drf-jsonapi Documentation

Release 1.0

Chris Brantley

Feb 05, 2019

Contents:

CHAPTER 1

Quick Start

Note: This quickstart guide is based on the example Django project located at `/example` in the repository.

1.1 Installation

At the command line:

```
$ pip install vacasa-drf-jsonapi
```

1.2 Create a New Django Project

Let's get started with a fresh Django project:

```
$ django-admin startproject example
```

And let's add an app for our API:

```
$ cd example
$ django-admin startapp api
```

1.3 Settings

`drf_jsonapi` doesn't have any settings of its own, but because it is built on top of Django Rest Framework you'll need to specify some settings to direct DRF to leverage the special classes that `drf_jsonapi` provides.

Make the following changes to your `settings.py` file:

```
INSTALLED_APPS = [
    ...
    "api",
    "rest_framework",
]

REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": (),
    "DEFAULT_PERMISSION_CLASSES": (),
    "UNAUTHENTICATED_USER": None,
    "EXCEPTION_HANDLER": "drf_jsonapi.exception_handlers.jsonapi_exception_handler",
    "DEFAULT_RENDERER_CLASSES": (
        "drf_jsonapi.renderers.JSONRenderer",
        "drf_jsonapi.renderers.BrowsableAPIRenderer",
    ),
    "DEFAULT_PARSER_CLASSES": (
        "drf_jsonapi.parsers.JSONAPIParser",
        "rest_framework.parsers.JSONParser",
    ),
}
```

There are several things going on here. First, we're adding our new `api` app and `rest_framework` to the list of installed apps.

Next, we're specifying some settings for django rest framework. First we're disabling authentication on the API. Obviously this isn't something you'd want to do in production but it's fine for this example and makes things much simpler.

Note: See the Django Rest Framework Documentation for more information on securing your API.

The next part overrides the default exception handler, renderer, and parser classes to use the `drf_jsonapi` versions.

1.4 Models

While not required, `drf_jsonapi` is designed to work conveniently with Django models and the Django ORM. In this example we'll create some models with many-to-many relationships to show off JSON-API's ability to work with graph-like data structures.

Add the following to your `api/models.py` file:

```
from django.db import models


class Publisher(models.Model):
    name = models.CharField(max_length=128)


class Author(models.Model):
    name = models.CharField(max_length=128)


class Book(models.Model):
    title = models.CharField(max_length=128)
    authors = models.ManyToManyField(Author, related_name="books")
```

(continues on next page)

(continued from previous page)

```
    publisher = models.ForeignKey(
        Publisher, related_name="books", on_delete=models.CASCADE
    )
```

1.5 Serializers

Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into JSON. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.

Note: For more information on serializers see the Django Rest Framework documentation here.

`drf-jsonapi` provides the `ResourceModelSerializer` base class that you can use to create your own serializers. If you're using Django models and the Django ORM (as we are here) then your serializers can be very simple.

If you're building an API for other data structures such as in-memory data or data fetched from a NOSQL database you'll want to use the `ResourceSerializer` base class and manually define how the data should be serialized and deserialized.

Let's start with the serializer for the *Publisher* model. Create a new file at `api/serializers.py` and add the following:

```
from drf_jsonapi.serializers import ResourceModelSerializer
from drf_jsonapi.relationships import RelationshipHandler

from api.models import Publisher, Author, Book


class PublisherSerializer(ResourceModelSerializer):
    class Meta:
        type = "publisher"
        basename = "publishers"
        model = Publisher
        id_field = "pk"
        fields = ("name",)

    @staticmethod
    def define_relationships():
        return {"books": RelationshipHandler(BookSerializer, "books", many=True)}
```

If you're familiar with Django Rest Framework you might have noticed that this is very similar to ModelSerializer with some additional attributes.

- The `type` attribute describes the JSON-API resource type.
- `basename` describes the base path for this resources. This is an optional attribute that, if omitted, will default to the resource type. I prefer types to be singular and basenames to be plural so this gives you that option.
- `model` is simply a reference to the Model class that this serializer is for.
- `id_field` is the model field that should be used as the identifier for each resource. This is also optional and will default to `pk` but it's useful if you have an alternative field (such as a UUID field) that should be used instead of `pk`.

- `fields` is a tuple (or list) of fields that should be included as attributes for the resource. One thing to note is that you don't want to include any foreign key relationships here. These should be direct attributes of the model.

The `define_relationships()` static method is used to describe how this resource relates to other resources. It should return a `dict` where the key is the name of the relationship ("books" in this case) and the value is an instance of `RelationshipHandler`. The `RelationshipHandler` constructor takes 3 arguments: the related resources serializer class, the lookup field for the relationship, and whether it's a "To-Many" relationship.

For simple relationships using the Django ORM this is all you need. For more complex relationships you'll want to create your own RelationshipHandler class by sub-classing *RelationshipHandler*.

Using these same concepts we can flesh out the serializers for the other models:

```
class AuthorSerializer(ResourceModelSerializer):
    class Meta:
        type = "author"
        basename = "authors"
        model = Author
        id_field = "pk"
        fields = ("name",)

    @staticmethod
    def define_relationships():
        return {"books": RelationshipHandler(BookSerializer, "books", many=True)}


class BookSerializer(ResourceModelSerializer):
    class Meta:
        type = "book"
        basename = "books"
        model = Book
        id_field = "pk"
        fields = ("title",)

    @staticmethod
    def define_relationships():
        return {
            "authors": RelationshipHandler(AuthorSerializer, "authors", many=True),
            "publisher": RelationshipHandler(PublisherSerializer, "publisher"),
        }
```

1.6 Views

Django Rest Framework has a concept called Viewsets which are a form of class-based View. `drf_jsonapi` expands on this concept with specialized `ViewSets` for handling JSON-API requests and responses.

Let's get started with a basic ViewSet for Publishers. Edit your `api/views.py` and add the following code:

```
from drf_jsonapi.viewsets import ReadWriteViewSet

from api.models import Publisher
from api.serializers import PublisherSerializer


class PublisherViewSet(ReadWriteViewSet):
    serializer_class = PublisherSerializer
    collection = Publisher.objects.all()
```

This is the bare minimum implementation of a `ViewSet`. We're sub-classing `ReadWriteViewSet` here. (There is also a `ReadOnlyViewSet` that does not allow POST, PATCH, or DELETE requests.) We declare two class attributes: The `serializer_class` of the resource and the default `collection` which, in this case, is a queryset of all `Publisher` objects.

This `ViewSet` will work fine for now but it is very limited. We'll come back to this later to make some improvements.

1.7 The Router

At this point we have our Models to describe our data. We've also written some serializers to define how render those models according to JSON-API spec. We've also written a simple ViewSet for Publishers to handle requests and responses.

Now we need to bring it all together by writing some URL routing rules so we can try out our API!

Edit your `api/urls.py` file and add the following:

```
from drf_jsonapi.routers import Router

from api.views import PublisherViewSet

router = Router(trailing_slash=False)
router.register(PublisherViewSet)

urlpatterns += router.urls
```

You'll also want to edit `example/urls.py` to include these urls:

```
from django.urls import path, include

urlpatterns = [
    path("", include("api.urls"))
]
```

The `Router` class is a sub-class of Django Rest Framework's `DefaultRouter` class. This class does all the work of creating the url configurations so requests to `/publishers` and `/publishers/1` are correctly routed to your `ViewSets`.

To do this just create an instance of `Router` and register your `ViewSets` with it. Then you populate `urlpatterns` with `router.urls` and you're ready to go.

1.8 Testing out your API

At this point we're almost ready to try out our API. Before we can do that though we need a database so we can persist some data. Fortunately Django provides out-of-the-box support for Sqlite. That's perfect for our little example.

Let's create our migrations so we can set up the database schema:

```
$ python manage.py makemigrations api
```

Then we need to apply the migrations:

```
$ python manage.py migrate
```

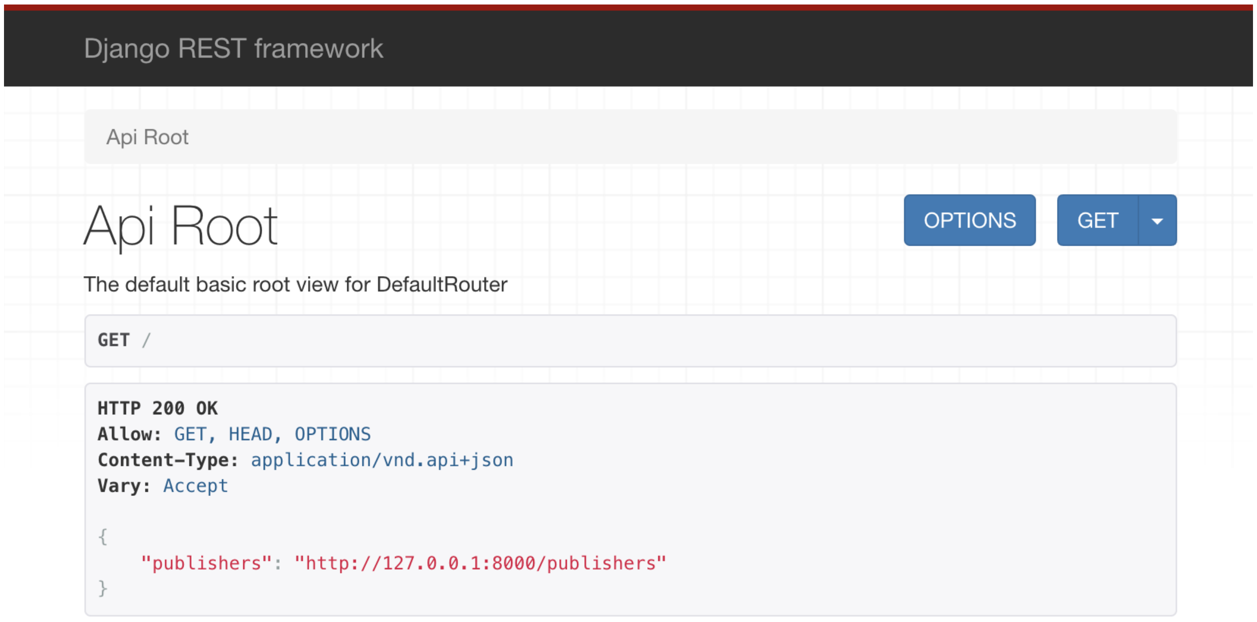
Now we have an empty database which is somewhat boring. We can use Django's fixture support to load in some sample data. Copy `/example/api/fixtures/example.json` to your project and load it into the database with the following command:

```
$ python manage.py loaddata example
```

Now that we have some data let's test out the API! First we need to launch the development server:

```
$ python manage.py runserver
```

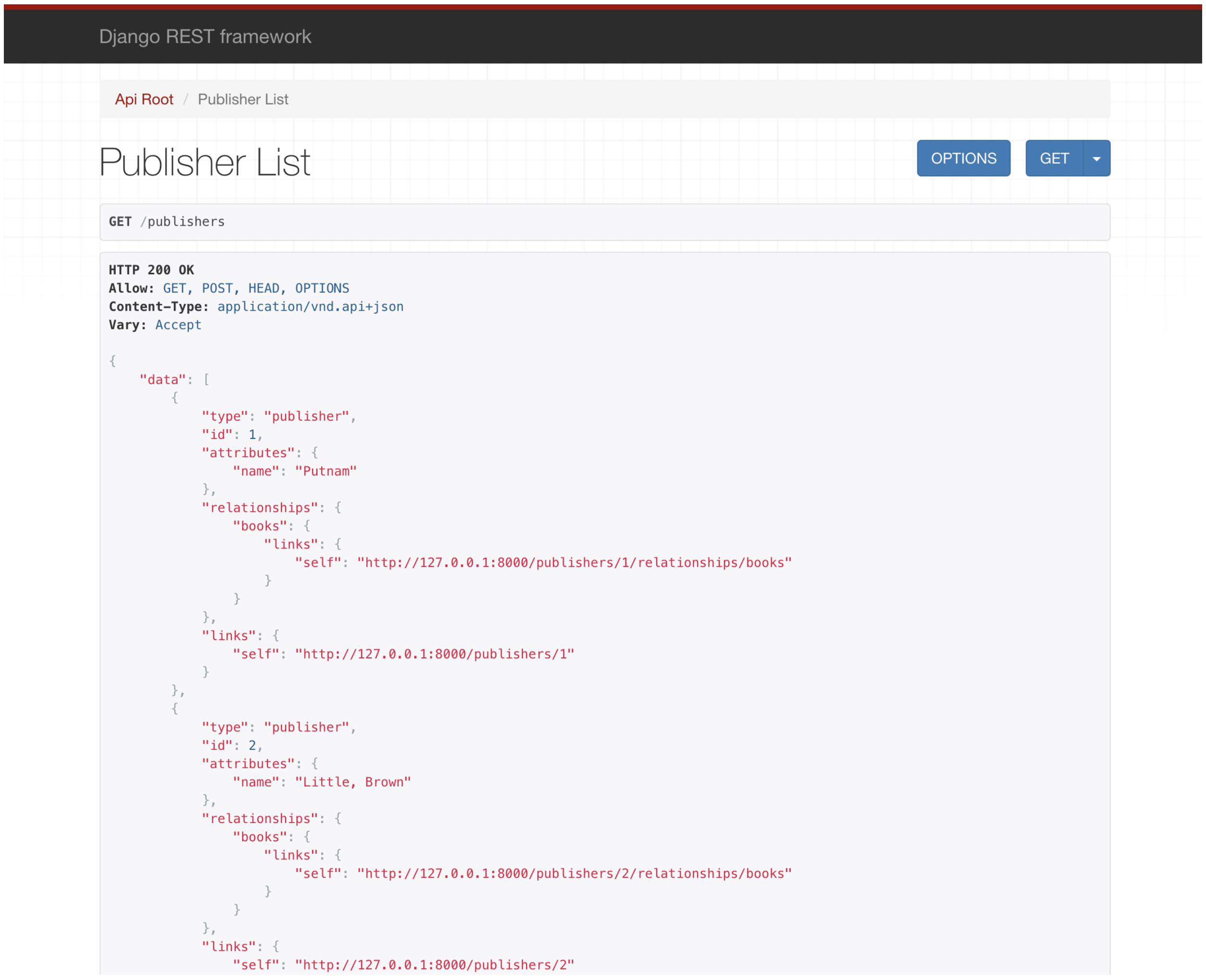
You should now have a development server up and running at `http://127.0.0.1:8000/`. Load that up in a browser and take a look. You should see something like this:



```
GET /
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/vnd.api+json
Vary: Accept

{
    "publishers": "http://127.0.0.1:8000/publishers"
}
```

What you're seeing is Django Rest Framework's browsable API. This is the root view which lists links for each top-level resource. At the moment we only have a single resource: `/publishers`. Follow the link to see the list of publishers. You should see something like this:



```
GET /publishers
```

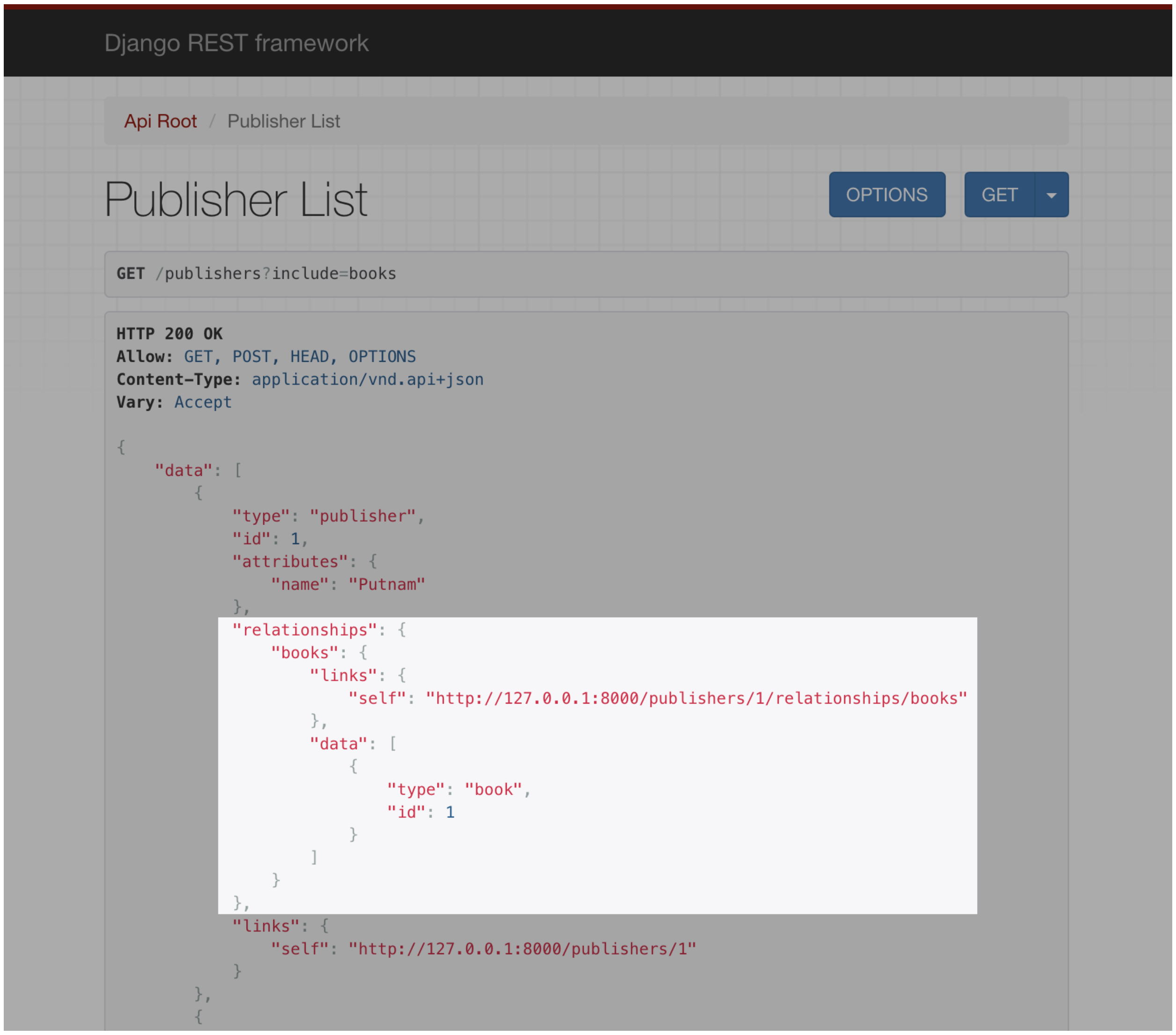
```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/vnd.api+json
Vary: Accept

{
    "data": [
        {
            "type": "publisher",
            "id": 1,
            "attributes": {
                "name": "Putnam"
            },
            "relationships": {
                "books": {
                    "links": {
                        "self": "http://127.0.0.1:8000/publishers/1/relationships/books"
                    }
                }
            },
            "links": {
                "self": "http://127.0.0.1:8000/publishers/1"
            }
        },
        {
            "type": "publisher",
            "id": 2,
            "attributes": {
                "name": "Little, Brown"
            },
            "relationships": {
                "books": {
                    "links": {
                        "self": "http://127.0.0.1:8000/publishers/2/relationships/books"
                    }
                }
            },
            "links": {
                "self": "http://127.0.0.1:8000/publishers/2"
```

Here's a list of all Publishers in a nice JSON-API response.

1.9 Compound Documents

One of the nice features of JSON-API is the ability to include related resources into compound documents. `drf-jsonapi` makes it easy to support this feature and we've already configured our serializers to support these included relationships (that's what the `define_relationships` and `RelationshipHandlers` are for).

To try this out just add `?include=books` to the Publisher list url. This tells the API to include related books in the response. You should see two changes to the response. First, each Publisher resource now contains a collection of ResourceIdentifiers (basically just an ID and a type) for each related book.

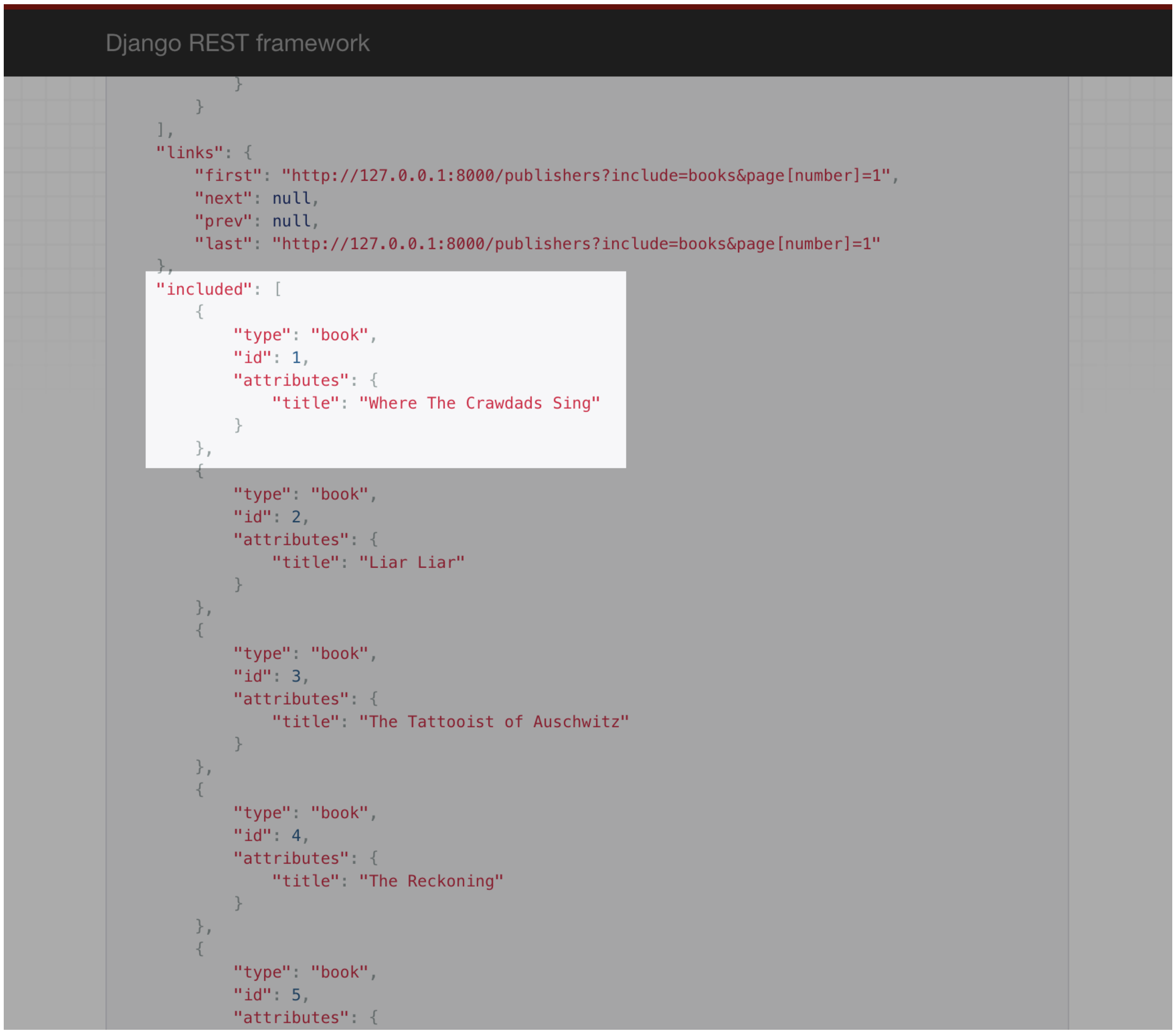


Publisher List

```
GET /publishers?include=books
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/vnd.api+json
Vary: Accept

{
    "data": [
        {
            "type": "publisher",
            "id": 1,
            "attributes": {
                "name": "Putnam"
            },
            "relationships": {
                "books": {
                    "links": {
                        "self": "http://127.0.0.1:8000/publishers/1/relationships/books"
                    },
                    "data": [
                        {
                            "type": "book",
                            "id": 1
                        }
                    ]
                }
            },
            "links": {
                "self": "http://127.0.0.1:8000/publishers/1"
            }
        },
        {
```

Also, each Book resource now appears in the `included` section of the of the response.



```
            }
        }
    ],
    "links": {
        "first": "http://127.0.0.1:8000/publishers?include=books&page[number]=1",
        "next": null,
        "prev": null,
        "last": "http://127.0.0.1:8000/publishers?include=books&page[number]=1"
    },
    "included": [
        {
            "type": "book",
            "id": 1,
            "attributes": {
                "title": "Where The Crawdads Sing"
            }
        },
        {
            "type": "book",
            "id": 2,
            "attributes": {
                "title": "Liar Liar"
            }
        },
        {
            "type": "book",
            "id": 3,
            "attributes": {
                "title": "The Tattooist of Auschwitz"
            }
        },
        {
            "type": "book",
            "id": 4,
            "attributes": {
                "title": "The Reckoning"
            }
        },
        {
            "type": "book",
            "id": 5,
            "attributes": {
```

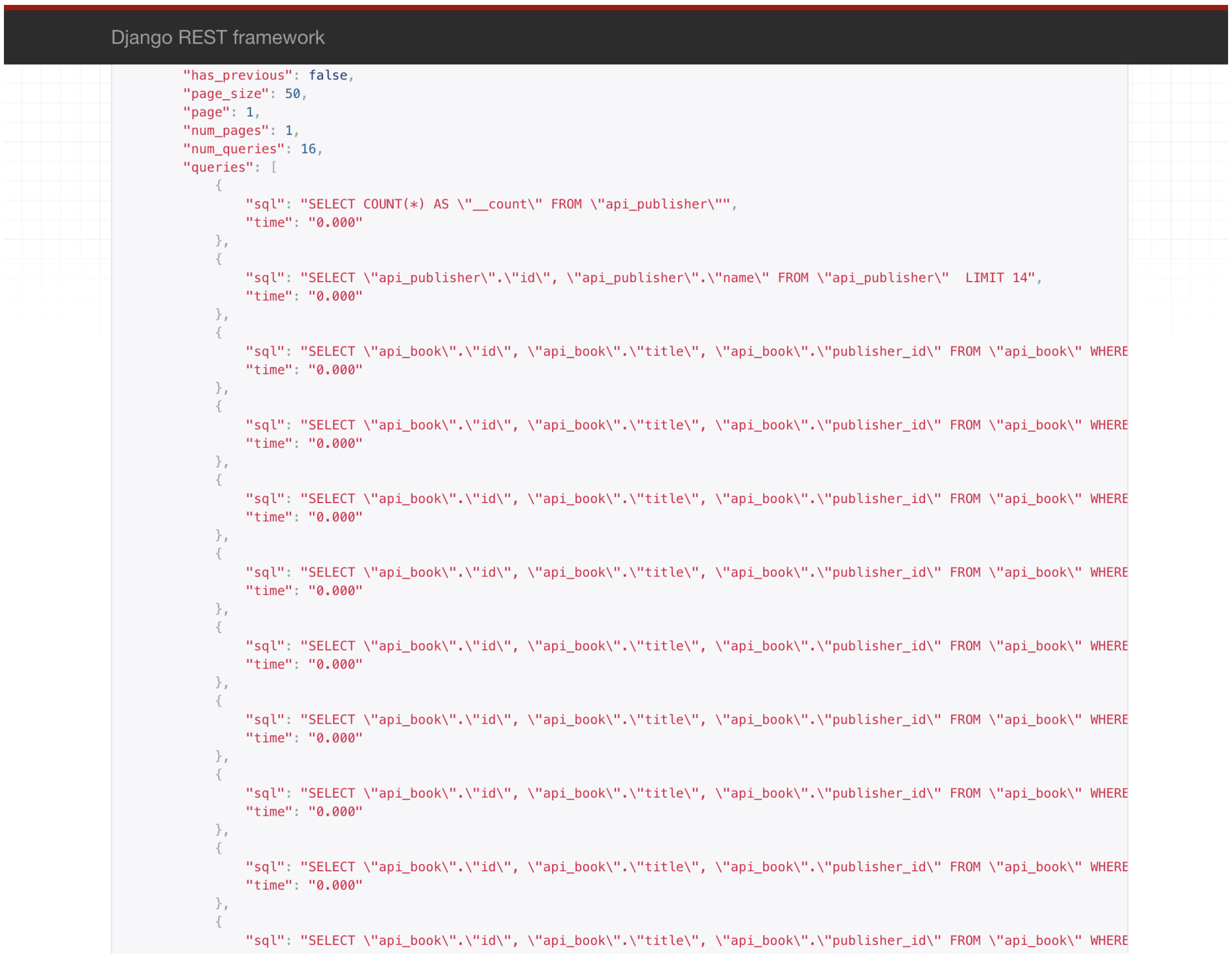
1.10 N+1 Query Problems

With great power comes great responsibility. As cool as it is to automatically include related books there is a problem with our implementation that we need to fix. To understand what that problem is we need to take a look at the database queries we're making to create our response.

Update `api/views.py` with the following changes:

```
from drf_jsonapi.mixins import DebugMixin

class PublisherViewSet(DebugMixin, ReadWriteViewSet):
...
```

The `DebugMixin` adds some additional metadata to responses. Reload `http://127.0.0.1/publishers` and scroll to the bottom. You should see this:



```
        "has_previous": false,
        "page_size": 50,
        "page": 1,
        "num_pages": 1,
        "num_queries": 16,
        "queries": [
            {
                "sql": "SELECT COUNT(*) AS \"__count\" FROM \"api_publisher\"",
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_publisher\".\"id\", \"api_publisher\".\"name\" FROM \"api_publisher\"  LIMIT 14",
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_book\".\"id\", \"api_book\".\"title\", \"api_book\".\"publisher_id\" FROM \"api_book\" WHERE
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_book\".\"id\", \"api_book\".\"title\", \"api_book\".\"publisher_id\" FROM \"api_book\" WHERE
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_book\".\"id\", \"api_book\".\"title\", \"api_book\".\"publisher_id\" FROM \"api_book\" WHERE
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_book\".\"id\", \"api_book\".\"title\", \"api_book\".\"publisher_id\" FROM \"api_book\" WHERE
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_book\".\"id\", \"api_book\".\"title\", \"api_book\".\"publisher_id\" FROM \"api_book\" WHERE
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_book\".\"id\", \"api_book\".\"title\", \"api_book\".\"publisher_id\" FROM \"api_book\" WHERE
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_book\".\"id\", \"api_book\".\"title\", \"api_book\".\"publisher_id\" FROM \"api_book\" WHERE
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_book\".\"id\", \"api_book\".\"title\", \"api_book\".\"publisher_id\" FROM \"api_book\" WHERE
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_book\".\"id\", \"api_book\".\"title\", \"api_book\".\"publisher_id\" FROM \"api_book\" WHERE
```

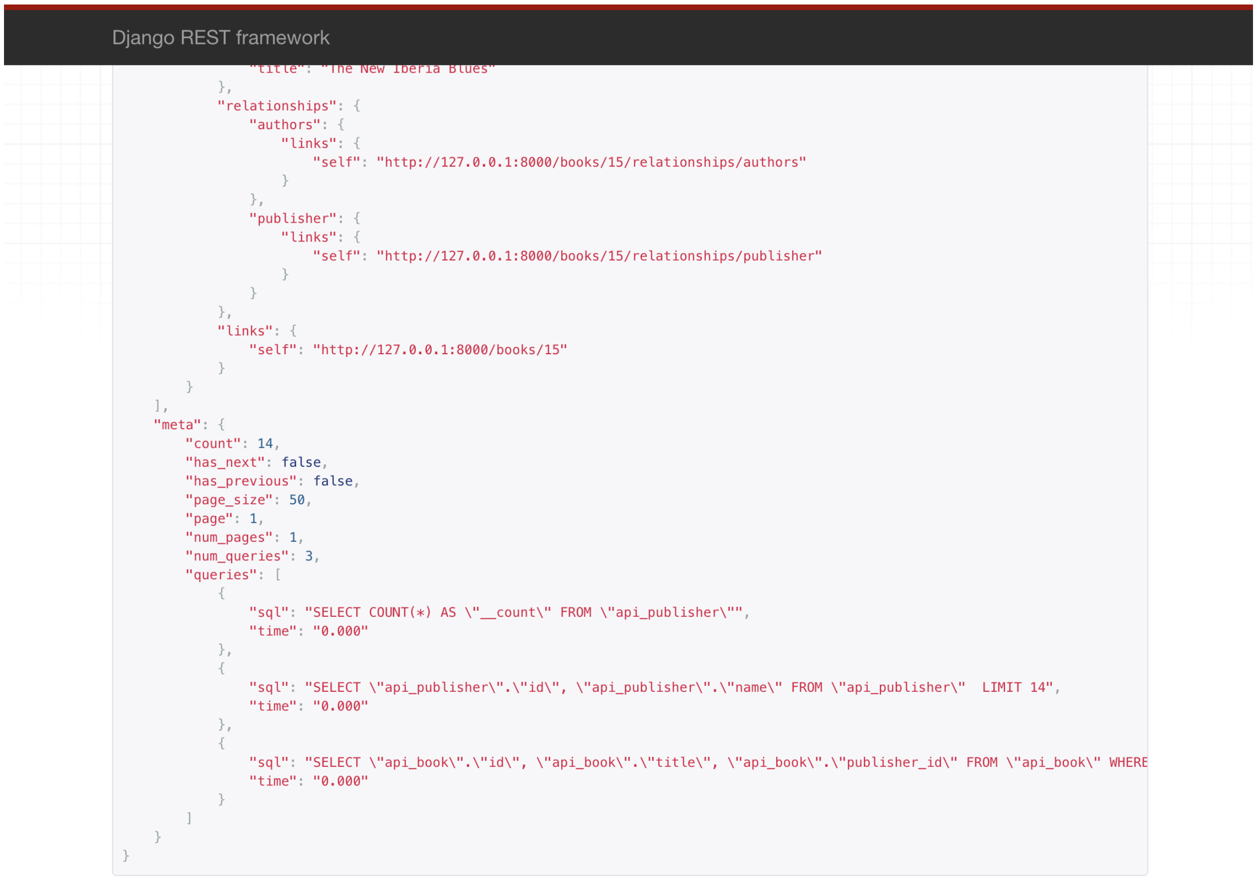
Ouch! That's a lot of queries. We're executing a separate query to fetch the related books for every publisher. This is the classic N+1 Query Problem and it can cause major performance problems in a production app.

We can solve this problem by eagerly loading all the related books in advance if we know we are going to need them. For example:

```python
class PublisherViewSet(DebugMixin, ReadWriteViewSet):
    serializer_class = PublisherSerializer

    def get_collection(self, request):
        collection = Publisher.objects.all()

        if "books" in request.include:
            collection = collection.prefetch_related("books")

        return collection
```

First off, we're replacing the `collection` class attribute with a method called `get_collection` which accepts the current request as an argument. This will allow us to check which relationships are to be included so we can modify our collection `QuerySet` to prefetch any related books.

Reload your page and you should see way fewer database queries.



Django REST framework

```
                "title": "The New Iberia Blues"
            },
            "relationships": {
                "authors": {
                    "links": {
                        "self": "http://127.0.0.1:8000/books/15/relationships/authors"
                    }
                },
                "publisher": {
                    "links": {
                        "self": "http://127.0.0.1:8000/books/15/relationships/publisher"
                    }
                }
            },
            "links": {
                "self": "http://127.0.0.1:8000/books/15"
            }
        }
    ],
    "meta": {
        "count": 14,
        "has_next": false,
        "has_previous": false,
        "page_size": 50,
        "page": 1,
        "num_pages": 1,
        "num_queries": 3,
        "queries": [
            {
                "sql": "SELECT COUNT(*) AS \"__count\" FROM \"api_publisher\"",
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_publisher\".\"id\", \"api_publisher\".\"name\" FROM \"api_publisher\"  LIMIT 14",
                "time": "0.000"
            },
            {
                "sql": "SELECT \"api_book\".\"id\", \"api_book\".\"title\", \"api_book\".\"publisher_id\" FROM \"api_book\" WHERE
                "time": "0.000"
            }
        ]
    }
}
```

Ah! Much better!

CHAPTER 2

Indices and tables

- genindex
- modindex
- search