Creating an API in Django without DRF

Create a new app

```
python manage.py startapp api
```

Add the new app to the `settings.py` file:

```
INSTALLED_APPS = [
    # "django.contrib.admin",
    # "django.contrib.auth",
    # "django.contrib.contenttypes",
    # "django.contrib.sessions",
    # "django.contrib.messages",
    # "django.contrib.staticfiles",
    # "store.apps.StoreConfig",
    "api",
]
```

Create the first api view

Inside the api folder, edit your `views.py` file as follows:

```
from django.http import JsonResponse

def api_home(request):
    return JsonResponse({"detail": "Hi there, this is your Django API response!"})
```

Inside the api folder, create a `urls.py` file, and insert the following:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.api_home)
]
```

Now we must include this urls file in our project's main `urls.py` file:

```
urlpatterns = [
    # path("admin/", admin.site.urls),
    # path("", include("store.urls")),
    path("api/", include("api.urls")),
]
```

Test

1. Start the web server: `python manage.py runserver`
2. Go to http://127.0.0.1:8000/api and see if you get the message

Add params to the request

Most of the times when we use API's we need to pass in data for the backend to process and return a specific response regarding that data. Let's take care of that in our view.

```
from django.http import JsonResponse

def api_home(request):
    params = dict(request.GET)
    if params:
        message = 'Params: '
        for param_key, param_value in params.items():
            message += f"{param_key}: {param_value}, "
        return JsonResponse({"detail": message})
    else:
        return JsonResponse({"detail": "Hmm, you didn't pass any params, did you?"})
```

Test

1. Go to http://127.0.0.1:8000/api?letter=A and see if you get the param.
2. Go to http://127.0.0.1:8000/api?letter=A&number=1 and see if you get the params.
3. Go to http://127.0.0.1:8000/api and see if you get the no params message.

Retrieving data from the database

Now let's see how we could return a row from the products table in our store, based on a given id. We'll completely rewrite the view, so the previous code will be commented out.

```
from django.http import JsonResponse
from django.forms.models import model_to_dict
from store.models import Product

def api_home(request):
    params = dict(request.GET)
    # if params:
    #     message = 'Params: '
    #     for param_key, param_value in params.items():
    #         message += f"{param_key}: {param_value}, "
    #     return JsonResponse({"message": message})
    # else:
    #     return JsonResponse({"message": "Hmm, you didn't pass any params, did you?"})
    if params.get("pid"):
        try:
            id = int(params["pid"][0])
        except Exception:
            return JsonResponse({"detail": "Sorry, the product id must be a number."})
        try:
            product = Product.objects.get(pk=id)
        except Exception:
            return JsonResponse({"detail": "Sorry, a product with that id doesn't exist."})
        data = model_to_dict(product, fields=["id", "name", "price"])
        return JsonResponse({"product": data})
    else:
        return JsonResponse({"detail": "Hmm, you didn't pass the product id, did you?"})
```

Test

1. Go to http://127.0.0.1:8000/api?pid=1 and see if you get the product info.
2. Go to http://127.0.0.1:8000/api?pid=9999 and see if you get the product doesn't exist message.
3. Go to http://127.0.0.1:8000/api?pid=abc and see if you get the product id must be a number message.

4. Go to http://127.0.0.1:8000/api and see if you get the no params message.

Introducing the Django Rest Framework

The code above is totally fine, but let's see what advantages the Django Rest Framework (DRF) brings us.

DRF install

```
poetry add djangorestframework
```

Add the new app to the `settings.py` file:

```
INSTALLED_APPS = [
    # "django.contrib.admin",
    # "django.contrib.auth",
    # "django.contrib.contenttypes",
    # "django.contrib.sessions",
    # "django.contrib.messages",
    # "django.contrib.staticfiles",
    # "store.apps.StoreConfig",
    # "api",
    "rest_framework",
]
```

To use the DRF, we just need some small changes in our view:

```
# from django.http import JsonResponse # this gets replaced by:
from rest_framework.response import Response
# from django.forms.models import model_to_dict
# from store.models import Product

from rest_framework.decorators import api_view

@api_view(['GET'])
# def api_home(request):
#     params = dict(request.GET)
#     if params.get("pid"):
#         try:
#             id = int(params["pid"][0])
#         except Exception:
            return Response({"detail": "Sorry, the product id must be a number."})
#         try:
#             product = Product.objects.get(pk=id)
#         except Exception:
            return Response({"detail": "Sorry, a product with that id doesn't exist."})
#         data = model_to_dict(product, fields=["id", "name", "price"])
        return Response({"product": data})
#     else:
        return Response({"detail": "Hmm, you didn't pass the product id, did you?"})
```

Test

Perform the same tests as before and you should see the same results but in a totally new way!

1. Go to http://127.0.0.1:8000/api?pid=1 and see if you get the product info.
2. Go to http://127.0.0.1:8000/api?pid=9999 and see if you get the product doesn't exist message.
3. Go to http://127.0.0.1:8000/api?pid=abc and see if you get the product id must be a number message.

4. Go to http://127.0.0.1:8000/api and see if you get the no params message.

Using DRF's model serializers instead of `model_to_dict`

Remember that we created the property `image_url` in the `Product` model? Well, if you're using `model_to_dict` you'll never be able to access the property, obviously because it's not part of the model (there are ways to get it though).

With model serializers this will be possible and much easier. Let's create a serializer?

Create a new file inside the `store` app called `serializers.py` and type the following:

```
from rest_framework import serializers
from .models import Product


class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = ["id", "name", "price", "image_url"]
```

And now back in our view make the following changes:

```
# from rest_framework.response import Response
# from django.forms.models import model_to_dict
# from store.models import Product
#
# from rest_framework.decorators import api_view
from store.serializers import ProductSerializer
#
# @api_view(['GET'])
# def api_home(request):
#     params = dict(request.GET)
#     if params.get("pid"):
#         try:
#             id = int(params["pid"][0])
#         except Exception:
#             return Response({"detail": "Sorry, the product id must be a number."})
#         try:
#             product = Product.objects.get(pk=id)
#         except Exception:
#             return Response({"detail": "Sorry, a product with that id doesn't exist."})
          data = ProductSerializer(product).data
#         return Response({"product": data})
#     else:
#         return Response({"detail": "Hmm, you didn't pass the product id, did you?"})
```

Test

1. Go to http://127.0.0.1:8000/api?pid=1 and see if you get the image_url property.

Sending a post request

To better test post requests, we'll create a very basic python client to do those requests. For that we'll use a package called `requests` , so we'll have to start off by installing it:

```
poetry install requests
```

Now create a `py_client.py` file anywhere inside your project (for example, next to your `manage.py` file), and insert the following:

```
import requests

endpoint = "http://127.0.0.1:8000/api/"
response = requests.post(endpoint, json={"name": "Watch"})

print(response.json())
```

Now in the `views.py` file, create a new endpoint:

```
@api_view(["POST"])
def api_home(request):
    data = request.data
    return Response(data)
```

In a terminal, browse to the folder of the script, and run:

```
python py_client.py
```

You should see the output: `{"name":"Watch"}`

Using the serializer to validade the data

Serializers aren't only used to build a response model, they can also be used as validators. Let's see how to do that.

Change the post view as follows:

```
def api_home(request):
    serializer = ProductSerializer(data=request.data)
    if serializer.is_valid():
        return Response(serializer.data)
    else:
        return Response(None)
```

Nothing is being returned, why is that? The serializer is not valid because we're only passing the title attribute, and our product model specifies that the price attribute is also required (check the `models.py` file in the `store` app to confirm).

So if we change our python client to what's below and run it again:

```
response = requests.post(endpoint, json={"name": "Watch", "price": "99,99"})
```

It still doesn't work. Why? To know why, let's ask for the errors found in the serializer. Add the following line in the view:

```
# def api_home(request):
#     serializer = ProductSerializer(data=request.data)
#     if serializer.is_valid():
#         return Response(serializer.data)
#     else:
          print(serializer.errors)
#         return Response(None)
```

And there's the culprit:

> {'price': [ErrorDetail(string='A valid number is required.', code='invalid')]}

Ok, let's change change our python client to what's below and run it again:

```
response = requests.post(endpoint, json={"name": "Watch", "price": 99.99})
```

Hooray! But we can do even better and automatically return the errors if they exist, so that we can delete the entire `else` clause:

```
# def api_home(request):
      serializer = ProductSerializer(data=request.data, raise_exception=True)
#     if serializer.is_valid():
#         return Response(serializer.data)
```

Generic API Views - RetrieveAPIView

DRF has a series of generic views that you can use instead of building your own. We'll take a look for starters at the RetrieveAPIView.

So imagine that you want to show a product, and you want to get that product by it's `id` .

First let's create our view in the `views.py` file:

```
...
from rest_framework import generics

...

class ProductDetailAPIView(generics.RetrieveAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

Now let's add it to our `urls.py` file:

```
...
from .views import ProductDetailAPIView

#urlpatterns = [
#    path("", views.api_home),
    path("products/<int:pk>/", ProductDetailAPIView.as_view()),
#]
```

This means that we want to retrieve a product by using:http://127.0.0.1/api/products/

Ok, so let's change our python client to test this:

```
endpoint = "http://127.0.0.1:8000/api/products/1"
response = requests.get(endpoint)
```

It worked: `{'id': 1, 'name': 'Headphones', 'price': 179.99, 'image_url': '/images/headphones.jpg'}`

Test

1. Go ahead and run the same endpoint in your browser to see ithttp://127.0.0.1:8000/api/products/1
2. Also try with an inexistent id, say 999. You'll notice a built-in 'Not Found' view.

Generic API Views - CreateAPIView

So we've seen how we can leverage the generic api views to get data. Now let see how we can insert data.

First let's create our view in the `views.py` file:

```
...

class ProductCreateAPIView(generics.CreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

As you can see, the code is exactly the same as the previous view.

Now let's add it to our `urls.py` file:

```
...
from .views import ProductDetailAPIView, ProductCreateAPIView

#urlpatterns = [
#    path("", views.api_home),
#    path("products/<int:pk>/", ProductDetailAPIView.as_view()),
    path("products/", ProductDetailAPIView.as_view()),
#]
```

This means that we want to retrieve a product by using:http://127.0.0.1/api/products/

Ok, so let's change our python client to test this:

```
endpoint = "http://127.0.0.1:8000/api/products/"
response = requests.get(endpoint)
```

Test

1. Try to run the code; you'll get a 405 error saying the method is not allowed. Makes sense, if we want to create something we have to pass in the data, so it must be a POST request. Change it in the python client.
2. Using the POST method, you'll get another error saying some fields are required. Again, it makes sense. Change the client again to this:

```
endpoint = "http://127.0.0.1:8000/api/products/"
data = { "name": "Skateboard", "price": 49 }
response = requests.post(endpoint, json=data)
```

There you have it, a new product has been inserted: `{'id': 7, 'name': 'Skateboard', 'price': 49.0,` `'image_url': '/images/placeholder.png'}`

1. Try to run this endpoint on the browser as well to see what it happens.

Imagine that we needed to pass in more data, such as the user that is creating the new product, or we need access to the incoming data. The `CreateAPIView` has a method called `perform_create()` to help you achieve that (the code below is just demo, we won't be using it):

```
...

class ProductCreateAPIView(generics.CreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer

    def perform_create(self, serializer):
        # get the user info
        user = self.request.user
        # check if price is negative
        price = serializer.validated_data.get('price')
        if price < 0:
            price = 0
        serializer.save(user=user, price=price)
```

Generic API Views - ListAPIView and ListCreateAPIView

We're not going to use the `ListAPIView`, since we can join it together with the creation process using the `ListCreateAPIView`. So `ListAPIView` will be a GET request to list all rows of a model. `ListCreateAPIView`, or the other hand, supports GET and POST requests made to the same endpoint; if it's GET it lists, if it's POST it creates.

First let's create our view in the `views.py` file:

```
...

class ProductListCreateAPIView(generics.ListCreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

As you can see, the code is exactly the same as the previous view.

Now let's add it to our `urls.py` file and replace the `products/` endpoint:

```
...
from .views import ProductDetailAPIView, ProductCreateAPIView, ProductListCreateAPIView

#urlpatterns = [
#    path("", views.api_home),
#    path("products/<int:pk>/", ProductDetailAPIView.as_view()),
    path("products/", ProductListCreateAPIView.as_view()),
#]
```

Test

1. If you try it in the browser (GET method), you'll get a list of the products and a form to insert new ones.
2. If you use the python client (POST method), it will insert a new product.

Generic API Views - UpdateAPIView and DestroyAPIView

Let's update and delete the Skateboard product we inserted earlier using the `UpdateAPIView` and `DestroyAPIView`.

First let's create our views in the `views.py` file:

```
...

class ProductUpdateAPIView(generics.UpdateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    lookup_field = "pk"

class ProductDestroyAPIView(generics.DestroyAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

Now let's add them to our `urls.py` file:

```
...
from .views import (
    ProductDetailAPIView,
    ProductListCreateAPIView,
    ProductUpdateAPIView,
    ProductDestroyAPIView,
)

#urlpatterns = [
#    path("", views.api_home),
#    path("products/<int:pk>/", ProductDetailAPIView.as_view()),
    path("products/<int:pk>/update/", ProductUpdateAPIView.as_view()),
    path("products/<int:pk>/delete/", ProductDestroyAPIView.as_view()),
#    path("products/", ProductListCreateAPIView.as_view()),
#]
```

To update, change the python client to:

```
endpoint = "http://127.0.0.1:8000/api/products/7/update/"
data = {"name": "Skateboard video", "price": 49}
response = requests.put(endpoint, json=data)
```

To delete:

```
endpoint = "http://127.0.0.1:8000/api/products/7/delete/"
response = requests.delete(endpoint)

print(response.status_code) # if successfull, this will be 204
```

Session authentication and permissions

The API we've been building is working correctly, but it's also totally public and open to everyone, and that's not a good thing in the majority of situations. Let's see what DRF has baked in for us in terms of authentication and permissions.

Let's start off with a quick example. Image that you want to only allow access to an endpoint for registered users, for example the `products/` endpoint.

Make the following changes in the `views.py` file:

```
...
from rest_framework import generics, permissions

...

#class ProductListCreateAPIView(generics.ListCreateAPIView):
#    queryset = Product.objects.all()
#    serializer_class = ProductSerializer
    permission_classes = [permissions.IsAuthenticated]
...
```

Now if you try to reach that endpoint in the browser you'll get `"detail": "Authentication credentials were not provided."`

So how do we add authentication to our API? Add this to the `views.py` file:

```
...
from rest_framework import generics, permissions, authentication

...

#class ProductListCreateAPIView(generics.ListCreateAPIView):
#    queryset = Product.objects.all()
#    serializer_class = ProductSerializer
    authentication_classes = [authentication.SessionAuthentication]
#    permission_classes = [permissions.IsAuthenticated]
...
```

Now go back to the browser and click the login button in the upper right corner. The credentials are still the same:

```
Username: superuser
Password: #imtheBOSS!
```

And there you go, now you can see the data again, because you're authenticated.

User and Group permissions using DjangoModelPermissions

Start by creating a new user in the admin panel with these credentials:

```
Username: api_user
Password: #imtheBOSS!
```

Then edit the user and give him the `Staff status` permission.

Now if you log in with this user (use an incognito tab in your browser to keep both user sessions open), you'll notice that he doesn't see anything.

Edit this user again using the admin account. Go to `User permissions` and add `store|product|Can view product`. Now your `api_user` can view the products.

Now let's a new group called `StaffProductEditor` that can add and change products. With the admin account go to `Groups` and create it.

Now go to `Users` and add the `api_user` to that group.

Ok, now let's use this configurations in our api. Back in `views.py` :

```
#class ProductListCreateAPIView(generics.ListCreateAPIView):
#    queryset = Product.objects.all()
#    serializer_class = ProductSerializer
#    authentication_classes = [authentication.SessionAuthentication]
    permission_classes = [permissions.DjangoModelPermissions]
```

Try to reach the `products` endpoint, and everything looks the same. But now remove the `StaffProductEditor` group from the user and refresh. Now you can see the list of products but the form in the bottom is gone, since the user doesn't have permission to add new products.

Global permissions

So everything seems to be working, right? Actually it's not. If the `api_user` tries to reach the `products/1/update` endpoint he'll be able to update to update the product, even not having permission to do so.

Why? Because we've only declared the use of DjangoModelPermissions in one view (`ProductListCreateAPIView`), so all the other views are still public. We could of course set the permissions on all views, but what we'll do right now is create a custom permission and set it as the global permission for all models we may use in our views.

In the `api` app folder create a new `permissions.py` file:

```
from rest_framework import permissions


class IsStaffEditorPermission(permissions.DjangoModelPermissions):
    perms_map = {
        "GET": ["%(app_label)s.view_%(model_name)s"],
        "OPTIONS": [],
        "HEAD": [],
        "POST": ["%(app_label)s.add_%(model_name)s"],
        "PUT": ["%(app_label)s.change_%(model_name)s"],
        "PATCH": ["%(app_label)s.change_%(model_name)s"],
        "DELETE": ["%(app_label)s.delete_%(model_name)s"],
    }

    def has_permission(self, request, view):
        user = request.user
        if not user.is_staff:
            return False

        return super().has_permission(request, view)
```

Now let's bring this permission to our `views.py` :

```
...
#from rest_framework import generics, permissions, authentication
from .permissions import IsStaffEditorPermission
...
#class ProductListCreateAPIView(generics.ListCreateAPIView):
#    queryset = Product.objects.all()
#    serializer_class = ProductSerializer
#    authentication_classes = [authentication.SessionAuthentication]
    permission_classes = [IsStaffEditorPermission]
...
```

Before performing the tests below, make sure that `api_user` only has the permission to view products.

Test

1. Try the `products` endpoint. It should list but not have the form to add new products
2. Add the user again to the `StaffProductEditor` group, and reload the endpoint. You should see the form, since the group has permissions to add new products.
3. Remove the user from the `Staff status` permission. Since he's no longer part of the staff, he should not see anything in the endpoint.

The test #3 happens because of our code: `if not user.is_staff:`. We will refactor our code and achieve the same goal using built-in permissions.

In the `permissions.py` file:

```
...
class IsStaffEditorPermission(permissions.DjangoModelPermissions):
    ...

    # this method is no longer needed
    # def has_permission(self, request, view):
    #     user = request.user
    #     if not user.is_staff:
    #         return False

    #    return super().has_permission(request, view)
```

In the `views.py` file:

```
...
#class ProductListCreateAPIView(generics.ListCreateAPIView):
#    queryset = Product.objects.all()
#    serializer_class = ProductSerializer
#    authentication_classes = [authentication.SessionAuthentication]
    permission_classes = [permissions.IsAdminUser, IsStaffEditorPermission]
...
```

> The order of the permissions matter, they should go from the most broad to the most fine grained.

Test

Just perform the last tests in reverse, and everything will work as expected.

Tokens

However, if you try to reach the same endpoint using the python client, with the following code:

```
endpoint = "http://127.0.0.1:8000/api/products/"
response = requests.get(endpoint)
print(response.json())
```

You'll still get the authentication issue because the client doesn't know anything about the browser's session. This is where token authentication comes in.

First, let's edit the `settings.py` file:

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

This change needs to create new models in the database, so let's run the migrations:

```
python manage.py migrate
```

If you go to the admin area you'll now see an `AUTH TOKEN` section where you could generate tokens for any given user.

Let's change the `urls.py` file:

```
#from django.urls import path
from rest_framework.authtoken.views import obtain_auth_token

...

#urlpatterns = [
#     path("", views.api_home),
#     path("products/<int:pk>/", ProductDetailAPIView.as_view()),
#     path("products/<int:pk>/update/", ProductUpdateAPIView.as_view()),
#     path("products/<int:pk>/delete/", ProductDestroyAPIView.as_view()),
#     path("products/", ProductListCreateAPIView.as_view()),
     path("auth/", obtain_auth_token),
#]
```

And add this type of authentication to our `views.py` :

```
#class ProductListCreateAPIView(generics.ListCreateAPIView):
#     queryset = Product.objects.all()
#     serializer_class = ProductSerializer
    authentication_classes = [
        authentication.SessionAuthentication,
        authentication.TokenAuthentication
    ]
#     permission_classes = [permissions.IsAdminUser, IsStaffEditorPermission]
```

Test

To test it, change the python client to:

```
auth_endpoint = "http://127.0.0.1:8000/api/auth/"
auth_response = requests.post(
    auth_endpoint, json={"username": "api_user", "password": "#imtheBOSS!"}
)

print(auth_response.json())

if auth_response.status_code == 200:
    token = auth_response.json()["token"]
    headers = {"Authorization": f"Token {token}"}
    endpoint = "http://127.0.0.1:8000/api/products/"
    response = requests.get(endpoint, headers=headers)
    print(response.json())
```

And you should see the list of products in the terminal.

1. Try to change the method to POST: `response = requests.get(endpoint, headers=headers)` . You should see a permission error.
2. But if you add the user back to the `StaffProductEditor` group, it will work.

Global permissions revisited

Now that we have both authentication and permission classes done, we want to set them as default for all views, so that we don't have to replicate the code below on all views.

```
authentication_classes = [
    authentication.SessionAuthentication,
    authentication.TokenAuthentication
]
permission_classes = [permissions.IsAdminUser, IsStaffEditorPermission]
```

We do that by adding our classes to the `settings.py` file:

```
REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": [
        "rest_framework.authentication.SessionAuthentication",
        "rest_framework.authentication.TokenAuthentication",
    ],
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.IsAuthenticatedOrReadOnly"
    ],
}
```

We've given a different default permission class because the classes we're using above are too restrictive. Better to keep the default less restrictive and change only the views that are different.

So going back to the `views.py` , we can now delete this:

```
authentication_classes = [
    authentication.SessionAuthentication,
    authentication.TokenAuthentication
]
```

So now we should copy `permission_classes = [permissions.IsAdminUser, IsStaffEditorPermission]` to all our views, since we we don't want just the default, right? We could, but let's take a look at Mixins and solve it that way.

Create a new file in the `api` app called `mixins.py` :

```
from rest_framework import permissions
from .permissions import IsStaffEditorPermission

class StaffEditorPermissionMixin:
    permission_classes = [permissions.IsAdminUser, IsStaffEditorPermission]
```

And now, in every view of the `views.py` file, just add the mixin. Here's an example using the `ProductDetailAPIView` :

```
from .mixins import StaffEditorPermissionMixin

class ProductDetailAPIView(generics.RetrieveAPIView, StaffEditorPermissionMixin):
```

Don't forget to delete the `permission_classes` variable from the `ProductListCreateAPIView` view.

Url's, reverse and serializers

Imagine that you want to add an url to the detail of a product and another url to link to the product image instead of the real value in the database. Here's how you could do it.

First, change the `serializers.py` file to the following:

```
...
#class ProductSerializer(serializers.ModelSerializer):
    detail_url = serializers.HyperlinkedIdentityField(
        view_name="product-detail", lookup_field="pk"
    )
    image_url = serializers.SerializerMethodField(read_only=True)

#    class Meta:
#        model = Product
        fields = ["id", "name", "price", "image_url", "detail_url"]

    def get_image_url(self, obj):
        request = self.context.get("request")
        if request is None:
            return None
        return request.build_absolute_uri(obj.image_url)
```

Now in the `urls.py` file, we have to add the `name` to the view:

```
urlpatterns = [
#    path("", views.api_home),
    path("products/<int:pk>/", ProductDetailAPIView.as_view(), name="product-detail"),
#    path("products/<int:pk>/update/", ProductUpdateAPIView.as_view()),
#    path("products/<int:pk>/delete/", ProductDestroyAPIView.as_view()),
#    path("products/", ProductListCreateAPIView.as_view()),
#    path("auth/", obtain_auth_token),
]
```

Test

Go to the `products` endpoint. You should see in every product something like `"detail_url": "http://127.0.0.1:8000/api/products/1/"`

Custom validation in serializers

Besides all the automatic validations that will be made according to the model information, very often custom validations need to be in place. For instance, let's pretend that the product title cannot contain bad words such as "stupid":

Every field that needs to be validated has to have it's own method with the name `validade_` +field_name.

So back in our `serializers.py` :

```
...
#class ProductSerializer(serializers.ModelSerializer):
    ...
    def validate_name(self, value):
        stop_words = ["stupid", "free", "new"]
        for word in value.split():
            if word.lower() in stop_words:
                raise serializers.ValidationError(
                    f"The title cannot contain words such as '{word}'"
                )
```

Pagination

It's always a good practice to paginate results, it's less weight for the database and a better user experience as well. We'll add pagination to our views in globally, so that all the api benefits from it.

Just go to `settings.py` and add the following:

```
REST_FRAMEWORK = {
#     "DEFAULT_PERMISSION_CLASSES": [
#         "rest_framework.permissions.IsAuthenticatedOrReadOnly"
#     ],
     "DEFAULT_PAGINATION_CLASS": "rest_framework.pagination.LimitOffsetPagination",
     "PAGE_SIZE": 5
}
```

That's great, but notice that the user can still change the parameters as he wishes simply by changing the url parameters `limit=5&offset=5` to whatever values they wish.

There are two ways to restrict this:

1. Simply change the `DEFAULT_PAGINATION_CLASS` to `PageNumberPagination` instead of `LimitOffsetPagination`
2. Write your own implementation of the `LimitOffsetPagination` and use the `max_limit` attribute, like this:

Create a new `pagination.py` in the `api` app folder and type:

```
from rest_framework.pagination import LimitOffsetPagination

class CustomLimitOffsetPagination(LimitOffsetPagination):
    max_limit = 5
```

Now return to `settings.py` and change the `DEFAULT_PAGINATION_CLASS` to `api.pagination.CustomLimitOffsetPagination` .

How about changing the response format? By default we have:

```
{
    "count": 7,
    "next": "http://127.0.0.1:8000/api/products/?limit=5&offset=5",
    "previous": null,
    "results": []
}
```

What if we wanted:

```
{
    "total": 7,
    "data": [],
    "links": {
        "previous": null,
        "next": "http://127.0.0.1:8000/api/products/?limit=5&offset=5",
    }
}
```

Here's how (`pagination.py`):

```
#from rest_framework.pagination import LimitOffsetPagination
from rest_framework.response import Response


class CustomLimitOffsetPagination(LimitOffsetPagination):
#    max_limit = 5

    def get_paginated_response(self, data):
        return Response(
            {
                "total": self.count,
                "data": data,
                "links": {
                    "previous": self.get_previous_link(),
                    "next": self.get_next_link(),
                },
            }
        )
```

JSON Web Token Authentication

So far we've used a very simple python client to test our api, but in the real world we'll need a proper and robust client for others to consume our api.

We won't be using the session authentication nor the token authentication we've built before, but instead JSON Web Tokens (JWT). JWT is a fairly new standard which can be used for token-based authentication. Unlike the built-in TokenAuthentication scheme, JWT Authentication doesn't need to use a database to validate a token, because it encodes data needed for authentication in the tokens themselves. Your app will be able to authenticate users after decoding tokens with data embedded in it.

We'll use the djangorestframework-simplejwt package since it already has a lot of the logic built-in.

```
poetry add djangorestframework-simplejwt
```

Now add the package to installed apps in `settings.py` :

```
INSTALLED_APPS = [
    ...
#    "api",
#    "rest_framework",
#    "rest_framework.authtoken",
    "rest_framework_simplejwt",
]
```

And to the rest framework settings also in `settings.py` :

```
REST_FRAMEWORK = {
#    "DEFAULT_AUTHENTICATION_CLASSES": [
#        "rest_framework.authentication.SessionAuthentication",
#        "rest_framework.authentication.TokenAuthentication",
        "rest_framework_simplejwt.authentication.JWTAuthentication",
    ],
    ...
}
```

And while we're here, add this to the beginning of the file:

```
import datetime
```

And these to the end of the file:

```
SIMPLE_JWT = {
    "ACCESS_TOKEN_LIFETIME": datetime.timedelta(seconds=30),
    "REFRESH_TOKEN_LIFETIME": datetime.timedelta(minutes=1),
}
```

Finally, we need to add the url paths to our `urls.py`

```
...
from rest_framework_simplejwt.views import (
    TokenObtainPairView,
    TokenRefreshView,
    TokenVerifyView,
)

urlpatterns = [
    ...
#    path("auth/", obtain_auth_token),
    path("token/", TokenObtainPairView.as_view(), name="token_obtain_pair"),
    path("token/refresh/", TokenRefreshView.as_view(), name="token_refresh"),
    path("token/verify/", TokenVerifyView.as_view(), name="token_verify"),
]
```

Ok, now let's take a look at our JWT client. Create a new file called `jwt_client.py` next to `manage.py` and `py_client.py` .

Start off by importing all the packages we need:

```python
from dataclasses import dataclass
import requests
from getpass import getpass
import pathlib
import json
```

Now this first block is a class that contains the whole logic of our client:

- Authorization (login with user/password to retrieve tokens)
- Authentication (verifying the token and if invalid refreshing it or asking for a new one)
- Saving tokens in a file
- Clearing tokens from file
- A request to an endpoint in our API

```python
@dataclass
class JWTClient:
    """
    Use a dataclass decorator to simplify the class construction
    """
    access: str = None
    refresh: str = None
    header_type: str = "Bearer"
    # this assumes you have DRF running on 127.0.0.1:8000
    base_endpoint: str = "http://127.0.0.1:8000/api"
    # the credentials file
    cred_path: pathlib.Path = pathlib.Path("creds.json")

    def __post_init__(self):
        if self.cred_path.exists():
            """
            You have stored creds, let's verify them and refresh them.
            If that fails, restart login process.
            """
            try:
                data = json.loads(self.cred_path.read_text())
            except Exception:
                print("Assuming creds has been tampered with")
                data = None
            if data is None:
                """
                Clear stored creds and run login process
                """
                self.clear_tokens()
                self.perform_auth()
            else:
                """
                `creds.json` was not tampered with
                Verify token -> if necessary, Refresh token -> if necessary, Run login process
                """
                self.access = data.get("access")
                self.refresh = data.get("refresh")
                token_verified = self.verify_token()
                if not token_verified:
                    """
                    This can mean the token has expired or is invalid.
                    Either way, attempt a refresh.
                    """
                    refreshed = self.perform_refresh()
                    if not refreshed:
                        """
                        This means the token refresh also failed. Run login process
                        """
                        print("invalid data, login again.")
                        self.clear_tokens()
                        self.perform_auth()
        else:
            """
            Run login process
            """
            self.perform_auth()

    def get_headers(self, header_type=None):
        """
        Default headers for HTTP requests including the JWT token
        """
        _type = header_type or self.header_type
        token = self.access
        if not token:
            return {}
        return {"Authorization": f"{_type} {token}"}

    def perform_auth(self):
        """
        Simple way to perform authentication Without exposing
        password during the collection process.
        """
        endpoint = f"{self.base_endpoint}/token/"
        username = input("What is your username? (type 'api_user')\n")
        password = getpass("What is your password? (type '#imtheBOSS!')\n")
        r = requests.post(endpoint, json={"username": username, "password": password})
        if r.status_code != 200:
            raise Exception(f"Access not granted: {r.text}")
        print("access granted")
        self.write_creds(r.json())

    def write_creds(self, data: dict):
        """
        Store credentials as a local file and update instance with correct data.
        """
        if self.cred_path is not None:
            self.access = data.get("access")
            self.refresh = data.get("refresh")
            if self.access and self.refresh:
                self.cred_path.write_text(json.dumps(data))

    def verify_token(self):
        """
        Simple method for verifying your token data.
        This method only verifies your access token.
        A 200 HTTP status means success, anything else means failure.
        """
        data = {"token": f"{self.access}"}
        endpoint = f"{self.base_endpoint}/token/verify/"
        r = requests.post(endpoint, json=data)
        return r.status_code == 200

    def clear_tokens(self):
        """
        Remove any/all JWT token data from instance as well as stored creds file.
        """
        self.access = None
        self.refresh = None
        if self.cred_path.exists():
            self.cred_path.unlink()

    def perform_refresh(self):
        """
        Refresh the access token by using the correct auth headers and the refresh token
        Refresh the access token by using the correct auth headers and the refresh token.
        """
        print("Refreshing token.")
        headers = self.get_headers()
        data = {"refresh": f"{self.refresh}"}
        endpoint = f"{self.base_endpoint}/token/refresh/"
        r = requests.post(endpoint, json=data, headers=headers)
        if r.status_code != 200:
            self.clear_tokens()
            return False
        refresh_data = r.json()
        if not "access" in refresh_data:
            self.clear_tokens()
            return False
        stored_data = {"access": refresh_data.get("access"), "refresh": self.refresh}
        self.write_creds(stored_data)
        return True

    def list(self, endpoint=None, limit=5):
        """
        Here is an actual api call to a DRF View that requires
        our simplejwt authentication to be working correctly.
        """
        headers = self.get_headers()
        if endpoint is None or self.base_endpoint not in str(endpoint):
            endpoint = f"{self.base_endpoint}/products/?limit={limit}"
        r = requests.get(endpoint, headers=headers)
        if r.status_code != 200:
            raise Exception(f"Request not complete {r.text}")
        data = r.json()
        return data
```

The second block is just a simple program to demo the client:

```
if __name__ == "__main__":
    """
    Here's a simple example of how to use our client above.
    """

    # this will either prompt a login process
    # or just run with current stored data
    client = JWTClient()

    # simple instance method to perform an HTTP
    # request to our /api/products/ endpoint
    lookup_1_data = client.list(limit=5)
    # We used pagination at our endpoint so we have:
    results = lookup_1_data.get("data")
    next_url = lookup_1_data.get("links").get("next")
    print("First lookup result length", len(results))
    if next_url:
        lookup_2_data = client.list(endpoint=next_url)
        results += lookup_2_data.get("data")
        print("Second lookup result length", len(results))
```

Test

1. Run this file on a terminal and check the results
2. Wait a minute and try again. You should not longer have valid tokens and be asked to login again (because of the lifetime settings in `settings.py`).

> If you whish to debug the client's code you can't use the Django debugger we've set up. With the

> `jwt_client.py` file open, press `CONTROL+SHIFT+p` and search for `Python: Debug Python File`. A new terminal will start and run the file with debugging enabled.

Javascript client

The purpose of this section is to show how we can create a web client to access our api.

Create a folder `js_client` inside the `ecommerce-app` folder (alongside our `ecommerce` Django project).

Inside that folder create a `index.html` file and a `client.js` file.

Add this to the `index.html` file:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>API Client</title>
</head>
<body>
    <form id="login-form" method="post">
        <input type="text" name="username" placeholder="Your username">
        <input type="password" name="password" placeholder="Your password">
        <input type="submit" value="Login">
    </form>

    <script src="./client.js"></script>
</body>
</html>
```

Test

1. Open a new terminal and browse to the `js_client` folder.
2. Type `python -m http.server 8111`
3. Open http://127.0.0.1:8111 in the browser. You should see the form.

Now let's write the javascript client:

```
const loginForm = document.getElementById('login-form')
const baseEndpoint = "http://127.0.0.1:8000/api"

if(loginForm) {
    loginForm.addEventListener('submit', handleLogin)
}

function handleLogin(event) {
    event.preventDefault()
    // get data from the form and prepare it to be sent
    let loginFormData = new FormData(loginForm)
    let loginObjectData = Object.fromEntries(loginFormData)
    console.log(loginObjectData)
    let bodyStr = JSON.stringify(loginObjectData)
    console.log(bodyStr)
    // Set up the request details
    const loginEndpoint = `${baseEndpoint}/token/`
    const options = {
        method: "POST",
        headers: {
            "Content-Type": "application/json"
        },
        body: bodyStr
    }
    // Make the request
    fetch(loginEndpoint, options)
    // Handle the response
    .then(response => {
        console.log(response)
        return response.json()
    })
    .then(x => {
        console.log(x)
    })
}
```

Test

1. Restart the server in the terminal by pressing `CONTROL+c` and then calling again `python -m http.server 8111`
2. In the browser, open `Developer Tools` in the `Console` tab.
3. Refresh the page, fill the form and submit.
4. If you look in the console, you'll see an error: Access to fetch at 'http://127.0.0.1:8000/api/token/' from origin 'http://127.0.0.1:8111' has been blocked by CORS policy.

That is a built-in security feature in Django and other frameworks, that by default don't allow POST methods coming from other adresses. We already saw it in the `settings.py` file: **ALLOWED_HOSTS = []**

There's a package for Django that will help us handle these CORS (Cross Origin Resource Sharing) security features.

Install the package:

```
poetry add django-cors-headers
```

Add it to the installed apps in `settings.py` :

```
INSTALLED_APPS = [
     ...
#    "rest_framework.authtoken",
#    "rest_framework_simplejwt",
     "corsheaders",
]
```

Still in `settings.py`, find middleware and the following **above** `"django.middleware.common.CommonMiddleware",` :

```
MIDDLEWARE = [
#    "django.middleware.security.SecurityMiddleware",
#    "django.contrib.sessions.middleware.SessionMiddleware",
     "corsheaders.middleware.CorsMiddleware",
#    "django.middleware.common.CommonMiddleware",
```

Finally, still in `settings.py`, let's define two new constants to allow requests to our api from an external adress:

```
# ROOT_URLCONF = "ecommerce.urls"
  CORS_URLS_REGEX = r"^/api/.*"
  CORS_ALLOWED_ORIGINS = ["http://127.0.0.1:8111", "https://127.0.0.1:8111"]
```

Test

1. Restart the server in the terminal by pressing `CONTROL+c` and then calling again `python -m http.server 8111`
2. In the browser, open `Developer Tools` in the `Console` tab.
3. Refresh the page, fill the form and submit.
4. If you look in the console, you should see your access and refresh tokens.

Using JWT with the Javascript client

Ok, now that we know that it works, let's continue and save those tokens.

Paste this function into the `client.js` file:

```
function handleAuthData(authData) {
    localStorage.setItem('access', authData.access)
    localStorage.setItem('refresh', authData.refresh)
}
```

This will save the tokens in the browser's storage. Now change the code in the `handleLogin` function to:

```
//    fetch(loginEndpoint, options)
//    .then(response => {
//        return response.json()
//    })
    .then(handleAuthData)
//}
```

Test

1. In the browser, open `Developer Tools` in the `Application` tab and open `Local Storage`.

2. Refresh the page, fill the form and submit.
3. If you look in the console, you should see your access and refresh tokens.

> Bear in mind that this type of storage might have security risks. Refer tothis article for more information on this subject.

It's time to create the request to list the products in the html page. In the `index.html` add this after the `form` and before the `script` tag:

```
<button class="get-products">List products</button>
<div id="content-container"></div>
```

Now add this code to the `client.js` file:

```
...
// const baseEndpoint = "http://127.0.0.1:8000/api"
const getBtn = document.querySelector('.get-products')
const contentContainer = document.getElementById('content-container')

// if(loginForm) {
//     loginForm.addEventListener('submit', handleLogin)
// }

if(getBtn) {
    getBtn.addEventListener('click', getProductList)
}

...

function getProductList() {
    const endpoint = `${baseEndpoint}/products/`
    const options = {
        method: "GET",
        headers: {
            "Content-Type": "application/json"
        }
    }
    fetch(endpoint, options)
        .then(response => {
            return response.json()
        })
        .then(data => {
            writeToContainer(data)
        })
}

function writeToContainer(data) {
    contentContainer.innerHTML = "<pre>" + JSON.stringify(data, null, 4) + "</pre>"
}
```

Test

1. Refresh the page and click the `List products` button. You should get the message `{"detail":"Authentication credentials were not provided."}`

That's happenning because we're not passing our token with the request. In the `getProductList` function add the this to the `headers` :

```
"Authorization": "Bearer abc"
```

Obviously, `abc` is an invalid token. Go ahead and try clicking the button again. Now there should be a total different message:

```
{
    "detail": "Given token not valid for any token type",
    "code": "token_not_valid",
    "messages": [
        {
            "token_class": "AccessToken",
            "token_type": "access",
            "message": "Token is invalid or expired"
        }
    ]
}
```

So now it has credentials but they're invalid. Let's provide the correct token from the `LocalStorage` :

```
"Authorization": `Bearer ${localStorage.getItem('access')}`
```

If we try again we still get the same error. Why? Because our tokens are expired (remember that we've set the access token lifetime for 30sec and the refresh for 1min in the `settings.py` file).

So this means that we have to deal with those rules as well in our code.

```
function getProductList() {
//    const endpoint = `${baseEndpoint}/products/`
//    const options = {
//        method: "GET",
//        headers: {
//            "Content-Type": "application/json",
//            "Authorization": `Bearer ${localStorage.getItem('access')}`
//        }
//    }
//    fetch(endpoint, options)
//        .then(response => {
//            return response.json()
//        })
//        .then(data => {
                if (data.code && data.code === 'token_not_valid') {
                    refreshToken()
                        .then(is_refreshed => {
                            if (!is_refreshed) {
                                contentContainer.innerHTML = "<pre>" +
                                    JSON.stringify({
                                        "detail": "Could not obtain token. Please login."
                                    }, null, 4) +
                                    "</pre>"
                                return false
                            } else {
                                getProductList()
                            }
                        })
                } else {
                    writeToContainer(data)
                }
//        })
}

...

function refreshToken() {
    return new Promise((resolve) => {
        const endpoint = `${baseEndpoint}/token/refresh/`
        const options = {
            method: "POST",
            headers: {
                "Content-Type": "application/json",
            },
            body: JSON.stringify({"refresh": localStorage.getItem('refresh')})
        }
        fetch(endpoint, options)
            .then(response => response.json())
            .then(data => {
                if(data.code && data.code === 'token_not_valid') {
                    resolve(0)
                } else {
                    handleAuthData(data)
                    resolve(1)
                }
            })
    })
}
```

Now if we try again, we'll get the proper message saying that we need to login again. To arrive to that message the code tried to use the access token (invalid), then tried to refresh it using the refresh token (also invalid), so the only option now is to get new tokens.

Test

Let's try a different scenario now, remembering the lifetime of our tokens (30sec for access, 1min for refresh). Use a stopwatch to check the time elapsed if needed.

1. Login to get new tokens and start the stopwatch.
2. Click the button to get the list of products.
3. When the stopwatch reaches the 35sec mark, try the button again. You should still see the product list, but by now the code refreshed the access token
4. Wait until you pass the 1min15sec mark. None of the tokens should work anymore, and you have to login again.

Generating documentation

It's paramount to your API to have documentation. It's where the developers using it can know what requests can me, what data has to be passed, etc.

There's a package that can help you have this docs automatically generated for you. Let's install it and see how it can help us.

```
poetry add drf-spectacular
```

Add it to the `INSTALLED_APPS` in `settings.py` :

```
INSTALLED_APPS = [
    ...
#    "rest_framework_simplejwt",
#    "corsheaders",
     "drf_spectacular",
]
```

And in the `REST_FRAMEWORK` also in `settings.py` :

```
REST_FRAMEWORK = {
    ...
#    "DEFAULT_PAGINATION_CLASS": "api.pagination.CustomLimitOffsetPagination",
#    "PAGE_SIZE": 5,
     "DEFAULT_SCHEMA_CLASS": "drf_spectacular.openapi.AutoSchema",
}
```

Finally, still in `settings.py` , we can add settings for the package itself:

```
SPECTACULAR_SETTINGS = {
    'TITLE': 'Products API',
    'DESCRIPTION': 'My first DRF API',
    'VERSION': '1.0.0',
    # OTHER SETTINGS
}
```

Now we have to add the package's urls to our api. In the `urls.py` inside the api app:

```
...
from drf_spectacular.views import (
    SpectacularAPIView,
    SpectacularRedocView,
    SpectacularSwaggerView,
)

urlpatterns = [
     ...
#    path("token/refresh/", TokenRefreshView.as_view(), name="token_refresh"),
#    path("token/verify/", TokenVerifyView.as_view(), name="token_verify"),
     path('schema/', SpectacularAPIView.as_view(), name='schema'),
     path('schema/swagger/', SpectacularSwaggerView.as_view(url_name='schema'), name='swagger'),
     path('schema/redoc/', SpectacularRedocView.as_view(url_name='schema'), name='redoc'),
]
```