Sunscrapers Guide to the Django REST Framework

sunscrapers

Table of Contents

ABOUT THE AUTHOR

Dominik has been fascinated with computers throughout his entire life. His two passions are coding and teaching - he is a programmer AND a teacher. He specializes mostly in backend development and training junior devs. He chose to work with Sunscrapers because the company profoundly supports the open-source community. In his free time, Dominik is an avid gamer.

Dominik Kozaczko

Backend Engineer

INTRODUCTION

When joining a project that uses the Django REST Framework (DRF), I often encountered problems like spaghetti code or antipatterns. These issues kept on repeating, and at some point, I began to wonder where the problem comes from.

The DRF documentation is comprehensive and well-organized, so it should help developers write better code.

Let's imagine a developer who has a project which is more or less ready. Now they want to add a REST API to it. Usually, their first stop will be the official Django REST Framework tutorial.

That's when it dawned on me:

The DRF tutorial is written in reverse order. And that's where the problem comes from.

Have a look at it. You'll see that the tutorial shows low-level versatility first, instead of explaining high-level acronyms instead. When we read the tutorial, we first learn about the details of views, serializers, and, only at the very end, about ViewSets - a wonderfully compact way for binding everything into a neat, transparent and manageable whole.

But the majority of developers never get to this point.

By then, they already have a relatively functional API and, most of the time, decide to abandon the tutorial in favor of the API Guide, searching for ways to implement the project requirements.

That's why I decided to write my own guide to Django REST Framework.

Over the following seven chapters, I take you through DRF step by step, from the general to detailed overview of its different aspects.

Read this ebook, and you're guaranteed to get clear and highly manageable code that won't bring you shame when you show or transfer it to others.

Note: Here's a link to the repository where you'll find all the code I'm going to discuss in this ebook.

CHAPTER 1: CRUD

If you're reading this, you probably already have the application wireframe and now want to add a REST API to enable basic operations on objects such as Create, Retrieve, Update and Delete (CRUD).

That's what we're going to do in this chapter.

To start, let's prepare an example application: we're going to build an app for managing items that we lend to our friends.

We need to install the Django REST Framework.

```
# cd your project's directory and activate its virtualenv
$ ./manage.py startapp rental
$ pip install djangorestframework
```

Next, we modify the INSTALLED_APPS parameter in settings.py file.

settings.py

```
INSTALLED_APPS = [
# previous apps

'rental',
'rest_framework',
]
```

rental/models.py

```
from django.db import models

class Friend(models.Model):
    name = models.CharField(max_length=100)

class Belonging(models.Model):
    name = models.CharField(max_length=100)

class Borrowed(models.Model):
    what = models.ForeignKey(Belonging, on_delete=models.CASCADE)
    to_who = models.ForeignKey(Friend, on_delete=models.CASCADE)
    when = models.DateTimeField(auto_now_add=True)
    returned = models.DateTimeField(null=True, blank=True)
```

To make our objects available through the API, we need to perform serialization to reflect the data contained in the object textually. The default format here is JSON, although DRF allows serialization to XML or YAML. The reverse process is called deserialization.

rental/serializers.py

```
from rest_framework import serializers
from . import models

class FriendSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.Friend
        fields = ('id', 'name')

class BelongingSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.Belonging
        fields = ('id', 'name')

class BorrowedSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.Borrowed
        fields = ('id', 'what', 'to_who', 'when', 'returned')
```

Now we need to create views that are going to handle the operations we want to perform on our objects.

Consider their names and possible ways of combining them with methods available in the HTTP standard:

Create - this one is rather straightforward. Standard support for it comes from the HTTP POST method. Since we're creating a set element here (the object's ID will be only determined now), we treat this method as an operation on the list: creating an element.

Retrieve - we have two options here: we can download a list of objects of a given type (list) or one specific object (retrieve). In both cases, GET will be the adequate HTTP method.

Update - two HTTP methods are available here: PUT and PATCH. The difference between them is that, according to its definition, PUT requires all attributes of the object - including those that haven't changed. PATCH, on the other hand, allows entering only those fields that have changed. That's why it's more popular. Using the PUT or PATCH method to update multiple objects is rare and DRF only supports updating single object in its default CRUD.

Delete - this deletes one or many objects. The HTTP method here will be DELETE. In practice, for security reasons, it's usually not possible to remove several objects at the same time. DRF only supports this operation on single objects in its default CRUD.

Let's summarize all of the above:

Operation	HTTP method	Endpoint type
Create	POST	list
Retrieve many	GET	list
Retrieve one	GET	detail
Update	PUT / PATCH	detail
Delete	DELETE	detail

To support such a set of operations, DRF provides a handy tool called a ViewSet. It takes the idea behind the standard class-based views from Django to the next level. It packs the above set into one class and creates appropriate URL paths automatically.

Let's see how it works now.

To start, let's create a ViewSet that will support our models. DRF provides the ModelViewSet thanks to which we can reduce the required amount of code to a minimum:

rental/api_views.py

```
from rest_framework import viewsets
from . import models
from . import serializers

class FriendViewset(viewsets.ModelViewSet):
    queryset = models.Friend.objects.all()
    serializer_class = serializers.FriendSerializer

class BelongingViewset(viewset.ModelViewSet):
    queryset = models.Belonging.objects.all()
    serializer_class = serializers.BelongingSerializer
```

And now we can move on to the last part: connecting all of that to the URL tree of our project.

We can take advantage of a very convenient tool: routers. The DRF provides two of the most important classes that differ only in that one of them shows the API structure when downloading / (root), and the other doesn't.

Our viewsets will be hooked up as follows:

api.py (global, next to settings.py)

```
from rest_framework import routers
from rental import api_views as rental_views

router = routers.DefaultRouter()
router.register(r'friends', rental_views.FriedViewSet)
router.register(r'belongings', rental_views.BelongingViewSet)
router.register(r'borrowings', rental_views.BorrowedViewSet)
```

urls.py (global)

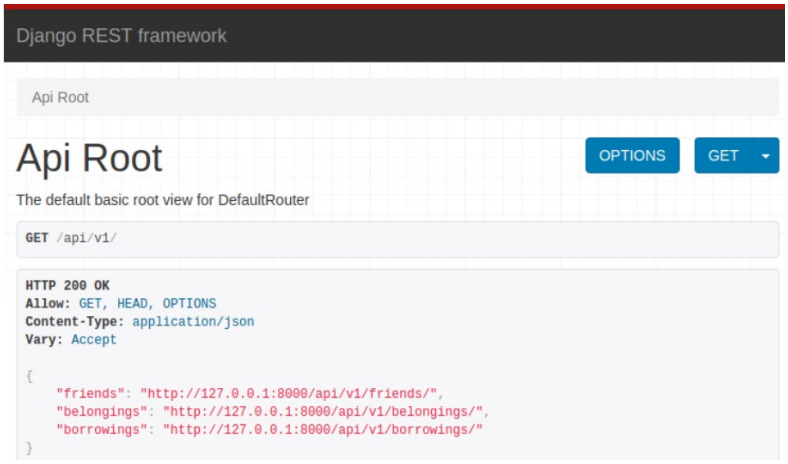
```
from django.urls import include, path
from django.contrib import admin
from .api import router

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include(router.urls)),
]
```

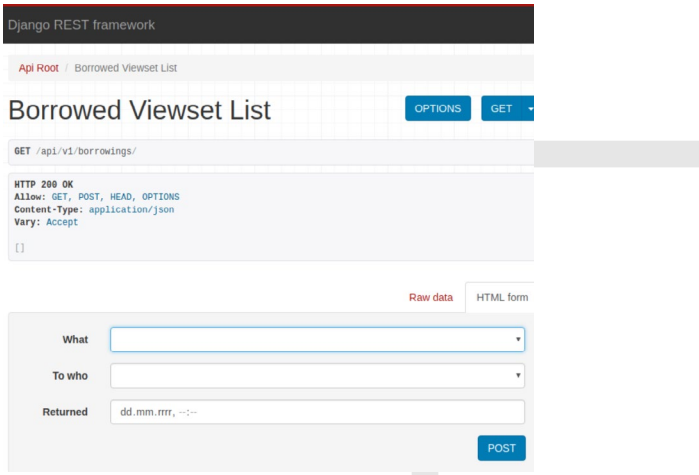
Let's test the API we created.

$ ./manage.py makemigrations
$ ./manage.py migrate
$ ./manage.py runserver

Open this address in your browser (in standard configuration): https://127.0.0.1:8000/api/v1/

DRF automatically creates views that allow performing API queries directly from the browser:





Experiment and check the effects of your work in the django-admin panel.

That's how we get an API that supports CRUD for our models. Note that we don't have any security against unauthorized access in place yet.

In the next chapter, I'm going to take a closer look at user login and registration process.

CHAPTER 2: Login and Authentication

In previous chapter I showed you how to prepare an API that implements basic CRUD on objects. This time, I'm going to take a closer look at logging into the API and regulating permissions.

We can distinguish between two dominant groups among REST API use cases: (1) single-page applications (SPA) that take advantage of browser's capabilities, and (2) mobile applications.

For the former, all we need is a standard session support mechanism provided by Django and supported by DRF by default. Unfortunately, we can't use this mechanism in mobile applications where it's much more common to log in with a token. Here's how it goes: when running the application, we provide login details, the application connects to the API that generates a token which is then saved. That way, users don't have to remember the login and password - or have the device remember this data and expose app users to risk.

DRF provides a token authentication mechanism, and you can read about it in the official documentation. The description is too detailed for our purposes, but it's a good idea to have a look at it once you finish reading my guide.

In what follows, we're going to use `djoser` library.

```
$ pip install djoser
```

Let's start with some basic configuration:

settings.py

```
INSTALLED_APPS = [
...
    'djoser',
    'rest_framework.authtoken',
]

REST_FRAMEWORK = {
...
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.TokenAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    ),
}
```

We can also add the token login to our URLs:

urls.py (global)

```
urlpatterns = [
    path('admin/', admin.site.urls),
```

```
    path('api/v1/', include(router.urls)),
    path('api/auth/', include('djoser.urls.authtoken'),
]
```

To finish, we complete the migration related to tokens:

```
$ ./manage.py migrate
```

From this moment, we can get the login token with the help of the REST API:

```
$ curl -X POST -d '{"username": "admin","password": "top_secret"}' -H 'Content-Type: application/json'
http://127.0.0.1:8000/api/auth/token/login/
```

You'll get something like this in response:

```
{"auth_token":"fe9a080cf91acb8ed1891e6548f2ace3c66a109f"}
```

Let's secure our views from unauthorized access now.

DRF offers several permission classes we can use to protect our API against unauthorized access.

The default permission is 'rest_framework.permissions.AllowAny' which - as the name suggests - allows everyone to do everything.

Let's protect the API to allow access only to logged-in users.

To do this, we need to modify settings.py by adding the following entry:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
}
```

Let's check whether our API is protected:

```
$ curl -X GET http://127.0.0.1:8000/api/v1/friends/

{"detail":"Authentication credentials were not provided."}
```

It works! Without a token, all we got was an error. Let's use the token we got earlier.

```
$ curl -X GET http://127.0.0.1:8000/api/v1/friends/ -H 'Authorization: Token fe9a080cf91acb8ed1891e6548f2ace3c66a109f'

[{"id":1,"name":"John Doe"}]
```

NOTE: For security reasons, it's critical that production API is made available only through https.

However, setting a default class that manages permissions for the entire API isn't the only option. We can also set different ways to handle permissions individually for each ViewSet by setting the permission_classes attribute:

```
class MyViewSet(viewsets.ModelViewSet):
    permission_classes = [permissions.DjangoModelPermissions]
```

Have a look at DRF documentation to learn more about default permissions classes.

Determining permissions is based on request analysis and returning bool value (True / False).

Let's follow the example I showed you in the previous chapter: the application that helps to manage items we lend to our friends.

What happens if our friends get interested in our app and would like to use it as well? To manage that, we need to enable user registration. But first, we have to prepare permissions to ensure that only item owners can modify the said items.

We can carry out user registration in many ways and I will leave out this stage, so that you can figure it out on your own. Hint: It's a good idea to use djoser or rest_auth libraries.

Let's add some information about the item owners to our models. The most convenient way is creating a mixin or an abstract model:

models.py

```
class OwnedModel(models.Model):
    owner = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)

    class Meta:
        abstract = True


class Belonging(OwnedModel):
    name = models.CharField(max_length=100)

# continue with other models
```

Remember to migrate the database afterwards:

```
$ ./manage.py makemigrations
$ ./manage.py migrate
```

It's time to prepare the permission checking class.

To implement it, we can use one of the following methods (or both): 'has_permission' and 'has_object_permission'.

General permissions are always checked and the object is only checked once the general permissions are accepted. These methods must return True if permission has been granted and False if it hasn't. The default value returned by both methods is True. If we use several permission validation classes in the view, all of them must pass the test successfully (the results are combined using AND).

Our view will check the permissions for an item, so we need to implement the permissions as follows:

permissions.py

```
from rest_framework import permissions
class IsOwner(permissions.BasePermission):
   message = "Not an owner."

   def has_object_permission(self, request, view, obj):
       return request.user == obj.owner
```

The message attribute allows setting a personalized error message when permission isn't granted to a user.

If we would like to allow all users to see our app's content, then the class will look like this:

```
class IsOwner(permissions.BasePermission):
    message = "Not an owner."

    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True
        return request.user == obj.owner
```

permissions.SAFE_METHODS contains a list of HTTP methods that don't write, i.e. GET, OPTION, and HEAD.

The next step is remembering the logged-in user as the owner of the newly created resource. DRF provides us with a useful method here. Note: That method is discussed in the documentation under a completely different topic.

serializers.py

```
class FriendSerializer(serializers.ModelSerializer):
    owner = serializers.HiddenField(
        default=serializers.CurrentUserDefault()
    )

    class Meta:
        model = models.Friend
        fields = ('id', 'name')

# and so on for other serizalizers
```

To finish, let's use our new permissions class in ViewSets:

```
from .permissions import IsOwner

class FriendViewset(viewsets.ModelViewSet):
    queryset = models.Friend.objects.all()
    serializer_class = serializers.FriendSerializer
    permission_classes = [IsOwner]

# and so on with other ViewSets
```

Note that creating an object only checks the 'has_permission' permission, so you may want to limit it to the logged-in users. To do that, import 'rest_framework.permissions.IsAuthenticated' and add it to the 'permission_classes' attribute. Remember - both must return True for the permission to be granted.

DRF provides us with basic permission classes, and their names speak for themselves: AllowAny,

- IsAuthenticated,
- IsAuthenticatedOrReadOnly,
- IsAdminUser (the is_staff attribute is checked here, not is_superuser!),
- DjangoModelPermissions,
- DjangoModelPermissionsOrAnonReadOnly,
- DjangoObjectPermissions.

The last three ones are based on Django and I think they're particularly interesting. While the first two implement standard model permissions, the last one requires using a library like django-guardian.

You can read more about this in DRF documentation. You can also use other libraries but I prefer DRY REST permissions because it follows the 'fat models' principle recommended by Django creators.

In the next chapter, I'm going to talk about additional information in serializers - in particular, in dynamically-generated fields.

CHAPTER 3: Custom Fields

In this chapter, I'm going to show you how to add dynamic fields to our serializer model.

Dynamic fields are the fields that aren't part of our database and their value is calculated on a regular basis. For example,
I use the amazing pendulum library for datetime service.

Let's assume that we want to show the information that a friend is due to give us back a borrowed item right next their name.

Here's what downloading this information into our data model will look like. Let's say I want to know if a Friend has been holding onto my items for more than two months.

```
friend = Friend.objects.get(id=1)
friend.borrowed_set.filter(returned__isnull=True,
when=pendulum.now().subtract(months=2)).exists()
```

This piece of code works great for a single friend. If we wanted to display this data for a group of people, we would inevitably flood the database with hundreds (or even thousands) of queries. And as Raymond Hettinger would say: There must be a better way!

And fortunately, there is. We can use the mechanism of annotation here. The following query will add the 'ann_overdue' field to all queryset elements.

Note: You'll see why I'm not using 'has_overdue' in a minute.

```
Friend.objects.annotate(
   ann_overdue=models.Case(
       models.When(borrowed__returned__isnull=True,
                   borrowed__when__lte=pendulum.now().subtract(months=2),
                   then=True),
       default=models.Value(False),
       output_field=models.BooleanField()
   )
)
```

Let's analyze this in detail. I use the Case function which takes any number of When functions as a parameter - in our case, one is enough. Inside **When,** I enter a condition - **if** any of Borrowed objects related to that Friend has empty (NULL) 'returned' field and also its 'when' field contains a date that is at least 'two months ago', **then** the value will be True. The **default** value is False, and the result is to be mapped to the BooleanField field. That's simple, right? ;)

You can find out more about conditional expressions in querysets in the Django documentation.

Now here's another question: where do we put this code?

Usually, if we follow the concept of 'fat models' we'd have to put the entire logic inside the model. However, that's not always possible. Our models (or at least a part of them) can come from third-party apps.
In our case, we have full control over the models, so I will follow this path. I'll still point out where we need to change something if dealing with a different situation:

models.py

```
class FriendQuerySet(models.QuerySet):
    def with_overdue(self):
        return self.annotate(
            ann_overdue=models.Case(
                models.When(borrowed__when__lte=pendulum.now().subtract(months=2),
                            then=True),
                default=models.Value(False),
                output_field=models.BooleanField()
            )
        )


class Friend(OwnedModel):
    name = models.CharField(max_length=100)

    objects = FriendQuerySet.as_manager()

    @property
    def has_overdue(self):
        if hasattr(self, 'ann_overdue'):  # in case we deal with annotated object
            return self.ann_overdue
        return self.borrowed_set.filter(  # 1
            returned__isnull=True, when=pendulum.now().subtract(months=2)
        ).exists()
```

views.py

```
class FriendViewset(viewsets.ModelViewSet):
    queryset = models.Friend.objects.with_overdue()
    serializer_class = serializers.FriendSerializer
    # and so on...
```

serializers.py

```
class FriendSerializer(serializers.ModelSerializer):
    owner = serializers.HiddenField(
        default=serializers.CurrentUserDefault()
    )

    class Meta:
        model = models.Friend
        fields = ('id', 'name', 'has_overdue')
```

If we can't modify models, the view would need to have its 'get_queryset' method overwritten and serializers.py would have the 'has_overdue' logic:

views.py

```
class FriendViewset(viewsets.ModelViewSet):
    queryset = models.Friend.objects.all()
    serializer_class = serializers.FriendSerializer
    # and so on…

    def get_queryset(self):
        return super().get_queryset().annotate(
            ann_overdue=models.Case(
                models.When(borrowed__when__lte=pendulum.now().subtract(months=2),
                            then=True),
                default=models.Value(False),
                output_field=models.BooleanField()
            )
        )
```

serializers.py

```
class FriendSerializer(serializers.ModelSerializer):
    owner = serializers.HiddenField(
        default=serializers.CurrentUserDefault()
    )
    has_overdue = serializers.SerializerMethodField()

    class Meta:
        model = models.Friend
        fields = ('id', 'name', 'has_overdue')

    def get_has_overdue(self, obj):
        if hasattr(obj, 'ann_overdue'):
            return obj.ann_overdue
        return obj.borrowed_set.filter(
            returned__isnull=True, when=pendulum.now().subtract(months=2)
        ).exists()
```

As you can see, I define a 'has_overdue' field (or property), put it in the serializer's Meta.fields attribute and, if required, explain to DRF how to get the value.

Notice how I check for 'ann_overdue' attribute. That way, I get very universal code - if the annotation has been used, the value has already been calculated and we can re-use it. If it wasn't - well, we need to do the heavy lifting ourselves.

Now how does this change the database hit count? I've prepared a sample dataset that contains 1000 Friends and 10000 Belongings (items) distributed randomly within a six-month-long period. Then I carried our a test (I removed print results for clarity):

```
/Side note: I'm using ipython to make my ./manage.py shell even more awesome./
```

```
In [1]: from django.db import connection

In [2]: from core.models import Friend
In [3]: for f in Friend.objects.all():
   ...:     print(f.has_overdue)
   ...:

In [4]: len(connection.queries)
Out[4]: 1000

# another try using with_overdue method

In [1]: from django.db import connection

In [2]: from core.models import Friend

In [3]: for f in Friend.objects.with_overdue():
   ...:     print(f.has_overdue)
   ...:

In [4]: len(connection.queries)
Out[4]: 1
```

So there you have it. For 'fat models,' you just need to put the method or property name in the serializer's Meta.fields attribute. An extra benefit of this is that the manager and 'has_overdue' property can also be used in admin panel so you get two serious improvements inside one slim package.

If your models come from a third-party app, you should put the required logic inside the serializer as it can be used by more than one view. Unfortunately, in our case the logic needs to operate (annotate) the queryset, so the logic lands inside a view. But if it needed to be reused in other views, we could make a mixin just for the 'get_queryset' method.

Also, remember to always anticipate the impact on the database and consider using 'select_related' and/or 'prefetch_related' when getting the queryset.

In the next chapter I'll take a closer look at pagination.

CHAPTER 4: Pagination

In this chapter, I'm going to deal with yet another interesting topic: pagination.

Why pagination?

Have a look at the standard answer at one of our endpoints.

```
$ curl http://127.0.0.1:8000/api/v1/friends/
[{“id”:1,“name”:“John Doe”,“has_overdue”:true},{“id”:2,“name”:“Frank Tester”,“has_overdue”:false}]
```

We got a standard list of items. So far, so good.

But what if our endpoint returns thousands of items? The serialization and transmission of such a data volume may take long enough for the user to notice the app's lagging.

We can solve this problem by using pagination; the division of results into pages of fixed size. The best strategy here is to use 'limit + offset' method, where the parameter 'limit' passed in GET specifies the number of elements per page, and 'offset' determines the offset in relation to the beginning of the list.

That's the universal way to handle the more traditional transition between subpages, as well as the sometimes desirable method called 'infinite scroll.'

The documentation recommends adding the following entries to the REST_FRAMEWORK settings in settings.py:

```
REST_FRAMEWORK = {
     ....
‘DEFAULT_PAGINATION_CLASS’: ‘rest_framework.pagination.LimitOffsetPagination’,
‘PAGE_SIZE’: 100,
...
}
```

That results in the following (I used the limit parameter and formatted the result for readability using json_pp):

```
$ curl http://127.0.0.1:8000/api/v1/friends/?limit=1 | json_pp
{
   “previous” : null,
   “next” : “http://127.0.0.1:8000/api/v1/friends/?limit=1&offset=1”,
   “count” : 2,
   “results” : [
     {
             “id” : 1,
```

```
            "has_overdue" : true,
            "name" : "John Doe"
        }
    ]
}
```

As you can see, the results were enveloped. This practice is justified when the client can't handle HTTP headers, but it's slowly becoming obsolete today.

The most recent guides to best practices recommend the transmission of metadata in headlines while allowing enveloping on demand. We will implement this solution below.

To do that, we'll follow these assumptions:

- The endpoint must return a list of items in the same structure as initially.
- Pagination is carried out using the 'limit' and 'offset' parameters.
- Additional metadata is included in the appropriate headers.
- The code needs to use enveloping on demand (provided by the parameter).
- The code always contains links in the headers - even if enveloping has been chosen.

Note: There already exists a [django-rest-framework-link-header-pagination] library, but it doesn't implement the limit / offset mechanism which is of interest to us here.

The simplest solution will be inheriting the class rest_framework.pagination.LimitOffsetPagination because we have most of the logic implemented there.

To begin, let's handle the parameter that turns enveloping on:

```
from collections import OrderedDict

from rest_framework.pagination import LimitOffsetPagination
from rest_framework.response import Response
from rest_framework.utils.urls import replace_query_param, remove_query_param


class HeaderLimitOffsetPagination(LimitOffsetPagination):
    def paginate_queryset(self, queryset, request, view=None):
        self.use_envelope = False
        if str(request.GET.get('envelope')).lower() in ['true', '1']:
            self.use_envelope = True
            return super().paginate_queryset(queryset, request, view)
```

We can later write a method that returns data:

```
def get_paginated_response(self, data):
    next_url = self.get_next_link()
    previous_url = self.get_previous_link()
```

```
    links = []
    for url, label in (
        (previous_url, 'prev'),
        (next_url, 'next'),
    ):
        if url is not None:
            links.append('<{}>; rel="{}"'.format(url, label))

       headers = {'Link': ',' .join(links)} if links else {}

       if self.use_envelope:
            return Response(OrderedDict([  # here
                ('count', self.count),
                ('next', self.get_next_link()),
                ('previous', self.get_previous_link()),
                ('results', data)
            ]), headers=headers)
     return Response(data, headers=headers)
```

To make it all work in line with best practices, we only need links to the first and last page.

Let's add these two methods:

```
def get_first_link(self):
   if self.offset <= 0:
       return None
   url = self.request.build_absolute_uri()
   return remove_query_param(url, self.offset_query_param)

def get_last_link(self):
   if self.offset + self.limit >= self.count:
       return None
   url = self.request.build_absolute_uri()
   url = replace_query_param(url, self.limit_query_param, self.limit)
   offset = self.count - self.limit
   return replace_query_param(url, self.offset_query_param, offset)
```

All that remains is completing the 'get_paginated response' method like this:

```
def get_paginated_response(self data):
   next_url = self.get_next_link()
   previous_url = self.get_previous_link()
   first_url = self.get_first_link()
   last_url = self.get_last_link()

   links = []
   for label, url in (
           ('first', first_url),
           ('next', next_url),
           ('previous', previous_url),
           ('last', last_url),
```

```
    ):
        if url is not None:
            links.append('<{}>; rel="{}"'.format(url, label))

    headers = {'Link': ',' .join(links)} if links else {}

    if self.use_envelope:
        return Response(OrderedDict([
            ('count', self.count),
            ('first', first_url),
            ('next', next_url),
            ('previous', previous_url),
            ('last', last_url),
            ('results', data)
        ]), headers=headers)
    return Response(data, headers=headers)
```

Where to put all that code?

The best place to put this code is a separate file that can be easily imported from anywhere in the project.

Let's assume that we create a 'pagination.py' file containing the above class in our book rental application. We will change the REST_FRAMEWORK configuration to this:

```
REST_FRAMEWORK = {
…
    'DEFAULT_PAGINATION_CLASS': 'rental.pagination.HeaderLimitOffsetPagination',
    'PAGE_SIZE': 100,
}
```

You can also use the library I prepared with the code above by installing 'pip install hedju' and later as DEFAULT_PAGINATION_CLASS you can use 'hedju.HeaderLimitOffsetPagination'.

Since everything is ready, all that's left is API testing; curl with the -v parameter will show us headers (I've removed irrelevant information):

```
$ curl "http://127.0.0.1:8000/api/v1/friends/?limit=1" -v
*   Trying 127.0.0.1...
...
< Content-Type: application/json
<   Link:   <http://127.0.0.1:8000/api/v1/friends/?limit=1&offset=1>;   rel="next",   <http://127.0.0.1:8000/api/v1/
friends/?limit=1&offset=1>; rel="last"

[{"id":1,"name":"John Doe","has_overdue":true}]

$ curl "http://127.0.0.1:8000/api/v1/friends/?limit=1&envelope=true"

{
   "last" : "http://127.0.0.1:8000/api/v1/friends/?envelope=true&limit=1&offset=1",
   "next" : "http://127.0.0.1:8000/api/v1/friends/?envelope=true&limit=1&offset=1",
```

```
    "first" : null,
    "results" : [
      {
              "id" : 1,
              "has_overdue" : true,
              "name" : "John Doe"
      }
    ],
    "previous" : null,
    "count" : 2
  }
```

Done!

As a bonus, I'd like to mention the support for navigation through headers in the requests library:

```
In [1]: import requests

In [2]: result = requests.get('http://127.0.0.1:8000/api/v1/friends/?limit=1')

In [3]: result.links
Out[3]:
{'next': {'url': 'http://127.0.0.1:8000/api/v1/friends/?limit=1&offset=1',
  'rel': 'next'},
 'last': {'url': 'http://127.0.0.1:8000/api/v1/friends/?limit=1&offset=1',
  'rel': 'last'}}
```

That's all, folks! In the next chapter, I'm going to discuss the subject of filtering the data list.

CHAPTER 5: Filtering

Ready to start working on filtering? Let's jump in!

In the previous chapter, we limited the volume of simultaneously downloaded data by pagination. This time, let's think about how we can filter and search our resources easily.

The rental list endpoint (/api/v1/borrowed/) displays all items regardless of whether they've been returned or not.

It makes sense to filter this list according to the field 'returned'. By doing that, we can check which items haven't been returned yet. The parameter specifying such filtering will be transmitted via GET, e.g.

```
$ curl http://127.0.0.1:8000/api/v1/borrowings/?missing=true
```

We can complete this task relatively simply in the `get_queryset` method.

```
class BorrowedViewset(viewsets.ModelViewSet):
    queryset = models.Borrowed.objects.all()
    serializer_class = serializers.BorrowedSerializer
    permission_classes = [IsOwner]

    def get_queryset(self):
        qs = super().get_queryset()
        only_missing = str(self.request.query_params.get('missing')).lower()
        if only_missing in ['true', '1']:
            return qs.filter(returned__isnull=True)
        return qs
```

BTW. Let me remind you that the 'returned' field is a date field. If it contains NULL, it means that the item hasn't been returned yet. That's why we use filtering here.

Such an implementation is sufficient for simple use-cases. But with more variables that we may want to filter, it can quickly become a mess that is difficult to manage. There must be a better way of handling this, right?

Fortunately, there is. Django REST Framework allows developers to use the django-filter library which drastically simplifies the definition and management of filters.

First, we need to install the library using pip:

```
$ pip install django-filter  # Note the lack of "s" at the end!
```

Then we update our settings:

settings.py

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': ('django_filters.rest_framework.DjangoFilterBackend',)
    ….v
}
```

The easiest way to complete the task would be adding the fields by which we want to filter to the attribute 'filterset_fields' in the appropriate view, e.g.

```
class BorrowedViewset(viewsets.ModelViewSet):
   queryset = models.Borrowed.objects.all()
   serializer_class = serializers.BorrowedSerializer
   permission_classes = [IsOwner]
   filterset_fields = ('to_who', )  # here
```

This allows us to filer by the person who borrowed an item from us.

```
$ curl http://127.0.0.1:8000/api/v1/borrowings/?to_who=1

[{"id":1,"what":1,"to_who":1,"when":"2018-01-01T12:00:00Z","returned":null},{"id":3,"what":1,"to_
who":1,"when":"2019-04-17T06:35:22.000236Z","returned":null},{"id":4,"what":1,"to_who":1,"when":"2019-04-17T06:35:
36.546848Z","returned":null}]
```

Unfortunately, this method has one very serious limitation: we can only give specific values except for NULL.

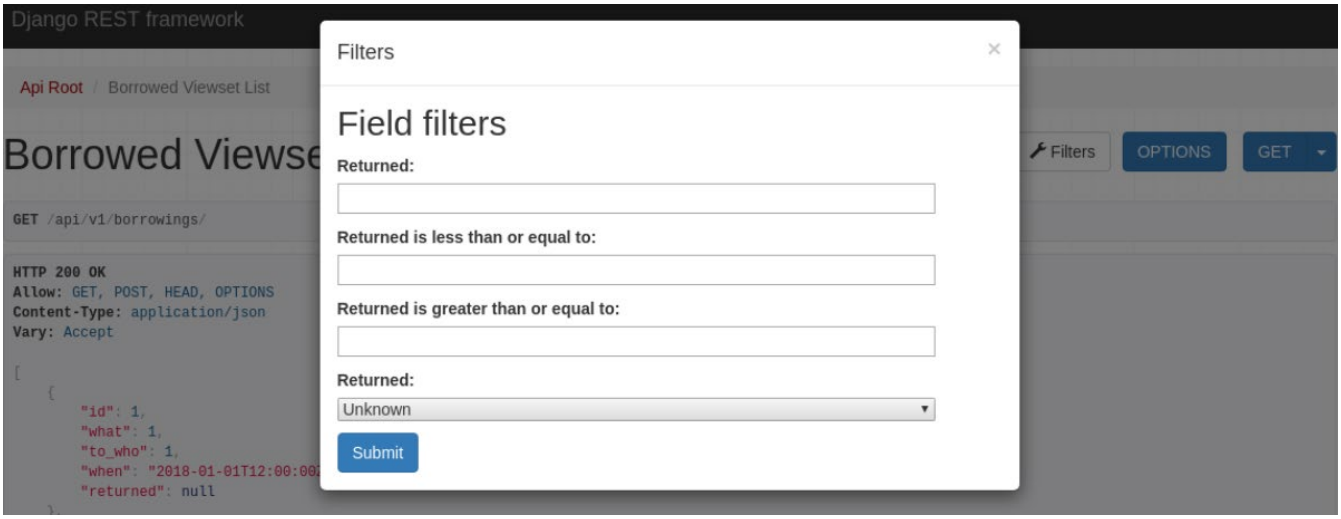
A partial solution to this problem is replacing the filterset_fields with a dictionary. The keys are field names, and the value is a list of acceptable subfilters compatible with the Django notation, e.g.:

```
class BorrowedViewset(viewsets.ModelViewSet):
   queryset = models.Borrowed.objects.all()
   serializer_class = serializers.BorrowedSerializer
   permission_classes = [IsOwner]
   filterset_fields = {
       'returned': ['exact', 'lte', 'gte', 'isnull']
   }
```

This allows filtering the list of rental items by the return date, including some useful sub-filters like 'lte' and 'gte', as well as 'isnull' to display the backlog:

```
$ curl http://127.0.0.1:8000/api/v1/borrowings/?returned__isnull=True

[{"id":1,"what":1,"to_who":1,"when":"2018-01-01T12:00:00Z","returned":null},{"id":2,"what":2,"to_
who":2,"when":"2019-04-16T18:46:16.646649Z","returned":"2019-04-16T18:46:13Z"},{"id":3,"what":1,"to_
who":1,"when":"2019-04-17T06:35:22.000236Z","returned":null},{"id":4,"what":1,"to_who":1,"when":"2019-04-17T06:35:
36.546848Z","returned":null}]
```

Moreover, this solution allows developers to display filtering options when browsing the API in the HTML mode (note that the hints can sometimes be confusing).



Writing your own FilterSet definition

Everything I described above is nothing compared to what we can achieve by writing our own FilterSet definition.

Let's start with a simple example.

To see how it works, we're going to implement the previous functionality: searching for unreturned items.

But now we're going to do that in a way that doesn't reveal the Django notation underneath.

To achieve that, we need to use BooleanFilter field that will parse the transferred value. In its parameters, we define the field we want to view and the specific expression to which the value will be passed:

```
class BorrowedFilterSet(django_filters.FilterSet):
    missing = django_filters.BooleanFilter(field_name='returned', lookup_expr='isnull')

    class Meta:
        model = models.Borrowed
        fields = ['what', 'to_who', 'missing']
```

Then we pass our BorrowedFilterSet to the ViewSet:

```
class BorrowedViewset(viewsets.ModelViewSet):
    queryset = models.Borrowed.objects.all()
    serializer_class = serializers.BorrowedSerializer
    permission_classes = [IsOwner]
    filterset_class = BorrowedFilterSet  # here
```

We can now do this:

```
$ curl http://127.0.0.1:8000/api/v1/borrowings/?missing=True
```

The result should be the same as previously.

Creating a field that allows filtering of outdated rental items is only a bit more work. We will also create a BooleanFilter field, but this time we will pass it the name of the method (it can also be callable) which will perform the filtering on the passed QuerySet.

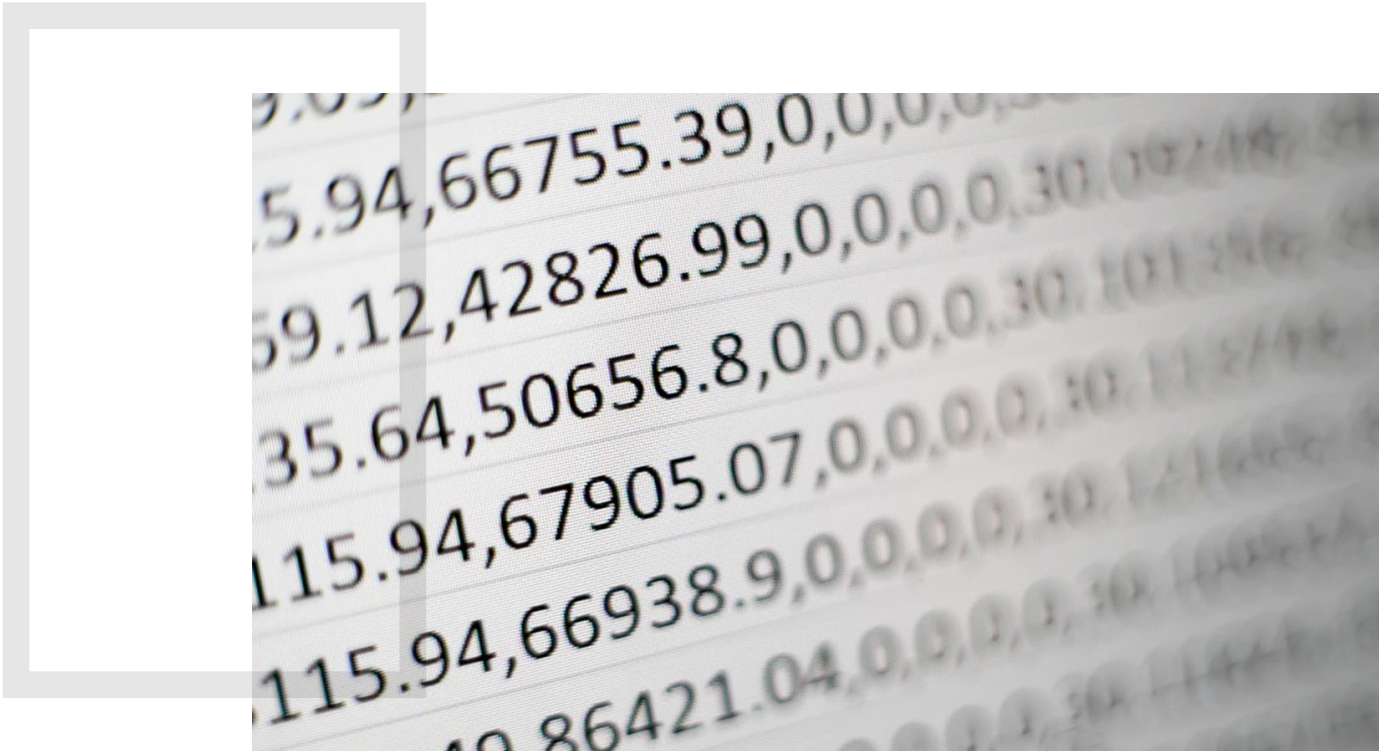
The entire thing may look like this (note that I omit part of the code from the previous example for clarity, so remember to add a field to Meta.fields):

```
class BorrowedFilterSet(django_filters.FilterSet):
   overdue = django_filters.BooleanFilter(method='get_overdue', field_name='returned')

   def get_overdue(self, queryset, field_name, value, ):
       if value:
           return queryset.filter(when__lte=pendulum.now().subtract(months=2))
       return queryset
```

As a homework assignment, try to simplify this fragment by adding the filtering of expired rental items to the QuerySet/Model Manager of the model, just like in the third chapter. You can find the solution in our repository.

Of course, there are more filter types available, and the easiest way is to explore them is by reading the documentation.

In the next chapter, I'm going to deal with functional endpoints and nesting.



CHAPTER 6: Functional Endpoints and API Nesting

In principle, functional endpoints (which in the Django Rest Framework are called actions) perform tasks that don't fall under CRUD - for example, sending a password reset request.

Throughout this guide, we've been building an application that allows managing loaned items. Today, we will build a functional endpoint that sends a reminder to people who borrowed our items.

Actions

Let's supplement our Friend model with a field that contains the email address:

We can complete this task relatively simply in the 'get_queryset' method.

```
# models.py

class Friend(OwnedModel):
   email = models.EmailField(default='')  # remember to add emails using django-admin
# rest of model's code
```

and update the database:

```
$ python manage.py makemigrations && python manage.py migrate
```

We can now proceed with completing the loan view. We decorate the functional endpoints with the @ action decorator. By the way, we can decide whether the action will apply to a single item or the entire list - if so, the path where the endpoint appears will change. We'll consider both cases below.

Let's look at the single item option first. We want to remind our friend about a specific loan by sending them an email. We want to do that after sending a POST order to the right endpoint. An example implementation looks like this:

```
from rest_framework.decorators import action

class BorrowedViewset(viewsets.ModelViewSet):
# rest of the code
@action(detail=True, url_path='remind', methods=['post'])
def remind_single(self, request, *args, **kwargs):
   obj = self.get_object()
   send_mail(
       subject=f"Please return my belonging: {obj.what.name}",
       message=f'You forgot to return my belonging: "{obj.what.name}" that you borrowed on {obj.when}. Please
return it.',
       from_email="me@example.com",  # your email here
       recipient_list=[obj.to_who.email],
```

```
        fail_silently=False
    )
    return Response("Email sent.")
```

The defined endpoint will appear under the path /api/v1/borrowings/{id}/remind/. Let's check if the email is sent correctly. If you don't want to configure the actual sending of emails, use the following configuration that comes in handy during experiments:

```
# settings.py

EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

We can now test our endpoint. In my local environment, the loan with id=1 hasn't been returned yet. Adjust the example to your specific case:

Our API is secure and we need to send authorized requests. Go back to the second chapter for more information about that.

```
$ curl -X POST http://127.0.0.1:8000/api/v1/borrowings/1/remind/  -H 'Authorization: Token
fe9a080cf91acb8ed1891e6548f2ace3c66a109f'
```

The result should be "Email sent." In the console where you have your server running, you should get a message similar to the following:

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Please return my belonging: Snatch
From: me@example.com
To: friend@example.com
Date: Thu, 25 Apr 2019 08:13:23 -0000
Message-ID: <155618000311.9173.14157612795944796688@mindstorm>

You forgot to return my belonging: "Snatch" that you borrowed on 2018-01-01 12:00:00+00:00. Please return it.
-------------------------------------------------------------------------------
[25/Apr/2019 08:13:23] "POST /api/v1/borrowings/1/remind/ HTTP/1.1" 200 13
```

The second case would be to send a reminder to all debtors at once. The endpoint should be available under this address: /api/v1/belongings/remind/ and the following decorator should do the trick:

```
@action(detail=False, url_path='remind', methods=['post'])
def remind_many(self, request, *args, **kwargs):
# ...
```

Implement this method as a homework assignment. The send_mail function will return 1 if the email has been sent, so you can easily count the number of sent emails and return it in the response.

Nesting

Let's now move to a slightly more advanced topic: nested endpoints. This functionality is not part of the Django Rest Framework, as some guides advise against using it. In my opinion, however, it's a very convenient way to filter out data.

Several libraries implement data nesting - you can find a list in DRF documentation. Personally, I like drf-extensions most because it's a compact package containing many useful things, not just nesting.

In the example below, we'll make a list of all the items borrowed by a specific person. Let's start with the library installation:

```
$ pip install drf-extensions
```

Next, we need to add the appropriate mixin to our ViewSets. That's how we ensure proper filtering. We add it to all the views that are 'on our way':

```
# views.py

from rest_framework_extensions.mixins import NestedViewSetMixin
# …
class FriendViewset(NestedViewSetMixin, viewsets.ModelViewSet):
# …
# …
class BorrowedViewset(NestedViewSetMixin, viewsets.ModelViewSet):
# …
# …
```

Next, we need to extend the Router in the file defining the API structure:

```
# api.py

from rest_framework import routers
from rest_framework_extensions.routers import NestedRouterMixin
from core import views as myapp_views

class NestedDefaultRouter(NestedRouterMixin, routers.DefaultRouter):
   pass

router = NestedDefaultRouter()
```

The modified router allows giving names to specific paths and record nestings in them:

```
# api.py

router = NestedDefaultRouter()
friends = router.register(r'friends', myapp_views.FriendViewset)
friends.register(
   r'borrowings',
   myapp_views.BorrowedViewset,
```

```
    base_name='friend-borrow',
    parents_query_lookups=['to_who']
)
```

As you can see, the syntax is similar to the standard register. At the beginning we have the name of the endpoint and the view that supports it. The parameters base_name and parents_query_lookups are new.

The first determines the base for url names if we'd like to use the reverse() function. The second contains a list of fields relative to the previous models in the queue. The values captured from the url will be juxtaposed with these names and used as a parameter of the filter() method - in our case it will look like this: queryset = Borrowed.objects.filter(to_who={value parsed from url}).

Now we can check which items have been borrowed by one of our friends:

```
$ curl http://127.0.0.1:8000/api/v1/friends/1/borrowings/

[{"id":1,"what":1,"to_who":1,"when":"2018-01-01T12:00:00Z","returned":null},{"id":3,"what":1,"to_
who":1,"when":"2019-04-17T06:35:22.000236Z","returned":null},{"id":4,"what":1,"to_who":1,"when":"2019-04-17T06:35:
36.546848Z","returned":null}]
```

There's one more aspect of nesting worth mentioning: displaying related models. How we retrieve the list of loaned items so that the details of related models are displayed immediately - for example, who borrowed the item (name, email) and what was the loaned item (name).

And that's going to be topic of the next chapter.

CHAPTER 7: Selective Fields and Related Objects

I want to present this topic in two ways.

The first one will follow the flow of the previous chapters. The second approach will ruin that order - and then bring about a new one. But let's not get ahead of ourselves. ;)

Selective fields

For this function, we need the drf-dynamic-fields library:

```
$ pip install drf-flex-fields
```

Next, we replace the current serializer class (a mixin is also available).

```
# serializers.py
from rest_flex_fields import FlexFieldsModelSerializer
from rest_framework import serializers

from . import models

class FriendSerializer(FlexFieldsModelSerializer):
    owner = serializers.HiddenField(
        default=serializers.CurrentUserDefault()
    )

    class Meta:
        model = models.Friend
        fields = ('id', 'name', 'owner', 'has_overdue')
```

From now on, we can point to specific fields we need:

http://127.0.0.1:8000/api/v1/friends/?fields=id,name

or list fields that we'd like to omit:

http://127.0.0.1:8000/api/v1/friends/?omit=has_overdue

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

[
    {
            "id": 1,
```

```
            "name": "John Doe"
        },
        {
            "id": 2,
            "name": "Frank Tester"
        }
]
```

Adding related objects

The simplest way to add related objects is using the right serializer in the relation field.

```
# serializers.py

class BorrowedSerializer(FlexFieldsModelSerializer):
    what = BelongingSerializer()
    to_who = FriendSerializer()

    class Meta:
        model = models.Borrowed
        fields = ('id', 'what', 'to_who', 'when', 'returned')
```

But we need to remember about two things here:

1. To avoid burdening our database, let's fill the default queryset with select_related:

```
# views.py

class BorrowedViewset(NestedViewSetMixin, viewsets.ModelViewSet):
    queryset = models.Borrowed.objects.all().select_related('to_who', 'what')
    # ...
```

2. Relations defined in that way are default only for reading and adding, the saving option will require more work. We can, however, avoid that by using a library that allows expanding fields when needed:

First, we point to the serializers that will be used to expand fields.

```
class BorrowedSerializer(FlexFieldsModelSerializer):
    expandable_fields = {
        'what': (BelongingSerializer, {'source': 'what'}),
        'to_who': (FriendSerializer, {'source': 'to_who'})
    }

    class Meta:
        model = models.Borrowed
        fields = ('id', 'what', 'to_who', 'when', 'returned')
```

Next, we need to work with the viewset. At this point, we should change the base class (a mixin is also available) and add a list of expandable fields.

```
class BorrowedViewset(NestedViewSetMixin, FlexFieldsModelViewSet):
   queryset = models.Borrowed.objects.all().select_related('to_who', 'what')
   permit_list_expands = ['what', 'to_who']
   # ...
```

NOTE: Make sure to remember about select_related in queryset. It will reduce the burden on the database significantly. You can get similar result with using prefetch_related for the ManyToMany relation.

We can now expand our fields.

http://127.0.0.1:8000/api/v1/borrowings/?expand=what,to_who

```
GET /api/v1/borrowings/?expand=what,to_who

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
            "id": 1,
            "what": {
                    "id": 1,
                    "name": "Snatch"
            },
            "to_who": {
                    "id": 1,
                    "name": "John Doe",
                    "has_overdue": true
            },
            "when": "2018-01-01T12:00:00Z",
            "returned": null
    },
    {
            "id": 2,
            "what": {
                    "id": 2,
                    "name": "Boondock Saints"
            },
            "to_who": {
                    "id": 2,
                    "name": "Frank Tester",
                    "has_overdue": false
            },
```

CHAPTER 7: Try another approach

Finally, I wanted to show you another approach that ruins the order we've built so far. I'm going to talk about it using one of the most well-developed DRF extensions, DREST (Dynamic REST). You can find the code for this section in a separate branch of our repository called drest and under the part07-drest tag.

The DREST documentation is very comprehensive, so I'm only going to focus on the most important fragments related to our topic.

Installation and configuration:

```
$ pip install dynamic-rest
```

```
INSTALLED_APPS = [
    # ...
    "'dynamic_rest'",
]
```

One of the first things we should add after the installation is filling out our browsable API with a list of all the available endpoints:

```
REST_FRAMEWORK = {
    'DEFAULT_RENDERER_CLASSES': [
        'rest_framework.renderers.JSONRenderer',
        'dynamic_rest.renderers.DynamicBrowsableAPIRenderer',
    ],
}
```

To take full advantage of DREST, we should switch the currently used ModelSerializer class to DynamicModelSerializer. We're also going to change the ModelViewSet class to DynamicModelViewSet.

```
# serializers.py
```

```
from rest_framework import serializers
from dynamic_rest.serializers import DynamicModelSerializer
# ...

class FriendSerializer(DynamicModelSerializer):
    # ...

class BelongingSerializer(DynamicModelSerializer):
    # ...

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

```
# views.py

import django_filters
from django.core.mail import send_mail
from dynamic_rest.viewsets import DynamicModelViewSet
# ...


class FriendViewset(NestedViewSetMixin, DynamicModelViewSet):
   # ...

class BelongingViewset(DynamicModelViewSet):
   # ...

class BorrowedViewset(NestedViewSetMixin, DynamicModelViewSet):
   # ...
```

Unfortunately, at the moment of writing this article the library is incompatible with nested routers. However, it partially realizes this function automatically, allowing us to view ManyToMany relations. For the sake of structure, let's get rid of the code responsible for nesting:

```
# api.py

from dynamic_rest.routers import DynamicRouter
from rest_framework_extensions.routers import NestedRouterMixin

from core import views as myapp_views


router = DynamicRouter()
friends = router.register(r'friends', myapp_views.FriendViewset)
router.register(r'belongings', myapp_views.BelongingViewset)
router.register(r'borrowings', myapp_views.BorrowedViewset)
```

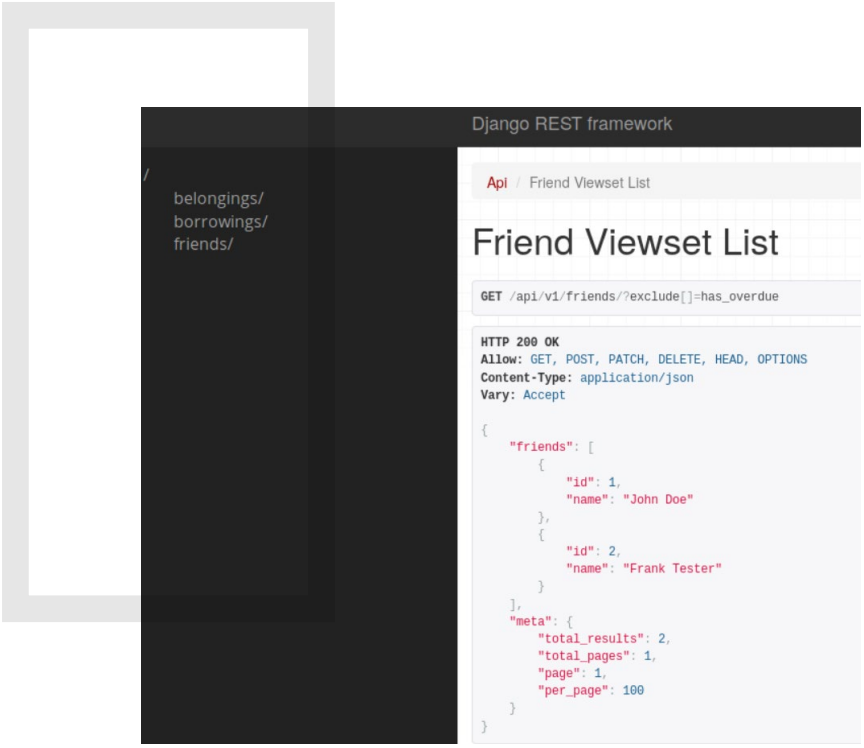
We can now begin to view the available functionalities:

Selective fields

Computing the value of specific fields in our serializers can be very costly. That's why we may want to omit them in certain cases or add them only upon request. DREST allows to realize both of these paths.

Deleting fields we don't need is easy. All it takes is adding the parameter exclude[] in our query together with the field's name.

http://127.0.0.1:8000/api/v1/friends/?exclude[]=has_overdue



It's worth to pay attention to the change of the returned data default format by the DynamicModelViewSet. Such a structure is in line with the REST best practices.

As for the fields added upon request, we can mark them with the help of the deferred_fields parameter:

```
class FriendSerializer(DynamicModelSerializer):
   owner = serializers.HiddenField(default=serializers.CurrentUserDefault())

   class Meta:
       model = models.Friend
       fields = ("id", "name", "owner", "has_overdue")
       deferred_fields = ("has_overdue",)
```

That is how we add the field to the response:

http://127.0.0.1:8000/api/v1/friends/?include[]=has_overdue

Another key matter is...

Adding related objects

DREST comes with a handy functionality that additionally optimizes query execution time.

Every relation we mark with the DynamicRelationField field can be expanded and the list of related elements will be returned together with the original result.

For example, the following query will include friends and items in the results:

http://127.0.0.1:8000/api/v1/borrowings?include[]=to_who.*&include[]=what.*

```
GET /api/v1/borrowings?include[]=to_who.*&include[]=What.*

HTTP 200 OK
Allow: GET, POST, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
     “belongings”: [
        {
             “id”: 1,
             “name”: “Snatch”
        },
        {
             “id”: 2,
             “name”: “Boondock Saints”
        }
     ],
     “friends”: [
        {
             “id”: 1,
             “name”: “John Doe”
        },
        {
             “id”: 2,
             “name”: “Frank Tester”
        }
     ],
     “borroweds”: [
        {
             “id”: 1,
             “what”: 1,
             “to_who”: 1,
             “when”: “2018-01-01T12:00:00Z”,
             “returned”: null
        },
        {
             “id”: 2,
             “what”: 2,
             “to_who”: 2,
             “when”: “2019-04-16T18:46:16.646649Z”,
             “returned”: “2019-04-16T18:46:13Z”
     },
```

Yes, I do realize that the default name of the borrowed items isn't that great...

If our application has a more complex structure, we can use that method for including objects on any level of nesting. However, we need to remember about the burden we place on the database as it becomes greater with every level.

This chapter finishes my series about the Django REST Framework where I touched upon almost every aspect of working on the REST API.

Just to remind you, here's a link to the repository where you'll find all the code I discussed in this ebook.

If the above isn't enough for you, you have two options. You can either forge your own path and share your experience with others, or turn to GraphQL (but that's a topic for another ebook).

But for now... So long and thanks for all the fish!

Want to become part of our team?

See open positions

Are you looking for a Django development team?

Hire us!

sunscrapers

Unrivaled Python Engineers

Sunscrapers Sp. z o.o.
Pokorna 2/947 (entrance 9)
00-199 Warsaw

New Business
hello@sunscrapers.com
Careers
careers@sunscrapers.com

TOP PYTHON & DJANGO Clutch DEVELOPERS 2019

TOP DEVELOPERS Clutch POLAND 2017