api.serializers.PolymorphicSerializerMetaclass(name, bases, attrs)
Bases: rest_framework_json_api.serializers.SerializerMetaClass
This metaclass ensures that the polymorphic_serializers is correctly defined on a PolymorphicSerializer class and make a cache of model/serializer/type mappings.

class rest_framework_json_api.serializers.PolymorphicModelSerializer(*args, **kwargs)
Bases: rest_framework_json_api.serializers.ModelSerializer
A serializer for polymorphic models. Useful for “lazy” parent models. Leaves should be represented with a regular serializer.

get_fields()
Return an exhaustive list of the polymorphic serializer fields.

classmethod get_polymorphic_serializer_for_instance(instance)
Return the polymorphic serializer associated with the given instance/model. Raise NotImplementedError if no serializer is found for the given model. This usually means that a serializer is missing in the class’s polymorphic_serializers attribute.

classmethod get_polymorphic_model_for_serializer(serializer)
Return the polymorphic model associated with the given serializer. Raise NotImplementedError if no model is found for the given serializer. This usually means that a serializer is missing in the class’s polymorphic_serializers attribute.

classmethod get_polymorphic_serializer_for_type(obj_type)
Return the polymorphic serializer associated with the given type. Raise NotImplementedError if no serializer is found for the given type. This usually means that a serializer is missing in the class’s polymorphic_serializers attribute.

classmethod get_polymorphic_model_for_type(obj_type)
Return the polymorphic model associated with the given type. Raise NotImplementedError if no model is found for the given type. This usually means that a serializer is missing in the class’s polymorphic_serializers attribute.

classmethod get_polymorphic_types()
Return the list of accepted types.

to_representation(instance)
Retrieve the appropriate polymorphic serializer and use this to handle representation.

to_internal_value(data)
Ensure that the given type is one of the expected polymorphic types, then retrieve the appropriate polymorphic serializer and use this to handle internal value.

rest_framework_json_api.settings module

This module provides the json_api_settings object that is used to access Django REST framework JSON:API settings, checking for user settings first, then falling back to the defaults.
class rest_framework_json_api.settings.JSONAPISettings(user_settings=<LazySettings
    "example.settings">,
    defaults={'FORMAT_FIELD_NAMES':
        False, 'FORMAT_RELATED_LINKS': False,
        'FORMAT_TYPES': False,
        'PLURALIZE_TYPES': False,
        'UNIFORM_EXCEPTIONS': False})

Bases: object

A settings object that allows JSON:API settings to be access as properties.

rest_framework_json_api.settings.reload_json_api_settings(*args, **kwargs)

rest_framework_json_api.utils module

rest_framework_json_api.utils.get_resource_name(context, expand_polymorphic_types=False)
    Return the name of a resource.

rest_framework_json_api.utils.get_serializer_fields(serializer)
rest_framework_json_api.utils.format_field_names(obj, format_type=None)
    Takes a dict and returns it with formatted keys as set in format_type or JSON_API_FORMAT_FIELD_NAMES
    
    Format_type Either ‘dasherize’, ‘camelize’, ‘capitalize’ or ‘underscore’

rest_framework_json_api.utils.undo_format_field_names(obj)
    Takes a dict and undo format field names to underscore which is the Python convention but only in case
    JSON_API_FORMAT_FIELD_NAMES is actually configured.

rest_framework_json_api.utils.format_field_name(field_name)
    Takes a field name and returns it with formatted keys as set in JSON_API_FORMAT_FIELD_NAMES

rest_framework_json_api.utils.undo_format_field_name(field_name)
    Takes a string and undos format field name to underscore which is the Python convention but only in case
    JSON_API_FORMAT_FIELD_NAMES is actually configured.

rest_framework_json_api.utils.format_link_segment(value, format_type=None)
    Takes a string value and returns it with formatted keys as set in format_type or
    JSON_API_FORMAT_RELATED_LINKS.
    
    Format_type Either ‘dasherize’, ‘camelize’, ‘capitalize’ or ‘underscore’

rest_framework_json_api.utils.undo_format_link_segment(value)
    Takes a link segment and undos format link segment to underscore which is the Python convention but only in case
    JSON_API_FORMAT_RELATED_LINKS is actually configured.

rest_framework_json_api.utils.format_value(value, format_type=None)
rest_framework_json_api.utils.format_resource_type(value, format_type=None, pluralize=None)
rest_framework_json_api.utils.get_related_resource_type(relation)
rest_framework_json_api.utils.get_resource_type_from_model(model)
rest_framework_json_api.utils.get_resource_type_from_queryset(qs)
rest_framework_json_api.utils.get_resource_type_from_instance(instance)
rest_framework_json_api.utils.get_resource_type_from_manager(manager)
rest_framework_json_api.utils.get_resource_type_from_serializer(serializer)
rest_framework_json_api.utils.get_included_resources(request, serializer=None)
Build a list of included resources.

rest_framework_json_api.utils.get_default_included_resources_from_serializer(serializer)

rest_framework_json_api.utils.get_included_serializers(serializer)

rest_framework_json_api.utils.get_relation_instance(resource_instance, source, serializer)

rest_framework_json_api.utils.is_relationship_field(field)

class rest_framework_json_api.utils.Hyperlink(url, name)
Bases: str
A string like object that additionally has an associated name. We use this for hyperlinked URLs that may render as a named link in some contexts, or render as a plain URL in others.

Comes from Django REST framework 3.2 https://github.com/tomchristie/django-rest-framework

is_hyperlink = True

rest_framework_json_api.utils.format_drf_errors(response, context, exc)

rest_framework_json_api.utils.format_error_object(message, pointer, response)

rest_framework_json_api.utils.format_errors(data)

rest_framework_json_api.views module

class rest_framework_json_api.views.PreloadIncludesMixin
Bases: object
This mixin provides a helper attributes to select or prefetch related models based on the include specified in the URL.

__all__ can be used to specify a prefetch which should be done regardless of the include

```
# When MyViewSet is called with ?include=author it will prefetch author and authorbio

class MyViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    prefetch_for_includes = {
        '__all__': [],
        'category.section': ['category']
    }
    select_for_includes = {
        '__all__': [],
        'author': ['author', 'author__authorbio'],
    }
```

get_select_related(include)

get_prefetch_related(include)

get_queryset(*args, **kwargs)

class rest_framework_json_api.views.AutoPrefetchMixin
Bases: object

get_queryset(*args, **kwargs)

This mixin adds automatic prefetching for OneToOne and ManyToMany fields.

3.1. rest_framework_json_api package
class rest_framework_json_api.views.RelatedMixin
    Bases: object
    This mixin handles all related entities, whose Serializers are declared in “related_serializers”
    retrieve_related(request, *args, **kwargs)
    get_related_serializer(instance, **kwargs)
    get_related_serializer_class()
    get_related_field_name()
    get_related_instance()

class rest_framework_json_api.views.ModelViewSet(**kwargs)
    http_method_names = ['get', 'post', 'patch', 'delete', 'head', 'options']

class rest_framework_json_api.views.ReadOnlyModelViewSet(**kwargs)
    http_method_names = ['get', 'post', 'patch', 'delete', 'head', 'options']

class rest_framework_json_api.views.RelationshipView(**kwargs)
    Bases: rest_framework_json_api.views.ResourceIdentifierObjectSerializer
    serializer_class
        alias of rest_framework_json_api.serializers.ResourceIdentifierObjectSerializer
    self_link_view_name = None
    related_link_view_name = None
    field_name_mapping = {}
    http_method_names = ['get', 'post', 'patch', 'delete', 'head', 'options']
    get_serializer_class()
    get_url(name, view_name, kwargs, request)
        Given a name, view name and kwargs, return the URL that hyperlinks to the object.
        May raise a NoReverseMatch if the view_name and lookup_field attributes are not configured to correctly match the URL conf.
    get_links()
    get(request, *args, **kwargs)
    remove_relationships(instance_manager, field)
    patch(request, *args, **kwargs)
    post(request, *args, **kwargs)
    delete(request, *args, **kwargs)
    get_related_instance()
    get_related_field_name()
    get_resource_name()
set_resource_name(value)

property resource_name
Django REST framework JSON:API (aka DJA) should be easy to contribute to. If anything is unclear about how to contribute, please submit an issue on GitHub so that we can fix it!

Before writing any code, have a conversation on a GitHub issue to see if the proposed change makes sense for the project.

### 4.1 Setup development environment

#### 4.1.1 Clone

To start developing on Django REST framework JSON:API you need to first clone the repository:

```
git clone https://github.com/django-json-api/django-rest-framework-json-api.git
```

#### 4.1.2 Testing

To run tests clone the repository, and then:

```
# Setup the virtual environment
python3 -m venv env
source env/bin/activate
pip install -r requirements.txt

# Format code
black .

# Run linting
flake8

# Run tests
pytest
```
4.1.3 Running against multiple environments

You can also use the excellent tox testing tool to run the tests against all supported versions of Python and Django. Install tox globally, and then simply run:

```
tox
```

4.1.4 Setup pre-commit

pre-commit hooks is an additional option to check linting and formatting of code independent of an editor before you commit your changes with git.

To setup pre-commit hooks first create a testing environment as explained above before running below commands:

```
pip install pre-commit
pre-commit install
```

4.2 For maintainers

4.2.1 Create release

To upload a release (using version 1.2.3 as the example) first setup testing environment as above before running below commands:

```
python setup.py sdist bdist_wheel
twine upload dist/*
git tag -a v1.2.3 -m 'Release 1.2.3'
git push --tags
```

4.2.2 Add maintainer

In case a new maintainer joins our team we need to consider to what of following services we want to add them too:

- Github organization
- Read the Docs project
- PyPi project
- Google Groups security mailing list
• genindex
• search
**PYTHON MODULE INDEX**

```plaintext
rest_framework_json_api, 27
rest_framework_json_api.django_filters, 27
rest_framework_json_api.django_filters.backends, 27
rest_framework_json_api.exceptions, 31
rest_framework_json_api.filters, 31
rest_framework_json_api.metadata, 32
rest_framework_json_api.pagination, 32
rest_framework_json_api.parsers, 33
rest_framework_json_api.relations, 33
rest_framework_json_api.renderers, 36
rest_framework_json_api.schemas, 28
rest_framework_json_api.schemas.openapi, 28
rest_framework_json_api.serializers, 37
rest_framework_json_api.settings, 39
rest_framework_json_api.utils, 40
rest_framework_json_api.views, 41
```
INDEX

A
AutoPrefetchMixin (class in rest_framework_json_api.views), 41
AutoSchema (class in rest_framework_json_api.schemas.openapi), 28

B
bind() (rest_framework_json_api.relations.SerializerMethodFieldBase method), 35
BrowsableAPIRenderer (class in rest_framework_json_api.renderers), 37
build_json_resource_obj() (rest_framework_json_api.renderers.JSONRenderer class method), 36
build_link() (rest_framework_json_api.pagination.JsonApiPageNumberPagination method), 32

C
Conflict, 31
conflict() (rest_framework_json_api.relations.ResourceRelatedField method), 34
content_types (rest_framework_json_api.schemas.openapi.AutoSchema attribute), 28

D
default_detail (rest_framework_json_api.exceptions.Conflict attribute), 31
default_error_messages (rest_framework_json_api.relations.PolymorphicResourceRelatedField attribute), 35
default_error_messages (rest_framework_json_api.relations.ResourceRelatedField attribute), 34
default_error_messages (rest_framework_json_api.serializers.ResourceIdentifierObjectSerializer attribute), 37
delete() (rest_framework_json_api.views.RelationshipView method), 42
determine_metadata() (rest_framework_json_api.metadata.JSONAPIMetadata method), 32

E
exception_handler() (in module rest_framework_json_api.exceptions), 31
extract_attributes() (rest_framework_json_api.renderers.JSONRenderer class method), 36
extract_included() (rest_framework_json_api.renderers.JSONRenderer class method), 36
extract_meta() (rest_framework_json_api.renderers.JSONRenderer class method), 36
extract_relation_instance() (rest_framework_json_api.renderers.JSONRenderer class method), 36
extract_relationships() (rest_framework_json_api.renderers.JSONRenderer class method), 36
extract_root_meta() (rest_framework_json_api.renderers.JSONRenderer class method), 36

F
field_name_mapping (rest_framework_json_api.views.RelationshipView attribute), 42
filter_queryset() (rest_framework_json_api.filters.QueryParameterValidationFilter method), 31
filter_regex (rest_framework_json_api.django_filters.backends.DjangoFilterBackend attribute), 28
format (rest_framework_json_api.renderers.JSONRenderer attribute), 36
format_drf_errors() (in module rest_framework_json_api.utils), 41
format_error_object() (in module rest_framework_json_api.utils), 41
format_errors() (in module rest_framework_json_api.utils), 41
format_field_name() (in module rest_framework_json_api.utils), 40

DjangoFilterBackend (class in rest_framework_json_api.django_filters.backends), 27
get_resource_name() (in module rest_framework_json_api.views.RelationshipView method), 42
get_resource_name() (rest_framework_json_api.views.RelationshipView method), 42
get_resource_type_from_included_serializer() (rest_framework_json_api.relations.ResourceRelatedField method), 34
get_resource_type_from_instance() (in module rest_framework_json_api.utils), 40
get_resource_type_from_manager() (in module rest_framework_json_api.utils), 40
get_resource_type_from_model() (in module rest_framework_json_api.utils), 40
get_resource_type_from_queryset() (in module rest_framework_json_api.utils), 40
get_resource_type_from_serializer() (in module rest_framework_json_api.utils), 40
get_schema() (rest_framework_json_api.schemas.openapi.SchemaGenerator method), 29
get_schema_operation_parameters() (rest_framework_json_api.django_filters.backends.DjangoFilterBackend method), 28
get_select_related() (rest_framework_json_api.views.PreloadIncludesMixin method), 41
get_serializer_class() (rest_framework_json_api.views.RelationshipView method), 42
get_serializer_fields() (in module rest_framework_json_api.utils), 40
get_serializer_info() (rest_framework_json_api.metadata.JSONAPIMetadata method), 32
get_url() (rest_framework_json_api.utils.Hyperlink method), 34
get_url() (rest_framework_json_api.views.HyperlinkedRelatedField method), 42
get_url() (rest_framework_json_api.views.RelationshipView method), 42

http_method_names (rest_framework_json_api.views.ModelViewSet attribute), 42
http_method_names (rest_framework_json_api.views.ReadOnlyModelViewSet attribute), 42
http_method_names (rest_framework_json_api.views.RelationshipView attribute), 42
Hyperlink (class in rest_framework_json_api.views), 41
HyperlinkedMixIn (class in rest_framework_json_api.relations), 33
HyperlinkedModelSerializer (class in rest_framework_json_api.serializers), 38
HyperlinkedRelatedField (class in rest_framework_json_api.relations), 34

IncludedResourcesValidationMixin (class in rest_framework_json_api.serializers), 37
includes_template (rest_framework_json_api.renderers.BrowsableAPIRenderer attribute), 37
is_hyperlink (rest_framework_json_api.utils.Hyperlink attribute), 41
is_relationship_field() (in module rest_framework_json_api.utils), 41
is_serializer() (rest_framework_json_api.relations.ResourceRelatedField method), 35

jsonapi_components (rest_framework_json_api.schemas.openapi.SchemaGenerator attribute), 29
JsonApiLimitOffsetPagination (class in rest_framework_json_api.pagination), 32
JSONAPIMetadata (class in rest_framework_json_api.metadata), 32
JsonApiPageNumberPagination (class in rest_framework_json_api.pagination), 32
JSONAPISettings (class in rest_framework_json_api.settings), 39
JSONParser (class in rest_framework_json_api.parsers), 33
JSONRenderer (class in rest_framework_json_api.renderers), 36
LazySerializersDict (class in rest_framework_json_api.serializers), 37
limit_query_param (rest_framework_json_api.pagination.JsonApiLimitOffsetPagination attribute), 32

many_cls (rest_framework_json_api.relations.SerializerMethodHyperlinkedRelatedField attribute), 36
many_cls (rest_framework_json_api.relations.SerializerMethodResourceRelatedField attribute), 35
many_init() (rest_framework_json_api.relations.HyperlinkedRelatedField class method), 34
many_init() (rest_framework_json_api.relations.SerializerMethodHyperlinkedRelatedField class method), 35
many_init() (rest_framework_json_api.relations.SerializerMethodResourceRelatedField class method), 35
ManyRelatedFieldWithNoData (class in rest_framework_json_api.serializers), 33
ManySerializerMethodHyperlinkedRelatedField (class in rest_framework_json_api.relations), 33
ManySerializerMethodResourceRelatedField (class in rest_framework_json_api.relations), 35

Index 53
<table>
<thead>
<tr>
<th>function/method</th>
<th>module</th>
<th>line</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove_invalid_fields()</td>
<td>rest_framework_json_api.views.OrderingFilter</td>
<td>31</td>
</tr>
<tr>
<td>remove_relationships()</td>
<td>rest_framework_json_api.views.RelationshipView</td>
<td>42</td>
</tr>
<tr>
<td>render()</td>
<td>rest_framework_json_api.renderers.JSONRenderer</td>
<td>37</td>
</tr>
<tr>
<td>render_errors()</td>
<td>rest_framework_json_api.renderers.JSONRenderer</td>
<td>37</td>
</tr>
<tr>
<td>render_relationship_view()</td>
<td>rest_framework_json_api.renderers.JSONRenderer</td>
<td>37</td>
</tr>
<tr>
<td>rendered_with_json_api()</td>
<td>in rest_framework_json_api.exceptions</td>
<td>31</td>
</tr>
<tr>
<td>renderer_class()</td>
<td>rest_framework_json_api.parsers.JSONParser</td>
<td>33</td>
</tr>
<tr>
<td>ReservedFieldNamesMixin</td>
<td>in rest_framework_json_api.serializers</td>
<td>37</td>
</tr>
<tr>
<td>resource_name()</td>
<td>in rest_framework_json_api.views.RelationshipView</td>
<td>43</td>
</tr>
<tr>
<td>ResourceIdentifierObjectSerializer</td>
<td>in rest_framework_json_api.serializers</td>
<td>37</td>
</tr>
<tr>
<td>ResourceRelatedField</td>
<td>in rest_framework_json_api.relations</td>
<td>34</td>
</tr>
<tr>
<td>rest_framework_json_api</td>
<td>module</td>
<td>27</td>
</tr>
<tr>
<td>rest_framework_json_api.django_filters</td>
<td>module</td>
<td>27</td>
</tr>
<tr>
<td>rest_framework_json_api.django_filters.backends</td>
<td>module</td>
<td>28</td>
</tr>
<tr>
<td>rest_framework_json_api.exceptions</td>
<td>module</td>
<td>31</td>
</tr>
<tr>
<td>rest_framework_json_api.filters</td>
<td>module</td>
<td>31</td>
</tr>
<tr>
<td>rest_framework_json_api.metadata</td>
<td>module</td>
<td>32</td>
</tr>
<tr>
<td>rest_framework_json_api.pagination</td>
<td>module</td>
<td>32</td>
</tr>
<tr>
<td>rest_framework_json_api.parsers</td>
<td>module</td>
<td>33</td>
</tr>
<tr>
<td>rest_framework_json_api.relations</td>
<td>module</td>
<td>33</td>
</tr>
<tr>
<td>rest_framework_json_api.renderers</td>
<td>module</td>
<td>36</td>
</tr>
<tr>
<td>rest_framework_json_api.schemas</td>
<td>module</td>
<td>28</td>
</tr>
<tr>
<td>rest_framework_json_api.schemas.openapi</td>
<td>module</td>
<td>28</td>
</tr>
<tr>
<td>rest_framework_json_api.serializers</td>
<td>module</td>
<td>37</td>
</tr>
<tr>
<td>rest_framework_json_api.settings</td>
<td>module</td>
<td>39</td>
</tr>
<tr>
<td>rest_framework_json_api.utils</td>
<td>module</td>
<td>40</td>
</tr>
<tr>
<td>retrieve_related()</td>
<td>in rest_framework_json_api.views.RelatedMixin</td>
<td>42</td>
</tr>
<tr>
<td>SchemaGenerator</td>
<td>in rest_framework_json_api.schemas.openapi</td>
<td>28</td>
</tr>
<tr>
<td>search_param()</td>
<td>in rest_framework_json_api.django_filters.backends.DjangoFilterBackend</td>
<td>28</td>
</tr>
<tr>
<td>self_link_view_name</td>
<td>in rest_framework_json_api.views.RelationshipView</td>
<td>42</td>
</tr>
<tr>
<td>set_resource_name()</td>
<td>in rest_framework_json_api.views.RelationshipView</td>
<td>42</td>
</tr>
<tr>
<td>Serializer</td>
<td>in rest_framework_json_api.serializers</td>
<td>38</td>
</tr>
<tr>
<td>serializer_class()</td>
<td>in rest_framework_json_api.views.RelationshipView</td>
<td>42</td>
</tr>
<tr>
<td>serializer_related_field</td>
<td>in rest_framework_json_api.serializers.ModelSerializer</td>
<td>38</td>
</tr>
<tr>
<td>SerializerMetaclass</td>
<td>in rest_framework_json_api.serializers</td>
<td>37</td>
</tr>
<tr>
<td>SerializerMethodFieldBase</td>
<td>in rest_framework_json_api.relations</td>
<td>35</td>
</tr>
<tr>
<td>SerializerMethodHyperlinkedRelatedField</td>
<td>in rest_framework_json_api.relations</td>
<td>35</td>
</tr>
<tr>
<td>SerializerMethodResourceRelatedField</td>
<td>in rest_framework_json_api.relations</td>
<td>35</td>
</tr>
<tr>
<td>set_resource_name()</td>
<td>in rest_framework_json_api.views.RelationshipView</td>
<td>42</td>
</tr>
<tr>
<td>SkipDataMixin</td>
<td>in rest_framework_json_api.relations</td>
<td>33</td>
</tr>
<tr>
<td>SparseFieldsetsMixin</td>
<td>in rest_framework_json_api.relations</td>
<td>37</td>
</tr>
<tr>
<td>status_code()</td>
<td>in rest_framework_json_api.exceptions.Conflict</td>
<td>31</td>
</tr>
<tr>
<td>template</td>
<td>in rest_framework_json_api.renderers.BrowsableAPIRenderer</td>
<td>37</td>
</tr>
<tr>
<td>to_internal_value()</td>
<td>in rest_framework_json_api.relations.PolymorphicResourceRelatedField</td>
<td>35</td>
</tr>
<tr>
<td>to_internal_value()</td>
<td>in rest_framework_json_api.relations.ResourceRelatedField</td>
<td>35</td>
</tr>
</tbody>
</table>

Index 55
method), 34

to_internal_value()
  (rest_framework_json_api.serializers.PolymorphicModelSerializer
   method), 39

to_internal_value()
  (rest_framework_json_api.serializers.ResourceIdentifierObjectSerializer
   method), 37

to_representation()
  (rest_framework_json_api.relations.ManySerializerMethodResourceRelatedField
   method), 35

to_representation()
  (rest_framework_json_api.relations.ResourceRelatedField
   method), 34

to_representation()
  (rest_framework_json_api.relations.SkipDataMixin
   method), 33

to_representation()
  (rest_framework_json_api.serializers.PolymorphicModelSerializer
   method), 39

to_representation()
  (rest_framework_json_api.serializers.ResourceIdentifierObjectSerializer
   method), 37

type_lookup (rest_framework_json_api.metadata.JSONAPIMetadata
   attribute), 32

U

undo_format_field_name() (in module
  rest_framework_json_api.utils), 40

undo_format_field_names() (in module
  rest_framework_json_api.utils), 40

undo_format_link_segment() (in module
  rest_framework_json_api.utils), 40

use_pk_only_optimization()
  (rest_framework_json_api.relations.PolymorphicResourceRelatedField
   method), 35

use_pk_only_optimization()
  (rest_framework_json_api.relations.ResourceRelatedField
   method), 34

V

validate_query_params()
  (rest_framework_json_api.filters.QueryParameterValidationFilter
   method), 31