
FREE eBook
LEARNING
django-rest-framework
Free unaffiliated eBook created from
Stack Overflow contributors.
#django-rest-framework

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: django-rest-framework

It is an unofficial and free django-rest-framework ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official django-rest-framework.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with django-rest-framework

Remarks

Django REST Framework is a toolkit for building Web Apps. It helps the programmer to do *REST APIs*, but it can handle less mature API levels. For further information on API maturity levels, search for Richardson's Maturity Model.

In particular, Django REST Framework does not endorse any particular Hypermedia-level layout, and it is up to the programmer (or other projects, such as srf-hal-json) if they want to pursue a HATEOAS API implementation, to lay down their opinions outside of the framework. Thus, it is possible to implement a HATEOAS API in Django REST Framework, but there are no readily-available utilities already in place.

Versions

Version	Release Date
3.5.3	2016-11-07

Examples

Install or Setup

Requirements

- Python (2.7, 3.2, 3.3, 3.4, 3.5, 3.6)
- Django (1.7+, 1.8, 1.9, 1.10, 1.11)

Install

You can either use `pip` to install or clone the project from github.

- **Using `pip`:**

```
pip install djangorestframework
```

- **Using `git clone`:**

```
git clone git@github.com:tomchristie/django-rest-framework.git
```

After installing, you need to **add `rest_framework` to your `INSTALLED_APPS` settings.**

```
INSTALLED_APPS = (
    ...
    'rest_framework',
)
```

If you're intending to use the browsable API you'll probably also want to add REST framework's login and logout views. Add the following to your root urls.py file.

```
urlpatterns = [
    ...
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

Example

Let's take a look at a quick example of using REST framework to build a simple model-backed API.

We'll create a read-write API for accessing information on the users of our project.

Any global settings for a REST framework API are kept in a single configuration dictionary named `REST_FRAMEWORK`. Start off by adding the following to your `settings.py` module:

```
REST_FRAMEWORK = {
    # Use Django's standard `django.contrib.auth` permissions,
    # or allow read-only access for unauthenticated users.
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'
    ]
}
```

We're ready to create our API now. Here's our project's root `urls.py` module:

```
from django.conf.urls import url, include
from django.contrib.auth.models import User
from rest_framework import routers, serializers, viewsets

# Serializers define the API representation.
class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ('url', 'username', 'email', 'is_staff')

# ViewSets define the view behavior.
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer

# Routers provide an easy way of automatically determining the URL conf.
router = routers.DefaultRouter()
router.register(r'users', UserViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
```

```
    url(r'^', include(router.urls)),
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

You can now open the API in your browser at `http://127.0.0.1:8000/`, and view your new 'users' API. If you use the login control in the top right corner you'll also be able to add, create and delete users from the system.

Read Getting started with django-rest-framework online: https://riptutorial.com/django-rest-framework/topic/1875/getting-started-with-django-rest-framework

Chapter 2: Authentication

Examples

Writing Custom Authentications

from django.contrib.auth.models import User from rest_framework import authentication from rest_framework import exceptions

This example authentication is straight from the official docs here.

```
class ExampleAuthentication(BaseAuthentication):
    def authenticate(self, request):
        username = request.META.get('X_USERNAME')
        if not username:
            return None

        try:
            user = User.objects.get(username=username)
        except User.DoesNotExist:
            raise AuthenticationFailed('No such user')

        return (user, None)
```

There are four parts to a custom authentication class.

1. Extend it from BaseAuthentication class provided in `from rest_framework. authentication import BaseAuthentication
2. Have a method called `authenticate` taking `request` as first argument.
3. Return a tuple of (user, None) for a successful authentication.
4. Raise `AuthenticationFailed` exception for a failed authentication. This is available in rest_framework.authentication.

```
class SecretAuthentication(BaseAuthentication):
    def authenticate(self, request):
        app_key = request.META.get('APP_KEY')
        app_secret = request.META.get('APP_SECRET')
        username = request.META.get('X_USERNAME')
        try:
            app = ClientApp.objects.match_secret(app_key, app_secret)
        except ClientApp.DoesNotExist:
            raise AuthenticationFailed('App secret and key does not match')
        try:
           user = app.users.get(username=username)
        except User.DoesNotExist:
            raise AuthenticationFailed('Username not found, for the specified app')
        return (user, None)
```

The authentication scheme will return HTTP 403 Forbidden responses when an unauthenticated request is denied access.

API-KEY base request authentication

You can use `django rest framework` permission classes to check request headers and authenticate user requests

- Define your secret_key on project settings

```
API_KEY_SECRET = 'secret_value'
```

note: a good practice is to use environment variables to store this secret value.

- Define a permission class for API-KEY authentication

create permissions.py file on your app dir with below codes:

```
from django.conf import settings

from rest_framework.permissions import BasePermission

class Check_API_KEY_Auth(BasePermission):
    def has_permission(self, request, view):
        # API_KEY should be in request headers to authenticate requests
        api_key_secret = request.META.get('API_KEY')
        return api_key_secret == settings.API_KEY_SECRET
```

- Add this permission class to views

```
from rest_framework.response import Response
from rest_framework.views import APIView

from .permissions import Check_API_KEY_Auth


class ExampleView(APIView):
    permission_classes = (Check_API_KEY_Auth,)

    def get(self, request, format=None):
        content = {
            'status': 'request was permitted'
        }
        return Response(content)
```

Or, if you're using the @api_view decorator with function based views

```
from rest_framework.decorators import api_view, permission_classes
from rest_framework.response import Response

from .permissions import Check_API_KEY_Auth


@api_view(['GET'])
@permission_classes((Check_API_KEY_Auth, ))
def example_view(request, format=None):
    content = {
        'status': 'request was permitted'
    }
```

```
    return Response(content)
```

Read Authentication online: https://riptutorial.com/django-rest-framework/topic/5451/authentication

Chapter 3: Authentication

Examples

Setting up authentication globally

Authentication in Django REST Framework can be configured globally as a subkey of the `REST_FRAMEWORK` variable in settings.py, just like the rest of the default framework configurations.

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

Setting up authentication for a specific APIView

Authentication can be set for an specific APIView endpoint, by using the `authentication_classes` variable:

```
from rest_framework.authentication import SessionAuthentication, BasicAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from rest_framework.views import APIView

class ExampleView(APIView):
    authentication_classes = (SessionAuthentication, BasicAuthentication)
    permission_classes = (IsAuthenticated,)

    def get(self, request):
        content = {
            'user': unicode(request.user),  # `django.contrib.auth.User` instance.
            'auth': unicode(request.auth),  # None
        }
        return Response(content)
```

Using basic token-based authentication

Django REST Framework provides a basic token-based authentication mechanism which needs to be configured as an application in Django before being usable, so that tokens are created in the database, and their lifecycle handled.

Add Token-based authentication to settings.py

```
INSTALLED_APPS = (
    ...
    'rest_framework.authtoken'
)
```

Run the database migration

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

Create tokens for your users

Somehow, you will have to create a token and return it:

```
def some_api(request):
    token = Token.objects.create(user=request.user)
    return Response({'token': token.key})
```

There is already an API endpoint in the token application, so that you can simply add the following to your urls.py:

urls.py

```
from rest_framework.authtoken import views
urlpatterns += [
    url(r'^auth-token/', views.obtain_auth_token)
]
```

Clients can now authenticate

Using a `Authorization` header like:

```
Authorization: Token 123456789
```

Prefixed by a literal "Token" and the token itself after whitespace.

The literal can be changed by subclassing `TokenAuthentication` and changing the `keyword` class variable.

If authenticated, `request.auth` will contain the `rest_framework.authtoken.models.BasicToken` instance.

Setting up OAuth2 authentication

OAuth is not handled by Django REST Framework, but there are a couple of pip modules that implement an OAuth client. The REST Framework documentation suggests one of the following

modules:

- Django OAuth Toolkit
- Django REST Framework OAuth

Django OAuth Toolkit

```
pip install django-oauth-toolkit
```

settings.py

```
INSTALLED_APPS = (
    ...
    'oauth2_provider',
)

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'oauth2_provider.ext.rest_framework.OAuth2Authentication',
    )
}
```

urls.py

```
urlpatterns = patterns(
    ...
    url(r'^o/', include('oauth2_provider.urls', namespace='oauth2_provider')),
)
```

Django REST Framework OAuth

```
pip install djangorestframework-oauth django-oauth2-provider
```

settings.py

```
INSTALLED_APPS = (
    ...
    'provider',
    'provider.oauth2',
)

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.OAuth2Authentication',
    )
}
```

urls.py

```
urlpatterns = patterns(
    ...
    url(r'^oauth2/', include('provider.oauth2.urls', namespace='oauth2')),
)
```

Admin

Go to the admin panel and create a new `Provider.Client` to have a `client_id` and a `client_secret`.

Using better Token-based authorization with several client-tokens

The most interesting package for managing real tokens is django-rest-knox which supports multiple tokens per user (and cancelling each token independently), as well as having support for token expiration and several other security mechanisms.

`django-rest-knox` depends on `cryptography`. You can find more information on how to install it at: http://james1345.github.io/django-rest-knox/installation/

Installing knox

```
pip install django-rest-knox
```

settings.py

```
INSTALLED_APPS = (
  ...
  'rest_framework',
  'knox',
  ...
)
```

Apply migrations:

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

Read Authentication online: https://riptutorial.com/django-rest-framework/topic/6276/authentication

Chapter 4: Filters

Examples

Filtering Examples, from Simple To More Complex ones

Plain Vanilla Filtering

To filter any view, override its `get_queryset` method to return a filtered query set

```
class HREmployees(generics.ListAPIView):
    def get_queryset(self):
        return Employee.objects.filter(department="Human Resources")
```

All the API functions will then use the filtered query set for operations. For example, the listing of the above view will only contain employees whose department is Human Resources.

Accessing query parameters in get_queryset

Filtering based on request parameters is easy.

`self.request`, `self.args` and `self.kwargs` are available and point to the current request and its parameters for filtering

```
def DepartmentEmployees(generics.ListAPIView):
    def get_queryset(self):
        return Employee.objects.filter(department=self.kwargs["department"])
```

Letting the API parameters decide what to filter

If you want more flexibility and allow the API call to pass in parameters to filter the view, you can plugin filter backends like Django Request Framework(installed via pip)

```
from rest_framework import filters

class Employees(generics.ListAPIView):
    queryset=Employee.objects.all()
    filter_backends = (filters.DjangoFilterBackend,)
    filter_fields = ("department", "role",)
```

Now you can make an API call `/api/employees?department=Human Resources` and you'll get a list of

employees that belong only to the HR department, or `/api/employees?role=manager&department=Human Resources` to get only managers in the HR department.

You can combine query set filtering with Django Filter Backend, no problemo. The filters will work on the filtered query set returned by `get_queryset`

```
from rest_framework import filters

class HREmployees(generics.ListAPIView):
    filter_backends = (filters.DjangoFilterBackend,)
    filter_fields = ("department", "role",)

    def get_queryset(self):
        return Employee.objects.filter(department="Human Resources")
```

FilterSets

So far, you can get by with simple type matches in the above cases.

But what if you want something more complex, like a list of HR employees who are between 25 and 32 years in age?

Answer to problem: Filtersets

Filter sets are classes that define how to filter various fields of the model.

Define em like so

```
class EmployeeFilter(django_filters.rest_framework.FilterSet):
    min_age = filters.django_filters.NumberFilter(name="age", lookup_expr='gte')
    max_age = filters.django_filters.NumberFilter(name="price", lookup_expr='lte')

    class Meta:
        model = Employee
        fields = ['age', 'department']
```

`name` points to the field which you want to filter

`lookup_expr` basically refers to the same names you use while filtering query sets, for example you can do a "starts with" match using `lookup_expr="startswith"` which is equivalent to `Employee.objects.filter(department__startswith="Human")`

Then use them in your view classes by using `filter_class` instead of `filter_fields`

```
class Employees(generics.ListAPIView):
    queryset=Employee.objects.all()
    filter_backends = (filters.DjangoFilterBackend,)
    filter_class = EmployeeFilter
```

Now you can do `/api/employees?department=Human Resources&min_age=25&max_age=32`

Non-exact matches and relationships

Filter classes and expressions are very similar to how you specify filtering in query sets

You can use the "__" notation to filter fields in relationships, For example, if department was a foreign key from employee, you can add

```
filter_fields=("department__name",)
```

and then you can do `/api/employees?department__name=Human Resources`

Or more elegantly, you can create a filter set, add a filter variable called `dept` and set its name to `department__name`, allowing you to do `/api/employees?dept=Human Resources`

Read Filters online: https://riptutorial.com/django-rest-framework/topic/8144/filters

Chapter 5: Mixins

Introduction

The mixin classes provide the actions that are used to provide the basic view behavior. Note that the mixin classes provide action methods rather than defining the handler methods, such as `.get()` and `.post()`, directly. This allows for more flexible composition of behavior. -Official Django rest Framework Documentation-

Examples

[Introductory] List of Mixins And Usage on Views/Viewsets

List of available mixins:

- **ListModelMixin:** provides a `.list()` method to the view/viewset
- **RetrieveModelMixin:** provides a `.retrieve()` method to the view/viewset
- **CreateModelMixin:** provides a `.create()` method to the view/viewset
- **UpdateModelMixin:** provides a `.update()` method to the view/viewset
- **DestroyModelMixin:** provides a `.destroy()` method to the view/viewset

---

We can mix and match the mixins in our generic views and viewsets, in order to give them the corresponding utility that we need:

1. An API view with `.list()` `.create()` and `.destroy()` methods?
   Inherit from the `GenericAPIView` and combine the appropriate mixins:

```
from rest_framework import mixins, generics

from myapp.models import MyModel
from myapp.serializers import MyModelSerializer

class MyCustomAPIView(mixins.ListModelMixin,
                      mixins.CreateModelMixin,
                      mixins.DestroyModelMixin,
                      generics.GenericAPIView):

    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)

    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```

2. A viewset with only a `.list()` and `.update()` methods?
   Inherit from the `GenericViewSet` and add the appropriate mixins:

```
from rest_framework import mixins

class MyCustomViewSet(mixins.ListModelMixin,
                      mixins.UpdateModelMixin,
                      viewsets.GenericViewSet):
    pass
```

   Yes, it was that easy!!

To conclude, we can mix and match every mixin we need and utilize it to customize our views and their methods in any possible combination!

[Intermediate] Creating Custom Mixins

DRF offers the chance to further customize the behavior of the generic views/viewsets by allowing the creation of custom mixins.

How to:

To define a custom mixin we just need to create a class inheriting from `object`.

Let's assume that we want to define two separate views for a model named `MyModel`. Those views will share the same `queryset` and the same `serializer_class`. We will save ourselves some code repetition and we will put the above in a single mixin to be inherited by our views:

- `my_app/views.py` (that is not the only file option available to place our custom mixins, but it is the less complex):

```
from rest_framework.generics import CreateAPIView, RetrieveUpdateAPIView
from rest_framework.permissions import IsAdminUser

class MyCustomMixin(object):
    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer

class MyModelCreateView(MyCustomMixin, CreateAPIView):
    """
    Only an Admin can create a new MyModel object
    """
    permission_classes = (IsAdminUser,)

    Do view staff if needed...

class MyModelCreateView(MyCustomMixin, RetrieveUpdateAPIView):
    """
    Any user can Retrieve and Update a MyModel object
    """
    Do view staff here...
```

Conclusion:

Mixins are essentially blocks of reusable code for our application.

Read Mixins online: https://riptutorial.com/django-rest-framework/topic/10083/mixins

Chapter 6: Pagination

Introduction

Pagination is the splitting of large datasets into separated and autonomous pages.

On django rest framework, pagination allows the user to modify the amount of data in each page and the style by which the split is applied.

Examples

[Introductory] Setup Pagination Style Globally

In order to set the pagination style for the entire project, you need to set the `DEFAULT_PAGINATION_CLASS` and `PAGE_SIZE` on the project settings.

To do so, go to `settings.py` and on the `REST_FRAMEWORK` variable, add the following:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.DESIRED_PAGINATION_STYLE',
    'PAGE_SIZE': 100
}
```

In place of the `DESIRED_PAGINATION_STYLE` one of the following must be placed:

- `PageNumberPagination`: Accepts a single `page` number in the request query parameters.

  ```
  http://your_api_url/a_table/?page=2
  ```

- `LimitOffsetPagination`: Accepts a `limit` parameter, which indicates the maximum number of items that will be returned and an `offset` parameter which indicates the starting position of the query in relation to the dataset. `PAGE_SIZE` does not need to be set for this style.

  ```
  http://your_api_url/a_table/?limit=50&offset=100
  ```

- `CursorPagination`: Cursor based pagination is more complex than the above styles. It requires that the dataset presents a fixed ordering, and does not allow the client to navigate into arbitrarily positions of the dataset.

- Custom pagination styles can be defined in place of the above.

[Intermediate] Override Pagination style and setup Pagination per class

Override Pagination Style:

Every available pagination style can be overridden by creating a new class that inherits from one of the available styles and then alters its parameters:

```
class MyPagination(PageNumberPagination):
    page_size = 20
    page_size_query_param = 'page_size'
    max_page_size = 200
    last_page_strings = ('the_end',)
```

Those parameters (as listed on the pagination official documentation) are:

PageNumberPagination

- `page_size`: A numeric value indicating the page size. If set, this overrides the `PAGE_SIZE` setting. Defaults to the same value as the `PAGE_SIZE` settings key.
- `page_query_param`: A string value indicating the name of the query parameter to use for the pagination control.
- `page_size_query_param`: If set, this is a string value indicating the name of a query parameter that allows the client to set the page size on a per-request basis. Defaults to `None`, indicating that the client may not control the requested page size.
- `max_page_size`: If set, this is a numeric value indicating the maximum allowable requested page size. This attribute is only valid if `page_size_query_param` is also set.
- `last_page_strings`: A list or tuple of string values indicating values that may be used with the `page_query_param` to request the final page in the set. Defaults to `('last',)`
- `template`: The name of a template to use when rendering pagination controls in the browsable API. May be overridden to modify the rendering style, or set to `None` to disable HTML pagination controls completely. Defaults to `"rest_framework/pagination/numbers.html"`.

LimitOffsetPagination

- `default_limit`: A numeric value indicating the limit to use if one is not provided by the client in a query parameter. Defaults to the same value as the `PAGE_SIZE` settings key.
- `limit_query_param`: A string value indicating the name of the "limit" query parameter. Defaults to `'limit'`.
- `offset_query_param`: A string value indicating the name of the "offset" query parameter. Defaults to `'offset'`.
- `max_limit`: If set this is a numeric value indicating the maximum allowable limit that may be requested by the client. Defaults to `None`.
- `template`: Same as *PageNumberPagination*.

CursorPagination

- `page_size`: Same as *PageNumberPagination*.
- `cursor_query_param`: A string value indicating the name of the "cursor" query parameter. Defaults to `'cursor'`.
- `ordering`: This should be a string, or list of strings, indicating the field against which the cursor based pagination will be applied. For example: `ordering = 'slug'`. Defaults to `-created`. This value may also be overridden by using `OrderingFilter` on the view.
- `template`: Same as *PageNumberPagination*.

Setup Pagination per Class:

In addition to the ability to setup the Pagination style globally, a setup per class is available:

```
class MyViewSet(viewsets.GenericViewSet):
    pagination_class = LimitOffsetPagination
```

Now only `MyViewSet` has a `LimitOffsetPagination` pagination.

A custom pagination style can be used in the same way:

```
class MyViewSet(viewsets.GenericViewSet):
    pagination_class = MyPagination
```

[Intermediate] Complex Usage Example

Lets assume that we have a complex api, with many generic views and some generic viewsets. We want to enable `PageNumberPagination` to every view, except one (either generic view or viewset, does not make a difference) for which we want a customized case of `LimitOffsetPagination`.

To achieve that we need to:

1. On the `settings.py` we will place our default pagination in order to enable it for every generic view/viewset and we will set `PAGE_SIZE` to 50 items:

   ```
   REST_FRAMEWORK = {
       'DEFAULT_PAGINATION_CLASS':
           'rest_framework.pagination.PageNumberPagination',
       'PAGE_SIZE': 50
   }
   ```

2. Now in our `views.py` (or in another `.py` ex: `paginations.py`), we need to override the `LimitOffsetPagination`:

   ```
   from rest_framework.pagination import LimitOffsetPagination

   class MyOffsetPagination(LimitOffsetPagination):
       default_limit = 20
       max_limit = 1000
   ```

   A custom `LimitOffsetPagination` with `PAGE_ZISE` of 20 items and maximum `limit` of 1000 items.

3. In our `views.py`, we need to define the `pagination_class` of our special view:

   ```
   imports ...

   # ==================================
   #    PageNumberPagination classes
   # ==================================
   ```

```
class FirstView(generics.ListAPIView):
    ...

class FirstViewSet(viewsets.GenericViewSet):
    ...

...

# ===================================
#     Our custom Pagination class
# ===================================

class IAmSpecialView(generics.ListAPIView):
    pagination_class = MyOffsetPagination
    ...
```

Now every generic view/viewset in the app has `PageNumberPagination`, except `IAmSpecial` class, which is indeed *special* and has its own customized `LimitOffsetPagination`.

[Advanced] Pagination on Non Generic Views/Viewsets

This is an advanced subject, do not attempt without first understanding the other examples of this page.

As stated in the official Django Rest Framework on pagination:

> Pagination is only performed automatically if you're using the generic views or viewsets. If you're using a regular APIView, you'll need to call into the pagination API yourself to ensure you return a paginated response. See the source code for the mixins.ListModelMixin and generics.GenericAPIView classes for an example.

But what if we want to use pagination on a non generic view/viewset?

Well let's go down the rabbit hole:

1. First stop is the official Django Rest Framework's repository and specifically the django-rest-framework/rest_framework/generics.py. The specific line this link is pointing at, shows us how the developers of the framework deal with pagination in their generics.
   That is exactly what we are going to use to our view as well!

2. Let's assume that we have a global pagination setup like the one shown in the introductory example of this page and lets assume as well that we have an `APIView` which we want to apply pagination to.

3. Then on `views.py`:

```
from django.conf import settings
from rest_framework.views import APIView

class MyView(APIView):
    queryset = OurModel.objects.all()
    serializer_class = OurModelSerializer
    pagination_class = settings.DEFAULT_PAGINATION_CLASS # cool trick right? :)
```

```
    # We need to override get method to achieve pagination
    def get(self, request):
        ...
        page = self.paginate_queryset(self.queryset)
        if page is not None:
            serializer = self.serializer_class(page, many=True)
            return self.get_paginated_response(serializer.data)

        ... Do other stuff needed (out of scope of pagination)

    # Now add the pagination handlers taken from
    #  django-rest-framework/rest_framework/generics.py

    @property
    def paginator(self):
        """
        The paginator instance associated with the view, or `None`.
        """
         if not hasattr(self, '_paginator'):
             if self.pagination_class is None:
                 self._paginator = None
             else:
                 self._paginator = self.pagination_class()
         return self._paginator

     def paginate_queryset(self, queryset):
         """
         Return a single page of results, or `None` if pagination is disabled.
         """
         if self.paginator is None:
             return None
         return self.paginator.paginate_queryset(queryset, self.request, view=self)

     def get_paginated_response(self, data):
         """
         Return a paginated style `Response` object for the given output data.
         """
         assert self.paginator is not None
         return self.paginator.get_paginated_response(data)
```

Now we have an `APIView` that handles pagination!

[Intermediate] Pagination on a function based view

We have seen in those examples (ex_1, ex_2) how to use and override the pagination classes in any generic class base view.
What happens when we want to use pagination in a function based view?

Lets also assume that we want to create a function based view for `MyModel` with `PageNumberPagination`, responding only to a `GET` request. Then:

```
from rest_framework.pagination import PageNumberPagination


@api_view(['GET',])
def my_function_based_list_view(request):
```

```
    paginator = PageNumberPagination()
    query_set = MyModel.objects.all()
    context = paginator.paginate_queryset(query_set, request)
    serializer = MyModelSerializer(context, many=True)
    return paginator.get_paginated_response(serializer.data)
```

We can do the above for a custom pagination as well by changing this line:

```
paginator = PageNumberPagination()
```

to this

```
paginator = MyCustomPagination()
```

provided that we have defined `MyCustomPagination` to override some default pagination

Read Pagination online: https://riptutorial.com/django-rest-framework/topic/9950/pagination

Chapter 7: Routers

Introduction

Routing is the process of mapping the logic (view methods etc.) to a set of URLs. REST framework adds support for automatic URL routing to Django.

Syntax

- router = routers.SimpleRouter()
- router.register(prefix, viewset)
- router.urls # the generated set of urls for the registered viewset.

Examples

[Introductory] Basic usage, SimpleRouter

Automatic routing for the DRF, can be achieved for the `ViewSet` classes.

1. Assume that the ViewSet class goes by the name of `MyViewSet` for this example.

2. To generate the routing of `MyViewSet`, the `SimpleRouter` will be utilized.
   On `myapp/urls.py`:

   ```
   from rest_framework import routers

   router = routers.SimpleRouter()         # initialize the router.
   router.register(r'myview', MyViewSet)   # register MyViewSet to the router.
   ```

3. That will generate the following URL patterns for `MyViewSet`:

   - `^myview/$` with name `myview-list`.
   - `^myview/{pk}/$` with name `myview-detail`

4. Finally to add the generated patterns in the `myapp`'s URL patterns, the django's `include()` will be used.
   On `myapp/urls.py`:

   ```
   from django.conf.urls import url, include
   from rest_framework import routers

   router = routers.SimpleRouter()
   router.register(r'myview', MyViewSet)

   urlpatterns = [
       url(r'other/prefix/if/needed/', include(router.urls)),
   ]
   ```

Read Routers online: https://riptutorial.com/django-rest-framework/topic/10938/routers

Chapter 8: Serializers

Examples

Speed up serializers queries

Let's say we have model `Travel` with many related fields:

```
class Travel(models.Model):

    tags = models.ManyToManyField(
        Tag,
        related_name='travels', )
    route_places = models.ManyToManyField(
        RoutePlace,
        related_name='travels', )
    coordinate = models.ForeignKey(
        Coordinate,
        related_name='travels', )
    date_start = models.DateField()
```

And we want to build CRUD in `/travels` via view ViewSet.
Here is the simple viewset:

```
class TravelViewset(viewsets.ModelViewSet):

    queryset = Travel.objects.all()
    serializer_class = TravelSerializer
```

Problem with this ViewSet is we have many related fields in our `Travel` model, so Django will hit db for every `Travel` instance. We can call select_related and prefetch_related directly in `queryset` attribute, but what if we want to separate serializers for `list`, `retrieve`, `create`.. actions of ViewSet. So we can put this logic in one mixin and inherit from it:

```
class QuerySerializerMixin(object):
    PREFETCH_FIELDS = [] # Here is for M2M fields
    RELATED_FIELDS = [] # Here is for ForeignKeys

    @classmethod
    def get_related_queries(cls, queryset):
        # This method we will use in our ViewSet
        # for modify queryset, based on RELATED_FIELDS and PREFETCH_FIELDS
        if cls.RELATED_FIELDS:
            queryset = queryset.select_related(*cls.RELATED_FIELDS)
        if cls.PREFETCH_FIELDS:
            queryset = queryset.prefetch_related(*cls.PREFETCH_FIELDS)
        return queryset


    class TravelListSerializer(QuerySerializerMixin, serializers.ModelSerializer):

        PREFETCH_FIELDS = ['tags'']
```

```
        RELATED_FIELDS = ['coordinate']
        # I omit fields and Meta declare for this example


    class TravelRetrieveSerializer(QuerySerializerMixin, serializers.ModelSerializer):

        PREFETCH_FIELDS = ['tags', 'route_places']
```

Now rewrite our `ViewSet` with new serializers

```
class TravelViewset(viewsets.ModelViewSet):

    queryset = Travel.objects.all()

    def get_serializer_class():
        if self.action == 'retrieve':
            return TravelRetrieveSerializer
        elif self.action == 'list':
            return TravelListSerializer
        else:
            return SomeDefaultSerializer


    def get_queryset(self):
        # This method return serializer class
        # which we pass in class method of serializer class
        # which is also return by get_serializer()
        q = super(TravelViewset, self).get_queryset()
        serializer = self.get_serializer()
        return serializer.get_related_queries(q)
```

Updatable nested serializers

Nested serializers by default don't support create and update. To support this without duplicating DRF create/update logic, it is important to *remove* the nested data from `validated_data` before delegating to `super`:

```
# an ordinary serializer
class UserProfileSerializer(serializers.ModelSerializer):
    class Meta:
        model = UserProfile
        fields = ('phone', 'company')


class UserSerializer(serializers.ModelSerializer):
    # nest the profile inside the user serializer
    profile = UserProfileSerializer()

    class Meta:
        model = UserModel
        fields = ('pk', 'username', 'email', 'first_name', 'last_name')
        read_only_fields = ('email', )

    def update(self, instance, validated_data):
        nested_serializer = self.fields['profile']
        nested_instance = instance.profile
        # note the data is `pop`ed
```

```
        nested_data = validated_data.pop('profile')
        nested_serializer.update(nested_instance, nested_data)
        # this will not throw an exception,
        # as `profile` is not part of `validated_data`
        return super(UserDetailsSerializer, self).update(instance, validated_data)
```

In the case of `many=True`, Django will complain that `ListSerializer` does not support `update`. In that case, you have to handle the list semantics yourself, but can still delegate to `nested_serializer.child`.

Order of Serializer Validation

In DRF, serializer validation is run in a specific, undocumented order

1. Field deserialization called (`serializer.to_internal_value` and `field.run_validators`)
2. `serializer.validate_[field]` is called for each field.
3. Serializer-level validators are called (`serializer.run_validation` followed by `serializer.run_validators`)
4. Finally, `serializer.validate` is called to complete validation.

Getting list of all related children objects in parent's serializer

Assume that, we implement a simple API and we have the following models.

```
class Parent(models.Model):
    name = models.CharField(max_length=50)


class Child(models.Model):
    parent = models.ForeignKey(Parent)
    child_name = models.CharField(max_length=80)
```

And we want to return a response when a particular `parent` is retrieved via API.

```
{
'url': 'https://dummyapidomain.com/parents/1/',
'id': '1',
'name': 'Dummy Parent Name',
'children': [{
        'id': 1,
        'child_name': 'Dummy Children I'
    },
    {
        'id': 2,
        'child_name': 'Dummy Children II'
    },
    {
        'id': 3,
        'child_name': 'Dummy Children III'
    },
    ...
],
```

}

For this purpose, we implement the corresponding serializers like this:

```
class ChildSerializer(serializers.HyperlinkedModelSerializer):

    parent_id =
serializers.PrimaryKeyRelatedField(queryset=Parent.objects.all(),source='parent.id')

    class Meta:
        model = Child
        fields = ('url','id','child_name','parent_id')

    def create(self, validated_data):
        subject = Child.objects.create(parent=validated_data['parent']['id'],
child_name=validated_data['child_name'])

        return child


class ParentSerializer(serializers.HyperlinkedModelSerializer):
    children = ChildSerializer(many=True, read_only=True)
    class Meta:
        model = Course
        fields = ('url','id','name','children')
```

To make this implementation work properly we need to update our `Child` model and add a **related_name** to `parent` field. Updated version of our `Child` model implementation should be like this:

```
class Child(models.Model):
parent = models.ForeignKey(Parent, related_name='children')  # <--- Add related_name here
child_name = models.CharField(max_length=80)
```

By doing this, we'll be able to get the list of all related children objects in parent's serializer.

Read Serializers online: https://riptutorial.com/django-rest-framework/topic/2377/serializers

Chapter 9: Serializers

Introduction

According to DRF official documentation, serializers helps to convert complex data like querysets and model instance to native python data types so that it could be rendered as JSON, XML and other content types.

The serializers in DRF are more like django **Form** and **ModelForm** class. **Serializer** class provide us a custom way to handle the data like django Form. And **ModelSerializer** class provide us a easy way to handle model based data.

Examples

Basic Introduction

Let's say we have a model called product.

```
class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.IntegerField()
```

Now we are going to declare a model serializers for this model.

```
from rest_framework.serializers import ModelSerializer

class ProductSerializers(ModelSerializer):
    class Meta:
        model = Product
        fields = '__all__'
        read_only_fields = ('id',)
```

By this ProductSerializers class we have declared a model serializers. In Meta class, by model variable we told the ModelSerializer that our model will be the Product model. By fields variable we told that this serializer should include all the field of the model. Lastly by read_only_fields variable we told that id will be a 'read only' field, it can't be edited.

Let's see what's in our serializer. At first, import the serializer in command line and create a instance and then print it.

```
>>> serializer = ProductSerializers()
>>> print(serializer)
ProductSerializers():
        id = IntegerField(label='ID', read_only=True)
        name = CharField(max_length=100)
        price = IntegerField(max_value=2147483647, min_value=-2147483648)
```

So, our serializer grab all the field form our model and create all the field it's own way.

We can use ProductSerializers to serialize a product, or a list of product.

```
>>> p1 = Product.objects.create(name='alu', price=10)
>>> p2 = Product.objects.create(name='mula', price=5)
>>> serializer = ProductSerializers(p1)
>>> print(serializer.data)
{'id': 1,'name': 'alu', 'price': 10}
```

At this point we've translated the model instance into Python native datatypes. To finalize the serialization process we render the data into json.

```
>>> from rest_framework.renderers import JSONRenderer
>>> serializer = ProductSerializers(p1)
>>> json = JSONRenderer().render(serializer.data)
>>> print(json)
'{"id": 1,"name": "alu", "price": 10}'
```

Read Serializers online: https://riptutorial.com/django-rest-framework/topic/8871/serializers

Chapter 10: Token Authentication With Django Rest Framework

Examples

ADDING AUTH FUNCTIONALITY

Django REST Framework has some authentication methods already built in, one of them is Token based, so first thing to do is to tell our project we're going to use rest framework's authentication. Open `settings.py` file and add the highlighted line.

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'rest_framework.authtoken',
    'test_app',
)
```

As we've added a new app to the project, we must synchronize

```
python manage.py migrate
```

CREATING A SUPERUSER IN DJANGO

In order to use authentication, we can rely on django users model, so the first thing to do is to create a superuser.

```
python manage.py createsuperuser
```

GETTING THE USER TOKEN

Token authentication functionality assigns a token to a user, so each time you use that token, the request object will have a user attribute that holds the user model information. Easy, isn't it?

We'll create a new POST method to return the token for this user, as long as the request holds a correct user and password. Open **views.py** located at test_app application folder.

```
from rest_framework.response import Response

from rest_framework.authtoken.models import Token
from rest_framework.exceptions import ParseError
from rest_framework import status
```

```
from django.contrib.auth.models import User


# Create your views here.
class TestView(APIView):
    """
    """

    def get(self, request, format=None):
        return Response({'detail': "GET Response"})

    def post(self, request, format=None):
        try:
            data = request.DATA
        except ParseError as error:
            return Response(
                'Invalid JSON - {0}'.format(error.detail),
                status=status.HTTP_400_BAD_REQUEST
            )
        if "user" not in data or "password" not in data:
            return Response(
                'Wrong credentials',
                status=status.HTTP_401_UNAUTHORIZED
            )

        user = User.objects.first()
        if not user:
            return Response(
                'No default user, please create one',
                status=status.HTTP_404_NOT_FOUND
            )

        token = Token.objects.get_or_create(user=user)

        return Response({'detail': 'POST answer', 'token': token[0].key})
```

USING THE TOKEN

Let's create a new View that requires this authentication mechanism.

We need to add these import lines:

```
from rest_framework.authentication import TokenAuthentication
from rest_framework.permissions import IsAuthenticated
```

and then create the new View in the same `views.py` file

```
class AuthView(APIView):
    """
    Authentication is needed for this methods
    """
    authentication_classes = (TokenAuthentication,)
    permission_classes = (IsAuthenticated,)

    def get(self, request, format=None):
        return Response({'detail': "I suppose you are authenticated"})
```

As we did on previous post, we need to tell our project that we have a new REST path listening, on `test_app/urls.py`

```
from rest_framework.urlpatterns import format_suffix_patterns
from test_app import views

urlpatterns = patterns('test_app.views',
    url(r'^', views.TestView.as_view(), name='test-view'),
    url(r'^auth/', views.AuthView.as_view(), name='auth-view'),
)

urlpatterns = format_suffix_patterns(urlpatterns)
```

Using CURL

If a curl would be run against this endpoint

```
curl http://localhost:8000/auth/
op : {"detail": "Authentication credentials were not provided."}%
```

would return a **401 error UNAUTHORIZED**

but in case we get a token before:

```
curl -X POST -d "user=Pepe&password=aaaa"  http://localhost:8000/
{"token": "f7d6d027025c828b65cee5d38240aec60dffa150", "detail": "POST answer"}%
```

and then we put that token into the header of the request like this:

```
curl http://localhost:8000/auth/ -H 'Authorization: Token
f7d6d027025c828b65cee5d38240aec60dffa150'

op: {"detail": "I suppose you are authenticated"}%
```

Read Token Authentication With Django Rest Framework online: https://riptutorial.com/django-rest-framework/topic/9087/token-authentication-with-django-rest-framework

Chapter 11: Using django-rest-framework with AngularJS as front-end framework.

Introduction

In this topic we will look at how to use Django REST Framework as a backend API for a AngularJS app. The main issues that arise between using DRF and AngularJS together generally revolve around the HTTP communication between the two technologies, as well as the representation of the data on both ends, and finally how to deploy and architect the application/system.

Examples

DRF View

```
class UserRegistration(APIView):
    def post(self, request, *args, **kwargs):
        serializer = UserRegistrationSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.to_representation(instance=serializer.instance),
status=status.HTTP_201_CREATED)
```

Angular Request

```
$http.post("\<domain\>/user-registration/", {username: username, password: password})
     .then(function (data) {
         // Do success actions here
     });
```

Read Using django-rest-framework with AngularJS as front-end framework. online: https://riptutorial.com/django-rest-framework/topic/10893/using-django-rest-framework-with-angularjs-as-front-end-framework-

Credits

S. No	Chapters	Contributors
1	Getting started with django-rest-framework	Abhishek, Chitrank Dixit, Community, davyria, itmard, Majid, Rahul Gupta, Saksow, ssice
2	Authentication	iankit, itmard
3	Filters	Bitonator
4	Mixins	John Moutafis
5	Pagination	John Moutafis
6	Routers	John Moutafis
7	Serializers	hnroot, Ivan Semochkin, Louis Barranqueiro, Silly Freak
8	Token Authentication With Django Rest Framework	Ali_Waris, Cadmus
9	Using django-rest-framework with AngularJS as front-end framework.	mattjegan